Detection and Correction of Android-specific Code Smells and Energy Bugs: An Android Lint Extension

Iffat Fatima[a], Hina Anwar[b], Dietmar Pfahl[b] and Usman Qamar[a]

[a] *College of Electrical and Mechanical Engineering, National University of Sciences and Technology,Islamabad, Pakistan*
[b] *Institute of Computer Science, University of Tartu, Tartu, Estonia*

Abstract

Context: While Android applications suffer from code smells and energy drain issues there is still a lack of tools that help developers improve energy consumption and maintainability of Android applications. Objective: Our research aims to provide tool support to Android developers helping them to create greener and more maintainable applications by eliminating Android-specific code smells/energy bugs. The proposed tool support integrates routine code smell detection with energy bug detection so that developers can do both at the same time. Method: We extend 'Android Lint' (AL) with custom rules to detect and correct 12 code smells (nine are new and three are improved) and three energy bugs (two are new and one is improved). In addition, for the improved and newly introduced code smells, we compared the performance of our tool with the open version of the 'PAPRIKA' tool. Result: We evaluated our tool on nine open-source Android applications. Our tool detects the specified code smells and energy bugs with an average precision, average recall and F1 score of 0.93, 0.96, and 0.94, respectively. It accurately corrects 84% of selected code smells and energy bugs. The performance of the new and improved code smell detection is better than that achieved by 'PAPRIKA'. Conclusion: Our tool is a useful extension to the existing 'AL' tool with better performance than 'PAPRIKA'.

Keywords

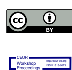
Green Software Development, Android, Energy Optimization, Code Smell, Energy Bug, Android Lint, Detection, Refactoring, Static Analysis

1. Introduction

Recently, the focus of research has shifted towards sustainable and green software development with a focus on energy optimized programming and energy optimization at the application level [1]. Software is now being built not only keeping performance, dependability, and maintainability in mind but also the principles of green software engineering aiming at the development of sustainable software with less negative impact on the environment. With the fast-paced emergence of mobile technologies in the past decade, mobile applications are being widely used. 3.5 billion people use smartphones around the world [2], and Android has 75% of the market share.

Code smells and energy bugs have been identified as causes of abnormal energy consumption in Android applications. Significant research has been carried out on the impact of object-oriented smells in Java applications [3, 4, 5, 6]. In our previous work [1] we identified support tools that aid green Android development. We further identified the coverage of code smells and energy bugs by those tools and identified their limitations. We concluded that there is a lack of guidelines for Android developers to write sustainable software. Current state of the art tools lack in providing complete coverage for Android code smells and energy bugs. Moreover, they lack in usability, IDE integration and effective refactoring approach. The aim of this research is to create a tool that solves these issues by aiding developer to solve energy related problems during development of the application for improved performance and maintainability.

'Android Lint' (AL) is the default static analysis tool in Android Studio IDE, hence used by most Android developers. Code smells detected and prioritized by 'AL' tend to disappear faster from code base as compared to other code smells detection tools. Moreover, a lint tool integrated in Android Studio IDE not only encourages the developers to correct code smells on the go but also plays a role in developer education [7]. We chose a custom implementation of 'AL' API to 1) maximize coverage of Android code smells/energy bugs, 2) provide recommendations to developers for refactoring, 3) provide a preview of the detected and corrected code, and

QuASoQ 2020: 8th International Workshop on Quantitative Approaches to Software Quality, December 1st, 2020, Singapore
iffat.fatima@ce.ceme.edu.pk (I. Fatima); hina.anwar@ut.ee (H. Anwar); dietmar.pfahl@ut.ee (D. Pfahl); usmanq@ceme.nust.edu.pk (U. Qamar)
0000-0002-4725-4636 (H. Anwar); 0000-0003-2400-501X (D. Pfahl)

© 2020 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-2767/09-QuASoQ-2020.pdf

4) provide an interface consistent with the Android Studio IDE.

The 'extended Android Lint' (xAL) tool is evaluated on open-source Android applications to detect and correct code smells and energy bugs. The current evaluation has resulted in average precision, average recall and F1 score of 0.93, 0.96, and 0.94, respectively. Whereas, 84% of the suggested corrections applied to the applications under test, resulted in the smooth functioning of applications. However, results cannot be generalized based on these statistics due to the small scale of the evaluation setup. Our tool also provides better code smell/energy bug coverage, and usability compared to 'PAPRIKA' tool.

Section 2 provides related work. Section 3 contains the tool design and implementation details. Section 4 presents the evaluation plan and results. Section 5 discusses threats to validity. Section 6 concludes the study.

2. Related Work

There are several publications in which Android application analysis tools have been presented that detect or correct Android-specific code smells and energy bugs. The aim of this study is to provide a solution at the development stage hence we look into only those tools that perform static analysis, with the aim to optimize applications in terms of their energy consumption.

The 'HOT-PEPPER' toolkit [8] (based on 'PAPRIKA' tool [9]) detected and refactored a set of Android-specific code smells and produced a corrected version of the APK without developer intervention.

'Statedroid' tool [10] performed a taint-like analysis using specified resource protocols to detect energy leaks caused by Wakelock Bugs and Resource Leaks.

In [11], the authors used a combination of 'Eclipse Refactoring API', 'PMD', and 'AL' to build a tool that optimizes Android applications for CPU usage. Their rule set covered only a limited number of Android code smells. However, the tool offered developers the flexibility to add their own rules.

The 'aDOCTOR' tool [12] detected 15 code smells causing energy drains by traversing the abstract syntax tree. The code smells were removed manually by authors and correlated to energy consumption. Energy estimation was done using 'PETrA'. The 'aDOCTOR' tool has a precision and recall of 98%.

Jiang et al.[13] used 'SAAF' for resource leak analysis and 'AL' for layout defect analysis in Android applications. They detected energy bugs like Camera Leak, Memory leak, Multimedia Leak, sensor Leak, and layout defects.

Olivier Le Goaër [14] presented an automated tool based on 'AL' which detected 11 energy greedy Android patterns such as Draw Allocation, Wakelock, Recycle, Obsolete Layout Parameter, HashMap Usage, Member Ignoring Method, Excessive Method Calls and some Resource Leaks. The tool 'AutoRefactor' was used for refactoring and the impact of those refactoring on energy consumption of open-source Android applications was measured.

The 'E-Debitum' [15] tool (based on 'SonarQube') detected six energy code smells and calculated their energy debt.

Comprehensive coverage of Android-specific code smells and energy bugs within one single tool is missing in existing tools. In the tools mentioned above, most commonly detected code smells were Member Ignoring Method, Internal Getter and Setter methods whereas most commonly detected energy bugs were Wakelock Bug and Resource Leaks. Existing tools have low usability due to lack of Android Studio IDE integration. Tools in studies [9, 8, 13] provided a command line interface while in the study [11] integration with Eclipse IDE was provided instead of Android Studio (which is the official IDE for Android development [7]). Tools compatible with Android Studio [14, 16], were not open source. Tools in studies [9, 13, 12, 11, 16] did not refactor applications while tools in studies [8, 15] provided completely automated refactoring hence reducing the control of the developer during refactoring process.

3. Design

In this section, we describe how we selected the baseline tool for extension and how we enhanced it.

3.1. Baseline Tool Selection

The aim of this study is to create a tool that covers the limitations of previously developed tools in terms of providing a comprehensive coverage of the Android code smells and energy bugs and solving the usability issues such as IDE integration, flexibility and ease of use for developer, open source availability etc. Based on this objective, we set the following criteria for selecting a tool for an extension:

- The tool should be open source or provide the ability to extend or customize it to add

rules for detection and refactoring of code smells/energy bugs.
- The tool should be able to perform static analysis.
- The tool should be integrate-able with Android studio IDE as it is the official IDE for Android development [7].
- The tool should provide an inline warning on code smell/energy bug detection inside Android Studio code editor.
- The tool should provide a mechanism to list detected code smells/energy bugs along with their description.

The above criteria were applied on industry-standard tools such as 'AL[1]' (AL), 'PMD', 'SpotBugs[2]' (SB), 'SonarLint[3]' (SL), 'SonarQube[4]' (SQ), and 'SOOT[5]' (ST) (see Table 1). A similar tool, 'FindBugs', was not considered as its successor is 'SpotBugs'. 'Eclipse refactoring engine' was also excluded as it is not integratable with Android Studio. Tools that perform static analysis but focus on code styling such as 'CheckStyle' were excluded. We also considered detection and optimization tools identified in our previous work [1]. Even though many of these tools were built on top of open-source static analysis tools such as 'SOOT', 'SPARK', 'SAAF', 'ASM', 'PMD', and 'Lint', we did not select them for an extension because they were not designed to be integrated with Android Studio IDE. The tools using dynamic analysis were also excluded as they require the application to be built every time. Long average build time for Android applications is a known and significant issue among the development community[6]. Table 1 shows a comparison of tools in terms of selection criteria. After this comparison, 'SpotBugs', 'SonarQube', and 'SOOT' were excluded as they built the application every time a code smell/energy bug needs to be detected.

Next, we compared the three shortlisted tools: 'AL', 'PMD', and 'SL' for Android-specific code smell and energy bug coverage (See Table 2 and 3). 'AL', 'PMD', and 'SonarLint' can cover many different types of issues in code (code smell, bug or error is referred to as 'issue' in these tools). For example, 'AL' can detect 261 different types of Android-specific issues[7]. Majority of these issues are related to syntax and styling of the code. We could

[1]http://tools.android.com/tips/lint-custom-rules
[2]https://spotbugs.github.io/
[3]https://www.sonarlint.org/features/
[4]https://docs.sonarqube.org/latest/extend
[5]https://github.com/Sable/soot/
[6]https://developer.android.com/studio/build/optimize-your-build
[7]http://tools.android.com/lint/overview

Table 1
Comparison of tools in terms of selection criteria

Criteria	AL	PMD	SB	SL	SQ	ST
Open Source	✓	✓	✓	✓	✓	✓
Customization API	✓	✓	X	✓	✓	X
Source Code Analysis	✓	✓	X	✓	X	X
Byte Code Analysis	✓	X	✓	X	✓	✓
Android Studio Integration	✓	✓	✓	✓	✓	X
Inline issue warning/hint	✓	X	X	✓	✓	X
List of detected code smells/energy bugs	✓	✓	✓	✓	✓	✓
Allows adding refactoring rules	✓	X	X	X	X	X

AL = Android Lint, SB = SpotBugs,SL= SonarLint, SQ = SonarQube, ST= SOOT

not find any evidence in the literature about the energy impact of the issues already covered by the above shortlisted tools, therefore, we only compared them for the coverage of 25 Android-specific code smells and nine energy bugs listed in [1].

In Tables 2 and 3, '*' represents that the code smell/energy bug is detected but based on the definition of code smell/energy bug[8] the detection coverage has room for improvement. ✓represents that code smell/energy bug is detected, × represents that code smell/energy bug is not covered by the tool yet. From Tables 2 and 3, we can see that 'AL' already covers 13 code smells and six energy bugs. Therefore, an effort towards improvement in the small number of undetected code smells/energy bugs will result in a single tool with maximum coverage. In addition, 'AL' provides offline documentation for rules and allows the developer to choose whether to correct a specific code smell/energy bug or not. Based on the above data, 'AL' is a feasible tool for an extension.

3.2. Android Lint Extension

In this section, we explain the 'AL' API and implementation details of our new 'extended Android Lint' (xAL) tool.

API Overview. 'AL' provides an embedding API that allows adding custom rules. In order to create custom rules, the 'AL' embedding API pro-

[8]https://figshare.com/s/84ae49a21551e6302d41

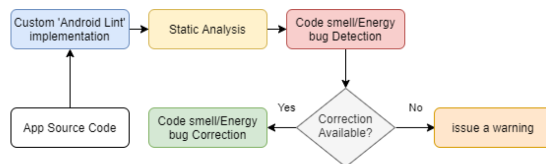


Figure 1: Overview of 'AL' API

Table 2
Code smell coverage by tools ‘AL’, ‘PMD’ and ‘SL’

	DTWC	DR	LIC	IDFP	ISQLQ	IDS	IGS	LT	MIM	NLMR	PD	RAM	SL	UC	LC	LWS	UHA	BFU	UIO	IWR	HAT	HSS	HBR	IOD	ERB
AL	X	✓	✓	✓	✓	✓	✓	*	✓	X	X	✓	✓	✓	X	✓	X	*	✓	X	X	X	X	✓	*
PMD	X	X	X	X	✓	X	✓	*	X	X	X	X	✓	✓	X	X	X	X	X	X	X	X	X	X	*
SL	X	✓	X	X	✓	X	✓	*	✓	X	✓	X	✓	✓	X	✓	X	X	X	X	X	X	✓	X	✓

DTWC=Data Transmission Without Compression, DR=Debuggable Release, LIC=Leaking Inner Class, IDFP=Inefficient Data Format and Parser, ISQLQ=Inefficient SQL Query, IDS=Inefficient Data Structure, IGS=Internal Getter and Setter, , 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

Table 3
Energy bug coverage by tools ‘AL’, ‘PMD’ and ‘SL’

	RL	WB	VBS	IB	TMV	TDL	NCD	UL	UP
AL	*	✓	X	X	✓	✓	✓	✓	✓
PMD	*	X	X	X	X	X	X	X	X
SL	*	X	X	X	X	X	X	X	X

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

vides many class APIs. In the ‘AL’ API, each code smell/energy bug has the following properties: Id, summary, explanation, category, severity, priority, and additional links[9]. This information is shown to the developer when a code smell/energy bug is detected. Each code smell/energy bug is registered in an issue registry class and is detected by a detector class. The functionalities of detector class used by our implementation are given in additional material[10]. Fig. 1 gives an overview of the ‘AL’ API (version 26.5.2 is used for the implementation of detectors). The complexity calculation for the code smells HAA, HSS and HBR are done using ‘Metrics Reloaded’ plugin for Android Studio[11].

Inclusion of New Code Smells/Energy Bugs. For all undetected and partially covered code smells/ energy bugs (see Table 2 and 3), detection and refactoring rules are defined based on the definitions provided in additional materials and Android development best practice guides[12] provided by Google. Table 4 shows a list of Android code smells and energy bugs that are implemented in our ’extended Android Lint’ (xAL) tool. In ’Implementation’ column ’novel’ refer to code smells/energy bugs that are not already present in the ‘AL’ tool. ’Improvement’ refers to code smells /energy bugs that are partially covered by the ‘AL’ tool and can be improved by inclusion of additional APIs/conditions in our new ’xAL’ tool. The last column of Table 4 shows whether a correction is suggested by ’xAL’ for the detected Android code smells and energy

[9]http://tools.android.com/tips/lint-custom-rules
[10]https://figshare.com/s/84ae49a21551e6302d41.
[11]https://github.com/BasLeijdekkers/MetricsReloaded
[12]https://developer.android.com/topic/performance

Table 4
Android code smells and energy bugs implemented in ’xAL’

Abbr.	Implementation	Detection	Correction offered
Code smells			
DTWC	novel	yes	No, just warning is shown
LT	improvement	yes	yes
NLMR	novel	yes	yes
PD	novel	yes	yes
LC	novel	yes	yes
UHA	novel	yes	yes
BFU	improvement	yes	No, just warning is shown
IWR	novel	yes	No
HAT	novel	yes	No, just warning is shown
HSS	novel	yes	No, just warning is shown
HBR	novel	yes	No, just warning is shown
ERB	improvement	yes	No, just warning is shown
Energy Bugs			
RL	improvement	yes	yes
VBS	novel	yes	yes
IB	novel	yes	yes

DTWC=Data Transmission Without Compression, LT=Leaking Thread, NLMR=No Low Memory Resolver, PD=Public Data, RAM=Rigid Alarm Manager, LC=Lifetime Containment, UHA=Unsupported Hardware Acceleration, BFU= Bitmap Format Usage, IWR=Invalidate Without Rect, HAT=Heavy AsyncTask, HSS=Heavy Service Start, HBR=Heavy Broadcast Receiver, ERB=Early Resource Binding, RL=Resource Leak, VBS=Vacuous Background Services, IB= Immortality Bug

bugs. Table 5 shows Android code smells/energy bugs that are partially covered by the original ‘AL’ tool and the improvements we implemented in our new ’xAL’ tool for each of them. Pseudo-code for implemented code smells and energy bugs are given in additional material[13]. For most of the detected code smells/energy bugs we offer corrections. In cases where we do not provide corrections to the developer, a warning is issued. Each implemented code smell/energy bug is tested on sample classes which contains possible variations in which a selected code smell/energy bug can be present in the code. In addition, we provide a description for each of the detected code smell/energy bug, with the aim to help developers in refactoring. We made our tool open-source [16]. The ’xAL’ tool is compiled

[13]https://figshare.com/s/84ae49a21551e6302d41
[16]https://figshare.com/s/63c5b3e957f390432edf

Table 5
Proposed improvements in the Android Code Smells and Energy bugs detection

CS/ EB	API/Class already detected by 'AL' tool	API/Class detected by 'xAL' tool as improvement
LT (CS)	Detects thread class leak only.	Detection of classes like android.os.Handler and java.lang.Runnable that can lead to a thread leak[14].
BFU (CS)	Detects bitmap duplication, creation in onDraw() and usability issues.	Detection of Bitmap using Bitmap.create(params. . .)[15].
ERB (CS)	Detects creation of objects during DrawAllocation which can be performed by lazy initialization	Detection of heavy APIs (of type: Android .location, android.media, android.database, android.hardware) that should be initialized in lazy fashion1.
RL (EB)	Detects resources such as IO, JDBC, static fields, wifi manager, StringBuffer etc.	Detection of resources like Camera, Media Player etc. [?]

CS =Code Smell, EB =Energy Bug, LTLeaking Thread, BFU= Bitmap Format Usage, ERB=Early Resource Binding, RL=Resource Leak

as a Jar file. The Jar file is placed in the .android/lint folder of the Android Studio installation, typically located in USER-HOME. Android Studio is restarted for new detectors to take effect. Applications can be analyzed for Android code smells and energy bugs in two ways[17], i.e., in-line analysis and whole-application analysis.

4. Evaluation

4.1. Evaluation Plan

We evaluated the 'xAL' tool in two steps. First, we evaluated the tool on a selection of open source apps, then we compared the tool's performance to that of the 'PAPRIKA' tool.

Evaluation on Open Source Apps. The evaluation includes testing on nine real-world applications[18] chosen from the F-Droid[19] repository. The applications were chosen if the source code was in Java and the number of line of code was less than 30,000 (for ease of manual verification). We checked that each application can be compiled and executed on a device without errors.

Comparison with 'PAPRIKA'. We considered state of the art tools such as 'PAPRIKA', 'aDOCTOR', 'SAAD', 'Chimer' and 'AL' (implementation by Goaër [14]) as possible comparison candidates. 'Chimer' and the 'AL' tool presented in [14] were not available in open source. We chose 'PAPRIKA' as it had the largest number of overlapping code smells/energy bugs with our 'xAL' tool. These are Heavy Async Task (HAT), Heavy Broadcast Receiver (HBR), Heavy Start Service (HSS), Invalidate without Rect (IWR), No Low Memory Resolver (NLMR) and Unused Hardware Acceleration (UHA) and Resource Leak (RL). JAR file for Paprika was downloaded from their open-source repository[20].

4.2. Evaluation Results

This section presents the results of testing the tool on open source projects and compares its coverage with PAPRIKA tool.

4.2.1. Evaluation on Open Source Apps

The tool was tested on the nine applications selected from the F-Droid repository. Table 6 shows the results of the evaluation on open source applications. It lists the application name against the code smells detected in it, true positives (TP), false positives (FP), false negatives (FN), precision (P), recall (R), total corrections available (TCA) and total applied correction (TAC) .

Some of the code smells/energy bugs (such as 'Early Resource Binding' (ERB), 'Heavy Async Task' (HAT) and 'Heavy Broadcast Receiver' (HBR)) did not appear in any of the applications under test.

In the case of application 'Kolabnotes' two false negatives were detected, i.e. two instances of code smell 'Lifetime Containment' (LC) code smell. A possible reason could be declarations of interfaces in non-lifecycle classes, which were not included in the LC code smell definition. In the case of application 'Sound Recorder' two false negatives were detected, i.e. two instances of code smells 'No Low Memory Resolver' (NLMR). A possible reason could be that the class used by this application is deprecated, hence not covered by the implementation of our 'xAL' tool. In the case of applications 'Kolabnotes' and 'Camera Roll' one false positive (i.e., one instance of the 'Data Transmission without Compression' (DTWC) code smell) was detected for each application. In the case of application 'Calorie Scope' one false positive was detected, i.e., one instance of 'Resource Leak (RL) for a camera instance energy bug. In the case of application, 'CameraColorPicker' five false positives (i.e. Lifetime Containment (LC) code smell (4 instances) and Resource Leak (RL) energy bug (1 instance)) were detected. In the case of application, 'Privacy- Friendly Weather' two false positives (i.e. two instances of Lifetime Containment (LC) code smell) were detected. In the case of code smell Lifetime Containment (LC), a possible reason for false positive could be that it flags abstract classes as interfaces as well. In the

[17] https://figshare.com/s/84ae49a21551e6302d41 (See Tool Walk-through)
[18] https://figshare.com/s/84ae49a21551e6302d41 (See Table 1)
[19] https://f-droid.com
[20] https://github.com/GeoffreyHecht/paprika

Table 6
Results of evaluation on open source app using 'xAL' tool

ID	App	CS/EB Detected	Detection Results					TCA	TAC
			TP	FP	FN	P	R		
1	Odyssey	UHA, NLMR, HSS, LC, IWR	11	0	0	1	1	6	6
2	Kolabnotes	UHA, BFU, LT, DTWC, IWR, NLMR	18	1	2	1	0.9	7	7
3	Calorie Scope	LC, NLMR, RL, VBS	12	1	0	0.9	1	13	13
4	Camera Roll	UHA, BFU, DTWC, IWR, LC, NLMR, PD	21	1	0	1	1	16	6
5	Bipol Alarm	UHA, HSS, DTWC, NLMR	4	0	0	1	1	4	4
6	Sound Recorder	UHA, PD	7	0	2	1	0.8	7	7
7	CameraColorPicker	UHA, BFU, NLMR, IWR, LC, RL	12	5	0	0.7	1	17	12
8	Privacy-Friendly Weather	UHA, LT, LC, NLMR	13	2	0	0.9	1	15	13
9	Reminders	UHA, DTWC, NLMR	5	0	0	1	1	5	5
		TOTAL	103	10	4	–	–	90	73

CS = Code Smell, EB = Energy Bug, TP = True Positive, FP = False positive, FN = False Negative, P= Precision, R= Recall, TCA = Total Corrections Available, TAC= Total Applied Corrections

case of code smell Data Transmission without Compression (DTWC), a possible reason could be that the 'xAL' tool does not track instances that were compressed in another class. In the case of energy bug Resource Leak (RL), a possible reason for false positives could be that code smell/energy bug was handled pro-actively by the developer. For example, for Resource Leak (RL), if the camera instance was closed proactively by the developer in a method other than onStop(). In this case, check on onStop() method is no longer required.

Corrections were available for 66.3% of the detected Android code smells and energy bug instances, out of which 84% of corrections were applied. 16% corrections were not applied, which include false positives and instances of Public Data (PD) code smell (correction of this code smell altered the functionality of the application under test). These numbers are dependent on the type of code smell/energy bug and the frequency of its instances in the application.

4.2.2. Comparison with PAPRIKA

The 'PAPRIKA' tool did not work on applications 1 to 4 that had AndroidX[21] dependencies. Due to this inherent limitation of 'PAPRIKA', it was only tested on applications 5 to 9. Refactoring of code smells/energy bugs was not applied in any test application as the accessible version of 'PAPRIKA' tool does not offer to refactor.

Table 7 shows the results of the evaluation on open source applications using 'PAPRIKA'. It lists the application name against the code smells detected in it, true positives (TP), false positives (FP), false negatives (FN), precision and recall. 'PAPRIKA' was able to detect three types of code smells namely: No Low Memory Resolver (NLMR), Heavy Start Service (HSS), Heavy Broadcast Receiver (HBR). For these smells, no false positives were detected. 'PAPRIKA' did not detect any instance of the code smells/energy bugs Unused Hardware Acceleration (UHA), Resource Leak (RL) and Invalidate without Rect (IWR), which could be seen in 'FN' column.

Table 8 shows a comparison of the code smells detected by both 'xAL' and 'PAPRIKA'. ✓represents that a code smell is detected in a test application, × represents that a code smell was not detected in a test application and empty cells show that the code smell was not present in a test application. Code smell Heavy Async Task (HAT) was not present in any of the test applications hence not detected by 'PAPRIKA' and our 'xAL' tool. 'xAL' was able to detect almost all the code smell instances that were detected by 'PAPRIKA' tool. However, two instances of 'No Low Memory Resolver' (NLMR) code smells were missed in application 'Sound Recorder' (as they used a deprecated class API). The 'Heavy Broadcast Receiver' (HBR) code smell was also missed by our tool as the value for an upper limit of computational complexity is set to five in 'PAPRIKA' and ten (as per McConnel [17]) in our 'xAL' tool.

'PAPRIKA' detected seven instances of false negatives for the code smells: Unused Hardware Acceleration (UHA) (5 instances) and Invalidate without Rect (IWR) (1 instance) and energy bug Resource Leak (RL) (1 instance). As compared to 'PAPRIKA', our tool was able to detect Invalidate without Rect (IWR) code smell and Resource Leak (RL) energy bug in 'CameraColorPicker' test application. Unused Hardware Acceleration (UHA) code smell was also detected in all five applications by our tool. Table 9 shows a compatibility comparison between 'PAPRIKA' and 'xAL' to gain a better understanding of the support offered by both tools as well as their limitations.

The average precision, recall and F1-score for our 'xAL' were 0.93, 0.96 and 0.94, respectively. The average precision, recall and F1-score for 'PAPRIKA' were 1.0, 0.74 and 0.85, respectively. Hence, our 'xAL' tool not only provides better Android code smell/energy bug coverage but also improves upon the usability aspects of the tool in comparison to 'PAPRIKA' tool.

5. Threats to Validity

Our 'xAL' tool only performs static source code analysis of Android applications. Since static source code analysis could be done during development re-

[21] https://developer.android.com/jetpack/androidx

Table 7
Results of evaluation on open source apps using 'PAPRIKA'

ID	Test App	CS/EB detected	TP	FP	FN	Precision	Recall
5	Bipol Alarm	NLMR, HSS	2	0	1	1	0.67
6	Sound Recorder	NLMR	2	0	3	1	0.67
7	CameraColor Picker	NLMR	6	0	1	1	0.67
8	PrivacyFriendly Weather	NLMR	11	0	1	1	0.92
9	Reminders	NLMR, HBR	4	0	1	1	0.8
	TOTAL		25	0	7	–	–

CS = Code Smell, EB = Energy Bug, TP = True Positive, FP = False positive, FN = False Negative, NLMR=No Low Memory Resolver, SS=Heavy Service Start, HBR=Heavy Broadcast Receiver

Table 8
Code smell detection comparison between 'PAPRIKA' and 'xAL'

App	Bipol Alarm		Sound Recorder		CameraColor Picker		PrivacyFriendly Weather		Reminders	
CS	P	xAL	P	xAL	P	xAL	P	xAL	P	xAL
HAT										
HBR							✓	×		
HSS	✓	✓								
IWR					×	✓				
NLMR	✓	✓	✓	×	✓	✓	✓	✓	✓	✓
UHA	×	✓	×	✓	×	✓	×	✓	×	✓
RL					×	✓				

CS= Code smell P='PAPRIKA', xAL= 'extended Android Lint', HAT=Heavy AsyncTask, HSS=Heavy Service Start, HBR=Heavy Broadcast Receiver, IWR=Invalidate Without Rect, NLMR=No Low Memory Resolver, UHA=Unsupported Hardware Acceleration, RL=Resource Leak

Table 9
Compatibility comparison between 'PAPRIKA' and 'xAL'

Compatibility criteria	PAPRIKA	xAL
Java version support	Java 7 only	versions >= Java7
Support for apps with Android X	No	Yes
In-line warnings in Code Editor	No	Yes
Navigation to LOC in Code Editor	No	Yes
Individual code smell analysis	No	Yes
Disabling detection of CS/EB	No	Yes

LOC=Line of code, CS/EB=CodeSmell/EnergyBug, xAL='extended Android Lint'

peatedly, the support provided by our tool could benefit developers. Implementation of a functionality may vary based an application's architecture/design and the coding style of a developer. Hence, our definitions of code smells/energy bugs might not cover every scenario related to a particular code smell, leading to false positives and false negatives in the results. For example, we only consider lifecycle classes, i.e. Activity and Fragment for lifecycle dependent issues like Resource Leak and Leaking Thread. However, depending on the application architecture, lifecycle dependent components might be called in non-lifecycle classes which may go undetected. Moreover, some corrections for code smells, and energy bugs might clash with the functional requirements of the application. For example, correction for Public Directory (PD) code smell changes public directory to a private directory. However, for an application that requires access to a public directory like Gallery, if it is changed to a private directory, the functional requirement will be in contradiction. But as the corrections can only be applied after the developer's consent, such issues are less likely to occur. Correction/recommendations are offered for 66% of detected code smells/energy bugs. For the rest of 34% code smells/energy bugs, only a warning is shown with hints for correction (cf section 3.2.2). The decision of applying the suggested correction is left for the developers. Our tool does not cover third-party libraries used in Android applications. During development, we tested our tool against sample classes, which were injected with code smells/energy bugs by the authors of this study. Hence there might be researcher bias in the introduction of those code smells/energy bugs (in terms of coding style, location and variety), leading to higher accuracy in results. To mitigate this threat, the tool was evaluated on nine open-source applications that already contained some of the code smells and energy bugs. During the evaluation, we did not physically measure the changes in energy consumption of the applications under test due to refactoring of code smells/energy bugs. The assumption that refactoring the selected code smells/energy bugs lead to energy optimization of Android applications is based on the related work such as [11, 14, 16].

6. Conclusion

We extended the tool 'AL' to detect and correct Android-specific code smells and energy bugs that may lead to energy optimization in Android applications. On top of the 261 issues already covered by 'AL', our extended tool 'xAL' provides coverage for 12 Android-specific code smells (nine new and three improved) and three energy bugs (two new and one improved). Moreover, 'xAL' integrates directly in Android Studio IDE and gives control to the developer for refactoring code smell/energy bugs, which was missing in other state of the art tools. We evaluated 'xAL' on nine open-source applications; it detects code smells and energy bugs with an average precision, average recall and F1 score of 0.93, 0.96, and 0.94 respectively. It accurately corrects 84% of selected code smells and energy bugs. Our tool offers better code smell and energy bug detection coverage as compared to 'PAPRIKA'. In the future, we aim to evaluate 'xAL' on a large data set of applications, which will also help in analyzing the

correlation between the frequency of occurrences of code smells/energy bugs, and impact on energy consumption due to their refactoring.

Acknowledgments

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

References

[1] I. Fatima, H. Anwar, D. Pfahl, U. Qamar, Tool Support for Green Android Development: A Systematic Mapping Study, in: 5th Int. Conf.on Softw. Technologies - ICSOFT, 2020, pp. 409–417.

[2] A. Turner, How many people have smarphones worldwide (Apr 2020), 2020. URL: https://www.bankmycell.com/blog/how-many-phones-are-in-the-world.

[3] R. Verdecchia, R. Aparicio Saez, G. Procaccianti, P. Lago, Empirical Evaluation of the Energy Impact of Refactoring Code Smells (2018) 345–365. doi:10.29007/dz83.

[4] H. Anwar, D. Pfahl, S. N. Srirama, Evaluating the Impact of Code Smell Refactoring on the Energy Consumption of Android Applications, in: 45th Euromicro Conf.on Softw. Eng. and Advanced Applications, SEAA, 2019, pp. 82–86. doi:10.1109/SEAA.2019.00021.

[5] A. V. Rodríguez, C. Mateos, A. Zunino, M. Longo, An analysis of the effects of bad smell-driven refactorings in mobile applications on battery usage, in: Modern Softw. Eng. Methodologies for Mobile and Cloud Environments, 2016. doi:10.4018/978-1-4666-9916-8.ch009.

[6] C. Sahin, L. Pollock, J. Clause, How do code refactorings affect energy usage?, Int'l Symposium on Empirical Softw. Eng. and Measurement - ESEM (2014) 1–10. doi:10.1145/2652524.2652538.

[7] S. Habchi, R. Rouvoy, N. Moha, On the survival of android code smells in the wild, Proceedings - 2019 IEEE/ACM 6th Int'l Conf. on Mobile Softw. Eng. and Systems, MOBILESoft 2019 (2019) 87–98. doi:10.1109/MOBILESoft.2019.00022.

[8] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, R. Rouvoy, Investigating the energy impact of Android smells, 24th IEEE Int'l Conf. on Softw. Analysis, Evolution, and ReEng. - SANER (2017) 115–126. doi:10.1109/SANER.2017.7884614.

[9] G. Hecht, R. Rouvoy, N. Moha, L. Duchien, Detecting Antipatterns in Android Apps, 2nd ACM Int'l Conf. on Mobile Softw. Eng. and Systems - MOBILESoft (2015) 148–149. doi:10.1109/MobileSoft.2015.38.

[10] Z. Xu, C. Wen, S. Qin, State-taint analysis for detecting resource bugs, Science of Computer Programming (2018) 93–109. doi:10.1016/j.scico.2017.06.010.

[11] V. N. Huynh, M. Inuiguchi, B. Le, B. N. Le, T. Denoeux, Improve the Performance of Mobile Applications Based on Code Optimization Techniques Using PMD and Android Lint, LNCS (including subseries LNAI and LNB) 9978 LNAI (2016) V–VI. doi:10.1007/978-3-319-49046-5.

[12] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, A. De Lucia, Lightweight detection of Android-specific code smells: The aDoctor project, SANER 2017 - 24th IEEE Int'l Conf. on Softw. Analysis, Evolution, and ReEng. (2017) 487–491. doi:10.1109/SANER.2017.7884659.

[13] H. Jiang, H. Yang, S. Qin, Z. Su, J. Zhang, J. Yan, Detecting Energy Bugs in Android Apps Using Static Analysis, LNCS (including subseries LNAI and LNB) 10610 LNCS (2017) 192–208. doi:10.1007/978-3-319-68690-5_12.

[14] O. L. Goaër, Enforcing green code with android lint, in: 34th IEEE/ACM Int. Conf. on Automated Softw. Eng. Workshop - ASEW, 2019. doi:10.1109/ASEW.2019.00018.

[15] D. Maia, M. Couto, J. Saraiva, E-Debitum : Managing Softw. Energy Debt (2020) 162–169.

[16] M. Couto, J. Saraiva, J. P. Fernandes, Energy Refactorings for Android in the Large and in the Wild (2020) 217–228. doi:10.1109/saner48275.2020.9054858.

[17] S. McConnell, Code Complete: A Practical Handbook of Softw. Construction 9 (2011).