Enforcing Green Code With Android Lint

Olivier Le Goaër
Computer Science Laboratory (LIUPPA)
University of Pau
Pau, France
olivier.legoaer@univ-pau.fr

***Abstract*—Nowadays, energy efficiency is recognized as a core quality attribute of applications (apps) running on Android-powered devices constrained by their battery. Indeed, energy hogging apps are a liability to both the end-user and software developer. Yet, there are very few tools available to help developers increase the quality of their native code by ridding it of energy-related bugs. Android Studio is the official IDE for millions of developers worldwide and there's no better place to enforce green coding rules in everyday projects. Indeed, Android Studio provides a code scanning tool called Android lint that can be extended with lacking green checks in order to foster the design of more eco-responsible apps.**

Keywords— green, Android, smells, lint, bugs, energy, battery

I. INTRODUCTION

Smartphones have become more and more popular since the introduction of iPhone and Android-based devices, and battery lifespan is unquestionably a hot concern. Most smartphones offer the possibility to add new applications, through distribution channels such as the Google Play for the Android platform or App Store for the iOS platform. These applications often take advantage of the sensors available, typically GPS and Internet connectivity to develop context-aware applications, accelerometer for motion tracking, Bluetooth for pairing with third-party objects, etc. While the fleet of devices is becoming increasingly mobile, many software developers have limited experience with energy-constrained portable embedded systems such as smartphones/tablets and wearables, which leads to unnecessarily power-hungry applications that rely overly on the operating system for power management. In addition, users struggle to determine which applications are energy-efficient, and typically users blame the operating system or hardware platform instead of unfortunate and unintentional software design decisions [1].

On the one hand, operating systems have responded by offering an increasingly intelligent energy management. Starting from Android 6.0, Android introduces two power-saving features called Doze and App Standby. Since Android 9.0, it introduces an AI feature called Adaptive Battery. Of course, equivalent functionalities exist for iOS. On the other hand, poor evaluations left on application stores by users [2] or sometimes massive uninstallations push developers to assume their share of responsibility for energy waste. The emergence of sustainability and the reduction of energy consumption in the social, political and technical agenda has accelerated this awareness in recent years. The noble objective of Green Software is that the small energy savings obtained at the scale of each device add up and contribute to reducing the ecological footprint of mobile software on a global scale. Thus, applications of all kinds and sizes must be concerned about their energy efficiency, and not merely world-class apps with their enormous base of users.

The Android platform is at the forefront of this ecological challenge because it is the undisputed leader in market share (about 85%), with 2 billion monthly active devices globally in 2017. Its Google Play application store has 2.6 million applications available in 2018, with an estimated download volume of 19 billion in 2017. Unfortunately, most developers only have little to no knowledge about energy-efficiency: common misconceptions are often made, and a general lack of energy-aware tooling has to be deplored [3]. And still: older applications would deserve to be reengineered to save energy, while newly created apps should make sure they are energy-friendly before being released in an ultra-competitive market.

Meanwhile, Android lint is a static code analysis tool enabled by default in the official Android Studio IDE to ensure the general quality of development project. It is thus the devoted companion of almost all native code developers whose recent study [4] showed that they have confidence in its inspection report and that they even use it to learn new things and improve themselves. From this point of view, a linter is much more effective than any kind of guideline in disseminating best practices to a large audience. For these reasons, this paper argues that Android lint is the ideal vector by which green coding habits can be gradually changed, through systematic hunting of recurring energy inefficiency bugs. Such a tool will allow to check if an application is green-by-design, far prior to its deployment on real devices.

The structure of this paper is as follows: Section II describes eleven green bugs that can be found in real-world development projects with the Android SDK. Section III explains how the bugs have been implemented as green checks with the Android lint framework. Section IV explains how to use the prototype in Android Studio before giving a brief feedback on this type of tool in Section V. Related work is mentioned in Section VI before concluding and providing perspectives in Section VII.

ASEW '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Association for Computing Machinery. ACM ISBN 978-1-4503-8128-4/20/09…$15.00
https://doi.org/10.1145/3417113.3422188

II. Android-Specific Green Bugs

A green bug (a.k.a energy bug [5]) is a defect in the native code written by the Android developer that can potentially shorten the battery life of a device while in use. It may be negligence or error on her part. Anyway, a green bug is an approximation of what will happen at the runtime exclusively on the basis of design-time. Green bugs are thus sometimes referred as bad smells or anti-patterns.

A. *Green Bugs Overview*

The list of the 11 green bugs considered in this paper is given in Table 1. They are intended to represent realistic energy-related problems that Android programmers may face.

TABLE I. ANDROID-SPECIFIC GREEN BUGS

Bug name	Severity	Artifact	Supply
Everlasting Service	ERROR	Source code	K
Dark UI	WARNING	Manifest, Style, Drawable	R
Battery-Efficient Location	INFO	Source code, Gradle	O
Sensor Leak	ERROR	Source code	O
Sensor Coalesce	WARNING	Source code	O
Bluetooth Low-Energy	INFO	Source code	O
Internet In The Loop	ERROR	Source code	R
Durable Wake Lock	WARNING	Source code	K
Uncompressed Data Transmission	WARNING	Source code	R
Rigid Alarm	WARNING	Source code, Gradle	K
Service at Boot-time	WARNING	Source code, Manifest	K

Discovering and evaluating green bugs tightly coupled to the Android framework is cumbersome in practice. To achieve this, it is possible to use the following suppliers:

- (O)fficial resources: some battery killers are explicitly mentioned in the Android developer guide[1] or are sometimes found in the API reference documentation[2], nestled in docstrings at the class or method level.
- (K)nowledgable communities: skilled Android developers report how they have dealt with excessive battery depletion and how they have solved them, whether in blog posts or on popular exchange platforms such as StackOverflow or Google Groups.
- (R)esearch literature: studies regularly focus on very specific aspects in order to identify hotspots, like cryptography [6], advertising [7], logging [8], network traffic [9], display [10], ML [11] etc.

In addition, the Android specificity means that bugs do not only reside in the source code but can possibly affect every artifacts that form an Android project, namely: manifest, resources, source code (*.kt, *.java), bytecode, Gradle files, ProGuard files, Property Files.

[1] `https://developer.android.com/guide`
[2] `https://developer.android.com/reference/`

Finally, the different levels of severity of green bugs are as follows: ERROR means a serious design problem. As it stands, the application should not be deployed. WARNING points out a potential problem that it would be more prudent to correct. INFO draws the developer's attention to the existence of better solutions elsewhere.

B. *Green Bugs Details*

In this section the technical description of each green bug is given using straight references to elements of the Android framework.

1) Everlasting Service: If someone calls `Context#startService()` then the system will retrieve the service (creating it and calling its `onCreate()` method if needed) and then call its `onStartCommand(Intent, int, int)` method with the arguments supplied by the client. The service will at this point continue running until `Context#stopService()` or `Service#stopSelf()` is called. Failing to call any of these methods leads to a serious energy leak.

2) Dark UI : Developers are allowed to apply native themes for their app, or derive new ones from the latter. This decision has a significant impact on energy consumption since displaying dark colors is particularly beneficial for mobile devices with (AM)OLED screens [12, 13]. Note that the recent generalization of dark mode apps on various platforms clearly goes in this direction. By default Android will set Holo to the Dark theme (style `Theme.Holo`) and hence switching to the light theme (style `Theme.Holo.Light`) within the manifest should be avoided.

3) Battery-Efficient Location: Location awareness is one of the most popular features used by apps. The fused location provider is one of the location APIs in Google Play services which combines signals from GPS, Wi-Fi, and cell networks, as well as accelerometer, gyroscope, magnetometer and other sensors. It is officially recommended to maximize battery life. Thus, developer has to set up Google Play Service in her gradle file and then to import from `com.google.android.gms.location` instead of the `android.location` package of the SDK.

4) Sensor Leak: Most Android-powered devices have built-in sensors that measure motion, orientation, and various environmental conditions. In addition to these are the image sensor (a.k.a Camera) and the geopositioning sensor (a.k.a GPS). The common point of all these sensors is that they are expensive while in use. Their common bug is to let the sensor unnecessarily process data when the app enters an idle state, typically when paused or stopped. Consequently, calls must be carefully pairwised: `SensorManager#registerListener()/unregisterListener()` for regular sensors, `Camera#open()/Camera#release()` for the camera and `LocationManager#requestLocationUpdates()/removeUpdates()` for the GPS.

5) Sensor Coalesce: With SensorManager#registerListener(SensorEventListener, Sensor, int) the events are delivered as soon as possible. Instead, SensorManager#registerListener (SensorEventListener, Sensor, int, int maxReportLatencyUs) allows events to stay temporarily in the hardware FIFO (queue) before being delivered. The events can be stored in the hardware FIFO up to maxReportLatencyUs microseconds. Once one of the events in the FIFO needs to be reported, all of the events in the FIFO are reported sequentially. Setting maxReportLatencyUs to a positive value allows to reduce the number of interrupts the AP (Application Processor) receives, hence reducing power consumption, as the AP can switch to a lower power state while the sensor is capturing the data.

6) Bluetooth Low-Energy: In contrast to classic Bluetooth, Bluetooth Low Energy (BLE) is designed to provide significantly lower power consumption. Its purpose is to save energy on both paired devices but very few developers are aware of this alternative API. From the Android client side, it means append android.bluetooth.le.* imports to android.bluetooth.* imports in order to benefits from low-energy features.

7) Internet In The Loop: Opening and closing internet connection continuously is extremely battery-inefficient since HTTP exchange is the most consuming operation of the network [14]. This bug typically occurs when one obtain a new HttpURLConnection by calling URL#openConnection() within a loop control structure (while, for, do-while, for-each). Also, this bad practice must be early prevented because it is the root of another evil that consists in polling data at regular intervals, instead of using push notifications to save a lot of battery power [15].

8) Durable Wake Lock: A wake lock is a mechanism to indicate that your application needs to have the device stay on. The general principle is to obtain a wake lock, acquire it and finally release it. Hence, the challenge here is to release the lock as soon as possible to avoid running down the device's battery excessively. Missing call to PowerManager#release() is a built-in check of Android lint (*Wakelock* check) but that does not prevent abuse of the lock over too long a period of time. This can be avoided by a call to PowerManager#acquire (long timeout) instead of PowerManager#acquire(), because the lock will be released for sure after the given timeout expires.

9) Uncompressed Data Transmission: In [15], Höpfner and Bunse discussed that transmitting a file over a network infrastructure without compressing it consumes more energy than with compression. More precisely, energy efficiency is improved in case the data is compressed at least by 10 %, transmitted and decompressed at the other network node. From the Android client side, it means making a post http request using a GZIPOutputStream instead of the classical OutputStream, along with the HttpURLConnection object.

10) Rigid Alarm: Applications are strongly discouraged from using exact alarms unnecessarily as they reduce the OS's ability to minimize battery use (i.e. Doze Mode). For most apps prior to API 19 (refer to the targetSdkVersion property of build.gradle), setInexactRepeating() is preferable over setRepeating(). When you use this method, Android synchronizes multiple inexact repeating alarms and fires them at the same time, thus reducing the battery drain. Similarly, setExact() and setExactAndAllowWhileIdle() can significantly impact the power use of the device when idle, so they should be used with care. High-frequency alarms are also bad for battery life but this is already checked by Android lint (*ShortAlarm* built-in check).

11) Service at Boot-time: Services are long-living operations, as components of the apps. However, they can be started in isolation each time the device is next started, without the user's acknowledgement. This technique should be discouraged because the accumulation of these silent services results in excessive battery depletion that remains unexplained from the end-user's point of view. In addition, end-users know how to kill applications, but more rarely how to kill services. Thus, any developer should avoid having a call to Context#startService() from a broadcast receiver component that has specified an intent-filter for the BOOT_COMPLETED action in the manifest.

III. Green Checks In Android Lint

The Android lint tool is a static code analysis tool that checks an Android project source files for potential bugs and optimization improvements for correctness, security, performance, usability, accessibility, and internationalization. Lint comes with many built-in checks almost without any consideration for energy savings. We describe how the previously mentioned green bugs were implemented as custom checks in a brand new category.

A. Lint Framework

Android lint was designed for a tight integration with Android Studio (based on IntelliJ IDEA). There are around 350 Android lint checks, and an even larger number of IDE inspections. Generally, lint tries to keep the checks mostly Android specific. The old checks were written in Java while the newer ones are now written in Kotlin.

Android Tools Lint API (com.android.tools.lint) provides a set of Java classes that define the internals of Android lint and its integration with the IDE. The general principle is to define issues (Issue) and declare them in a register (IssueRegistry). An Issue is described by a unique Id, a brief description, an explanation, a priority (from 1 to 10), a severity (among Ignore, Informational, Warning, Error, Fatal) and the category to which it belongs. Then, detectors (Detector) are responsible for detecting the occurrence of these issues.

The interesting thing is that a complete Android project can be scanned, covering different scopes. As a result, the following specialized scanners are available:

- `XmlScanner` - XML files (visit with DOM)
- `SourceCodeScanner` - Java and Kotlin files (visit with UAST)
- `ClassScanner` - .class files (bytecode, visit with ASM)
- `BinaryResourceScanner` - binaries like images
- `ResourceFolderScanner` - Android /res folders
- `GradleScanner` - Gradle build scripts
- `OtherFileScanner` - Other files

Created by JetBrains, UAST (Universal Abstract Syntax Tree) tries to "unify" Java/Kotlin syntax trees so that a check written with UAST will often work for both languages automatically. Please also note that UAST is just a wrapper around PSI (Program Structure Interface), the underlying API of IntelliJ IDEA.

B. *Greenness Category*

At first sight, energy saving is a special case of performance, which is an existing category. In fact, the gain at the battery level is just a side effect of performance, because the search for reactivity/speed generally implies that fewer operations are performed. Conversely, energy conservation considers the preservation of battery level as the primary intention, even if it means sacrificing other quality attributes if necessary (including speed).

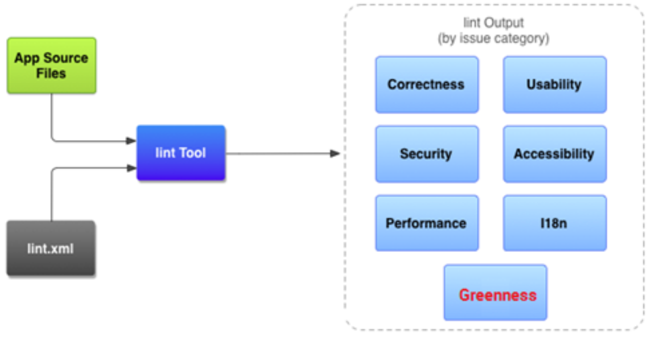


Fig. 1. Introducing a new “Greenness” category within Android lint.

The green checks should not blend with the other inspections because it seems important to understand what's really energy-related, and what's not. Consequently, we added a new category *Greenness* in addition to the 6 built-in categories (Fig. 1). The 11 checks will be placed in this category, which will help to better inform developers. Ideally, the two built-in checks *ShortAlarm* and *Wakelock* mentioned in Section II.B should be moved to this new category.

C. *Detectors*

Detectors are responsible for reporting problems by scanning certain types of files (the scope). As there are 11 issues, as many detectors have been created. However, a given detector may report different messages to the developer. For example, the *Sensor Leak* bug includes two separate messages corresponding to the following distincts situations: (1) the calls to `SensorManager# registerListener ()` are missing and/or (2) they are present but are not placed in the `onPause()` or `onStop()` activity’s callbacks.

When scanning human-readable files, a detector is based on visitor-style programming. For performance reasons, it is not feasible to traverse the abstract syntax tree for each check (as a reminder, there are hundreds of them). As a result, the Lint framework requires to first declare which node types you are interested in so that a single tree iteration takes place, and when the right node type is encountered the detectors concerned are called back throughout a specific handler.

D. *Quick fixes*

Fixes are a mechanism proposed by Android Tools Lint API that allows the developer to apply a single-click modification of her source code. An Android lint quick fix does not work at the UAST level but as text replacement, so that it is necessarily limited to trivial cases. Fixable green bugs are the following:

- *Dark UI*: in the AndroidManifest.xml, replace the value of `android:theme` attribute.
- *Battery-Efficient Location*: add a dependency entry to `build.gradle` for Google Play Service Location.
- *Sensor Coalesce*: replace the call to `registerListener()` by its queued alternative. If the argument is 0, switch to the default value 15000.
- *Durable Wake Lock*: replace the call to `acquire()` by its timed alternative.
- *Rigid Alarm*: replace the call to `setRepeating()` by its inexact alternative.

IV. Tool Walk-Through

The prototype developed to illustrate the ideas expounded in this paper can be downloaded from the following site:

http://green.pauware.com/android-lint/greenchecks.jar

A. *Installation*

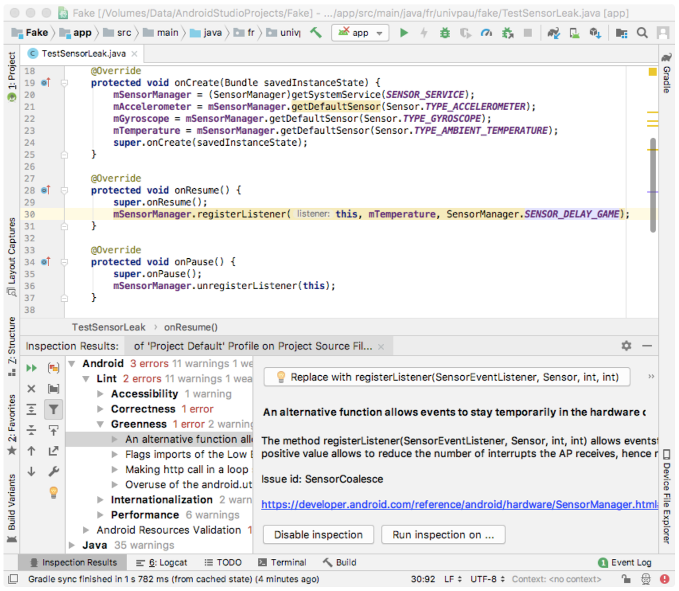
The file must be placed in the `~/.android/lint/` default directory, unless the environment variable `$ANDROID_LINT_JARS` is set. Then Android Studio must be restarded. If it fails, try *Invalid Caches/Restart* from the menu in the IDE.

If you are not using Android Studio, lint can also be run from the command-line and can even write its results in output files (HTML and XML).

B. *Inspection*

Because some green bugs cannot be checked on-the-fly, the preferred way is to manually run inspections (*Analyze > Inspect Code*). The results are displayed in the Inspection Results window in Android Studio (Fig. 2).

In the left pane tree view, view the inspection results by expanding and selecting the novel greenness category. The right pane displays the inspection report for the selected green bug and provides the name and location of the issue. Where applicable, the inspection report displays other information such as a green bug synopsis to help you correct the problem, and optionally a quick-fix at the top of the pane.



Fake [/Volumes/Data/AndroidStudioProjects/Fake] - .../app/src/main/java/fr/univpau/fake/TestSensorLeak.java [app]

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        mSensorManager = (SensorManager)getSystemService(SENSOR_SERVICE);
        mAccelerometer = mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
        mGyroscope = mSensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE);
        mTemperature = mSensorManager.getDefaultSensor(Sensor.TYPE_AMBIENT_TEMPERATURE);
        super.onCreate(savedInstanceState);
    }

    @Override
    protected void onResume() {
        super.onResume();
        mSensorManager.registerListener( listener: this, mTemperature, SensorManager.SENSOR_DELAY_GAME);
    }

    @Override
    protected void onPause() {
        super.onPause();
        mSensorManager.unregisterListener(this);
    }
```

Inspection Results: of 'Project Default' Profile on Project Source Fil...

```
▼ Android 3 errors 11 warnings 1 we
  ▼ Lint 2 errors 11 warnings 1 wea
    ▶ Accessibility 1 warning
    ▶ Correctness 1 error
    ▼ Greenness 1 error 2 warning
      ▶ An alternative function all
      ▶ Flags imports of the Low E
      ▶ Making http call in a loop
      ▶ Overuse of the android.ut
    ▶ Internationalization 2 warn
    ▶ Performance 6 warnings
  ▶ Android Resources Validation 1
▶ Java 35 warnings
```

Replace with registerListener(SensorEventListener, Sensor, int, int)

An alternative function allows events to stay temporarily in the hardware c

The method registerListener(SensorEventListener, Sensor, int, int) allows events t
positive value allows to reduce the number of interrupts the AP receives, hence r

Issue id: SensorCoalesce

https://developer.android.com/reference/android/hardware/SensorManager.html

Gradle sync finished in 1 s 782 ms (from cached state) (4 minutes ago) 30:92 LF UTF-8 Context: <no context>

Fig. 2. Defect highlighting for the green bug *Sensor Coalesce*.

V. Lessons Learned

The choice of the Android lint framework to address the problem of green bugs is not without its pitfalls. Here are three lessons learned from the design of the tool presented in this paper.

A. *A rather limited power of expression*

In principle, some well-known battery optimizations such as deferring and caching could be checked statically but they are almost impossible to formalize in practice. Deferring consists in postponing costly actions to a more appropriate time, typically when the device is in charge. As easy as it is to detect that a code chunk will be triggered by a "in charge" event broadcasted by the system, the payload of the latter cannot be analyzed. As for catching, its typical purpose is to cache downloaded data instead of repeatedly waking up the radio to re-download the data. Unfortunately, there is no easily detectable code pattern for this situation because the data structure for caching is freely defined by the programmer.

B. *False positives and false negatives*

It is important to avoid alerting the developer to bugs that are not bugs, but more importantly, it must be avoided missing the real bugs. In this respect, the writing of checks can quickly take a defensive form because it is necessary to anticipate various situations. False positives occur, for example, when you simply visit a method call by name, without making sure that the call concerns an instance of the correct Android API class. Indeed, it could be just a user-defined method with the same name. False negatives often occur because of indirection levels. For example, it is easy to detect that a given statement is located inside a given method, but it escapes the scrutiny of the detector when it is encapsulated in an intermediate method call.

C. *Undocumented and instable API*

Writing custom lint checks is like wading in undocumented waters: there is very little documentation about Android Tools Lint API and it is often deprecated. Indeed, lint has an unstable API so that developers ought to be prepared to adjust their code for the next tools release. In addition, writing a check can quickly become a headache. However, to avoid having to write things the "hard way", the framework provides helpers, evaluators (Java, Constant, Type, Resource) and utils (LintUtils, SdkUtils, UastLintUtils, XmlUtils). They allow for example: method resolving, type inference, class inheritance test, string distance, and so on.

VI. Related Works

Not surprisingly, the body of research on energy efficiency for mobile applications is quite recent, and the Android OS platform is over-represented. A lot of works require the execution of apps, which implies dealing with major hindrances: indeterministic runtime environment, automated yet realistic tests, invasive code instrumentation, hardly comparable nature of apps (e.g. Video Game vs. Social Network), etc. In contrast, these pitfalls are avoided by solutions at design-time. Nevertheless, the latter often suffer from the same flaw: they focus on well-known poor object-oriented designs (Blob Class, Feature Envy, Long Method...) and Java idioms (for loop instead of for-each loop...) instead of focusing on the power-intensive, high-level constructs of Android apps. Nevertheless, the most related work on this paper is as follows:

The PAPRIKA toolkit [17] is used for detecting 7 antipatterns. From an APK file, the tool (based on SOOT) produces a graph model stored in Neo4J database. Then, the graph is queried with CYPHER for the detection phase.

In [18], Android source code is modeled as a directed graph under the TGraphs representation format. The authors then use GReQL to both detect and restructure 7 energy code smells.

In [19] the authors base their work on 5 built-in checks of Android lint that they consider to be energy-related. Based on an existing tool chain, their solution called Leafactor automatically refactor Java source code and xml layouts into the Eclipse IDE.

With Enersave API [20], the authors circumvent the problem by suggesting to developers to use a specific API instead of the genuine Android API. This custom API is a wrapper that embeds timeouts and other optimizations to ensure energy savings when using the network, location, screen, maps and bluethooth.

In [21], the authors proposed a tool named Relda2 and written in Python. It disassembles the APK file into Dalvik bytecode and constructs a function call graph. Relda2 then analyses the resource request call and release calls for resource leak detection. Considered method calls are provided to the tool by the developer.

In [22], a technique called SAAD (Static Application Analysis Detector) focuses on resource leak and layout defect issues. To address the first issue, they decompile APK file into Dalvik bytecode and perform a components call relationship analysis. To address the second issue, they analyze the reports (xml output) of Android lint which comes with built-in checks for layouts.

During his thesis work, Reimann [23] laid the foundations for a catalogue of quality smells for Android, including energy efficiency. The technical solution chosen to detect and refactor them is based on model-driven engineering tooling.

VII. Conclusion

A linter is a tool that highlight suspicious code with the final objective to improve the global quality. Since energy efficiency became an important quality concern for application software developers [24], it becomes urgent to augment linters with green coding rules. To illustrate that, this paper focused on eleven green bugs encountered in real-life Android developments, and whose impact on energy consumption was assessed by various sources. Then they have been implemented as custom checks within the extensible Android lint framework for a seamless integration with Android Studio, the official IDE for the development of native apps.

The green software research field is still in its infancy, and there are virtually no tools with a sufficient level of acceptability and maturity for the mobile developer community. Leveraging from the well-known Android lint, the proposed solution has the great potential to be one of them. Of course, not all green bugs can be statically checked, but this contributes to the green-by-design principles.

The work in progress consists of expanding the catalog of Android-specific energy-related bugs/smells and making them publicly available (See [25]). Future work will consist of discovering unexpected or undocumented smells through learning techniques.

References

[1] L. Zhang, B. Tiwana, R. P. Dick, Z. Qian, Z. Morley Mao, Z. Wang, L Yang, “Accurate Online Power Estimation And Automatic Battery Behavior Based Power Model Generation for Smartphones”. CODES+ISSS’10, October 24-29, Scottsdale, Arizona, USA, 2010.

[2] C. Wilke, S. Richly, S. Götz, C. Piechnick and U. Aßmann, “Energy Consumption and Efficiency in Mobile Applications: A User Feedback Study,” 2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing, Beijing, 2013, pp. 134-141.

[3] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E Hassan. “What do programmers know about software energy consumption?”, IEEE Software, 33(3):83–89, 2016.

[4] Sarra Habchi, Xavier Blanc, and Romain Rouvoy. 2018. “On adopting linters to deal with performance concerns in Android apps.” In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018). ACM, New York, NY, USA, 6-16.

[5] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. 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). ACM, New York, NY, USA, Article 5, 6 pages.

[6] Jiehong Wu, I. Detchenkov and Yang Cao, “A study on the power consumption of using cryptography algorithms in mobile devices”, 2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS), Beijing, 2016, pp. 957-959.

[7] J. Gui, S. Mcilroy, M. Nagappan and W. G. J. Halfond, “Truth in Advertising: The Hidden Cost of Mobile Ads for Software Developers” 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, 2015, pp. 100-110.

[8] Shaiful Chowdhury, Silvia Di Nardo, Abram Hindle, and Zhen Ming Jiang. 2018. “An exploratory study on assessing the energy impact of logging on Android applications”. Empirical Softw. Engg. 23, 3 (June 2018), 1422-1456.

[9] Sanae Rosen, Ashkan Nikravesh, Yihua Guo, Z. Morley Mao, Feng Qian, and Subhabrata Sen. “Revisiting Network Energy Efficiency of Mobile Apps: Performance in the Wild”. In Proceedings of the 2015 Internet Measurement Conference (IMC '15). ACM, New York, USA, 339-345.

[10] M. Wan, Y. Jin, D. Li and W. G. J. Halfond, “Detecting Display Energy Hotspots in Android Apps”, 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), 2015, pp. 1-10.

[11] Andrea K. McIntosh, Safwat Hassan, Abram Hindle, “What can Android mobile app developers do about the energy consumption of machine learning?”, Empirical Software Engineering 24(2): 562-601 (2019).

[12] M. Linares-Vásquez, C. Bernal-Cárdenas, G. Bavota, R. Oliveto, M. Di Penta and D. Poshyvanyk, “GEMMA: Multi-objective Optimization of Energy Consumption of GUIs in Android Apps”, 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), Buenos Aires, 2017, pp. 11-14.

[13] T. Agolli, L. Pollock and J. Clause, “Investigating Decreasing Energy Usage in Mobile Apps via Indistinguishable Color Changes”, 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), Buenos Aires, 2017, pp. 30-34.

[14] D. Li, S. Hao, J. Gui and W. G. J. Halfond, “An Empirical Study of the Energy Consumption of Android Applications,” 2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, 2014, pp. 121-130.

[15] D. Burgstahler, U. Lampe, N. Richerzhagen and R. Steinmetz, “Push vs. Pull: An Energy Perspective (Short Paper),” 2013 IEEE 6th International Conference on Service-Oriented Computing and Applications, Koloa, HI, 2013, pp. 190-193.

[16] H. Höpfner and C. Bunse, “Towards an Energy-Consumption Based Complexity Classification for Resource Substitution Strategies”, Proceedings of the 22nd Workshop "Grundlagen von Datenbanken 2010", Bad Helmstedt, Germany, May 25-28, 2010.

[17] G. Hecht, O. Benomar, R. Rouvoy, N. Moha and L. Duchien, “Tracking the Software Quality of Android Applications Along Their Evolution (T),” 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, 2015, pp. 236-247.

[18] M. Gottschalk, M. Josefiok, J. Jelschen and A. Winter, “Removing Energy Code Smells with Reengineering Services”, in Informatik 2012, pp. 441-455.

[19] L. Cruz, R. Abreu and J. Rouvignac, “Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring,” 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), 2017, pp. 205-206.

[20] A.M. Muharum, V.T. Joyejob, V. Hurbungs, Y. Beeharry, “Enersave API: Android-based power-saving framework for mobile devices,” Future Computing and Informatics Journal, Volume 2, Issue 1, 2017, pp 48-64.

[21] T. Wu, J. Liu, X. Deng, J. Yan and J. Zhang, “Relda2: An effective static analysis tool for resource leak detection in Android apps,” 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore, 2016, pp. 762-767.

[22] Jiang H., Yang H., Qin S., Su Z., Zhang J., Yan J. (2017) “Detecting Energy Bugs in Android Apps Using Static Analysis”. In: Duan Z., Ong L. (eds) Formal Methods and Software Engineering. ICFEM 2017. Lecture Notes in Computer Science, vol 10610. Springer.

[23] Reimann, J., Brylski, M. & Aßmann, U. (2014). “A Tool-Supported Quality Smell Catalogue For Android Developers,” Softwaretechnik-Trends, 34.

[24] Pinto, Gustavo & Castor, Fernando. (2017). “Energy efficiency: A new concern for application software developers”, Communications of the ACM. 60, 68-75.

[25] Energy Smells For Android. (Online) https://pauware.univ-pau.fr/green/android-energy-smells/