iJRASET
International Journal For Research in Applied Science and Engineering Technology



INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 13 **Issue:** V **Month of publication:** May 2025

DOI: https://doi.org/10.22214/ijraset.2025.71004

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue V May 2025- Available at www.ijraset.com

Detection of Energy Leaks on Android Apps Using Machine Learning

Mrs. R. Jayalakshmi[1], Dr. R.G. Suresh Kumar[2], A. Divya Priya[3], A . Dheepika[4], D. Jenifer[5]
[1]*AssisantProfessor,*[2]*Head of the department,* [3, 4, 5]*UG Scholar, Dept of Computer Science and Engineering, Rajiv Gandhi College of Engineering and Technology, Puducherry, India*

Abstract: ***With the growing reliance on smartphones, optimizing the energy efficiency of mobile applications has become increasingly important due to the limited capacity of device batteries. Android, being the dominant mobile operating system, is particularly prone to energy inefficiencies caused by improper handling of power-intensive components.This paper introduces a machine learning-driven approach to identify energy inefficiencies in Android apps, with a focus on wake lock mismanagement and improper resource handling. The model is trained on a dataset composed of static features extracted from APKs, which reflect behavioral indicators such as the invocation of power-sensitive APIs, access to permissions, and usage of application components.***

Since direct energy leak annotations are not typically available in such datasets, proxy labels were applied based on observable behavior patterns, categorizing applications into Normal, Wake Lock Leak, and Resource Leak classes. To improve model performance on imbalanced data, the Synthetic Minority Oversampling Technique (SMOTE) was used. A Light Gradient Boosting Machine (LightGBM) classifier was trained and fine-tuned using RandomizedSearchCV, achieving a classification accuracy of 99%. The results indicate strong generalization and high performance across all categories, demonstrating the effectiveness of the proposed approach in identifying energy-related issues in Android apps.

Keywords: Energy Efficiency, Machine Learning, Android Apps,Wake Lock Mismanagement

I. INTRODUCTION

Android, as the dominant mobile operating system globally, powers more than 70% of smartphones in active use. Its open-source architecture, broad hardware compatibility, and extensive developer support have positioned it as the platform of choice for mobile application development. With millions of applications hosted on the Google Play Store, the Android ecosystem continues to expand, offering diverse functionalities and rich user experiences.Although there has been significant advancement in developing feature-rich applications, energy efficiency continues to be an often overlooked issue. Mobile devices are inherently limited by battery capacity, making efficient power consumption a critical metric for application performance. However, in practice, developers often prioritize functional features and user interfaces over energy optimization, inadvertently introducing inefficiencies that impact battery life and user satisfaction[7].

A key reason for this imbalance lies in the complexity of the Android application lifecycle and its event-driven architecture. Android apps are designed to handle diverse triggers, including user interactions, system-generated broadcasts, and asynchronous system events.This decentralized control flow can lead to overlooked execution paths, where developers may unintentionally hold system resources or fail to release them properly. These programming oversights often manifest as **energy leak**scenarios where an application consumes system power without serving user needs.[16]

A recent study investigated energy consumption issues in 32 highly rated mobile applications across 16 categories and found that all of them exhibited energy inefficiencies despite their positive user ratings. Out of 14,064 user reviews collected, 8,007 specifically mentioned concerns related to energy usage. Interestingly, many users gave favorable ratings while still expressing dissatisfaction with battery performance, with comments such as “Works well, drains battery” and “Good, battery usage is very high”. This underscores a growing awareness and sensitivity among users regarding the energy impact of mobile applications, emphasizing the need for developers to prioritize energy-efficient design and implementation[21].

Power inefficiencies in Android applications are typically divided into two key types: wake lock mismanagement and improper resource deallocation.Wake locks are used by developers to keep the device’s CPU or screen awake to complete critical tasks. However, if wake locks are acquired but not released correctly, the device remains in a high-power state, leading to excessive battery drain.

©IJRASET: All Rights are Reserved

Similarly, resource leaks occur when system resources such as file handles, network sockets, GPS modules, or sensors are allocated but not released, causing prolonged or redundant energy consumption[16].

These inefficiencies are often associated with code smells, which are indicative patterns of poor design or flawed implementation that do not necessarily produce immediate errors but compromise code maintainability and performance. In the context of energy management, code smells typically emerge from improper API usage, misinterpretation of the Android framework, or insufficient consideration of lifecycle state[20] [9].

As the demand for energy-efficient mobile applications increases, addressing these issues has become essential. Accurate detection of energy leaks, particularly wake lock and resource leaks, is a prerequisite for improving battery performance and ensuring sustainable application development practices in the Android ecosystem.

Improper management of power-related components and system resources in Android applications often results in energy leaks, which can severely impact device performance and battery longevity. A common example is a wake lock leak, where an application acquires a wake lock to prevent the device from sleeping, such as during music playback, but fails to release it properly after use. For instance, if a developer acquires a PARTIAL_WAKE_LOCK to keep the CPU active but omits the corresponding release() call, the device remains in a high-power state even after the task is complete, leading to unnecessary battery drain[7].

The following code snippet illustrates a potential wake lock leak arising from improper resource management within an Android application. The updateWakeLock() method is responsible for acquiring and releasing a partial wake lock based on the logging state of the application. When the logging state is active (Constants.LOGGING), a new wake lock is acquired using PowerManager.PARTIAL_WAKE_LOCK, ensuring that the CPU remains awake during data logging. However, the code registers a shared preference change listener without a corresponding unregistration and does not guarantee the release of the wake lock in all execution paths.When the method is triggered multiple times during an active logging state, it can lead to the accumulation of wake locks without proper release. This issue may arise if release conditions are skipped due to changes in the app's lifecycle or control flow [25].

This oversight can lead to the device remaining in a high-power state unnecessarily, thereby draining the battery even when the app is no longer performing critical tasks. Such issues are often introduced due to the complexity of Android's event-driven lifecycle and asynchronous behavior. This scenario exemplifies a wake lock leak, a prevalent form of energy inefficiency in Android applications, and highlights the necessity for developers to ensure all acquired system resources are explicitly and consistently released[5].

Similarly, resource leaks occur when applications allocate system resources such as file streams, GPS sensors, or network connections without releasing them. An example includes failing to close a FileInputStream after reading data or neglecting to stop GPS updates via removeUpdates() in a location-based app. These oversights can result in continuous resource usage in the background, depleting battery life and affecting overall performance [24].

Such programming flaws are often attributed to the complexities of Android's event-driven architecture and lifecycle, where multiple entry points and asynchronous callbacks increase the likelihood of missed release operations. These patterns are typically referred to as code smellsindicators of suboptimal coding practices that, while not always causing immediate failures, compromise the efficiency and maintainability of the application[15].

II. RELATEDWORKS

Early studies into Android power management uncovered critical issues related to improper use of system resources, particularly wake locks and sensors. Pathak et al. were among the first to identify "no-sleep" bugserrors caused by unreleased wake locks that prevent devices from entering low-power states. Their dataflow-based analysis provided key insights into wake-lock misuse[1]. Building on this, Liu et al. introduced *GreenDroid* [17], a tool that performs static analysis to automatically detect energy inefficiencies linked to sensor and wake lock management. In subsequent work, *Elite* was developed as a refined static analysis framework that identifies mismanagement of wake locks by recognizing common usage patterns [2]. These foundational efforts established the significance of systematic energy bug detection and laid the groundwork for the development of automated analysis tools.

Static analysis continues to be a dominant technique for identifying energy-related defects in mobile applications. Jiang et al. proposed *SAAD*, a framework that statically analyzes decompiled Dalvik bytecode to detect energy bugs through inter-procedural analysis[6]. Wu et al. developed *Relda2*, a lightweight tool that constructs Function Call Graphs and Callback Graphs to enhance precision in detecting resource leaks, thereby reducing false positives and negatives[10].

Xu et al. introduced *Statedroid*, which employs state-taint analysis to monitor resource usage transitions, effectively identifying complex energy leaks involving multiple resources[5]. Pereira et al. extended the capabilities of *EcoAndroid* by adding support for inter-procedural static analysis targeting key resources such as cursors, databases, cameras, and wake locks. More recently, Campelo et al. created *E-APK*, a rule-based detection system built on Kadabra's abstract syntax tree (AST) representation. Notably, *E-APK* can operate on both source code and decompiled APKs, making it useful even when full source access is unavailable[14][15].

While static tools offer scalability, dynamic analysis provides runtime accuracy by simulating real app behavior. Tools like *E-GreenDroid* and *NavyDroid* [9][10] monitor how applications interact with wake locks and sensors during execution, revealing misuse under various app states[3][4]. Hybrid approaches have also gained attention, offering a balance between coverage and precision. *EnergyPatch* combines static bug detection with dynamic validation to confirm the presence of energy leaks and proposes automated fixes[8]. Abbasi et al. explored *application tail energy bugs* (ATEBs)instances where apps continue to consume power post-exit—and developed a Java-based diagnostic tool leveraging Android Debug Bridge (ADB) to collect and analyze system-level data for identifying such inefficiencies[12].

In recent years, there has been a growing interest in utilizing machine learning methods to improve the detection of energy inefficiencies in mobile applications.Zhu et al. applied classification algorithms to system call traces to identify code changes that increase energy consumption. Their model supports early detection of potential regressions[6]. Khan et al. [14] introduced a testing framework that simulates realistic usage patterns to trigger energy issues, using clustering and filtering techniques to isolate anomalous power-draining processes. Their approach dynamically adjusts test cases based on observed behaviors, outperforming several traditional methods[7]. Li et al. further improved detection by integrating runtime context modeling and workload analysis to enhance accuracy and coverage in identifying inefficient addition to detection techniques, efforts have components[22].

In been made to quantify and model power usage in Android applications. Le et al. presented a power consumption automaton (PCA) that models transitions between various hardware power states (e.g., GPS, Wi-Fi, CPU). By refining these transitions algorithmically, the model estimates application-level energy usage and provides actionable feedback for optimizing resource consumption. This framework enables developers to understand and visualize the energy behavior of their apps, contributing to better design and power management strategies.

III. METHODOLOGY

We work presents a machine learning approach to identify energy-related issues in Android applications using static features extracted from APKs. The proposed workflow consists of five main stages: proxy labeling of the dataset, data preprocessing, class imbalance correction, model training and optimization, and final deployment through a user-accessible system.

we utilized an Android malware dataset to detect energy leaks in mobile applications. Although the dataset was originally designed for malware detection, many malicious apps exhibit behaviors that are also indicative of energy inefficiencies, such as prolonged wake locks, continuous background service usage, and frequent system calls. These overlapping characteristics make the dataset a useful proxy for studying energy-related issues. We adapted the dataset by focusing on features linked to energy consumption patterns, including resource usage and power-related API calls. This approach enabled us to train and validate our energy leak detection model effectively. While this dataset was not purpose-built for energy analysis, we acknowledge this as a limitation and propose the development of dedicated energy leak datasets as future work.

Following proxy labeling, the dataset was labeled based on heuristics derived from known energy leak patterns found in Android malware samples, including behaviors such as abnormal wake lock usage and improper resource management. This indirect labeling approach was necessary due to the lack of publicly available energy leak datasets. Once labeled, the dataset underwent a rigorous cleaning process to eliminate incomplete, inconsistent, or duplicate entries. Feature vectors were validated for structural consistency to ensure compatibility with downstream machine learning workflows. The processed dataset retained a binary matrix format, where each column represented a specific static feature and each row corresponded to an individualAPK[26].

Exploratory data analysis revealed significant class imbalance, with a predominance of energy-efficient (normal) applications. To address this, we applied the Synthetic Minority Oversampling Technique (SMOTE), which synthetically augments underrepresented classes (wake lock leaks and resource leaks) by interpolating between nearby samples. This improved class balance and helped prevent model bias during training.

The classification task was conducted using the Light Gradient Boosting Machine (LightGBM), chosen for its robustness with sparse, high-dimensional data and efficiency during training.The dataset was partitioned into 80% for training and 20% for testing to evaluate the model's performance effectively.

Model evaluation employed standard metrics such as accuracy, precision, recall, and F1-score to measure performance across all three classes, with a particular focus on detecting energy leaks accurately. Finally, the trained model was deployed via a Flask-based web interface, enabling users to upload APK-derived feature vectors and receive predictions indicating whether the app is likely affected by a wake lock leak, a resource leak, or is energy-efficient. To enhance practical usability, interpretability features were also included to highlight key indicators influencing each prediction, supporting developers in targeted debugging and optimization.

IV. DESIGN AND IMPLEMENTATION

The proposed system follows a modular architecture, as depicted in Figure 1. It begins with the acquisition of a dataset derived from static analysis of Android malware applications, followed by several stages: data preprocessing, class balancing using SMOTE, dataset splitting, model training using LightGBM, model evaluation, and deployment using a Flask-based web interface. The user interacts with the interface to upload APK-derived feature files, which are then classified by the trained model.

Each component of the system was developed with modularity in mind, ensuring independence, reusability, and scalability.This modular structure provides the flexibility to experiment with alternative models or preprocessing techniques without disrupting the overall workflow.

Since publicly available malware datasets typically lack direct labels for energy-related behaviors, we adopted a *proxy labeling* strategy to enable supervised learning. Static features were extracted from Android apps, including usage of wake locks, background services, sensor access, and power-intensive APIs—features commonly associated with energy inefficiencies. Based on observed patterns in the static code features and known energy misuse behaviors in malware, instances were heuristically labeled as Normal, Wake Lock Leak, or Resource Leak. This proxy labeling approach allowed us to infer energy leak classes using indirect evidence, compensating for the lack of explicit annotations.Although this introduces a level of abstraction, it enables meaningful classification and analysis of energy leaks within the context of Android malware analysis[26].

Data preprocessing was critical to ensure reliability. The dataset underwent a thorough cleaning process, including removal of missing values, duplicate records, and noisy or inconsistent entries. Categorical features were encoded using one-hot encoding, and irrelevant or highly correlated attributes were eliminated using correlation matrices and domain-specific insights.

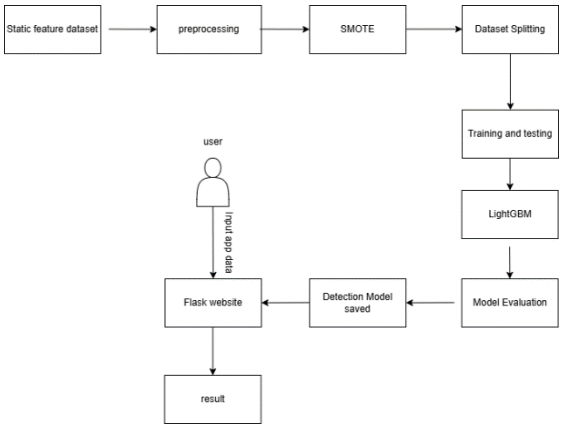


Fig 1. Architecture diagram

While LightGBM is robust to unscaled data, normalization was still applied to maintain uniform feature contributions. Notably, no feature reduction or transformation was performed—all available features were used to preserve maximum representational capacity for leak detection. This comprehensive approach, combined with proxy labels, provided the foundation for effective model training.

The dataset exhibited significant class imbalance: most samples were labeled "Normal," with relatively few representing the leak categories. To mitigate this issue, the Synthetic Minority Oversampling Technique (SMOTE) was utilized to balance the class distribution.. SMOTE generates synthetic examples for the minority classes by interpolating between existing samples, enhancing the model's ability to generalize and recognize underrepresented patterns such as Wake Lock and Resource Leaks.

To ensure a representative distribution of classes, the dataset was split into 80% training and 20% testing sets using stratified sampling.Stratification preserved the class distribution across both sets, providing a balanced evaluation. Additionally, k-fold cross-validation was used during training to ensure robustness and reduce the risk of overfitting.

LightGBM (Light Gradient Boosting Machine) was chosen for its speed, scalability, and superior performance on structured data. Without applying feature extraction, the model was tuned using randomized search for optimal hyperparameters. During training, signs of overfitting were observed, prompting regularization and parameter tuning to improve generalization.

The final model, trained on the proxy-labeled dataset, achieved 99% classification accuracy, with strong performance on minority leak classes. LightGBM demonstrated strong performance in the multi-class classification task, benefiting from features such as gradient-based one-side sampling, exclusive feature bundling, and integrated regularization techniques.

The model's performance was assessed using several evaluation metrics, including accuracy, precision, recall, F1-score, and an analysis of the confusion matrix.The model achieved an F1-score exceeding 0.98 for both leak categories, demonstrating reliable detection of energy issues even with proxy labels.

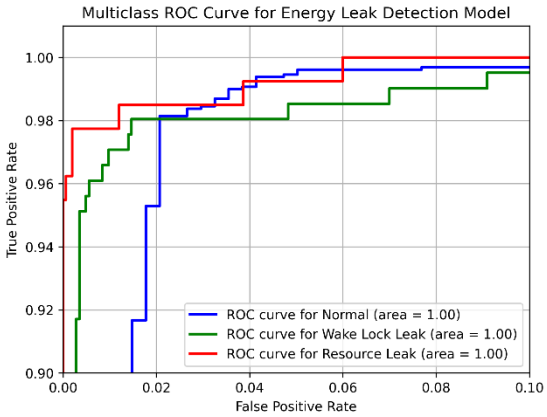


Fig 2. ROC curve for energy leak detection model

ROC curves were generated for each classNormal, Wake Lock Leak, and Resource Leakbased on the predicted probabilities. As shown in Figure 2, all ROC curves are concentrated in the top-left region, with Area Under the Curve (AUC) values approaching 1.00. This indicates excellent separability and high diagnostic accuracy using the selected static features.

The trained model was serialized using joblib, allowing for persistent reuse in a production environment. A web-based front end was developed using Flask to facilitate easy interaction. The system allows developers and analysts to upload a feature file extracted from an Android APK and receive immediate classification results.

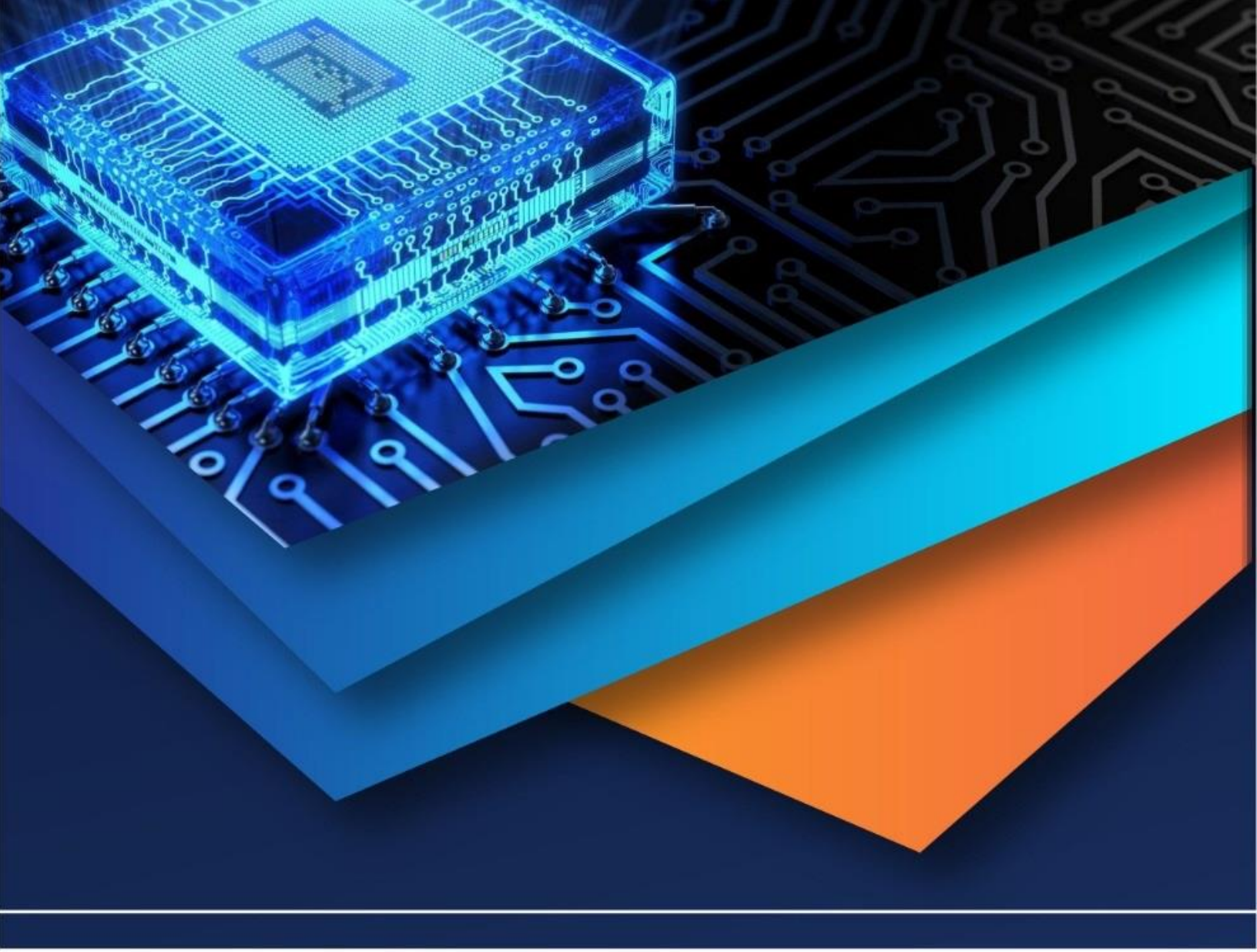
The output includes the predicted class, probability scores for each leak type, and a summary of feature contributions, offering insights into potential energy inefficiencies. This feedback mechanism supports rapid testing, debugging, and optimization of mobile applications from an energy consumption perspective**.**

V. CONCLUSION

We present a machine learning-based framework for detecting energy inefficiencies—specifically, wake lock leaks and resource leaks—in Android applications using a proxy-labeled dataset derived from static analysis of malware samples. The dataset comprises key indicators such as API calls, permission requests, and component usage patterns associated with power consumption behavior. Proxy labels were heuristically assigned based on known energy misuse patterns to facilitate supervised learning in the absence of ground truth annotations. Leveraging LightGBM, tuned via RandomizedSearchCV and enhanced with SMOTE to address class imbalance, the proposed model achieved a classification accuracy of 99%. It effectively distinguishes between Normal, Wake Lock Leak, and Resource Leak classes, offering actionable insights into energy-related issues in mobile applications. A Flask-based web interface was developed to ensure accessibility and ease of integration into development workflows. This tool empowers developers to proactively detect and mitigate energy inefficiencies during the software development lifecycle, contributing to improved battery performance and user experience. Future work may include expanding the feature set, incorporating real-time behavioral analysis, and detecting code smells that contribute to energy leaks**.**

REFERENCES

[1] Pathak, A., Jindal, A., Hu, Y. C., & Midkiff, S. P. (2012). What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys '12), Lake District, UK. ACM. https://dl.acm.org/doi/10.1145/2307636.2307661
[2] Liu, Y., Xu, C., Cheung, S. C., & Terragni, V. (2016). Understanding and detecting wake lock misuses for Android applications. In Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE '16). ACM. https://dl.acm.org/doi/10.1145/2950290.2950297
[3] Wang, J., Liu, Y., Xu, C., Ma, X., & Lu, J. (2016). E-GreenDroid: Effective energy inefficiency analysis for Android applications. In Proceedings of Internetware '16, Beijing, China. ACM. https://dl.acm.org/doi/10.1145/2993717.2993720
[4] Liu, Y., Wang, J., Xu, C., & Ma, X. (2017). NavyDroid: Detecting energy inefficiency problems for smartphone applications. In Proceedings of Internetware 2017, Shanghai, China. ACM.https://dl.acm.org/doi/10.1145/3131704.3131705
[5] Xu, Z., Wen, C., & Qin, S. (2018). State-taint analysis for detecting resource bugs. Science of Computer Programming, 162, 93-109. Elsevier. https://doi.org/10.1016/j.scico.2017.06.010
[6] Zhu, C., Zhu, Z., Xie, Y., Jiang, W., & Zhang, G. (2019). Evaluation of machine learning approaches for Android energy bug detection with revision commits. IEEE Access, 7, 85241–85252. https://doi.org/10.1109/ACCESS.2019.2924953
[7] Khan, M. U., Lee, S. U., Abbas, S., Abbas, A., & Bashir, A. K. (2021). Detecting wake lock leaks in Android apps using machine learning. IEEE Access, 9. https://doi.org/10.1109/ACCESS.2021.3110244
[8] Banerjee, A., & Roychoudhury, A. (2015). EnergyPatch: Repairing resource leaks to improve energy efficiency of Android apps. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (FSE), 37-49. https://doi.org/10.1145/2786805.2786827
[9] Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A., & De Lucia, A. (2019). On the impact of code smells on the energy consumption of mobile applications. Information & Software Technology, 105(105), 43–55.https://doi.org/10.1016/J.INFSOF.2018.08.004.
[10] Wu, T., Liu, J., Xu, Z., Guo, C., Zhang, Y., Yan, J., & Zhang, J. (2016). Light-weight, inter-procedural, and callback-aware resource leak detection for Android apps. IEEE Transactions on Software Engineering. https://doi.org/10.1109/TSE.2016.2547385
[11] Arnatovich, Y. L., Wang, L., Ngo, N. M., & Soh, C. (2018). A comparison of Android reverse engineering tools via program behaviors validation based on intermediate languages transformation. IEEE Access, 6. https://doi.org/10.1109/ACCESS.2018.2808340
[12] Abbasi, A. M., Al-Tekreeti, M., Naik, K., Nayak, A., Srivastava, P., & Zaman, M. (2018). Characterization and detection of tail energy bugs in smartphones. IEEE Access, 6. https://doi.org/10.1109/ACCESS.2018.2877395
[13] Jiang, H., Yang, H., Qin, S., Su, Z., Zhang, J., & Yan, J. (2017). Detecting energy bugs in Android apps using static analysis. In Proceedings of the IEEE International Conference. https://doi.org/10.1007/978-3-319-68690-5_12
[14] Pereira, R. B., Ferreira, J. F., Mendes, A., & Abreu, R. (2022). Extending EcoAndroid with Automated Detection of Resource Leaks. International Conference on Mobile Software Engineering and Systems, 17–27. https://doi.org/10.1145/3524613.3527815
[15] Campelo, F. P., Sousa, M. C. B. de O., & Nascimento, C. L. (2023). E-APK: Energy pattern detection in decompiled android applications. Journal of Computer Languages, 76, 101220. https://doi.org/10.1016/j.cola.2023.101220
[16] Khan, M. U., Abbas, S., Lee, S. U.-J., & Abbas, A. (2020). Energy-leaks in Android application development: Perspective and challenges. Journal of Theoretical and Applied Information Technology, 98(22), 2005–ongoing.
[17] Liu, Y., Xu, C., Cheung, S.-C., & Lu, J. (2014). GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications. IEEE Transactions on Software Engineering, 40(9), 911–940. https://doi.org/10.1109/TSE.2014.2323982
[18] H. Ahmed et al., "Evolution of Kotlin Apps in terms of Energy Consumption: An Exploratory Study," 2023 International Conference on ICT for Sustainability (ICT4S), Rennes, France, 2023, pp. 46-56, doi: 10.1109/ICT4S58814.2023.00014.
[19] Groza, C., Dumitru-Cristian, A., Marcu, M., & Bogdan, R. (2024). A Developer-Oriented Framework for assessing power consumption in mobile Applications: Android Energy Smells case Study. Sensors, 24(19), 6469. https://doi.org/10.3390/s24196469
[20] Fatima, I., Anwar, H., Pfahl, D., Qamar, U., College of Electrical and Mechanical Engineering, National University of Sciences and Technology, & Institute of Computer Science, University of Tartu. (2020). Detection and correction of Android-specific code smells and energy bugs: An Android Lint extension. QuASoQ 2020: 8th International Workshop on Quantitative Approaches to Software Quality, 71
[21] Sahin, C. (2024). Do popular apps have issues regarding energy efficiency? PeerJ, 10, e1891. https://doi.org/10.7717/peerj-cs.1891
[22] Li, X., Chen, J., Liu, Y., Wu, K., & Gallagher, J. J. (2022). Combatting Energy Issues for Mobile Applications. ACM Transactions on Software Engineering and Methodology, 32(1), 1–44. https://doi.org/10.1145/3527851
[23] Bhatt, B. N., & Furia, C. A. (2020). Automated Repair of Resource Leaks in Android Applications. arXiv: Software Engineering. https://doi.org/10.1016/j.jss.2022.111417
[24] Liu, Y., Wei, L., Xu, C., & Cheung, S.-C. (2016). DroidLeaks: Benchmarking Resource Leak Bugs for Android Applications. arXiv: Software Engineering. https://dblp.uni-trier.de/db/journals/corr/corr1611.html#LiuWXC16
[25] https://github.com/rcgroot/opengpstracker/blob/8ac7905a5ac78520c63adb864eb0765eca08cc56/application/src/nl/sogeti/android/gpstracker/logger/GPSLoggerService.java
[26] Yerima, Suleiman (2018). Android malware dataset for machine learning 2. figshare. Dataset. https://doi.org/10.6084/m9.figshare.5854653.v1
[27] Alkasassbeh, Mouhammd & Abbadi, Mohammad & Al-Bustanji, Ahmed. (2020). LightGBM Algorithm for Malware Detection. DOI:10.1007/978-3-030-52243-8_28.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)