Article

Wake Lock Leak Detection in Android Apps Using Multi-Layer Perceptron

Muhammad Umair Khan, Scott Uk-Jin Lee *, Zhiqiang Wu and Shanza Abbas

Department of Computer Science and Engineering, Hanyang University, Ansan 15588, Korea; mumairkhan@hanyang.ac.kr (M.U.K.); wzq0515@hanyang.ac.kr (Z.W.); shanza92@hanyang.ac.kr (S.A.)
* Correspondence: scottlee@hanyang.ac.kr

Abstract: With the proliferation of mobile devices, the popularity of Android applications (apps) has increased exponentially. Efficient power consumption in a device is essential from the perspective of the user because users want their devices to work all day. Developers must properly utilize the application programming interfaces (APIs) provided by Android software development kit to optimize the power consumption of their app. Occasionally, developers fail to relinquish the resources required by their app, resulting in a resource leak. Wake lock APIs are used in apps to manage the power state of the Android smartphone, and they frequently consume more power than necessary if not used appropriately (also called energy leak). In this study, we use a multi-layer perceptron (MLP) to detect wake lock leaks in Android apps because the MLP can solve complex problems and determine similarities in graphs. To detect wake lock leaks, we extract the call graph as features from the APK and embed the instruction and neighbor information in the node's label of the call graph. Then, the encoded data are input to an MLP model for training and testing. We demonstrate that our model can identify wake lock leaks in apps with 99% accuracy.

Keywords: wake lock; Android; oversampling; power consumption; multi-layer perceptron

Citation: Khan, M.U.; Lee, S.U.-J.; Wu, Z.; Abbas, S. Wake Lock Leak Detection in Android Apps Using Multi-Layer Perceptron. *Electronics* **2021**, *10*, 2211. https://doi.org/10.3390/electronics10182211

Academic Editor: Paulo Ferreira

Received: 24 July 2021
Accepted: 3 September 2021
Published: 9 September 2021

1. Introduction

We live in a technologically advanced world where the number of mobile devices is increasing rapidly owing to their mobility. However, these mobile devices have limited battery capacity and users need to operate these devices for the entire day. Different mobile companies are attempting to increase battery capacity; however, there are certain constraints on battery capacity (such as size and material) [1]. Therefore, efficient operating systems (OSs) and applications must be run on these devices. Android, iPhone OS and Windows OS are the most popular OSs in mobile devices, and manufacturers are working to make their OSs more power efficient by providing dark, and power efficiency modes [2]. Similarly, popular applications (apps) such as Facebook (https://play.google.com/store/apps/details?id=com.facebook.katana, accessed on 15 July 2021), WhatsApp (https://play.google.com/store/apps/details?id=com.whatsapp, accessed on 15 July 2021), and Google Chrome (https://play.google.com/store/apps/details?id=com.android.chrome, accessed on 15 July 2021) also provide a dark mode for increased power efficiency [3]. The number of Android devices has increased significantly [4], and the Android OS is the most popular OS in mobile devices because it is an open-source OS; moreover, most of the apps available are free for download, and are available for low-end mobile devices, thereby capturing large mobile user markets [5]. This also implies that a large community of developers is working to build new Android apps and update the existing apps to improve them. On average, 3700 apps are uploaded daily on Google Play Store [6]. Maintaining the quality of apps is necessary, which is measured based on the number of downloads, reviews, and ratings provided by the users [7]. The quality of certain apps suffers because developers of the app focus on functional requirements and ignore non-functional requirements such as performance [8], power consumption [9] and resource usage [10]. Apps should be power efficient, and developers must efficiently use the application programming interfaces (APIs) provided by the OS to control power-hungry components. The Android OS provides `WAKE_LOCK` API to control the power consumption of the apps [11].

These `WAKE_LOCK` API are used when the app needs to work in the background or prevent the CPU, display, and keyboard to go to sleep (if not used for some time). For example, when watching video on social media app or updating the app content in background when the phone is locked. If the lock acquired by these apps is left unattended (i.e., not released after completing work), they lead to unwanted power consumption. Carefully using wake lock is important, because it controls the power-hungry component (CPU, display) of the device [12].

The power consumption of the device and app can be measured using different tools [13]. To uncover unwanted power consumption, different tools are available for detecting power leaks, such as static [14], dynamic [15] or hybrid analysis [16]. Static analysis tools use function call graph (FCG) [17], control flow graph (CFG) [18] and, data flow graph (DFG) [19] to extract the information regarding the app. Li et al. [20] provided details of state-of-the-art tools that use static analysis in the Android app. They concluded, that Soot, a framework for optimizing Java bytecode [21], and Jimple [22], an intermediate language (IL), are adopted by most tools. This guide can be used to obtain the initial knowledge of static analysis tools. Code smells [23] are also used to detect the energy leak in Android apps; however the source code of the app is required to determine the code smells because the source code is analyzed at compile time to indicate the problems in the code. In contrast, dynamic analysis run the application to extract the flow of the app and identify the APIs used in the flow [24,25]. The primary disadvantage of using dynamic analysis is that it may not cover all the paths of the app because the app flow depends on the actions performed on the device. Dynamic analysis methods suffer from the construction of an execution environment and the creation of input data to inspect different paths [26].

These techniques have both advantages and disadvantages. The OS version is updated every year [27,28] with new features and changes to deprecate APIs to improve the quality of the OS. The developers of an app must make these changes and update their app to ensure its compatibility with the new version. The static and dynamic analysis tools will not adopt the new changes automatically and must be updated every time there is a change in the related APIs which is difficult. Notably, every time there is a major change in the API, a new tool is required to detect power consumption, such as in the case of GreenDroid [29] and Relda [30], which were upgraded to E-GreenDroid [24] and Relda2 [31], respectively, to provide functionality with the newer version. This problem can be overcome by a using the multi-layer perceptron (MLP) because it can be automatically trained using new data and can subsequently classify the apps.

MLP is a field of artificial intelligence that is currently used in different fields to determine the pattern and predict the output of a complex problem [32]. MLP minimizes manual human intervention to the largest extent and ensure that classification decisions are primarily dependent on the sample for automatic feature extraction and pattern recognition. It is also used for malware detection [33], user behavior prediction [34], app description analysis [35], and several other applications. In this study, we used MLP to detect wake lock leaks in Android apps and determined how it can effectively detect these leaks. Our work is the first to use MLP to detect wake lock leaks and can be extended to other resource leaks. The primary advantage of an MLP is that it can automatically learn the representation (features) from the raw data to perform the detection task.

In this study, we first determined how frequently a wake lock is used in the apps based on the permission declared in the apps. To train the MLP, we collected different apps from different studies and manually validated the problem of wake lock leaks from GitHub. The selected apps were preprocessed before training. Then, the MLP was applied to determine its efficiency in detecting wake lock leaks. We divided our study and answered the following questions:

- **RQ1: Are `WAKE_LOCK` permissions prevalent in Android apps?**
- **RQ2: Can MLP be used to detect wake lock leaks?**
- **RQ3: Is the performance of the MLP model better than that of traditional machine learning (ML) algorithms?**

We answer the RQ1 by collecting 800 popular apps from Google Play Store and we found that 98% of apps use `WAKE_LOCK` permission, which is second most popular permission in the dataset and the APIs of wake lock should be carefully used to prevent energy leakage. To answer the RQ2, we applied MLP and found that MLP is effective in detecting wake lock leak with high accuracy. We compared the accuracy of traditional ML algorithms with MLP in RQ3 and shows that ML algorithms can also be used to detect wake lock leak with a little less accuracy.

This paper is divided into different sections. In Section 2 covers the fundamentals of Android APK, wake lock leak with example, MLP, and synthetic minority oversampling technique (SMOTE). In Section 3, different related works are discussed. The wake lock leak detection model utilized in this study is presented in Section 4. The evaluation of the model is discussed in Section 5. In Section 6, we address limitations of our work, and in Section 7, we conclude our work.

2. Background

In this section, we provide an overview of Android apps, an example of wake lock leak, MLP and oversampling techniques.

2.1. Android Development Languages, Package and Components

Android apps are typically written in Java [36]. In 2011, a new programming language called Kotlin [37] was introduced and in 2017, Google announced support for Kotlin in the Android OS [38]. Kotlin was announced as the preferred language for Android app developers in 2019 by Google [39]. Android Native Development Kit [40] is a tool set that allows the implementation of a part of the app in native code using languages such as C and C++, which helps in the reuse of code libraries. The most popular Android app development environments are Android Studio (https://developer.android.com/studio, accessed on 15 July 2021) (which is the official Integrated Development Environment (IDE)) and Eclipse (https://www.eclipse.org/, accessed on 15 July 2021).

Developed apps are built into a package called APK format, which is a compressed file containing different files (classes.dex, resources.arsc and AndroidManifest.xml) and folders (res,lib, assets and META-INF) [41]. The different files and folders are as explained subsequently.

- META-INF: This folder contains the manifest file, signature, and a list of resources in the archive.
- lib: This folder contains native libraries that run on specific device architectures. Contains multiple directories, one for each supported CPU architecture.
- res: This folder contains resources, such as images that were not compiled into resources.arsc.
- assets: This folder contains raw resource files that developers bundle with the app.
- AndroidManifest.xml: This file describes the name, version, permissions, and contents of the APK file.
- classes.dex: This file includes the compiled Java classes to be run on the device.
- resources.arsc: This file comprises compiled resources, such as strings, colors or styles, used by the app.

The Android app contains the following four basic components:

- **Activity** is the only component that contains graphical user interfaces. An application may comprise multiple activities to provide a cohesive user experience.
- **Service** is a background component for performing long-running activities such as sensor reading. Services can be used to initiate activities and interact with them.

- **Broadcast receiver** describes how a program reacts to system-wide broadcast messages. It can be registered statically in the configuration file of an application or dynamically at runtime.
- **Content provider** handles shared application data and provide a query or modification interface for other components or apps.

A life cycle defines the production utilization, and disposal of an application component. A call to the onCreate() handler begins the life cycle of an activity, and it ends with a call to the onDestroy() handler. The foreground lifespan of an activity begins when the onResume() handler is triggered and ends when the onPause() handler is called when another activity comes to the forefront. The activity in the foreground can interact with the user. Its onStop() handler is invoked when it fades into the background and becomes invisible. The onResume() or onRestart() handler of an activity is triggered when users return to a paused or halted activity, and the activity returns to the foreground. Halted or interrupted activities can be terminated under unusual circumstances. A paused or halted activity may be terminated in rare circumstances to free memory for higher-priority programs.

2.2. Wake Lock Leak Example

The power consumption of Android devices can be controlled by the `PowerManger` API [42] provided by the OS. There are different functionalities provided by this API and `WAKE_LOCK` is one of the most important discussed in different research (explained in Section 3). Research shows that the display is the most power-consuming component of the device [12], and wake lock controls the display and CPU processing used by the device depending on the type of lock used [9]. We can use different levels of wake locks that demonstrate different effects on the power consumption of the device. These wake locks are important for developers because they need their app to work in the background or the display of the device to stay ON for a specific time. For example, if a user is playing a video on a video player app, the device should not go to sleep because the user is watching the video. If the device goes to sleep, then the user must unlock the device or touch the screen to prevent it from going to sleep. Wake locks can be acquired and released depending on the functionality of the app. These acquire, and release functions can be used in different methods and developers must carefully consider the flow of the program. If not carefully used, these methods can lead to a wake lock leak problem, which causes unwanted power consumption. We present an example of a wake lock leak that is not detected by Lint [43] because this tool only looks for a wake lock leak in the pause method and will not detect leaks in other paths of the program.

The sample wake lock leak presented in Listing 1 is explained subsequently. The example is from "VLC player" [44] having GitHub version "233C863". This is a fixed code that removes the wake lock problem. The red and green highlighted code illustrates the code that were removed and added in the revision, respectively. The app faces a wake lock problem when the play button is pressed multiple times. The functionality of the play button is to play the songs/videos and while playing, if the user wants to pause the songs/videos, the play button is pressed again. The OnClickListener() method (Lines 16–21) is responsible for capturing this event, which calls the doPausePlay() method. In the doPausePlay() method (Lines 22–28), if the player is playing the song/video, it will call the pause() method (Line 24); otherwise, it will call the play() method (Line 26). The play() method (Lines 29–34) plays the songs and acquires a wake lock; conversely, the pause() method (Lines 35–40) pauses the songs and releases the wake lock. However, if we press the play button multiple times, it will acquire the wake lock multiple times (Line 31) which is similar to the case with pause (Line 37), which will lead to unnecessary power consumption. To remove the wake lock leak, the developer added a code that checks if the wake lock is held, in the play() method (Lines 32–33). If it is not held, then the wake lock is acquired; otherwise, the song is merely played. Similarly, in the case with the pause() method, if the wake lock is held, then the wake lock is released, otherwise the song is merely paused (Lines 38–39).

Listing 1. Sample code of unnecessary wake-up in VLC player.

```

protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.player);
        // stop screen from dimming
        PowerManager pm = (PowerManager) getSystemService(Context.POWER_SERVICE);
        mWakeLock = pm.newWakeLock(PowerManager.SCREEN_BRIGHT_WAKE_LOCK, TAG);

    protected void onPause() {
        if (mLibVLC.isPlaying())
            pause();
            mLibVLC.pause();
        if (mWakeLock.isHeld())
            mWakeLock.release();
    }
private OnClickListener mPauseListener = new OnClickListener() {
        public void onClick(View v) {
            doPausePlay();
            showOverlay();
        }
    };
    private void doPausePlay() {
        if (mLibVLC.isPlaying()) {
            pause();
        } else {
            play();
        }
    }
private void play() {
        mLibVLC.play();
        mWakeLock.acquire();
        if (!mWakeLock.isHeld())
            mWakeLock.acquire();
    }
    private void pause() {
        mLibVLC.pause();
        mWakeLock.release();
        if (mWakeLock.isHeld())
            mWakeLock.release();
    }
}
```

2.3. Multi-Layer Perceptrons

MLP is a feedforward artificial neural network [45]. It is a supervised learning system composed of several basic components known as neurons or perceptrons. Each neuron can make basic decisions and feeds these decisions to other neurons, which are arranged in interconnected layers. The neural network (NN) can replicate virtually any function and answer almost any question if sufficient training data and processing power are provided. A "shallow" NN [46] has only three layers of neurons: an input layer, where the independent variables of the model or inputs are added, a hidden layer, and an output layer that generates predictions. Conversely, a "deep" NN [47] consists of multiple hidden layers, each of which comprises a significant number of artificial neurons. The neurons in each layer i are linked to layer $i + 1$, although the method of connection differs between the models. NN needs large amount of data that can be feed into the network to activate the neurons and update the weights for better training of input data and predict output. Sometimes the input data is not enough for the NN to train which lead to a data imbalance problem, and NN will not be able to predict the correct output. To solve the data imbalance problem in NN, oversampling techniques are used.

2.4. Synthetic Minority Oversampling Technique

SMOTE [48,49] is the most widely used approach to synthesize new examples from existing examples. SMOTE is used to produce minority class samples and is helpful in generating more data that can be used by ML algorithms or the MLP, which is also known as data imbalanced classification. It works by selecting a random example from the minority class. Then, the k nearest neighbors of a previously selected example are determined (default $k = 5$) and a neighbor is randomly selected. Now, we have two examples: one is selected randomly and the other is selected randomly from its neighbor in the feature space. The new synthetic instance is generated as a convex combination of the two selected instances. This process is repeated until the generated example of the minority classes is equal to the majority classes of the samples.

3. Related Work

Wake lock leaks are discussed in several other related works, and a brief overview of these works is presented subsequently. The methods for analyzing wake lock leaks are categorized into two main categories: static and dynamic analyses.

3.1. Static Analysis

In Android, static analysis can be applied to the source code or APK of the apps. Popular analysis techniques that use source code are based on identifying code smells. Code smell detection tools are used to analyze the source code of the app at compile time to identify the problem [50]. Android Lint is the official tool for detecting code smells when writing an app and it is integrated with Android Studio IDE. This tool identifies only two wake-lock-related code smells, which are "Incorrect Wakelock usage" (checks if the wake lock is released in `onPause()` not in `onDestroy()` method) and "Using wake lock without timeout" [51] (always used with timeout) [52]. aDoctor [53] is another tool that can detect 15 code smells, including durable wake lock. It is a fully automated tool and is based on the findings of Reimann et al. [54]. Palomba et al. [55] and Cruz et al. [23] identified 9 code smells and 22 design patterns, respectively, that affect the power consumption of mobile apps. Wake locks were common in these findings. This demonstrates that wake lock is a common pattern in energy leakage and must be detected and removed; however, these tools can only detect one or two wake lock problems. Liu et al. [56] provide the first detailed study on wake locks and identified eight patterns of wake lock misuse that can cause functional and nonfunctional issues. Only three patterns were used for detection in their study. These tools are used to guide developers in writing bug-free code by identifying resource leaks in their code. Developers should use these tools to minimize the power consumption of apps in their code. These tools can only detect the limited patterns defined in the tools.

Static analysis can also be performed on APK files, and different tools available to analyze the APK. Relda [30] and Relda-2 [31] are lightweight tools for locating resource leaks, including wake locks, and are based on the FCG to handle callbacks. Xu et al. [18] used state-taint analysis to detect "open-but-not-used" problems when a mobile device uses resources. In this study, the CFG is used as the input to track resource behaviors and identify the open resources in the different programs, which must be closed if unused. The authors compared the results with those of Relda [30,31] and GreenDroid [29] to indicate that their study could detect more energy leak patterns. Pathak et al. [19] used data flow analysis to detect no-sleep paths, which include wake lock, Global Positioning System (GPS), camera. They handled events based on their entry points, but only considered the open and closed states. Elite [56] and Verifier [57] use data flow analysis to detect wake lock leaks, but the false-positive rate is high while using these tools. Li et al. [20] showed that static analysis suffers from a lesser number of app analyses, high false-positive rates and the requirement of updating the tool to support a newer version of the OS or when there is a change in the APIs. Static features cannot accurately depict the run-time behavior of apps [58].

3.2. Dynamic Analysis

In dynamic analysis, the flow of the app is extracted by running the app on a mobile device or emulator and performing different operations at run-time. Pathak et al. [59] used a finite state machine-based power model to monitor the state of power consumed by different system calls. The app must be modified to insert the logging information of the different states, which must be manually inserted. Abbasi et al. [60] measured the power consumption of different wake lock types and their effect when the user performs different actions on the device (home button, back button, etc.). GreenDroid [29] and E-GreenDroid [24] perform state exploration using the Java Path Finder (JPF) to detect the wake lock deactivation function. Liu et al. [25] proposed the NavyDroid tool, which considers multiple patterns of wake lock misuses. It is an extension of E-GreenDroid [24] and does not address unnecessary wake-up patterns. Kurihara et al. [61] monitored the start and end of wake lock and GPS usage to estimate power consumption. They created their own benchmark apps to measure power consumption; however, no experiments were conducted using this tool while handling apps from the app store. Notably, dynamic analysis necessitates the installation of an app on the device and monitoring the status of the device when various operations are performed or when the user interacts with the app. This process is time consuming and requires more processing because the data are collected through a log file or Android Debug Bridge (ADB) [62].

To emphasize the issues and our contribution, we included some of the tools in Table 1. It listed the tools name, number of apps used in the evaluation, availability of tool online, detection technique used in the tool, and whether the tool operate on APK or source code. Table 1 listed some of the tools that were used for detecting wake lock leaks and other resource leaks. The main problem we see is that most of the tools are not open source (i.e., not available for use/comparison). Some of them analyze source code, which can be integrated with the development environment and used when developing the app. This study focuses on quality of APKs, available on app store. We also noticed that the number of apps used to evaluate tools were very less. The smaller number of apps cannot represent the vast market. This also shows that their tools cannot be used to process the vast number of apps such as app repositories. We also observe that these tools suffer when the relevant API changes or OS is updated to a newer version; they must be updated to reflect the new changes. These tools are not maintained on regular basis. To solve these issues, we used MLP, which can automatically adapt to new API changes and update its model with provided data. We used the large number of apps to train MLP and this trained MLP model can be used to detect wake lock leaks in large repositories such as Google Play Store. The source code of our model is available online (https://github.com/umkhanqta/MLPWake, accessed on 25 August 2021).

Table 1. Tools used to detect wake lock leaks.

Tool Name	No. of Apps	Availability	Detection Technique	App Types
aDoctor	18	Yes	Rule-based	Source Code
GreenDroid	13	On request	Model checking	Source Code
GreenDroid2	15	No	Model checking	Source Code
NavyDroid	17	No	Taint analysis	APK
Elite	31	Yes	Model checking	APK/Jar
Relda	98	No	Model checking	APK
Relda2	103	No	Model checking	APK
Verifier	328	No	Model checking	APK

4. Wake Lock Leak Detection Model

Our wake lock leak detection model consists of three stages, as shown in Figure 1. In the first stage, data is collected from different sources and labeled as "Clean APK" or "Leak APK". The labeling is performed manually. In the second stage, the labeled data is encoded by extracting useful information from the APK and preprocessing is performed. In the last stage, an MLP model is trained to detect an app having wake lock leaks. We elaborate on each stage as follows.

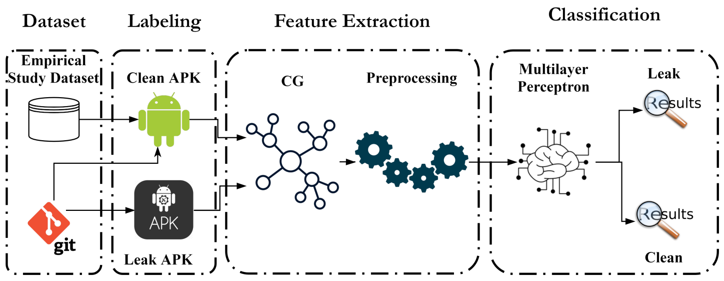


Figure 1. Multilayer perceptron (MLP) Model to Detect Wake Lock Leak.

4.1. Collecting and Labeling Data

Data collection is an important part of training the MLP. The apps that are used in training must be carefully identified. We used apps in the APK format to cover a large set of app data because most of the downloaded and highly rated apps are available in stores and can be freely downloaded; however, their source code is not available online. This is a problem faced by other studies in this field; consequently, not many apps have been identified as having wake lock leaks. To solve this problem, we used APK in our study; therefore, the MLP-trained model can be used to analyze apps from different app stores to ensure the quality of the app.

Identifying apps with excessive power consumption on the app store is challenging because only apps with significant battery drainage issues are reported as having energy leaks by users, who then award a low rating for the app [52]. These identified apps are then analyzed by the developer to reduce the power consumption. Most of the users do not comment on the primary problem and only enter general comments such as "bad app."

There are also certain apps whose source code is available in the GitHub (https://github.com/, accessed on 15 July 2021) repository, which can be used by the open-source community to enhance the capability of apps and reuse the code [63]. As these open-source codes are used by several other developers, bugs are identified and corrected early. When a bug is identified, it is assigned an issue number and is closed when it is resolved [64]. These issues have comments that define the problem in simple English with an error message for the developer; when the error is resolved, the developer includes comments on how the problem was resolved. This newly committed version of the code at GitHub is indicated with the added code in green, removed code in red; moreover, the files that are changed by the developer are also indicated. In our study, we collected apps from two sources:

- GitHub
- An empirical study on wake locks

We collected open-source apps from different studies and tools (as discussed in Section 3) that use the source code of the app to detect different resource leaks. We selected only the apps with wake lock leak problems from these studies and verified their GitHub versions manually to ensure that the downloaded apps have a wake lock leak in their code. We collected both clean and leaked versions of the identified apps. The collected GitHub dataset is summarized in Table 2. It lists the app name, fixed version, which contains the GitHub version of the app in which the wake lock leak has been rectified, leak version, which contains the version of the app where the problem of wake lock exists, and references to studies where the respective apps were used. The names of tools corresponding to the references are listed below the table. After collecting the data related to the clean/fixed and leak versions from GitHub repositories, we built the source code to generate the APK format using Android Studio.

Table 2. Applications having the wake lock leak problem and tools they were used.

App Name	Fixed Version	Leak Version	Tools
AndTweet	v. 0.2.6	v.0.2.4	[24,25,29]
CallMeter	4E9106C	10729EA	[10,31,56]
ConnectBot	540C693 F5D392e	669566E 76C4F80	[10,31,56]
CsipSimple	E50DF4E 1B2D2B6 45E35BC F6848D9 F1C8A2B 3AA981C C72156E 3246DE8	00EC304 75A269A 10D8D9A 352FC6C 04496E6 A388C20E B94C56F C72156E	[10,29,31,56], [10,24,65]
Cwac-wakeful	C7D440F	D984B89	[24,25,29]
Firefox	BE42FAE	———	[18,56,57,66]
IRCCloud	CE5822D	———	[10]
K9Mail	57E5573 58EFEE8 3171EE9 5918QQ3 F1232A1 ——— ——— ——— ———	E613228 0E03F26 2DF436E 7E15014 71A8FFC 378ACBD 1596DDF 3077E6A ACD1829	[10,31,56]
Open-GPSTracker	763B11E	8AC7905	[10,31,56]
OsmAnd	4D1C97F FE0060C	AC724B9 F314EA3	[10,18,56,57]
VLC	0493588 82FAE8A A9B911D 233C863	6A85C2A 14B18BC 19483DE ABE60F5	[10,31,56,67]

DroidLeak = [10], Relda = [18], E-GreenDroid = [24] , NavyDroid = [25], GreenDroid = [29], Relda2 = [31], Elite = [56], Verifier = [57], SENTINEL = [65], GreenMinning = [66], GreenOracle = [67], EnergyPatch = [68].

Clean apps were collected from the empirical study [56], which is a large database of apps with wake locks. This dataset contains 44,736 apps (http://sccpu2.cse.ust.hk/elite/downloadApks.html, accessed on 15 July 2021). We used 2778 apps that use wake lock in their code but do not have any wake lock leak. The distribution of wake lock permissions for these apps is shown in Figure 2. For easy readability of the figure, we only indicated the permissions that were requested over 1000 times in the dataset. Figure 2 illustrates that all the apps that were used from the empirical study dataset require `WAKE_LOCK` permission. We can add more apps from Google Play Store or other repositories, but if the app utilizes the wake lock permission, we cannot tell whether it has a wake lock leak. Therefore, we used data from the empirical study to ensure that the app does not have any wake lock leak.

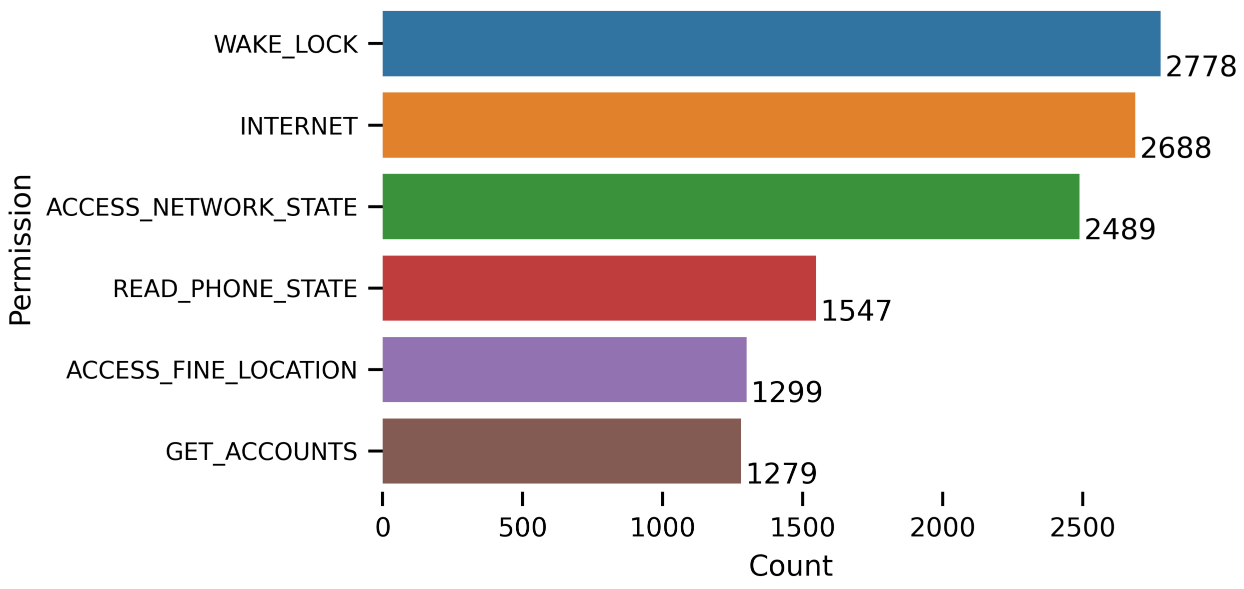


Figure 2. Wake Lock Permissions in Empirical Study Data.

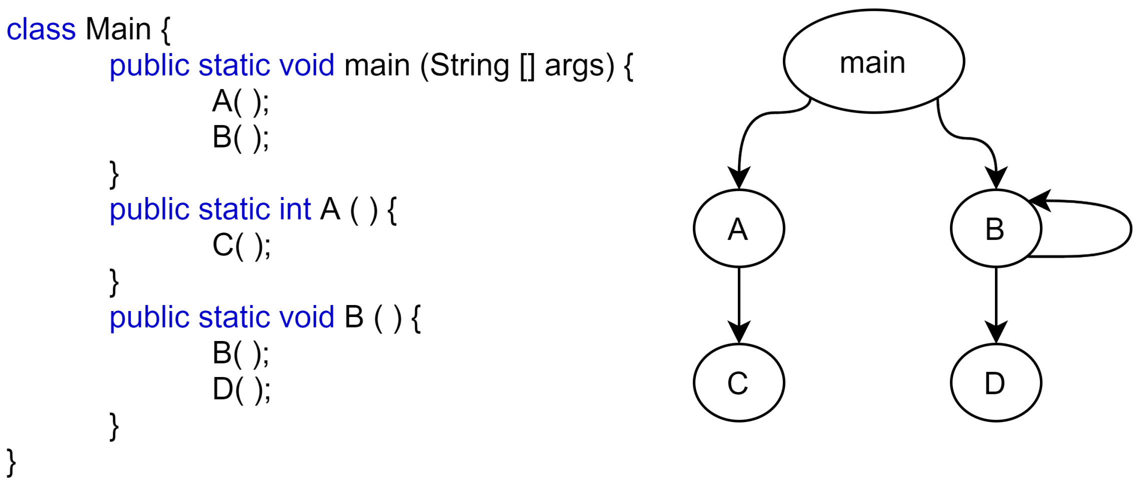
Three authors in the paper labeled the data manually by visiting the GitHub page of the app, finding the specific version, and reviewing the code changes by the developer. If the changes are related to wake lock API, then labeled the app as clean. We also find the previous version of the app and label it as a leak. The labeled data was finalized if two authors agreed on the same label for the app.

4.2. Features Extraction

After labeling the APK files, we extracted features from the apps so that they can be used as input for the MLP because APK files cannot be sent directly to the MLP. The APK contains .dex files, also known as Dalvik bytecode [69] (explained in detail in Section 2). Therefore, it is desirable to convert APK files into ILs. IL is the lowest-level human-readable programming language that is created automatically by reversing tools [70] by converting the executable code into its textual representation. There are different reverse engineering tools available for extracting information from the APK files, such as Soot (https://github.com/soot-oss/soot, accessed on 15 July 2021), APKtool (https://github.com/iBotPeaches/Apktool, accessed on 15 July 2021), Androguard (https://github.com/androguard/androguard, accessed on 15 July 2021). These tools convert APK files into ILs, such as Jimple, Jasmin, Smali. For example, Relda [30] and Relda2 [31] use Androguard to convert APK into Smali to detect wake lock leaks; similarly, APKtool and Soot can convert APK into Smali and Jimple, respectively. A comparison between these ILs is presented in a previous study [71], which concluded that Smali is the most accurate

IL, which maintains the program representation as it was written and is easily readable by humans.

We used Androguard to extract information from the APK files because it converts the APK into Smali, supports Python language (which we used for implementation), and extracts more user-friendly information. Androguard is used to extract a call graph (CG) from the APK files. A CG contains the information flow of the app, which illustrates how each method interacts with others. In Androguard, the CG is constructed using an `Analysis` object that generates a `DiGraph` (directed graph), which involves a pair $G = (V, E)$, where V is a set of vertices or nodes and E is a set of edges between different nodes. By default, these nodes are labeled as file names and method names. The instruction set of the method is stored in the attributes of the node. The labels of the nodes are important to identify which node represents which file, class, and method; however, for machines, the label is only considered as a string and does not provide much information; therefore, we updated these labels. The instructions contained in the node and connections between these nodes (edges) are important to provide a summary of the node and its surroundings. To encode the instructions and neighbor information of each node, we first encoded the labels of CGs using the instruction set contained in their node. In the CG, each method is represented as a node, and the interaction between these nodes is represented by edges, as shown in Figure 3. The figure shows the CG of the simple program that is shown on the left.



```
class Main {
        public static void main (String [] args) {
                A( );
                B( );
        }
        public static int A ( ) {
                C( );
        }
        public static void B ( ) {
                B( );
                D( );
        }
}
```

Figure 3. Sample of CG.

We consider basic Dalvik bytecode instructions [72] that are listed in Table 3. In Dalvik bytecode, there are 256 instructions; however, for simplicity, we only considered the basic instruction class; for example, *monitor* instruction has different variations such as *monitor-enter* and *monitor-exit*. The 15 bits are chosen because of two reasons, one they are most commonly used instructions and second is the study [73] proves that the comprehensive features are not suitable and shows that full opcode features have less accuracy and consume more time and space. The instruction class and labels can be represented as follows.

$$C = \{c_1, c_2, c_3, \ldots, c_m\} \tag{1}$$

$$nl(v) = [b_1, b_2, b_3, \ldots, b_m] \tag{2}$$

$$b_c(v) = \begin{cases} 1, & \text{if} f_v \text{ contains an instruction from } C \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

Here, C is the instruction class, as represented in Table 3. The label of the node v is represented as nl and the number of bits is represented by the field m. In our case, m is of 15 bits. f_v is a function associated with node v.

If we include more instructions, we require more bits (m) to represent them and will require greater memory and processing capability. These 15 instruction classes (based on Equation (1)) are represented using the 15 bit label of the node (according to Equation (2)). If a node contains the instructions from these classes, these bits are converted to one (according to Equation (3)). Table 4 lists a simple code, in which the "Instruction" column provides the instruction sequence used in the node, "Instruction Class" and "Bit" columns depict the equivalent instruction and bit representations, respectively, as listed in Table 3. From the sample code listed in Table 4, we can obtain the bit representation of the label, as listed in Table 5. Notably, only the bits pertaining to the instructions contained in the node were converted from "0" to "1". Furthermore, multiple "invoke" instructions present in the function would not affect the label bits once they were converted to "1", because instructions can be in a different order in different methods, but the methods perform the same functionality. This indicates that if the instruction sets in different nodes are the same, they will have the same label, which also reduces the complexity of the graph [17].

Table 3. Instruction class and corresponding bit representation.

Instruction Class	Bit	Instruction Class	Bit
new	1	invoke	9
monitor	2	staticop	10
test	3	instanceop	11
move	4	arrayop	12
return	5	branch	13
nop	6	binop	14
jump	7	unop	15
throw	8		

Table 4. Code example and representation in bit for each Instruction.

Instruction	Instruction Class	Bit
new-instance v0, Ljava/util/ArrayList;	new	1
invoke-direct{v0},Ljava/util/ArrayList;-><init>();	invoke	9
invoke-virtual{p0,p1,v0},Lcom/liato/urllib/Urllib;-> open(Ljava/lang/String;Ljava/util/List;)Ljava/lang/String;	invoke	9
move-result-object v0	move	4
return-object v0	return	5

Table 5. Code representation of label.

Bit	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Label	1	0	0	1	1	0	0	0	1	0	0	0	0	0	0

After encoding the instruction information of each node of the CG in the label, we must also provide neighborhood information of the node, which is computed using the neighborhood hash graph kernel (NHGK) [74]. It is a kernel that acts on the enumerable collection of sub-graphs of the labeled graph. It has low computational complexity and a highly expressive visual structure. The NHGK of each node can be computed by first identifying all its neighbors and then determining the XOR of their labels.

We can compute the hash of a given node v and the set of its adjacent nodes V_v using

$$h(v) = r(nl(v)) \oplus \left(\bigoplus_{z \epsilon V_v} nl(z) \right), \tag{4}$$

where r is a rotation to the left of a single bit and $\oplus$ represents a bit wise XOR on the binary labels. For each node, this computation can be performed in constant time, more precisely in $\mathcal{O}(md)$ time, where d is the maximum out-degree and m is the length of the binary label.

We can obtain greater details of a neighborhood by including a neighbor of the initially determined neighbor; however, this increases the complexity. NHGK is used to gather the neighborhood information of the function into a single hash value. The primary advantage of NHGK is that it runs in linear time on the number of nodes and processed graphs with thousands of nodes, such as CG. The label is replaced with the calculated hash value, and the number of bits of this hash value is identical to the label. Thus, we can obtain a hashed node that contains the information related to the instructions of the function and the neighborhood. After extracting and embedding the instructions and neighborhood information to the label of the node, we normalized the output in an array with 32,768 items, which was then used as the input to the MLP.

4.3. Classification Using Multi-Layer Perceptron

We chose the MLP because it has three different layers (input, output, and hidden layer). The signal to be processed is received by the input layer. The output layer completes the necessary operations, such as categorization. The real computational engine of the MLP consists of an arbitrary number of hidden layers positioned between the input and output layers. In an MLP, data travel in the forward direction from the input to the output layer, similar to a feedforward network [75]. The back propagation learning [76] technique was used to train the neurons in the MLP.

The following are the computations performed by each neuron in the output and hidden layers.

$$o(x) = G(b_2) + W_2 h(x) \tag{5}$$

$$h(x) = \Phi(x) = s(b_1 + W_1 x) \tag{6}$$

Here, $o(x)$ is the output layer, $h(x)$ is the hidden layer, b_1 and, b_2 are bias vectors, W_1 and, W_2 are weight matrices, and G and, s are activation functions. The parameters to be learned are W_1, b_1, W_2 and b_2.

As shown in Figure 4, our MLP consists of one input layer, one fully connected layer, and one output layer. We can increase the number of hidden layers, but this does not improve accuracy and may cause overfitting; therefore, we only used one fully connected layer [77].

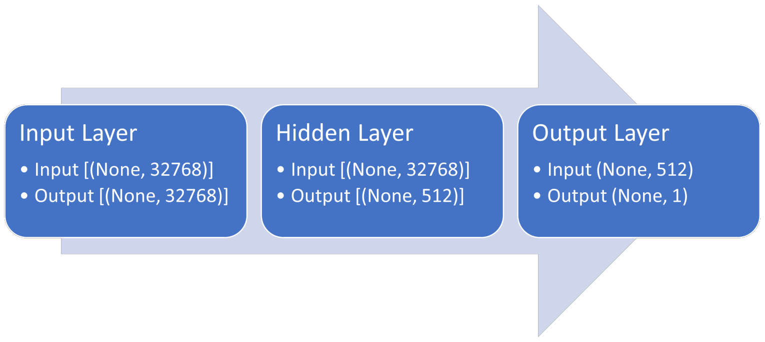


Figure 4. MLP Sequential Model.

For the input we randomly split the training and testing sets into 80 and 20%, respectively. We ensured that the training and testing data were balanced (i.e., the number of samples of "leak" and "clean" data were equal) using `stratified` distribution [78]. To avoid overfitting, `L1` regularization with $\lambda = 0.001$ and dropout of 0.3 was applied. `Adamax` optimizer with a `learning rate` of 0.1 was used. The `sigmoid` (σ) function was used in the output layer to classify the data. We trained MLP for 500 `epochs` and determine the validation accuracy to illustrate the accuracy of the MLP model.

5. Evaluation

We evaluated our approach by performing different experiments and our results are presented in this section. We aimed to answer each of the aforementioned research questions.

5.1. RQ1: Are `WAKE_LOCK` Permissions Prevalent in Android Apps?

To answer this question, we gathered popular apps from Google Play Store (https://play.google.com/store, accessed on 15 July 2021). We can obtain a list of the popular apps from other app stores such as APKPure (https://apkpure.com/, accessed on 15 July 2021), Androzoo [79], F-Droid (https://www.f-droid.org/, accessed on 15 July 2021), APKMirror (https://www.apkmirror.com/, accessed on 15 July 2021); however, Google Play Store is the official Android app store provided by Google and is the most used store to download apps. We first browsed through 800 apps from Google Play Store, which topped the charts in different categories, i.e., "Top free app," "Top gaming app," "Top grossing app," and "Top grossing game." To identify the top-rated apps, we used Selenium (https://www.selenium.dev/, accessed on 15 July 2021) to browse the Google Play Store website and extract the list of apps. Selenium is a tool that performs repetitive web tasks automatically, and we can control the state of the browser. We visited each page and collected the information regarding the app from this tool and parsed the available app-related information. We filtered the repeated apps because they were present in both categories (top free and top grossing apps). After removing duplicates, 731 apps remained, which were then downloaded from the "APKPure" repository using Selenium. As 63 apps were not present or had broken links in the repository, we retained only 638 apps.

We extracted the metadata using Androguard from the downloaded APK file, which is located in the AndroidManifest.xml file. This file contains basic information about the app [80]. Figure 5 illustrates a plot with the number of permission counts on the x-axis and permission names used in the apps on the y-axis. We removed the permissions that were used by less than 200 apps to obtain the most used permissions. As shown in Figure 5, the most used permissions are `ACCESS_NETWORK_STATE` and `INTERNET`, which are used in all 638 apps, because all apps must obtain updates from the internet. We can also observe that `WAKE_LOCK` is the second most-used permission in these apps which is used in 618 apps. This illustrates that wake lock is used in almost all apps; therefore, it is important to use wake lock carefully. It also illustrates that it is the common permission used by developers in their app to control the power state of the device. For example, a developer acquires a wake lock to ensure that the CPU does not go to sleep when the app must perform some calculation in the background or when the device goes to sleep. Similarly, when watching a video on the device, the screen must remain turned ON if the video is playing in the foreground.

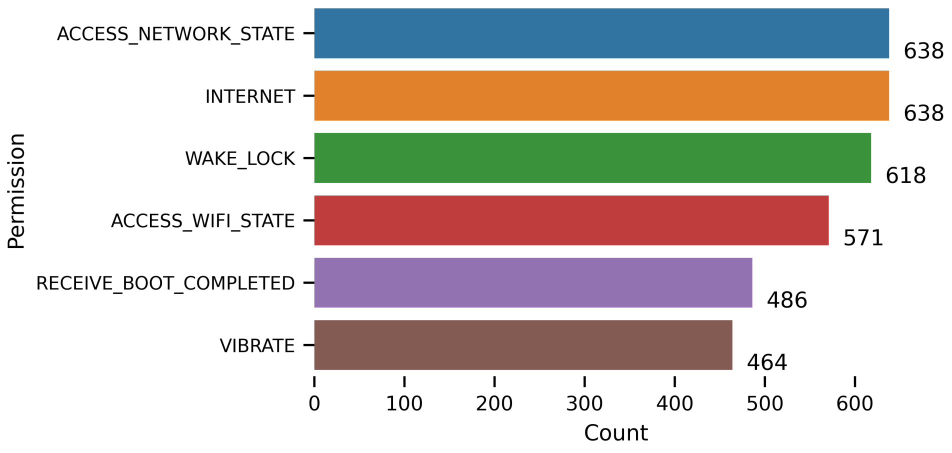


Figure 5. Wake Lock Permissions based on Google Play Store Data.

The result obtained by extracting permission of most popular apps shows that wake locks are important and `WAKE_LOCK` permission is used in almost every app nowadays. We see from our wake lock leak example in Section 2.2, that if wake locks APIs are not carefully used, they can lead to energy leakage in the app because acquiring and releasing wake locks occur in different methods and developers must take care when using these APIs. This also shows that the use of wake lock usage is increasing in the app and these apps want to control the state of components (CPU, display, and keyboard) to get better results of app usage from user points of view.

- **Answer for RQ1: Based on Figure 5, we can answer that wake locks are important, and 98% of the apps use WAKE_LOCK permission. Therefore, the careful use of the wake lock is important.**

5.2. RQ2: Can MLP Be Used to Detect Wake Lock Leaks?

To train the MLP, we require a large number of apps that can be used for training and testing. We must download apps with wake lock problems and train the MLP to detect wake lock leaks. Collecting apps with wake lock leaks is difficult because not many apps with wake lock leaks have been identified when compared to malware [81]. The list of apps with wake lock leaks and those that are clean is shown in Table 2 and Figure 2 of Section 4.1. We extracted features from the apps and encoded them, as stated in Section 4.2, to feed them to the MLP for detection.

The wake lock leak data we used in our experiment is not very large and we face data imbalanced problem in our dataset. One way to generate more data is to simply make copies of the original data, but it will not add any useful information. Another way is to use oversampling techniques such as ADASYN [82], and SMOTE. Results of the study [83] illustrate that both SMOTE and ADASYN can improves the performance and that there does not seem to be a pattern that speaks in favor of either SMOTE or ADASYN being consistently better than the other. There is also under-sampling [84] techniques that are used when we have more than enough labeled data and removing samples from negative examples will not hurt the data. In our case we have fewer examples of positive data (wake lock leak); therefore, we cannot use the under-sampling technique.

We applied SMOTE to oversample the minority classes and generate minority classes to balance the data. We had 2946 clean apps and 32 apps with leaks. After applying SMOTE, we obtained 2946 clean apps and 2946 leak apps. We used the Python imbalance library (https://imbalanced-learn.org/stable/index.html, accessed on 15 July 2021) for the implementation of SMOTE. This balanced data was used to train and test the MLP for classification. To demonstrate the performance of the MLP, we used the accuracy and loss graphs of the training and testing data. The accuracy graph in Figure 6 illustrates that MLP performed well and has an accuracy of 99.32% and a loss of 0.05, as shown in Figure 7.

We can see from Figure 6 that the accuracy is quite high, this is because of oversampling (SMOTE) consider near neighbors of random minority class sample to create new samples, so they have a pattern which is easily captured by the MLP. The main purpose of the study is to use MLP to detect wake lock leaks. If we have more data, we can apply MLP without oversampling which may result in lower accuracy as compared to our results. We added less data as a limitation of our work and highlighted in Section 6. Still, if the accuracy drops, we see that MLP can detect wake lock leak with high accuracy. We optimized the parameters of MLP to avoid overfitting and get high accuracy. We also compare the accuracy of the MLP with different ML algorithms.

- **Answer for RQ2: MLP can be used to detect wake lock leaks with high accuracy.**

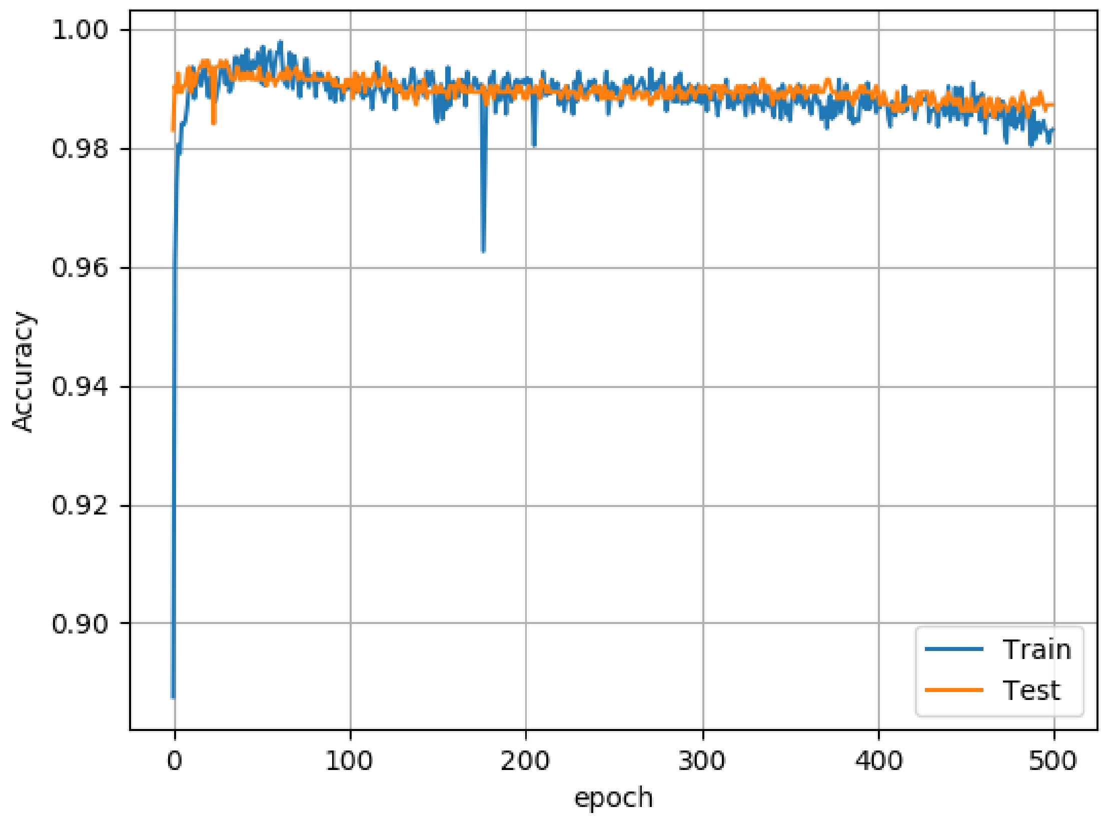


Figure 6. Accuracy of MLP Model.

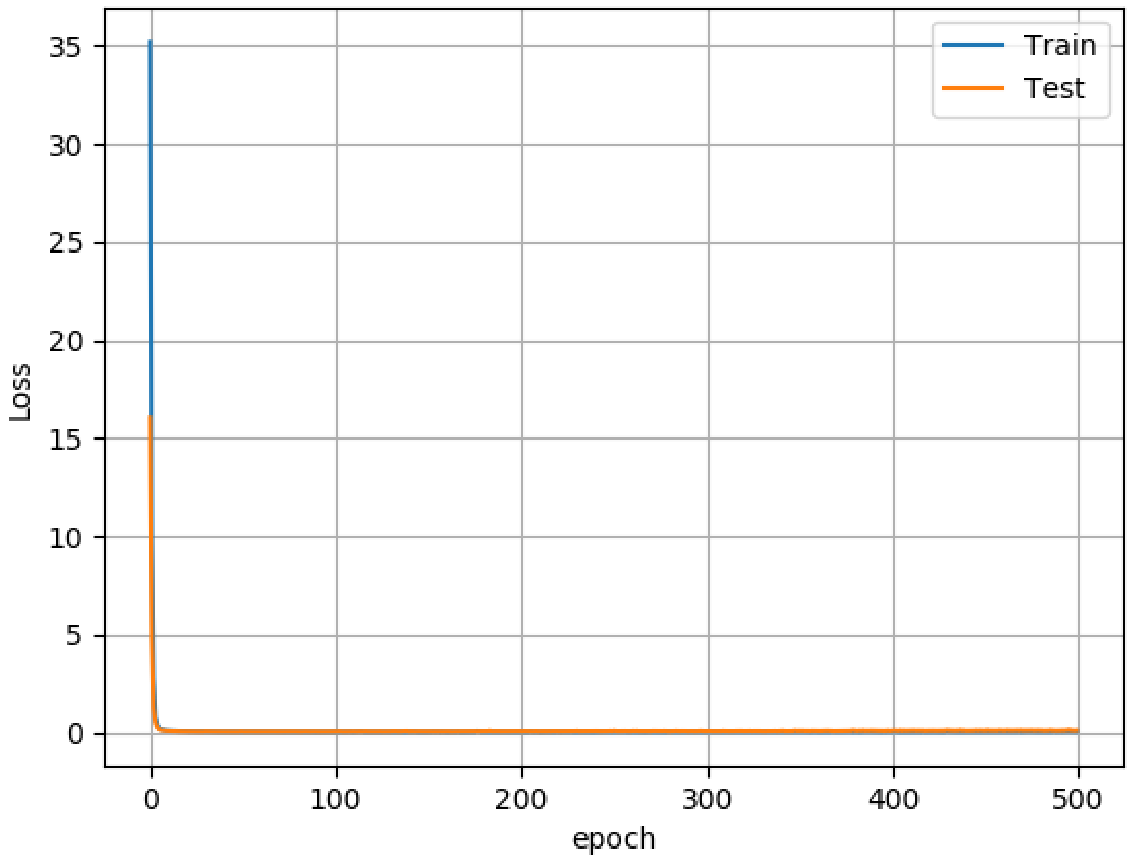


Figure 7. Loss of MLP Model.

5.3. RQ3: Is the Performance of the MLP Model Better Than That of Traditional ML Algorithms?

To answer RQ3, we compared the MLP model with traditional ML algorithms such as naïve-Bayes (NB), support vector machine (SVM), K-Nearest neighbor (KNN), logistic regression (LR), ridge classifier (RC), begged decision tree (BDT), random forest (RF) and stochastic gradient boosting (SGB). We used 80% of the data for training, 20% for testing, and 10-fold cross-validation to avoid overfitting. To remove noise, we performed the experiment 10 times and calculated the average of the accuracy results. The results are listed in Table 6, which show that the MLP model achieves higher accuracy than the ML algorithms.

Table 6. Comparison with machine learning algorithms.

Algorithm	NB	SVM	KNN	LR	RC	BDT	RF	SGB	MLP
Mean	90.16	93.81	91.77	94.23	92.70	94.06	93.89	94.06	**99.32**

NB = naïve Bayes, SVM = support vector machine, KNN = K-nearest neighbor, LR = logistic regression, RC = ridge classifier, BDT = begged decision tree, RF = random forest, SGB = stochastic gradient boosting, MLP = multi-layer perceptron.

The comparison in Table 6 shows that accuracy of NB is lowest as compared to other algorithms, this is because in probability calculation, it considers each feature independently. LR has highest accuracy in ML algorithms because NN representation can be perceived as stacking together a lot of little LR classifiers. This comparison is important because it shows that the ML algorithm can also be used to detect wake lock leaks. We see that these ML algorithms also perform well with high accuracy (more than 90%). MLP takes higher training time as compared to ML algorithms so we can use ML algorithms, when we have less computational resources in exchange of lower accuracy.

- **Answer for RQ3: The accuracy of MLP is significantly high while detecting wake lock leaks in Android apps. This clearly illustrates that the MLP model performs better than traditional ML algorithms.**

5.4. Comparison

We compared our method with a state-of-the-art tool, Elite [56], which is an open-source tool and requires APK files as input to detect the wake lock leaks in the Android apps. We randomly selected 10 apps from our labeled dataset. The Elite tool uses the Dex2Jar to decompile APK to Java bytecode and analyze each app to detect a wake lock leak. Next, we applied our trained MLP to detect wake lock leaks as described in Section 4.2. We just need to provide the path of the folder where APK files are present and run the trained MLP model. A comparison of the tools is listed in Table 7, which shows that MLP can accurately detect wake lock leaks in Android apps. The Elite tool was unable to identify three apps, two of which were detected as false-positive and one as false-negative. We conclude from the results, that our MLP model can detect wake lock leaks accurately.

Table 7. Comparison with Elite tool.

App (Version)	Elite	MLP	Ground Truth
AndTweet (0.2.4)	Leak	Leak	Leak
CallMeter (4E9106C)	**Leak**	Clean	Clean
ConnectBot (540C693)	Clean	Clean	Clean
CSipSimple (1B2D2B6)	**Leak**	Clean	Clean
Cwac-Wakeful (D984B89)	Leak	Leak	Leak
K9Mail (F1232A1)	Clean	Clean	Clean
Open-GPSTracker (763B11E)	Clean	Clean	Clean
OsmAnd (F314EA3)	**Clean**	Leak	Leak
VLC (A9B911D)	Clean	Clean	Clean
VLC (233C863)	Clean	Clean	Clean

6. Limitations

Data collection is the most important part of the research because most of the apps used in the evaluation of other tools were not available online or we did not find the appropriate version of the app. Certain apps were obtained, but they did not have any leaks. To remove this threat, we only used the apps that were obtained from GitHub and manually verified, if the wake lock leak was removed from their updated version. We converted these apps to the APK format because we processed only APK files in our experiment.

Imbalanced data were another problem in our research. The number of identified apps with wake lock leaks was significantly low when compared to the number of clean apps. To remove this threat, we used SMOTE, which is the most popular oversampling technique. There are certain other variations of oversampling methods, but SMOTE has the best performance [83].

To validate and compare with other tool we do not have enough data. Benchmark apps are required to overcome this problem which can be used to validate the tools and find the effectiveness of tool in detecting wake lock leaks.

Another limitation in this work is that we only considered 15 Dalvik bytecode instruction classes during feature extraction, which can lead to inaccurate representation of the method. We can remove this threat by including all the basic instructions; however, the memory and processing requirements will be considerably high and will affect the processing time.

7. Conclusions and Future Work

Reducing power consumption of mobile devices is important. Display and CPU of the device consumes most of the power. Wake lock APIs are used to control the state of the display, CPU, and keyboard which affect the power consumption when the app is running. We see from our RQ1, that more than 98% of the app uses `WAKE_LOCK` permission in their app and control the power state of the device when the app is running. If these APIs (`acquire(), and released()`) are not used properly, they will lead to unwanted power consumption.

To detect wake lock leaks in Android apps, we extracted CG from APK, encoded instructions and neighbor information of the node in their label for more descriptions about the node. The apps were collected from GitHub, an empirical study, and oversampled using SMOTE. This encoded and oversampled data was then input to train the MLP. After training, we tested the MLP and calculated the accuracy and loss. The results illustrate that MLP can detect wake lock leaks with high accuracy of 99%. We also compare the MLP model with other ML algorithms, which demonstrated that MLP outperforms the other ML algorithms in detecting wake lock leaks.

For our future work, we plan to include other resource leaks in the study and create a larger dataset that represents all resource leaks in Android app; then, we plan to evaluate the effectiveness of the MLP in detecting all resource leaks.

Author Contributions: Conceptualization, M.U.K.; methodology, M.U.K.; software, M.U.K.; validation, Z.W. and S.U.-J.L.; formal analysis, M.U.K.; investigation, M.U.K.; resources, S.U.-J.L.; data curation, M.U.K.; writing—original draft preparation, M.U.K.; writing—review and editing, Z.W., S.A.; visualization, M.U.K.; supervision, S.U.-J.L.; project administration, S.U.-J.L.; funding acquisition, S.U.-J.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the research fund of Hanyang University (HY-2021-1959).

Data Availability Statement: The data presented in this study are openly available in GitHub at https://github.com/umkhanqta/MLPWake, accessed on 15 July 2021.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

NN	Neural Network
MLP	Multi-Layer Perceptron
ML	Machine Learning
CG	Call Graph
CFG	Control Flow Graph
FCG	Function Call Graph
DFG	Data Flow Graph
OS	Operating System
IL	Intermediate Language
Apps	Applications
SMOTE	Synthetic Minority Oversampling Techniques
FSM	Finite State Machine
JPF	Java Path Finder
ADB	Android Debug Bridge
APK	Android Package Kit
NHGK	Neighborhood Hash Graph Kernel
ADASYN	ADAptive SYNthetic sampling
NB	Naïve Bayes
SVM	Support Vector Machine
KNN	K-Near Neighbor
LR	Logistic Regression
RC	Ridge Classifier
BDT	Begged Decision Tree
RF	Random Forest

References

1. Cheng, H.; Shapter, J.G.; Li, Y.; Gao, G. Recent progress of advanced anode materials of lithium-ion batteries. *J. Energy Chem.* **2020**, *57*, 451–468. [CrossRef]
2. Dash, P.; Hu, Y.C. How much battery does dark mode save? An accurate OLED display power profiler for modern smartphones. In Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services, online, 24 June–2 July 2021; pp. 323–335.
3. How to Set Up Dark Mode on Your Favorite Apps PCMag, (n.d.). Available online: https://www.pcmag.com/how-to/how-to-set-up-dark-mode-on-your-favorite-apps (accessed on 15 July 2021).
4. Smartphone Users 2020, Statista, (n.d.). Available online: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/ (accessed on 15 July 2021).
5. Mobile App Statistics To Know in 2021. Available online: https://mindsea.com/app-stats/ (accessed on 15 July 2021).
6. Top Google Play Store Statistics. Available online: https://appinventiv.com/blog/google-play-store-statistics/ (accessed on 15 July 2021).
7. App Store Ranking. Available online: https://www.preapps.com/blog/app-store-ranking-vs-google-play-store-ranking-545/ (accessed on 15 July 2021).
8. Mun, H.; Lee, Y. Appspeedxray: A mobile application performance measurement tool. In Proceedings of the 35th Annual ACM Symposium on Applied Computing, Brno, Czech Republic, 30 March–3 April 2020; pp. 1010–1012.
9. Khan, M.U.; Abbas, S.; Lee, S.U.J.; Abbas, A. Energy-leaks in android application development: Perspective and challenges. *J. Theor. Appl. Inf. Technol.* **2020**, *98*, 3591–3601.
10. Liu, Y.; Wang, J.; Wei, L.; Xu, C.; Cheung, S.C.; Wu, T.; Yan, J.; Zhang, J. DroidLeaks: A comprehensive database of resource leaks in Android apps. *Empir. Softw. Eng.* **2019**, *24*, 3435–3483. [CrossRef]
11. PowerManager.WakeLock. Available online: https://developer.android.com/reference/android/os/PowerManager.WakeLock (accessed on 15 July 2021).
12. Mittal, R.; Kansal, A.; Chandra, R. Empowering developers to estimate app energy consumption. In Proceedings of the 18th Annual International Conference on Mobile Computing and Networking, Istanbul, Turkey, 22–26 August 2012; pp. 317–328.
13. Khan, M.U.; Abbas, S.; Lee, S.U.J.; Abbas, A. Measuring Power Consumption in Mobile Devices for Energy Sustainable App Development: A Comparative Study and Challenges. *Sustain. Comput. Inform. Syst.* **2021**, *31*, 100589.
14. Payet, É.; Spoto, F. Static analysis of Android programs. *Inf. Softw. Technol.* **2012**, *54*, 1192–1201. [CrossRef]
15. Ball, T. *The Concept of Dynamic Analysis*; Software Engineering—ESEC/FSE'99; Springer: Berlin/Heidelberg, Germany, 1999; pp. 216–234.

16. Roundy, K.A.; Miller, B.P. Hybrid analysis and control of malware. In *Lecture Notes in Computer Science, Proceedings of the International Workshop on Recent Advances in Intrusion Detection, Ottawa, ON, Canada, 15–17 September 2010*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 317–338.
17. Gascon, H.; Yamaguchi, F.; Arp, D.; Rieck, K. Structural detection of android malware using embedded call graphs. In Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security, Berlin, Germany, 4 November 2013; pp. 45–54.
18. Xu, Z.; Wen, C.; Qin, S. State-taint analysis for detecting resource bugs. *Sci. Comput. Program.* **2018**, *162*, 93–109. [CrossRef]
19. Pathak, A.; Jindal, A.; Hu, Y.C.; Midkiff, S.P. 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, Windermere, UK, 25–29 June 2012; pp. 267–280.
20. Li, L.; Bissyandé, T.F.; Papadakis, M.; Rasthofer, S.; Bartel, A.; Octeau, D.; Klein, J.; Traon, L. Static analysis of android apps: A systematic literature review. *Inf. Softw. Technol.* **2017**, *88*, 67–95. [CrossRef]
21. Lam, P.; Bodden, E.; Lhoták, O.; Hendren, L. The Soot framework for Java program analysis: A retrospective. In Proceedings of the Cetus Users and Compiler Infastructure Workshop (CETUS 2011), Galveston Island, TX, USA, 10 October 2011; Volume 15, p. 35.
22. Bartel, A.; Klein, J.; Le Traon, Y.; Monperrus, M. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis, Beijing, China, 14 June 2012; pp. 27–38.
23. Cruz, L.; Abreu, R. Catalog of energy patterns for mobile applications. *Empir. Softw. Eng.* **2019**, *24*, 2209–2235. [CrossRef]
24. Wang, J.; Liu, Y.; Xu, C.; Ma, X.; Lu, J. E-greenDroid: Effective energy inefficiency analysis for android applications. In Proceedings of the 8th Asia-Pacific Symposium on Internetware, Beijing, China, 18 September 2016; pp. 71–80.
25. Liu, Y.; Wang, J.; Xu, C.; Ma, X. NavyDroid: Detecting energy inefficiency problems for smartphone applications. In Proceedings of the 9th Asia-Pacific Symposium on Internetware, Shanghai, China, 23 September 2017; pp. 1–10.
26. Li, Z.; Sun, J.; Yan, Q.; Srisa-an, W.; Bachala, S. Grandroid: Graph-based detection of malicious network behaviors in android applications. In *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Proceedings of the International Conference on Security and Privacy in Communication Systems, Singapore, 8–10 August 2018*; Springer: Cham, Switzerland, 2018; pp. 264–280.
27. Android Codenames, Tags, and Build Numbers. Available online: https://source.android.com/setup/start/build-numbers (accessed on 15 July 2021).
28. Android Version History. Available online: https://en.wikipedia.org/wiki/Android_version_history (accessed on 15 July 2021).
29. Liu, Y.; Xu, C.; Cheung, S.C.; Lü, J. Greendroid: Automated diagnosis of energy inefficiency for smartphone applications. *IEEE Trans. Softw. Eng.* **2014**, *40*, 911–940.
30. Guo, C.; Zhang, J.; Yan, J.; Zhang, Z.; Zhang, Y. Characterizing and detecting resource leaks in Android applications. In Proceedings of the 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), Silicon Valley, CA, USA, 11–15 November 2013; pp. 389–398.
31. Wu, T.; Liu, J.; Deng, X.; Yan, J.; Zhang, J. Relda2: An effective static analysis tool for resource leak detection in Android apps. In Proceedings of the 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore, 3–7 September 2016; pp. 762–767.
32. Hu, H.; Liu, Z.; An, J. Mining mobile intelligence for wireless systems: A deep neural network approach. *IEEE Comput. Intell. Mag.* **2020**, *15*, 24–31. [CrossRef]
33. Qiu, J.; Zhang, J.; Luo, W.; Pan, L.; Nepal, S.; Xiang, Y. A survey of Android malware detection with deep neural models. *ACM Comput. Surv. (CSUR)* **2020**, *53*, 1–36. [CrossRef]
34. Alamoudi, A.; Liu, M.; Payani, A.; Fekri, F.; Li, D. Predicting Mobile Users Traffic and Access-Time Behavior Using Recurrent Neural Networks. In Proceedings of the 2021 IEEE Wireless Communications and Networking Conference (WCNC), Nanjing, China, 29 March–1 April 2021; pp. 1–6.
35. Wu, Z.; Chen, X.; Lee, S.U.J. FCDP: Fidelity Calculation for Description-to-Permissions in Android Apps. *IEEE Access* **2020**, *9*, 1062–1075. [CrossRef]
36. The Java-se Tutorial. Available online: https://docs.oracle.com/javase/tutorial/index.html (accessed on 15 July 2021).
37. Develop Android Apps with Kotlin. Available online: https://developer.android.com/kotlin (accessed on 15 July 2021).
38. Kotlin on Android. Now official. Available online: https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/ (accessed on 15 July 2021).
39. Kotlin Is Now Googles Prefered Language for Android App Development. Available online: https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/ (accessed on 15 July 2021).
40. Get Started with the NDK. Available online: https://developer.android.com/ndk/guides (accessed on 15 July 2021).
41. Android Package File. Available online: https://fileinfo.com/extension/apk (accessed on 15 July 2021).
42. PowerManager. Available online: https://developer.android.com/reference/android/os/PowerManager (accessed on 15 July 2021).
43. Improve Your Code with Lint Checks. Available online: https://developer.android.com/studio/write/lint (accessed on 15 July 2021).
44. Vlc-Android. Available online: https://github.com/mstorsjo/vlc-android/commit/233c8639b20f3eaf3ae23d5d40ff7c1d559b8796 (accessed on 15 July 2021).

45. White, H. *Artificial Neural Networks*; Blackwell Cambridge: Cambridge, MA, USA, 1992.
46. Soltanolkotabi, M.; Javanmard, A.; Lee, J.D. Theoretical insights into the optimization landscape of over-parameterized shallow neural networks. *IEEE Trans. Inf. Theory* **2018**, *65*, 742–769. [CrossRef]
47. Gulli, A.; Pal, S. *Deep Learning with Keras*; Packt Publishing Ltd.: Birmingham, UK, 2017.
48. Mansourifar, H.; Shi, W. Deep Synthetic Minority Over-Sampling Technique. *arXiv* **2020**, arXiv:2003.09788.
49. Elreedy, D.; Atiya, A.F. A comprehensive analysis of synthetic minority oversampling technique (SMOTE) for handling class imbalance. *Inf. Sci.* **2019**, *505*, 32–64. [CrossRef]
50. Hecht, G.; Moha, N.; Rouvoy, R. An empirical study of the performance impacts of android code smells. In Proceedings of the International Conference on Mobile Software Engineering and Systems, Austin, TX, USA, 14–22 May 2016; pp. 59–69.
51. Android Lint Checks. Available online: http://tools.android.com/tips/lint-checks (accessed on 15 July 2021).
52. Cruz, L.; Abreu, R. Performance-based guidelines for energy efficient mobile applications. In Proceedings of the 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), Buenos Aires, Argentina, 22–23 May 2017; pp. 46–57.
53. Palomba, F.; Di Nucci, D.; Panichella, A.; Zaidman, A.; De Lucia, A. Lightweight detection of android-specific code smells: The adoctor project. In Proceedings of the 2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER), Klagenfurt, Austria, 20–24 February 2017; pp. 487–491.
54. Reimann, J.; Brylski, M.; Aßmann, U. A tool-supported quality smell catalogue for android developers. In Proceedings of the Conference Modellierung 2014 in the Workshop Modellbasierte und Modellgetriebene Softwaremodernisierung–MMSM, 2014; Volume 2014. Available online: http://akmda.ipd.kit.edu/mmsm/mmsm_2014/ (accessed on 15 July 2021).
55. Palomba, F.; Di Nucci, D.; Panichella, A.; Zaidman, A.; De Lucia, A. On the impact of code smells on the energy consumption of mobile applications. *Inf. Softw. Technol.* **2019**, *105*, 43–55. [CrossRef]
56. Liu, Y.; Xu, C.; Cheung, S.C.; Terragni, V. Understanding and detecting wake lock misuses for android applications. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, WA, USA, 13–18 November 2016; pp. 396–409.
57. Vekris, P.; Jhala, R.; Lerner, S.; Agarwal, Y. Towards verifying android apps for the absence of no-sleep energy bugs. In Proceedings of the 2012 Workshop on Power-Aware Computing and Systems (HotPower 12), Hollywood, CA, USA, 7 October 2012.
58. Cai, M.; Jiang, Y.; Gao, C.; Li, H.; Yuan, W. Learning features from enhanced function call graphs for Android malware detection. *Neurocomputing* **2021**, *423*, 301–307. [CrossRef]
59. Pathak, A.; Hu, Y.C.; Zhang, M. Where is the energy spent inside my app? Fine Grained Energy Accounting on Smartphones with Eprof. In Proceedings of the 7th ACM European Conference on Computer Systems, Bern, Switzerland, 10–13 April 2012; pp. 29–42.
60. Abbasi, A.M.; Al-Tekreeti, M.; Naik, K.; Nayak, A.; Srivastava, P.; Zaman, M. Characterization and Detection of Tail Energy Bugs in Smartphones. *IEEE Access* **2018**, *6*, 65098–65108. [CrossRef]
61. Kurihara, S.; Fukuda, S.; Kamiyama, T.; Fukuda, A.; Oguchi, M.; Yamaguchi, S. Estimation of power consumption of each application considering software dependency in Android. *J. Inf. Process.* **2019**, *27*, 221–232. [CrossRef]
62. Android Debug Bridge (ADB). Available online: https://developer.android.com/studio/command-line/adb.html (accessed on 15 July 2021).
63. Gharehyazie, M.; Ray, B.; Filkov, V. Some from here, some from there: Cross-project code reuse in github. In Proceedings of the 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), Buenos Aires, Argentina, 20–21 May 2017; pp. 291–301.
64. GitHub Issues. Available online: https://open-learning-exchange.github.io/#!pages/vi/vi-github-issues.md (accessed on 15 July 2021).
65. Wu, H.; Wang, Y.; Rountev, A. Sentinel: Generating GUI tests for Android sensor leaks. In Proceedings of the 2018 IEEE/ACM 13th International Workshop on Automation of Software Test (AST), Gothenburg, Sweden, 28–29 May 2018; pp. 27–33.
66. Hindle, A. Green mining: A methodology of relating software change and configuration to power consumption. *Empir. Softw. Eng.* **2015**, *20*, 374–409. [CrossRef]
67. Chowdhury, S.A.; Hindle, A. Greenoracle: Estimating software energy consumption with energy measurement corpora. In Proceedings of the 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), Austin, TX, USA, 14–15 May 2016; pp. 49–60.
68. Banerjee, A.; Chong, L.K.; Ballabriga, C.; Roychoudhury, A. Energypatch: Repairing resource leaks to improve energy-efficiency of android apps. *IEEE Trans. Softw. Eng.* **2017**, *44*, 470–490. [CrossRef]
69. Dalvik Bytecode. Available online: https://source.android.com/devices/tech/dalvik/dalvik-bytecode (accessed on 15 July 2021).
70. Müller, H.A.; Jahnke, J.H.; Smith, D.B.; Storey, M.A.; Tilley, S.R.; Wong, K. Reverse engineering: A roadmap. In Proceedings of the Conference on the Future of Software Engineering, Limerick, Ireland, 4–11 June 2000; pp. 47–60.
71. Arnatovich, Y.L.; Wang, L.; Ngo, N.M.; Soh, C. A comparison of android reverse engineering tools via program behaviors validation based on intermediate languages transformation. *IEEE Access* **2018**, *6*, 12382–12394. [CrossRef]
72. Dalvik Opcodes. Available online: http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html (accessed on 15 July 2021).

73. Dong, F.; Wang, J.; Li, Q.; Xu, G.; Zhang, S. Defect prediction in android binary executables using deep neural network. *Wirel. Pers. Commun.* **2018**, *102*, 2261–2285. [CrossRef]
74. Hido, S.; Kashima, H. A linear-time graph kernel. In Proceedings of the 2009 Ninth IEEE International Conference on Data Mining, Miami Beach, FL, USA, 6–9 December 2009; pp. 179–188.
75. Fine, T.L. *Feedforward Neural Network Methodology*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2006.
76. Hecht-Nielsen, R. Theory of the backpropagation neural network. In *Neural Networks for Perception*; Elsevier: Amsterdam, The Netherlands, 1992; pp. 65–93.
77. Liu, S.; Chen, L.; Dong, H.; Wang, Z.; Wu, D.; Huang, Z. Higher-order weighted graph convolutional networks. *arXiv* **2019**, arXiv:1911.04129.
78. Zeng, X.; Martinez, T.R. Distribution-balanced stratified cross-validation for accuracy estimation. *J. Exp. Theor. Artif. Intell.* **2000**, *12*, 1–12. [CrossRef]
79. Allix, K.; Bissyandé, T.F.; Klein, J.; Le Traon, Y. Androzoo: Collecting millions of android apps for the research community. In Proceedings of the 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), Austin, TX, USA, 14–15 May 2016; pp. 468–471.
80. Manifest Permission. Available online: https://developer.android.com/reference/android/Manifest.permission (accessed on 15 July 2021).
81. Jeon, S.; Moon, J. Malware-detection method with a convolutional recurrent neural network using opcode sequences. *Inf. Sci.* **2020**, *535*, 1–15. [CrossRef]
82. He, H.; Bai, Y.; Garcia, E.A.; Li, S. ADASYN: Adaptive synthetic sampling approach for imbalanced learning. In Proceedings of the 2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence), Hong Kong, China, 1–8 June 2008; pp. 1322–1328.
83. Gosain, A.; Sardana, S. Handling class imbalance problem using oversampling techniques: A review. In Proceedings of the 2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Udupi, India, 13–16 September 2017; pp. 79–85.
84. Liu, X.Y.; Wu, J.; Zhou, Z.H. Exploratory undersampling for class-imbalance learning. *IEEE Trans. Syst. Man Cybern. Part B (Cybern.)* **2008**, *39*, 539–550.