IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 5, 2021, accepted August 31, 2021, date of publication September 3, 2021, date of current version September 17, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3110244

Detecting Wake Lock Leaks in Android Apps Using Machine Learning

MUHAMMAD UMAIR KHAN[1], SCOTT UK-JIN LEE[1], (Member, IEEE), SHANZA ABBAS[1], ASAD ABBAS[2], AND ALI KASHIF BASHIR[3], (Senior Member, IEEE)

[1]Department of Computer Science and Engineering, Hanyang University, Seoul 15588, Republic of Korea
[2]Faculty of Information Technology, University of Central Punjab, Lahore 54000, Pakistan
[3]Department of Computing and Mathematics, Manchester Metropolitan University, Manchester M2 1WL, U.K.

Corresponding author: Scott Uk-Jin Lee (scottlee@hanyang.ac.kr)

This work was supported by the research fund of Hanyang University under Grant HY-2021-1959.

ABSTRACT The popularity of Android devices has increased exponentially with an increase in the number of mobile devices. Millions of online apps are used in these devices. Energy consumption of a device is a major concern for end-users, who want a long usage time on a single battery charge. The energy consumed by the app must be optimized by developers, and the available APIs must be used carefully. A wake-lock is used in apps to control the power state of the Android device and often leads to energy leakage. In this study, we detected wake-lock leaks in Android apps using machine learning. We pre-processed apps by extracting wake-lock related APIs to obtain the structural information of wake-lock usage and oversampled the data using the synthetic minority oversampling technique (SMOTE) to balance the dataset. The machine learning algorithms used to detect wake-lock leaks were first optimized using grid search to determine the best parameters. These parameters were then used in training to detect wake-lock leaks in these apps. We employed various machine learning algorithms and divided them into simple and ensemble algorithms to evaluate their efficacy. The support vector machine (SVM) and stochastic gradient boosting (SGB) were the most effective, producing 97 % and 98 % accuracy, respectively.

INDEX TERMS Android apps, call graphs, wake lock, support vector machine, over sampling.

I. INTRODUCTION

The number of smartphone users worldwide is increasing; they surpass three billion and are expected to increase more [1]. The primary function of a mobile device is to connect users to the rest of the world and provide basic functionalities to aid them in performing small tasks. These mobile devices have limited resources such as CPU, memory, battery, and display. Batteries in these devices are important because they provide power to all the components of the device; they require a sufficient amount of charge to run the device for an entire day. Research on rendering hardware components energy efficient and using high-capacity batteries is popular [2], but applications (apps) running on the device should also be energy efficient. The field of mobile app development is on the rise, and on average, 100,000 new Android apps are released in Google Play Store every month [3]. In this fast-growing market, developers of these apps focus on functional requirements and neglect non-functional requirements, such as batteries. To improve battery life, developers must write efficient codes that release all resources when they are not in use.

Research trends in mobile devices have increased dramatically over the past five years [4]. Detecting and removing energy leakage in mobile devices is one of the most widely published topics [4]. Notably, the Android operating system (OS), iOS, Windows, and most popular apps such as WhatsApp, Facebook, YouTube, and Chrome provide a dark mode to reduce the energy consumption of the device [5]. This shows that big companies are also aware of the importance of energy consumption in mobile devices. The developers of mobile apps must also take extra care when implementing apps, and monitor the energy used by the apps. If the energy used by an app is increasing or the app uses resources after being closed, then the app has some energy leak that requires some intervention to be resolved. Removing all resource leaks will lead to an efficient mobile app.

The associate editor coordinating the review of this manuscript and approving it for publication was Dominik Strzalka.

Mobile devices have different apps that can play video and music for which they need the screen of the device to stay working, or some apps must be constantly processed while operating in the background. For example, a music player must play music when the app is in the background, and the user is working on a different app or when the screen is locked. These types of apps must obtain special permission from the Android OS called `WAKE_LOCK`. Whenever an app must work in the background or a video player requires the device screen to stay awake to prevent it from going to sleep mode, it must acquire a wake-lock. This wake-lock acquire, and the release mechanism must be carefully used because this is a common reason for energy leakage in Android devices [6]. Among the different tools available to detect wake-lock leaks, the majority use static analysis, while others use dynamic analysis. Static analysis tools consider control flow graphs [7], function call graphs (FCG) [8], and data flow graphs [9] to detect wake-lock leaks. Dynamic analysis tools use an Android emulator to run the app and then process their log files to determine wake-lock leaks [10], [11].

These analysis techniques are useful for developers to detect flaws and bugs in the software, but they often have performance and precision issues. Static analysis can detect specifically defined patterns in the code to detect leaks. Any new pattern was missed. They also suffer from a high false-positive rate. Similarly, dynamic analysis detects bugs at runtime and requires a significant amount of computing resources to monitor the app when it is running on the emulator [12]. Notably, machine learning techniques have strongly established themselves in a number of research fields where they are used to effectively detect patterns in the provided data/features. Researchers continue to use machine learning techniques everywhere to determine the correlation in the data and extract useful information. The defects can be grouped together and calculated as a similarity measure. They can also be used to rank defects. Machine learning techniques seem to be useful for identifying patterns in the given data that influence the output. Notably, they are currently popular and are used to detect malware [13], code smells, [14], and permission violations [15], [16] in Android apps. Machine learning techniques are suitable for detecting wake-lock leaks because they exhibit patterns that can be characterized by machine learning. This motivates us to use machine learning techniques to detect wake-lock leaks in Android apps and analyze their performance because the wake-lock acquires, and the release mechanism must be monitored, and we must identify the flow of the program that leads to wake-lock leak.

In this study, wake-lock leaks were detected using machine learning algorithms. To apply machine learning, the apps were pre-processed by extracting the FCGs. These FCGs represent the paths where the wake-lock API uses `acquire()`, `release()`, and `isheld()`. These graphs are then encoded, containing information on the instructions and their neighbor nodes. After pre-processing, they were fed into a machine-learning algorithm for training. When the training was complete, the trained algorithm was tested, and its efficacy was compared through metrics such as accuracy, precision, recall, and f-measure. The results demonstrated that machine learning algorithms were effective in detecting wake-lock leaks with high accuracy. Different algorithms were compared to demonstrate the best-performing algorithm.

This study was divided into different sections. Section II summarizes the work related to wake-lock detection. In Section III we present the motivation example of a wake-lock leak occurrence in-app. The basic background of SMOTE and machine-learning algorithms is described in Section IV. Section V describes the methodology used to extract the wake-lock-related APIs from apps. The evaluation of the machine-learning algorithms is presented in Section VI. Section VII discusses the problems faced and how they can be resolved. We discuss threats to the validity in Section VIII. Finally, Section IX summarizes the work and presents scope for future work.

II. RELATED WORK

There are several related works on wake-lock energy leaks in mobile apps. A fully automated tool called aDoctor [17] was proposed to detect 15 code smells, including a durable wake-lock, based on the findings of Reimann *et al.* [18]. Palomba *et al.* [19] identified nine code smells that affect the energy consumption of mobile apps. Cruz and Abreu [20] provided 22 design patterns related to the energy efficiency of mobile apps. In these studies, wake-lock is a common pattern in the detection and removal of energy leaks. These tools are helpful during code development to write energy-efficient codes. They require source code to detect code smells, thus, covering a smaller number of apps. In our study, we use Application Package Kit (APK) files because most popular apps only provide APK files, and the number of apps available in APK files is much larger than apps with source code.

Liu *et al*. [6] is the first detailed study that identified eight patterns of wake-lock misuses that causes functional and non-functional issues. They developed a static analysis tool called ''ELITE'' to detect two common patterns in their study. This study is helpful in understanding the wake-lock leak pattern, but only six apps are used in the evaluation, which is very few. To solve this problem, DroidLeak [21] provided a comprehensive publicly available database on resource leaks in Android apps. They searched the repository comment to determine these resource leaks and energy-related words to obtain code revisions that raise the issue and solve the issue. Such resource leaks are not only restricted to code repositories; therefore, we must focus on APK files to cover a large number of available apps. The most common techniques used to detect wake-lock leaks in APK files are static and dynamic analyses.

A. STATIC ANALYSIS

Static analysis allows app analysis without actually running them. We can use source code, bytecode, binary code,

or intermediate representation (IR) to analyze the flow of the app. Li *et al*. [4] provided details of state-of-the-art works that used static analysis in the Android apps. They concluded that the soot framework [22] and Jimple IR [23] were adopted by most tools. Banerjee and Roychoudhury [24] generated a set of defect expressions to determine the violation of energy defects. Relda [25] and Relda2 [26] are lightweight static analysis tools developed to locate resource leaks that include wake-locks. They are based on FCGs to handle the callbacks of the Android framework. Relda2 supports two analysis techniques (for quick scanning, they use flow-insensitive and for accurate scanning they use flow-sensitive) with better control on the user interface. They suffer from overhead and false-positive problems. Xu *et al*. [7] used state-taint analysis to detect resource bugs and search for resources that are open by developers but not used in the program. They accepted control flow graphs as inputs to track resource behaviors and compared their results with those of Relda [25], [26] and GreenDroid [27] to show that their approach detects more energy leak patterns. Pathak *et al*. [9] used dataflow analysis to detect a no-sleep path that includes wake-lock, GPS, camera, and Wi-Fi. Data flow analysis is based on events, and the entry point of each event is used as the starting point of the analysis.

B. DYNAMIC ANALYSIS

In dynamic analysis, flow is extracted by running the apps and performing different operations at runtime. Dynamic analysis may miss some portions of app functionality or events, as it is not possible to run all the sequences of events. These sequences are intertwined with each other, which implies that to run such activities/services, a certain sequence of events must be followed. Dynamic analysis also requires a significant amount of computing resources to run the apps [12], [28]. Pathak *et al*. [29] used a power model to monitor the state of energy consumed by different system calls using a finite-state machine (FSM). However, their approach required modifying the app to perform automatic profiling. Abbasi *et al*. [30] measured the energy consumption of different types of wake-lock types (listed in Table.1) and their effect when a user performs different actions such as pushing the home button, back button, power button, and swipe. They developed their own apps to test these events without considering the real apps that existed. GreenDroid [27] and E-GreenDroid [10] used state exploration with Java PathFinder (JPF) to detect the wake-lock deactivation function. Liu *et al*. [11] proposed the NavyDroid tool, which monitors multiple patterns of wake-lock leaks. NavyDroid extends the functionality of E-GreenDroid [10] and includes a back event at the end of each event to ensure that the app terminates after consuming all events. In contrast, E-GreenDroid simulates the paused and killed states. Kurihara *et al*. [31] monitored the usage of GPS and wake-lock usage to forecast energy consumption. They built their own benchmark apps to measure energy consumption. These tools are helpful when testing the app because they display the exact behavior of the app on the device. To process a large number of apps, we must have multiple test cases for each app to cover the functionalities. The manual generation of these tests is time-consuming. Random test-generating tools can be used, but log files must be analyzed to extract useful information.

TABLE 1. Wake lock types.

WAKE_LOCK TYPE	CPU	Screen	Keyboard	Power Consumption
PARTIAL_ WAKE_LOCK	Yes	No	No	Very Low
SCREEN_DIM_ WAKE_LOCK	Yes	Dim	No	Low
SCREEN_BRIGHT_ WAKE_LOCK	Yes	Bright	No	High
FULL_ WAKE_LOCK	Yes	Bright	Yes	Very High

C. HYBRID ANALYSIS

Certain tools use a combination of the aforementioned techniques, which is known as hybrid analysis. EcoDroid [32] used both static and dynamic analyses to rank apps based on their energy consumption. Jabbarvand *et al*. [33] proposed an energy-aware test suite minimization approach. They identified energy greedy methods by statically analyzing the app and calculating the energy cost by executing the tests, which indicated the amount of energy used by the segment of the test. This technique is helpful in analyzing the apps because it produces in-depth knowledge acquired from static and dynamic analysis, but the analysis and processing of the logged information takes more time and increases with the number of apps processed.

None of the aforementioned tools and techniques use machine learning to detect wake-lock leaks. However, Zhu *et al*. [34] were the only researchers to use machine learning to detect energy bugs. They used machine learning to train and predict the energy consumption of the changes made in the revision of code commits. In this study, we focused on detecting wake lock energy leaks by first extracting wake lock information and then applying a machine learning algorithm to find wake lock leaks and evaluate their accuracy.

III. MOTIVATION

First, we briefly introduce wake locks and the way they are used, which can cause energy leakage. They are used by developers to control the power state of an Android device. The developer must declare `android.permission.WAKE_LOCK` permission in the `AndroidManifest.xml` file of the app [35], create a `PowerManager.WakeLock` instance in the source code, and specify its type as listed in Table.1. The energy consumption of each type of wake lock is different as presented in Table.1. The full wake-lock type causes CPU running and screen and keyboard backlight at full brightness, which consumes more energy than all other types. After creating

```
1  public class MusicActivity extends
       Activity implements...{
2  public void onCreate() {//start
       PlaybackService...
3  }
4  public void onTouch() {
5  PlaybackService.getInstance().playPause();...
6  }
7  public void onDestroy() {//stop
       PlaybackService...
8  }}
9  public class PlaybackService extends
       Service{
10 public void onCreate() {...
11  wakeLock.acquire(); //partial wake lock
12  mediaPlayer.start();...
13 }
14 public void playPause() {
15 if(mediaPlayer.isPlaying()) {
16 mediaPlayer.pause();
17  if(wakeLock.isHeld())wakeLock.release();
18 }
19 else {
20  if(!wakeLock.isHeld())wakeLock.acquire();
21 mediaPlayer.start(); }
22 }
23 public void onDestroy() {
24 if (wakeLock.isHeld()) wakeLock.release();
25 if (mediaPlayer.isPlaying())
       mediaPlayer.stop(); ...}}
```

LISTING 1. Sample code of unnecessary wakeup in Tomahawk app.

instances of wake-lock, certain APIs can be invoked to acquire and release a wake-lock. Once the wake lock is acquired, it will have long-lasting effects until it is released, or the specified timeout expires. The developer can also set certain flags when acquiring the wake locks. For example, the `ON_AFTER_RELEASE` flag can be set to cause the device screen to remain ON for a defined time after the wake-lock is released [36]. To avoid undesirable consequences, developers must carefully use wake-lock APIs in their code, as they directly affect the device hardware state.

The mechanism for acquiring and releasing the wake-lock is difficult to determine, as it may be located using different methods. For example, a wake lock is acquired in one method and released in another method (depending on some event). If the app is sent to the background without calling the release method, the app is left in a high-power state, causing an energy leak. Similarly, acquiring a wake lock too early or releasing it too late can cause unnecessary awake time. To understand this problem, we present a real issue in the Tomahawk app [6], which is a music player.

Listing. 1 illustrates the sample code of the app in which a wake-lock leak occurs. When the app is started and a user selects an album, Tomahawk's `MusicActivity` will start the `PlaybackService` to play music in the background (`Line 2`). The service acquires a partial wake lock when it is launched, sets up the media player, and starts playing music (`Lines 10–13`). Music players have pause or resume methods that can be called by the user by tapping the device screen (`Lines 4–6` and `14–22`). A wake-lock is released when the user closes the app, and `MusicActivity` and `PlaybackService` will be destroyed accordingly (`Lines 7` and `23–25`). The flow of the program looks functionally correct, and music can be played correctly in practice. However, the wake lock is held unnecessarily when the music playing is paused (`Line 14–22`), as the user may not use the player for a long time, and the acquired wake lock causes unnecessary energy waste. This energy waste must be prevented as it drains the battery of the device. The developers of the app later fixed the issue by removing the wake-lock acquired at the start of service (`Line 11–12`, shown in `red color`). They added a condition to check whether the wake-lock was held or not, and depending on the result, they released the lock when music was paused and acquired when the music started (`Lines 17` and `20` shown in `green color`). This is a simple example of a wake-lock leak and motivates us to detect wake lock leaks in Android apps.

IV. BACKGROUND

In this section, we discuss SMOTE, which is used to balance the dataset, as the data available to train machine learning algorithms are less. Then, we provide a short introduction to machine learning.

Before we go in-depth, we must first understand ways to gather apps and identify their contents. The Android app can be downloaded from different app stores such as Play Store [37], F-Droid [38], APKMirror [39], Androzoo [40], and APKPure [41]. These apps are in the Android Package Kit (APK) format, which is a zip or rar compressed package. This compressed APK file contains `AndroidManifest.xml`, `classes.dex` and `resources.arsc` files as well as `res`, `assets`, `META-INF`, and `lib` folders. `AndroidManifest.xml` is a binary file that provides various feature information needed by the device to run the application, such as permission features and component features [35]. The `classes.dex` are application codes compiled in dex format. A `dex` file is known as Dalvik bytecode [42], which is logically similar to the class file in Java. The `resources.arsc` are binary XML files that contain pre-compiled application resources. The `res` folder contains resources that are not compiled into `resources.arsc` file. The `assets` folder contains the original resources of the application, and AssetManager provides access to these asset files.[1] The `META-INF` folder contains a MANIFEST.MF file that stores metadata about the contents of JAR and APK signatures.[2] The `lib` folder contains the code compiled from the local library.[3] Most of the apps were written in Java programming language and compiled to Dalvik bytecode, but from 2019, Kotlin[4] is the official language declared by

[1] https://developer.android.com/guide/topics/resources/providing-resources
[2] https://developer.android.com/guide/topics/manifest/manifest-intro
[3] https://developer.android.com/studio/projects/configure-cmake
[4] https://developer.android.com/kotlin

Google for Android app development. C++ can also be used for app development using the Android Native Development Kit (NDK),[5] which is useful for managing native activities and accessing physical device components such as sensors and touch input. These languages can be used in Android Studio to write the app code. Android Studio [43] is the official IDE for Android development, which includes everything that is required to build Android apps.

A. SYNTHETIC MINORITY OVERSAMPLING TECHNIQUE (SMOTE)

SMOTE is the most widely used approach to synthesize new examples [44]. Imbalanced classification is used when there are too few examples of minority classes, and machine learning algorithms must be trained on them. The following steps were used to create the new examples.

1) Select a random example from the minority class.
2) Find "k" nearest neighbors from the selected point (default $k = 5$).
3) Select one neighbor randomly from the selected neighbors.
4) Generate a new example using a convex combination of examples selected from *Step-1* and *Step-3*.

The general downside of the approach is that synthetic examples are created without considering the majority class. This will not affect us because, in our case, we are more interested in minority classes (wake lock leak app) and have enough examples of majority classes.

There are three different types of variations for SMOTE, and we will discuss common types that can be adopted.

- *Oversampling:* Oversampling increases the number of instances in the minority class by selecting a random minority example and replicating it to present a higher number of sample representations of the minority class. We used oversampling in our evaluation because the number of examples of wake-lock leaks is less than that of a clean app. The advantage of using this sampling is that there is no information loss, and it outperforms under-sampling. The disadvantage of this technique is that it increases the likelihood of overfitting as it replicates the minority class. Random oversampling and adaptive synthetic (ADASYN) sampling are oversampling techniques.
- *Undersampling:* Undersampling aims to balance the class distribution by randomly eliminating the majority class examples. This is performed until the majority and minority class instances are balanced. This approach is effective in situations where the minority class has a sufficient number of examples, despite the severe imbalance. The disadvantage of this technique is that it randomly removes the examples from the dataset, which can lead to the loss of valuable information that resides in the removed data. Random under-sampling, Cluster, Tomek links, and under-sampling with ensemble learning are some undersampling techniques. We did not use under-sampling in our evaluation because we had a smaller number of examples in our dataset.
- *Over-Under Sampling:* This is a combination of both techniques. Suppose there are data of 10000 samples with a ratio of 99:1, that is, 9900 samples are classified as "0" and 100 samples are classified as "1". This is clearly an example of imbalanced data. To balance the dataset, the minority class is first oversampled to have 10 % of the number of examples of the majority class (that is, 990). For the majority class, random undersampling is used to reduce the number of examples, but they should double the oversampled minority classes (that is, from 9900 to 1980). The final class distribution is 2:1.
  We used Tomek links to observe the effects of over-undersampling for comparison. Tomek links are pairs of instances that are very close, but of opposite classes (one instance belongs to the minority class, and another instance belongs to the majority class). Removing the instances of the majority class from each pair increases the space between the two classes, thereby facilitating the classification process.

B. MACHINE LEARNING ALGORITHMS

Machine learning provides machines with the ability to learn autonomously based on experience, observations, and analyzing patterns within a given dataset without explicit programming [45]. Machine learning algorithms are applied in every field of science to extract useful information, and new machine learning approaches are being developed continuously. Naïve Bayes (NB) [46], support vector machine (SVM) [47], k-near neighbor (KNN) [48], logistic regression (LR) [49], and ridge classifier (RC) [45] are some of the important models used in common practice.

Some advanced algorithms are called ensemble learning algorithms. They are used to produce improved predictive efficiency compared to that obtained from any of the constituent learning algorithms alone, and an ensemble approach utilizes multiple learning algorithms [50]. These algorithms include bagging [51], random forest (RF) [52], and boosting [53].

V. METHODOLOGY

Detecting wake-lock leaks is difficult, as wake locks are concentrated on a small number of methods and are called in different methods. Wake locks must be acquired and released through the flow of the program and should not cause any energy leakage. As Android apps are event-driven, it is difficult to obtain the flow of the program. We used static analysis to address this problem and extracted the call graphs of the app for processing. The proposed model is shown in Fig. 1, and is divided into three main sections. We used a supervised machine-learning algorithm that required the labeled data. For this purpose, we first labeled the data as "clean" and

[5]https://developer.android.com/ndk

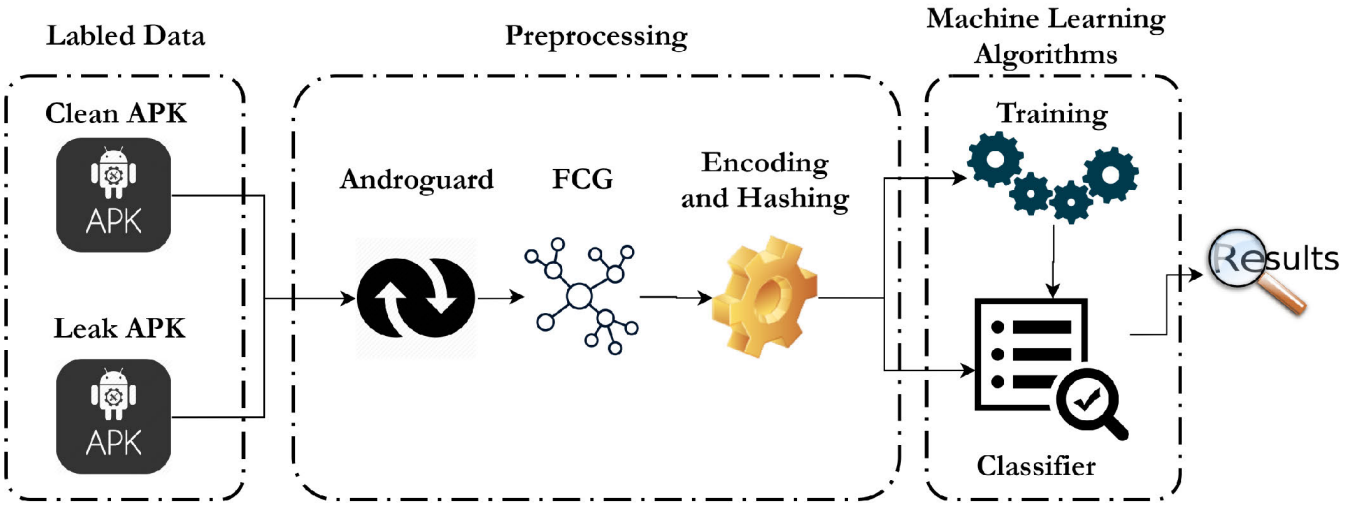


FIGURE 1. Wake lock detection model using machine learning.

"leak" APKs. These APK files were preprocessed in the second section to extract useful information before they were fed into the machine learning algorithms. The preprocessed information was split into training and testing data. Training data were used to train machine learning algorithms to determine similarities in these data, and then a trained machine learning algorithm was used to classify the testing data. The preprocessing is described in detail in the next section.

A. CALL GRAPH EXTRACTION, ENCODING, AND HASHING

In our experiment, we used the Android app in the APK format because they are available on different app stores [37]–[41] where most of them can be freely downloaded. Another reason for using APK files is that the source code of popular apps is typically not accessible online, which results in missing a large portion of the apps in our study. The APK files do not include the source code, as discussed in Section IV. Hence, we used a reverse engineering tool to extract useful information from APK. A reverse engineering tool cannot extract the same information encoded in the APK, but it tries to extract similar information. There are different tools available to process and convert APK files into an IR, where popular IRs such as Smali, Jasmin, and Jimple represent Dalvik bytecode into a human-readable format. Apktool,[6] dex2jar,[7] soot,[8] and Androguard[9] are the most popular Android reversing tools. A comparison between these tools can be found in the study [54], which confirms that the most accurate IR is Smali. We used `Androguard` [55] to extract call graphs (CG) from Android apps because it is an open-source tool used for the static analysis that converts the APK file into Smali IR. The extracted call graph contains information on the different methods and their interrelationships. Androguard builds CG using the class hierarchy analysis (CHA) algorithm [56]. The CHA algorithm [57] visits each callsite and obtains the name of the method the callsite is referencing. It then adds an edge from the method containing the callsite to every method with the same name as the callsite. The result of the CHA is a completely sound call graph.

Each app has multiple methods that are interconnected and are represented as CG. In the CG, each method is represented as a node, and these methods are interconnected with the edges. A small APK has more than 1000 nodes and edges in its CG, which increases exponentially with the size of the app [26]. The instructions contained in the method are represented by the attributes of the node that we use to encode nodes. The CG of an APK contains a large number of different method calls, which means a large number of nodes and edges. It requires more time and computational resources to process; therefore, we reduce the size by extracting the FCG. FCG can be used to determine correlations between samples and is resistant to certain obfuscation techniques [58]. We extracted the FCG of the wake-lock related API[10] (i.e. `acquire()`, `release()`, and `isheld()`). For example, we first determine the node (method) where the wake-lock API `acquire()` is called. This node contains instructions that use the wake-lock API. This method is important for us; therefore, we next determine all the ancestors of this method. Ancestors refer to all the methods in the chain that call each other to reach this API, as shown in Fig. 2. It shows all the ancestors of the methods that call the `acquire()` API of the wake-lock. The node of `acquire()` is shown in "red". For simplicity, we do not mention the full name of the method. Notably, `acquire()` is called in two methods "UpdateBlankingBehavior" and "updateWakeLock" which are called by other

[6]https://github.com/iBotPeaches/Apktool
[7]https://github.com/pxb1988/dex2jar
[8]https://github.com/soot-oss/soot
[9]https://github.com/androguard/androguard

[10]https://developer.android.com/reference/android/os/PowerManager.WakeLock

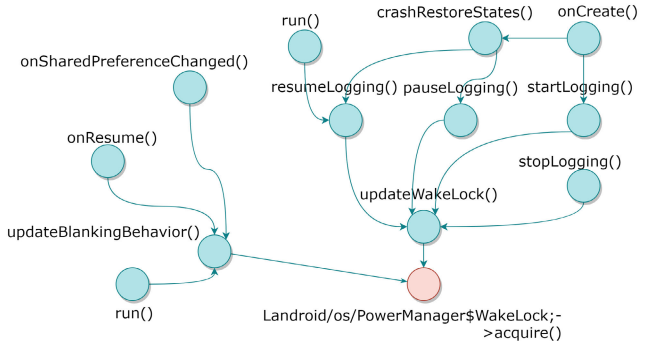


FIGURE 2. FCG of method acquire().

methods. Extracting FCG removes the complexity of the graph because it is much simpler as it only contains the nodes and edges related to the wake-lock API. Fig. 2 shows the graph of only `acquire()`; similarly, we extracted the FCG of `release()` and `isheld()` APIs of wake-lock. Some methods are common in these graphs because the `isheld()` can be called with acquire/release, which implies that they can be in the same method. Therefore, we merge the overlapping methods and create a single graph that represents the API calls a pattern of wake-lock. This removes the redundancy, and we obtained one graph containing all the method calls of wake-lock APIs. This step also filtered unrelated APIs. By extracting and pre-processing these FCGs, we reduced the dimension of the CG to include a graph that is related to wake-locks only.

These FCGs are directed graphs containing a node for each app's methods and edges from the caller to the callees. Encoding these graphs is an important task. We can have a unique encoded value for each graph; however, they do not exploit the properties shared between the methods. To embed the functional information, encoding was performed by applying the same encoded value to methods that share the same properties. App methods can have different lengths. To normalize them, we converted the method's label into a 15-bit array, and each bit represented the instructions presented in the method [8]. The Dalvik bytecode instructions [59] and the bits they represent are listed in Table.2. Notably, each instruction class was represented by a single bit, and these instruction classes represented all the variations in that instruction. For example, *move* represented all the instruction variations, such as *move-wide*, *move-object*, and *move-result*. In Table.3, we present an example code, where the "Instruction" column provides the sequence of instructions in the method and the "Instruction Class" and "Bit" columns represent the equivalent instruction and bit representations, respectively, according to Table.2. The bit representation of the label from the code example (Table.3) is presented in Table.4. In label we see that only the bits of those instructions were converted from "0" to "1" which are present in the method. Notably, multiple "invoke" and "move" instructions present in the method would have no effect on the label bit once they were converted to "1". The reason behind this is to capture

TABLE 2. Instruction classes and their bit representation.

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

TABLE 3. Example code and representation of instruction and bit.

Instruction	Instruction Class	Bit
invoke-virtual{p0},Ljava/lang/Object;->toString() Ljava/lang/String;	invoke	4
move-result-object v0	move	6
invoke-virtualp1,Ljava/lang/Object;->toString() Ljava/lang/String;	invoke	4
move-result-object v1	move	6
invoke-virtual{v0,v1},Ljava/lang/String;-> compareToIgnoreCase(Ljava/lang/String')I	invoke	4
move-result v0	move	6
return v0	return	15

TABLE 4. Output label that represents code.

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

the structure of the node, and for this, we only need which instructions are present in the method. This indicates that if the instruction sets in the different methods are the same, they will have the same label. If we consider the order of instruction and the number of times the instruction is present in the method, we need a more complex scheme to represent it. Encoding similar methods with the same label will capture the same functional properties and keep things simple.

After encoding the FCGs, we computed the neighborhood hash using the neighborhood hash graph kernel (NHGK) [52]. It is a kernel that operates over an enumerable set of subgraphs in the encoded graph. NHGK has high graphic structure expressiveness and low computational complexity. To calculate the NHGK, we first determine all the neighbors of the nodes. For example, in Fig. 2, `resumeLogging()` has neighbors `run()`, `crashRestoreStates()`, and `updateWakeLock()` where the encoded labels of these nodes are calculated in the previous step. The NHGK of method `resumeLogging()` is calculated by determining all the neighbor methods and then taking the XOR of their encoded label. We could go deeper into neighbors by including a neighbor's neighbor, but this would add to the complexity [52]. NHGK was used to condense the neighborhood information of the method into a single hash value. The key advantage of NHGK is that it analyzes networks with thousands of nodes in linear time, similar to FCG. The calculated

hash value was used to replace the encoded label, and this hash had the same bit length as the label. As a result, we had the hashed node, which held the function's instructions as well as neighborhood information. These hashes were fed into the machine learning algorithms as inputs.

B. IMBALANCED DATA

Training a machine learning algorithm requires more data, and the larger the dataset, the higher the accuracy. The number of apps containing wake-locks is very high, but we do not know whether they have wake-lock leaks, and we cannot use these apps in our experiment. We gathered apps containing wake-lock leaks from different sources, but the number of apps containing the wake-lock leaks was less than that of the clean apps. To detect the wake-lock leaks using machine learning, we must obtain more data. Therefore, we oversampled from the available app data. The data can be oversampled using the two techniques. First, by simply duplicating minority data multiple times, these examples do not add any new information to the model and simply overlap the previous values. Second, new data can be synthesized from the existing datasets. This is a type of data extension for the minority class and is referred to as SMOTE [44]. We used SMOTE to determine the effects of oversampling. The details of the SMOTE are discussed in Section IV-A.

VI. EVALUATION

In this section, we first introduce the experimental data and then show how machine learning algorithms are applied to detect wake-lock leaks. The dataset is classified as ''clean'' and ''leak''. We have used supervised machine learning, and the clean apps are labeled with the value of ''1'' and apps with wake-lock leaks are labeled as ''0''. Labeling is performed manually by creating two folders, one having apps with a wake-lock leak and the other having a clean app. When we ran the experiment, we labeled the apps according to their folder names. We further divided the data into training and test sets for machine learning models. The models were evaluated using an accuracy metric, and we further explained the results using a confusion matrix to demonstrate the effectiveness of the model in detecting the wake-lock leaks. Finally, we will also compare the accuracy, precision, recall, and f-measure of different machine learning algorithms. We also compared the accuracy of machine learning algorithms with other wake-lock detection tools.

A. DATASET

Collecting datasets for the wake-lock leak app is not an easy task; most of the tools that detect wake-lock leaks only check the limited number of apps to show the effectiveness of their tool [11], [26]. Other studies [21], [60], [63] have searched the comments of the repositories of apps to determine the energy leaks related to bugs. We do not have a large dataset of apps with wake-lock leaks as compared to a large dataset such as malware apps.[11] One option is to download a large number of apps from different app stores that use wake-lock permission in their app, but this will lead to a problem because we do not know whether the app has a wake-lock problem. Therefore, we cannot use random data from the app stores for training machine learning algorithms to detect wake-lock leaks. We must feed the apps with a wake-lock leak so that we can downloaded data from different sources and label them accordingly (as clean and leak). These downloaded apps were previously used in a similar type of research, and we knew the behavior of the app to label them. Table 5 summarizes the app data used in this study. The table is divided into three columns, and the `app name` is shown in the first column. The second column shows the Github version of the app that fixes the wake-lock leak problem, and we can search the version number of the app and determine the problem in the comments that they fix. The tools that use these apps for evaluation are shown in the third column. Notably, most of the studies used the same dataset because of the non-availability of a large number of identified apps. The source code of these apps is available online and can be downloaded for further studies. There are 32 apps with wake-lock problems, we collected both versions of each app having a wake-lock leak and the version that resolve this problem. The versions that removed the wake-lock problems were added to the

TABLE 5. Applications with version and tools.

App Name	Fixed Version	Tools
K-9 Mail	E6132286A0	[21], [26] , [6]
CallMeter	10729EA13B	[21], [26] , [6]
ConnectBot	10729EA13B, 669566E, D60794B08F	[21], [26] , [6]
OpenGPSTracker	8AC7905A5A	[21], [26] , [6]
VLC	14B18BC27F, ABE60F50F4, 6A85C2A2EC, 19483DE2A3	[21], [26], [6], [60]
CSipSimple	EAC20442BB, BF79346FCB, 88D62BC951, C72156E, 75A269A87D, 10D8D9A936, B94C56F0E2, 00EC304B0C, A388C20EB9	[21], [26], [6], [27], [10], [21], [61]
TomaHawk	B4F339BB24	[6], [7], [62]
MyTracks	17ECE1CD75, F2B4B968DF	[6], [7], [62], [27], [10]
Osmand	AC724B9, 41787D7E1E	[6], [7], [62], [21]
FBReader	15598C9, 769269336E	[6], [7], [62], [21]
Firefox	BE42FAE64E	[6], [7], [62], [63]
AndTweet	V-0.2.4	[11], [27]. [10]
BabbleSink	R-D12879A	[11], [27], [10] [64]
CWAC-Wakeful	R-D984B89	[11], [27], [10]
Ebookdroid	73b23550bb	[27], [10]
IRCCloud	CE5822D359	[21]

[21] = DroidLeak, [7] = Relda, [26] = Relda2, [6] = Elite, [60] = GreenOracle, [27] = GreenDroid, [10] = E-GreenDroid, [61] = SENTINEL, [62] = Verifier, [63] = GreenMinning, [11] = NavyDroid, [64] = EnergyPatch

[11] https://github.com/traceflight/Android-Malware-Datasets

clean app data. The apps with both clean and wake-lock leak versions will help the algorithm to narrow down the problem and determine the acute difference between wake-lock leaks and clean applications. The clean app dataset of 200 apps used in our experiment was also used in the ELITE tool [6] and identified as clean. These APK files used wake-lock in their apps but did not have any wake-lock issues causing energy leaks. After collecting data from different sources, we labeled the data accordingly.

B. EXPERIMENT

Different machine learning algorithms were used in the experiment to determine the best algorithm for classifying apps with wake-lock leaks. We used eight machine learning algorithms in our evaluation, five of which are common algorithms (NB, SVM, KNN, LR, and RC) and three ensemble algorithms (bagging decision tree (BDT), random forest (RF), and stochastic gradient boosting (SGB)). To remove the bias, we randomized our data and split the data into 80 % and 20 % for training and testing, respectively. Each machine learning algorithm requires different parameter values to be set, which play an important role in achieving high accuracy. To determine the optimized parameters, we performed a grid search and used these parameters. The grid search ran the training and testing data multiple times by changing the parameters of the algorithm to determine the optimal parameters. The grid search provided the best score, parameters, mean and standard deviation. We used the optimized parameters in our experiment, as listed in Table.6. In the grid search, we also cross-validated these algorithms using k-fold cross-validation. K-fold implies dividing the data into k sets; k-1 sets were used for training, and one remaining set was used for testing. This implies that the algorithm was trained and tested k times. In our evaluation, we used k = 10 and repeated the experiment three times using different training and testing data to remove noise. We used ''stratified k-fold'' because it ensures that the data used in each fold for training and the testing split is balanced. Table.6 shows the results of 10-fold cross-validation. The selected parameters that produce the highest accuracy are shown in the ''Parameters'' column. Optimized parameters were used for further testing. The accuracies of each algorithm are listed in Table.6.

To further explain the importance of machine learning algorithms we divided the experiment into two sub experiments.

1) NO OVER SAMPLING

In this experiment, we used the original 200 clean apps and 32 apps with wake-lock problems. After decompiling, extracting call graphs, encoding, and hashing (as discussed in Section V), we applied different machine learning algorithms to evaluate the results. We ran each algorithm 10 times to demonstrate their performance because each dataset split produced different accuracies, and we took the average of these results to remove the noise. It can be noted from Table. 7 that the accuracy of these machine learning algorithms is high (>90 %). This indicated that the machine was trained well on the data and correctly classified the apps. Our data were imbalanced. Therefore, our results should be evaluated further. A confusion matrix must be drawn to better demonstrate the consistency of the classified apps.

A confusion matrix is a matrix of two rows and two columns when there are only two classes. It reports the number of false positives (FP), false negatives (FN), true positives (TP), and true negatives (TN).

We selected the confusion matrix of the SVM algorithm to address this problem. The number of apps shown in the confusion matrix was 47, which is 20 % of the 232 apps used for testing because we randomly divided the training and testing sets into 80 % and 20 %, respectively. According to Table.8 ''34'' apps were detected as clean apps, which implied a high TN and indicated that the machine learning algorithm predicted clean apps more accurately. However, we were

TABLE 6. Grid search accuracy of algorithms with parameter.

Algorithm	Accuracy	Parameters
NB	0.85375	guassianNB
SVM	0.97375	c=10, gamma="scale", kernel="rbf"
KNN	0.97625	metric="euclidean", n_neighbors=3, weights="distance"
LR	0.97125	c=1.0, penalty="1", solver="liblinear"
RC	0.96875	alpha=0.7
BDT	0.98125	n_estimators=100
RF	0.98375	max_features="sqrt", n_estimators=100
SGB	0.98500	learning_rate=0.1, max_depth=9, n_estimators=1000, subsample=1.0

NB = Naïve-Bayes, SVM = Support Vector Machine, KNN = K-Near Neighbor, LR = Logistic Regression, RC = Ridge Classifier, BDT = Begged Decision Tree, RF = Random Forest, SGB = Stochastic Gradient Boosting.

TABLE 7. Accuracy of machine learning algorithms with no oversampling.

Algorithm	NB	SVM	KNN	LR	RC	BDT	RF	SGB
Iter 0	0.87	0.91	0.93	0.89	0.89	0.93	0.89	0.93
Iter 1	0.93	0.95	0.91	0.93	0.93	0.93	0.93	0.91
Iter 2	1.00	1.00	0.97	0.97	1.00	0.97	1.00	1.00
Iter 3	0.93	0.97	0.95	0.93	0.95	0.95	0.93	0.95
Iter 4	0.93	0.95	0.95	0.95	0.95	0.95	0.95	0.95
Iter 5	0.93	0.95	0.93	0.93	0.93	0.93	0.93	0.93
Iter 6	0.91	0.93	0.91	0.91	0.91	0.91	0.91	0.91
Iter 7	1.00	0.97	0.91	0.97	0.97	0.97	0.95	0.95
Iter 8	0.97	0.97	0.93	1.00	1.00	1.00	0.97	0.97
Iter 9	0.97	0.97	0.95	0.97	1.00	0.95	0.95	0.95
Mean	0.94	0.95	0.93	0.94	0.95	0.94	0.94	0.94

TABLE 8. Confusion matrix of a SVM.

		Predicted Class	
		Leak	Clean
Actual Class	Leak	**9**	1
	Clean	3	**34**

interested in apps containing wake-lock leaks. Notably, SVM detected only "9" apps as leaks owing to less data. Furthermore, "3" apps were detected as a leak but were clean (FP), and "1" app was detected as clean but contained leak app (FN), which demonstrated that the machine was unable to correctly predict apps with wake-lock leaks. The calculated recall, precision, accuracy, and f-measure were 0.90, 0.75, 0.91, and 0.81, respectively, indicating that the accuracy was high, but the precision and f-measure were not high. This is a common problem in an imbalanced dataset. This problem can be solved by oversampling the minority classes (apps with wake-lock leaks) and balancing the data, such that machine learning algorithms are trained correctly to classify the apps with a wake-lock problem.

2) OVERSAMPLING USING SMOTE

To solve the problem of imbalanced data, we used different oversampling techniques, as discussed in Section IV-A. In this experiment, we determined the effect of different sampling techniques on the accuracy of detecting apps with wake-lock problems.

First, we applied simple SMOTE, which uses the Python `imblearn` library to generate new samples of minority classes. Consequently, we have a dataset of 200 clean apps and 200 apps with wake-lock leaks. These generated samples were then passed through different machine-learning algorithms to obtain the classification accuracy.

We compared the accuracy of the oversampled data by applying different machine learning algorithms. Fig. 3 shows the accuracies of the various algorithms after applying SMOTE. The graph shows the number of iterations on the *x-axis* and the accuracy of the different algorithms on the *y-axis*. The number of iterations shown in the graph is 10 because training and testing were performed multiple times by splitting the data randomly for each iteration. Notably, NB performed the worst as it considered each feature independently for the calculation of the probability, and it did not consider any possible correlation between the features.

The accuracy of all the other algorithms was high, but the ensemble algorithms (BDT, RF, and SGB) had a slightly better predictive performance than the simple learning algorithms. Thus, the application of SMOTE produced the best results from the SGB algorithm. The testing results of oversampling using SMOTE were very accurate, but we also wanted to compare the effects of other oversampling techniques. These results are compiled in Table.9 which shows the accuracy of different oversampling algorithms (in the column) and their effect on the accuracy of different machine

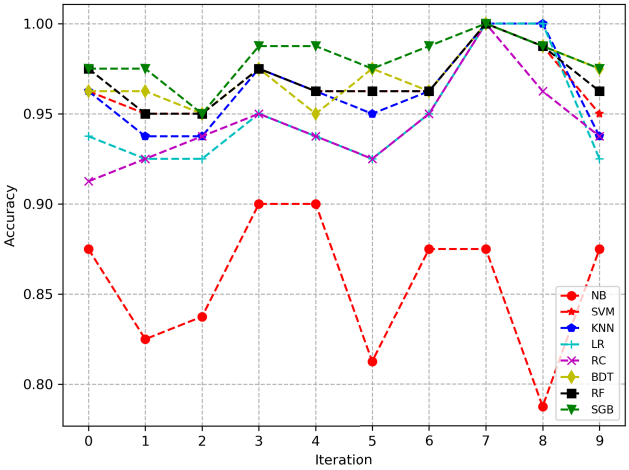


FIGURE 3. Accuracy using SMOTE oversampling.

TABLE 9. Accuracy of different machine learning algorithms with sampling.

Algorithm	No Oversampling	SMOTE	Random Oversampling	Oversampling Undersampling Tomek Link
NB	0.9489	0.8537	0.9450	0.8623
SVM	0.9638	**0.9762**	0.9712	**0.9724**
KNN	0.9404	0.9762	0.9687	0.9674
LR	0.9510	0.9712	0.9512	0.9662
RC	0.9553	0.9687	**0.9725**	0.9712
BDT	0.9553	0.9812	**0.9850**	0.9837
RF	0.9468	0.9837	0.9825	0.9825
SGB	0.9510	**0.9850**	0.9825	**0.9837**

learning algorithms (in rows). Notably, among ensemble algorithms, the SGB algorithm is highly accurate when "SMOTE" and "oversampling undersampling with Tomek link" were used. For "random oversampling", "BDT" performed a little better, but overall, the "SGB" algorithm had the highest accuracy for all the techniques.

On comparing basic machine learning algorithms, it can be noted that the accuracy of "SVM" using "SMOTE" and "oversampling undersampling Tomek Link" was the highest as compared to other machine learning algorithms (NB, KNN, LR, RC). The performance of "RC" was high when "random oversampling" was used. It can be concluded that oversampling using "SMOTE" was useful and the accuracy was the highest for both simple machine learning algorithms and ensemble algorithms, and it did not remove any samples such as in "oversampling undersampling Tomek Link". From the perspective of algorithms, it can be noted that "SVM" and "SGB" had the highest accuracy.

Oversampling with SMOTE may lead to overfitting. To show that our model does not overfit, we have presented the training and testing scores of SVM in Table. 10. Notably, both training and testing scores are almost the same; if we see a large difference between training and testing scores, then our model will be over-fitting. This implies that our model does not overfit and performs well on the data.

TABLE 10. Training and testing score of SVM.

Training Score	Testing Score
0.6275	0.6275
0.6275	0.6275
0.8525	0.8525
0.9625	0.9600
0.9818	0.9625

TABLE 11. Confusion matrix of SVM after SMOTE.

		Predicted Class	
		Leak	Clean
Actual Class	Leak	**46**	2
	Clean	0	**32**

TABLE 12. Classification report.

Algorithm	Accuracy	Precision	Recall	F-measure
NB	0.90	0.87	0.97	0.92
SVM	**0.97**	**1.0**	**0.96**	**0.98**
KNN	**0.97**	**1.0**	**0.96**	**0.98**
LR	0.95	1.0	0.92	0.96
RC	0.95	1.0	0.96	0.98
BDT	0.97	1.0	0.96	0.98
RF	0.97	1.0	0.96	0.98
SGB	**0.98**	**1.0**	**0.98**	**0.99**

To further illustrate the results, we consider the effectiveness of these algorithms in detecting wake-lock leak apps and draw a confusion matrix. Table.11 presents the confusion matrix of ''SVM'' after applying ''SMOTE''.

The confusion matrix shows 80 apps that were used for testing; there were a total of 400 apps after SMOTE and 320 were used for training. The confusion matrix indicated that the algorithm correctly detected ''46'' apps containing wake-lock leaks (TP), and only ''2'' apps were detected as FN. Moreover, it is well trained in detecting clean apps (TN). Similar results were observed with other algorithms, implying that applying oversampling (SMOTE) increased the accuracy of detecting wake-lock leaks with high precision and recall.

The classification report of each machine learning algorithm used is presented in Table.12. The accuracy, precision, recall, and f-measure are presented in Table.12 to illustrate the results. According to these results, SGB was the best performing algorithm with the highest accuracy, precision, recall, and f-measure. During testing, we observed that the SGB took a long time to train and test. This requires more processing time than simple machine learning algorithms. If we have low computational resources and a large amount of dataset for training, then we can use a simple machine learning algorithm and sacrifice a little accuracy for better performance. From simple machine-learning algorithms, we can choose an SVM.

To check whether machine learning algorithms perform better, we performed an additional evaluation on the entire CG of the app, which implies, not filtering and extracting FCG. The results of this evaluation are listed in Table. 13.

TABLE 13. Accuracy of machine learning algorithms with CG.

Algorithm	NB	SVM	KNN	LR	RC	BDT	RF	SGB
Iter 0	0.96	0.92	0.88	0.95	0.91	0.93	0.92	0.91
Iter 1	0.91	0.86	0.91	0.87	0.86	0.90	0.87	0.90
Iter 2	0.90	0.81	0.90	0.87	0.86	0.90	0.90	0.87
Iter 3	0.88	0.91	0.83	0.90	0.88	0.91	0.88	0.92
Iter 4	0.91	0.85	0.87	0.91	0.85	0.86	0.90	0.86
Iter 5	0.90	0.60	0.88	0.87	0.87	0.88	0.87	0.88
Iter 6	0.90	0.91	0.92	0.90	0.85	0.90	0.88	0.88
Iter 7	0.93	0.85	0.87	0.92	0.83	0.92	0.88	0.90
Iter 8	0.90	0.91	0.87	0.87	0.87	0.90	0.87	0.86
Iter 9	0.93	0.80	0.90	0.93	0.93	0.93	0.93	0.93
Mean	**0.91**	0.84	0.88	0.89	0.87	0.90	0.89	0.89

TABLE 14. Confusion matrix of ELITE tool.

		Predicted Class	
		Leak	Clean
Actual Class	Leak	**55**	17
	Clean	116	**1532**

We can see from the table that the accuracy of machine learning algorithms is much lower than the results obtained by extracting FCG. Notably, the accuracy of all machine learning decreased. The most accurate algorithm is now Naïve Bayes, and SVM has the lowest accuracy. On the other hand, the remaining algorithms had an accuracy of almost of 90 %. The overall accuracy of all the algorithms decreases because the machine learning algorithms perform better when the unrelated features are filtered out, which is also known as dimension reduction [65]. We conclude that extracting FCG from the CG helps to increase the accuracy of these machine learning algorithms. The source code for implementation is available online.[12]

C. COMPARISON WITH ELITE TOOL

We chose the ELITE tool [6] for two main reasons to compare the accuracy of the machine learning algorithms. First, it is open-source and available online for evaluation, whereas most of the other tools are not open-source [66]. Second, we use APKs in our evaluation and some tools to evaluate wake-lock leaks on the source code [17], [18], [20]. Gathering the source code of all apps is not feasible; therefore, we used the ELITE tool for comparison.

To obtain meaningful results, we could not apply random or new datasets for comparison. The dataset we used to evaluate the ELITE tool was the same as that used to train and test the machine learning algorithms. We used two datasets and divided the comparison into two stages. In the first stage, we apply the ELITE tool to 2,000 apps[13] used in their empirical study (which contains 44,736 apps and uses wake-lock). The list of apps we used in our comparison is also provided in our Github repository, which contains the `app IDs` and `permissions` used by the app. We found that the ELITE tool was able to analyze 1,642 apps; among them, 110 apps

[12]https://github.com/umkhanqta/MLwakelockleak

[13]http://sccpu2.cse.ust.hk/elite/downloadApks.html

TABLE 15. ELITE tool evaluation result.

App Name	Version	ELITE Output	Actual Output
AndTweet	v 0.2.4	Clean	Leak
Babblesink	12879A3	Clean	Leak
CallMeter	4E9106C	Leak	Clean
	10729EA	Clean	Leak
ConnectBot	2DFA7AE	Leak	Clean
	669566E	Leak	Clean
	76C4F80	Clean	Leak
CsipSimple	1B2D2B6	Leak	Clean
	A388C20	Clean	Leak
	10D8D9A	Clean	Leak
	C72156E	Clean	Leak
	00EC304	Clean	Leak
	352FC6C	Clean	Leak
	04496E6	Clean	Leak
	B94C56F	Clean	Leak
Cwac-wakeful	D984B89	Clean	Leak
K9Mail	0A07250	Leak	Clean
Open-GPSTracker	8AC7905	Clean	Leak
OsmAnd	AC724B9	Clean	Leak
	F314EA3	Clean	Leak
	3852E55	Leak	Clean
VLC	ABE60F5	Clean	Leak
	14B18BC	Clean	Leak
	19483DE	Clean	Leak
	6A85C2A	Clean	Leak

were detected as wake-lock leaks (FPs), which showed a high false-positive rate of 6.7 % and accuracy of 93 %. In the second stage, we further analyze the tool by checking both clean and leak apps from Table 5. These were a total of 78 apps, of which 32 were identified as leaks and 46 as clean. When we analyzed them using the ELITE tool, 23 apps were not correctly identified. After further investigation, we found that six apps were found to be FP, and 17 were FN.

In summary, we used a total of 1,720 apps, of which 17 were FN, 116 were FP, 55 were TP, and 1,532 were TN, as shown in Table. 14. The accuracy of the tool on these apps was 92 %, with a false-positive rate of 7.3 %. The app version that was incorrectly identified by the tool is shown in Table. 15 for reproducibility purposes. The table shows the name of the app, their Github commits/APK version used in the experiment, the classification outcome of the ELITE tool, and the actual outcome. We conclude from Table. 12 that the machine learning algorithms have higher accuracy (98 %), whereas the ELITE tool has an accuracy of 92 % and a high false-positive rate, as shown in Table. 14.

VII. DISCUSSION

Reducing the energy consumption of mobile devices is important [4]. This study plays a crucial role in detecting energy leaks related to wake-locks in Android apps. Notably, this field has not been extensively explored by the research community, especially in the field of machine learning. The main hurdle to applying machine learning in this field is the availability of large datasets. In this study, we attempted to overcome this problem using SMOTE. We used machine learning algorithms to detect wake-lock leaks in Android apps. This trained ML model can be used to process a large number of apps, such as in the Play Store, to detect wake-lock leaks in apps before publishing them online and ensuring the quality of the apps. There are some open issues that need to be resolved in our research.

We must obtain more apps with wake-lock leaks to train our model to effectively detect apps with wake-lock leaks. One way to obtain more data is to extrapolate them using different techniques [44], [67], [68]. In this study, we used SMOTE to generate new samples. Another way to obtain more data is to modify the original apps and insert code segments with wake-lock leaks using different techniques [69], [70]. We can extract data from the Github repository and from the comments of the Play Store to determine apps facing the wake-lock problems [21].

Our study provides a basic dataset of wake-lock leak apps that we collected from different sources and can be validated using the Github version. This dataset can be used to create an online database of wake-lock leaks because the available dataset is insufficient compared to the malware dataset.

We must have benchmark apps to verify the accuracy of the tool. This benchmark should evaluate the tool's accuracy in detecting different types of wake-lock leaks. We can create different apps with different wake-lock problems and use them as benchmarks. These apps need to be created carefully so that they cover each wake-lock leak type in detail.

A comparative study between different tools is required to determine which tool detects wake-lock leaks more efficiently with the pros and cons of each tool. A comparative study can be carried out when we have standard benchmark apps, or when we know the outcome of the app (wake-lock leak or clean). We obtained data from many tools in our study, and these tools demonstrate the usefulness of their tool on the same data; thus, we cannot apply the same data in a comparative analysis. However, we made our findings public, encouraging other researchers to conduct comparable studies.

Despite the aforementioned issues, we apply machine learning techniques to detect wake-lock leaks in Android apps. Our results are encouraging for researchers because they show that machine learning algorithms can be effectively used to detect wake-lock leaks in Android apps. Our dataset can be used to create an online database for wake-lock leak apps, which can further be used to compare the accuracy of different tools in detecting wake-lock leaks. The trained machine learning algorithm used in our study can be used to detect wake-lock leaks in big datasets (Play Store, F-droid, Androzoo, and other popular stores) to ensure the quality of the app.

VIII. THREATS TO VALIDITY

A. INTERNAL VALIDITY

Filtering the CG to FCG may affect the results because we use FCG to minimize the complexity and processing power of CG by filtering out unwanted APIs and using only APIs related to wake-lock. This process has an impact on

the outcomes of machine learning training. To mitigate this challenge, we evaluated the results by training the machine learning algorithm with the CG. Table. 13 demonstrates that the accuracy decreases when CG is used. This shows that filtering CG to FCG improves the accuracy of the results.

To ensure that we have sufficient data for the experiment, we rely on SMOTE to generate synthetic examples similar to the original apps, which may cause overfitting or underfitting. To check for overfitting and underfitting, we used Table. 10. Notably, the training and testing scores are close to each other, which indicates that our model is fit normally. If the difference between them is large, then we may have an overfitting or underfitting problem.

Our data are imbalanced, and to train machine learning algorithms, we must train them by splitting them equally. Therefore, we used `StratifiedKFold`, which split the samples of each class percentage equally and training is performed in an equal number of samples.

B. EXTERNAL VALIDITY

The dataset used to train and validate the machine learning algorithm plays a crucial role. Because we do not have a well-known dataset, the number of apps included in our experiment is restricted. Table. 5 summarizes the apps used in the experiment. We ensured that these apps have wake-lock problems by accessing their Github page and finding the comments of the version to remove ambiguity. The results may change when a large dataset is collected.

We used only APK in our dataset to represent a wide portion of the Android app market. Some apps come in.jar and.zip file formats, which can be extracted using the same approach. We may use source code from the repository to train machine learning, but this would limit our analysis because it could only be used with source code, and we did not want to limit our studies.

Another threat we face in our study is that Androguard has the limitation of using only the CHA algorithm to construct CGs and not providing any option to change the algorithm. There are some other algorithms, such as RTA, VTA, and SPARK, but the difference between the performance and memory cost is very small, indicating that this will not have a significant effect on our results.

IX. CONCLUSION AND FUTURE WORK

In this study, we used machine learning algorithms to detect wake-lock leaks and evaluate the results. To train the machine learning algorithms, we decompiled the Android app and constructed the FCG of wake-lock APIs, and encoded them. The dataset of wake-lock leak apps was small, which led to an imbalanced data problem. Different types of oversampling techniques were applied to balance imbalanced data problems, and it was noted that SMOTE oversampling performed well. This balanced data was then used to train different machine-learning algorithms, which were optimized using a grid search algorithm. The results indicated that SVM and SGB were the most effective and detected wake-lock leaks with high accuracy and precision.

The results demonstrated that machine learning algorithms were effective in detecting energy leakage in mobile apps. This opens several new research directions that can be applied to solve the energy leakage problem in mobile apps and encourage researchers to detect code smells that lead to energy leaks, detect other resource leaks, and observe the performance of other machine learning techniques. In future work, we will obtain more data and observe the effects of machine learning algorithms without oversampling, and include other energy leaks.

REFERENCES

[1] *Smartphone Users 2020, Statista*. Accessed: Dec. 30, 2020. [Online]. Available: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/

[2] H. Cheng, J. G. Shapter, Y. Li, and G. Gao, "Recent progress of advanced anode materials of lithium-ion batteries," *J. Energy Chem.*, vol. 57, pp. 451–468, Jun. 2021.

[3] *App-Stats*. Accessed: Jan. 26, 2021. [Online]. Available: https://mindsea.com/app-stats

[4] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of Android apps: A systematic literature review," *Inf. Softw. Technol.*, vol. 88, pp. 67–95, Aug. 2017.

[5] *How to Set Dark Mode on Your Favorite Apps PCMag*. Accessed: May 13, 2020. [Online]. Available: https://www.pcmag.com/how-to/how-to-set-up-dark-mode-on-your-favorite-apps

[6] Y. Liu, C. Xu, S.-C. Cheung, and V. Terragni, "Understanding and detecting wake lock misuses for Android applications," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2016, pp. 396–409.

[7] Z. Xu, C. Wen, and S. Qin, "State-taint analysis for detecting resource bugs," *Sci. Comput. Program.*, vol. 162, pp. 93–109, Sep. 2018.

[8] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of Android malware using embedded call graphs," in *Proc. ACM Workshop Artif. Intell. Secur.*, Nov. 2013, pp. 45–54.

[9] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps," in *Proc. 10th Int. Conf. Mobile Syst., Appl., Services*, 2012, pp. 267–280.

[10] J. Wang, Y. Liu, C. Xu, X. Ma, and J. Lu, "E-greendroid: Effective energy inefficiency analysis for Android applications," in *Proc. 8th Asia–Pacific Symp. Internetware*, 2016, pp. 71–80.

[11] Y. Liu, J. Wang, C. Xu, and X. Ma, "Navydroid: Detecting energy inefficiency problems for smartphone applications," in *Proc. 9th Asia–Pacific Symp. Internetware*, 2017, pp. 1–10.

[12] Z. Li, J. Sun, Q. Yan, W. Srisa-an, and S. Bachala, "Grandroid: Graph-based detection of malicious network behaviors in Android applications," in *Proc. Int. Conf. Secur. Privacy Commun. Syst.* Singapore: Springer, Aug. 2018, pp. 264–280.

[13] K. Liu, S. Xu, G. Xu, M. Zhang, D. Sun, and H. Liu, "A review of Android malware detection approaches based on machine learning," *IEEE Access*, vol. 8, pp. 124579–124607, 2020.

[14] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: Are we there yet?" in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Mar. 2018, pp. 612–621.

[15] L. Zhao, D. Li, G. Zheng, and W. Shi, "Deep neural network based on Android mobile malware detection system using opcode sequences," in *Proc. IEEE 18th Int. Conf. Commun. Technol. (ICCT)*, Oct. 2018, pp. 1141–1147.

[16] Z. Wu, X. Chen, and S. U.-J. Lee, "FCDP: Fidelity calculation for description-to-permissions in Android apps," *IEEE Access*, vol. 9, pp. 1062–1075, 2021.

[17] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Lightweight detection of android-specific code smells: The adoctor project," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Feb. 2017, pp. 487–491.

[18] J. Reimann, M. Brylski, and U. Aßmann, "A tool-supported quality smell catalogue for Android developers," in *Proc. Conf. Workshop Modellbasierte Modellgetriebene Softw. Modernisierung*, 2014, pp. 1–2.
[19] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "On the impact of code smells on the energy consumption of mobile applications," *Inf. Softw. Technol.*, vol. 105, pp. 43–55, Jan. 2019.
[20] L. Cruz and R. Abreu, "Catalog of energy patterns for mobile applications," *Empirical Softw. Eng.*, vol. 24, no. 4, pp. 2209–2235, Aug. 2019.
[21] Y. Liu, J. Wang, L. Wei, C. Xu, S.-C. Cheung, T. Wu, J. Yan, and J. Zhang, "DroidLeaks: A comprehensive database of resource leaks in Android apps," *Empirical Softw. Eng.*, vol. 24, no. 6, pp. 3435–3483, Dec. 2019.
[22] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for Java program analysis: A retrospective," in *Proc. Cetus Users Compiler Infastruct. Workshop (CETUS)*, vol. 15, 2011, p. 35.
[23] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Dexpler: Converting Android Dalvik bytecode to jimple for static analysis with soot," in *Proc. ACM SIGPLAN Int. Workshop State Art Java Program Anal.*, 2012, pp. 27–38.
[24] A. Banerjee and A. Roychoudhury, "Automated re-factoring of Android apps to enhance energy-efficiency," in *Proc. Int. Conf. Mobile Softw. Eng. Syst.*, May 2016, pp. 139–150.
[25] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang, "Characterizing and detecting resource leaks in Android applications," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2013, pp. 389–398.
[26] T. Wu, J. Liu, X. Deng, J. Yan, and J. Zhang, "Relda2: An effective static analysis tool for resource leak detection in Android apps," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2016, pp. 762–767.
[27] Y. Liu, C. Xu, S. C. Cheung, and J. Lu, "GreenDroid: Automated diagnosis of energy inefficiency for smartphone applications," *IEEE Trans. Softw. Eng.*, vol. 40, no. 9, pp. 911–940, Sep. 2014.
[28] A. Naway and Y. LI, "A review on the use of deep learning in Android malware detection," 2018, *arXiv:1812.10360*. [Online]. Available: http://arxiv.org/abs/1812.10360
[29] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app: Fine grained energy accounting on smartphones with Eprof," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 29–42.
[30] A. M. Abbasi, M. Al-Tekreeti, K. Naik, A. Nayak, P. Srivastava, and M. Zaman, "Characterization and detection of tail energy bugs in smartphones," *IEEE Access*, vol. 6, pp. 65098–65108, 2018.
[31] S. Kurihara, S. Fukuda, T. Kamiyama, A. Fukuda, M. Oguchi, and S. Yamaguchi, "Estimation of power consumption of each application considering software dependency in Android," *J. Inf. Process.*, vol. 27, no. 4, pp. 221–232, 2019.
[32] R. J. Behrouz, A. Sadeghi, J. Garcia, S. Malek, and P. Ammann, "Ecodroid: An approach for energy-based ranking of Android apps," in *Proc. IEEE/ACM 4th Int. Workshop Green Sustain. Softw.*, May 2015, pp. 8–14.
[33] R. Jabbarvand, A. Sadeghi, H. Bagheri, and S. Malek, "Energy-aware test-suite minimization for Android apps," in *Proc. 25th Int. Symp. Softw. Test. Anal.*, Jul. 2016, pp. 425–436.
[34] C. Zhu, Z. Zhu, Y. Xie, W. Jiang, and G. Zhang, "Evaluation of machine learning approaches for Android energy bugs detection with revision commits," *IEEE Access*, vol. 7, pp. 85241–85252, 2019.
[35] *Android Permissions in Manifest*. Accessed: Dec. 24, 2020. [Online]. Available: https://developer.android.com/reference/android/Manifest.permission?hl=en
[36] *Power Management*. Accessed: Apr. 28, 2020. [Online]. Available: https://developer.android.com/about/versions/pie/powerbucket-best
[37] *Google Play Store*. Accessed: Dec. 31, 2020. [Online]. Available: https://play.google.com/store,
[38] *F-Droid—Free and Open Source Android App Repository*. Accessed: Dec. 31, 2020. [Online]. Available: https://f-droid.org/en/
[39] *APKMirror—Free APK Downloads*. Accessed: Dec. 31, 2020. [Online]. Available: https://www.apkmirror.com/
[40] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of Android apps for the research community," in *Proc. IEEE/ACM 13th Work. Conf. Mining Softw. Repositories (MSR)*, May 2016, pp. 468–471.
[41] *Download APK Free Online Downloader*. Accessed: Dec. 31, 2020. [Online]. Available: https://apkpure.com/
[42] *Android Developers Guide and Documentation*. Accessed: Dec. 24, 2020. [Online]. Available: https://developer.android.com/docs
[43] *Android-Studio*. Accessed: Apr. 26, 2021. [Online]. Available: https://developer.android.com/studio
[44] H. Mansourifar and W. Shi, "Deep synthetic minority oversampling technique," 2020, *arXiv:2003.09788*. [Online]. Available: https://arxiv.org/abs/2003.09788
[45] A. Géron, *Hands-on Machine Learning With Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. Newton, MA, USA: O'Reilly Media, 2019.
[46] I. Rish, "An empirical study of the naive Bayes classifier," in *Proc. IJCAI Workshop Empirical Methods Artif. Intell.*, 2001, vol. 3, no. 22, pp. 41–46.
[47] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, 1995.
[48] S. A. Dudani, "The distance-weighted K-nearest-neighbor rule," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-6, no. 4, pp. 325–327, Apr. 1976.
[49] D. G. Kleinbaum, K. Dietz, M. Gail, M. Klein, and M. Klein, *Logistic Regression*. New York, NY, USA: Springer-Verlag, 2002. [Online]. Available: https://www.springer.com/gp/book/9781441917416
[50] R. Maclin and D. Opitz, "Popular ensemble methods: An empirical study," 2011, *arXiv:1106.0257*. [Online]. Available: http://arxiv.org/abs/1106.0257
[51] L. Breiman, "Bagging predictors," *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, 1996.
[52] S. Hido and H. Kashima, "A linear-time graph kernel," in *Proc. 9th IEEE Int. Conf. Data Mining*, Dec. 2009, pp. 179–188.
[53] J. H. Friedman, "Stochastic gradient boosting," *Comput. Statist. Data Anal.*, vol. 38, no. 4, pp. 367–378, 2002.
[54] Y. L. Arnatovich, L. Wang, N. M. Ngo, and C. Soh, "A comparison of Android reverse engineering tools via program behaviors validation based on intermediate languages transformation," *IEEE Access*, vol. 6, pp. 12382–12394, 2018.
[55] *Androguard*. Accessed: Dec. 11, 2020. [Online]. Available: https://code.google.com/archive/p/androguard/
[56] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! analyzing unsafe and malicious dynamic code loading in Android applications," in *Proc. NDSS*, vol. 14, 2014, pp. 23–26.
[57] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *Proc. Eur. Conf. Object-Oriented Program.* Aarhus, Denmark: Springer, Aug. 1995, pp. 77–101. [Online]. Available: https://www.springer.com/gp/book/9783540495383
[58] T. Wu, J. Liu, Z. Xu, C. Guo, Y. Zhang, J. Yan, and J. Zhang, "Light-weight, inter-procedural and callback-aware resource leak detection for Android apps," *IEEE Trans. Softw. Eng.*, vol. 42, no. 11, pp. 1054–1076, Mar. 2016.
[59] *Dalvik Bytecode Instruction*. Accessed: Dec. 11, 2020. [Online]. Available: https://source.android.com/devices/tech/dalvik/dalvik-bytecode
[60] S. A. Chowdhury and A. Hindle, "GreenOracle: Estimating software energy consumption with energy measurement corpora," in *Proc. 13th Int. Conf. Mining Softw. Repositories*, May 2016, pp. 49–60.
[61] H. Wu, Y. Wang, and A. Rountev, "Sentinel: Generating gui tests for Android sensor leaks," in *Proc. IEEE/ACM 13th Int. Workshop Autom. Softw. Test (AST)*, May 2018, pp. 27–33.
[62] P. Vekris, R. Jhala, S. Lerner, and Y. Agarwal, "Towards verifying Android apps for the absence of no-sleep energy bugs," in *Proc. Workshop Power-Aware Comput. Syst. (HotPower)*, 2012, pp. 1–5.
[63] A. Hindle, "Green mining: A methodology of relating software change and configuration to power consumption," *Empirical Softw. Eng.*, vol. 20, pp. 374–409, 2015. [Online]. Available: https://link.springer.com/article/10.1007/s10664-013-9276-6, doi: 10.1007/s10664-013-9276-6.
[64] A. Banerjee, L. K. Chong, C. Ballabriga, and A. Roychoudhury, "EnergyPatch: Repairing resource leaks to improve energy-efficiency of Android apps," *IEEE Trans. Softw. Eng.*, vol. 44, no. 5, pp. 470–490, May 2018.
[65] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning*. Cambridge, MA, USA: MIT Press, 2018.
[66] S. Habchi, "Understanding mobile-specific code smells," Ph.D. dissertation, Dept. Comput. Sci., Univ. de Lille, Lille, France, 2019. [Online]. Available: https://www.semanticscholar.org/paper/Understanding-Mobile-Specific-Code-Smells-Habchi/d053ab8424a9b8c6d088f5cdbd5417259e0c8060
[67] H. He, Y. Bai, E. A. Garcia, and S. Li, "ADASYN: Adaptive synthetic sampling approach for imbalanced learning," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, Jun. 2008, pp. 1322–1328.

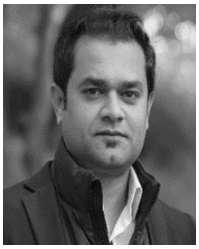
[68] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," 2014, *arXiv:1406.2661*. [Online]. Available: http://arxiv.org/abs/1406.2661
[69] L. Li, "Boosting static analysis of Android apps through code instrumentation," in *Proc. 38th Int. Conf. Softw. Eng. Companion*, May 2016, pp. 819–822.
[70] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar, "A platform for secure static binary instrumentation," in *Proc. 10th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, 2014, pp. 129–140.

MUHAMMAD UMAIR KHAN received the B.S. degree in computer engineering from Balochistan University of Information Technology, Engineering and Management Science, Pakistan, and the M.S. degree in computer engineering from Lahore University of Management Sciences, Pakistan. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, Hanyang University, South Korea, funded by the Higher Education Commission, Pakistan. His research interests include android development, program analysis, and machine learning.

SCOTT UK-JIN LEE (Member, IEEE) received the B.S. degree in software engineering and the Ph.D. degree in computer science from The University of Auckland, New Zealand. After the Ph.D. degree, he was a Postdoctoral Research Fellow with the Commissariat à l'Énergie Atomique et aux Énergies Alternatives, France. He has been with the Department of Computer Science and Engineering, Hanyang University ERICA Campus, South Korea, since 2011. He is currently serving as an Associate Professor major in bio artificial intelligence for the Department of Computer Science and Engineering. His research interests include software engineering, formal methods, and quality assurance. He is also a member of the Korean Institute of Information Scientists and Engineers and the Korean Society of Computer and Information. He has served as an editor, the technical chair, and a committee member for several journals and conferences.

SHANZA ABBAS received the B.S. degree in information technology from the University of the Punjab, Pakistan, and the M.S. degree in software engineering from the National University of Science and Technology, Pakistan. She is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, Hanyang University, South Korea, funded by the Higher Education Commission, Pakistan, in 2018. Her research interest includes natural language interface to database systems with machine learning.

ASAD ABBAS received the B.S. degree in information technology from the University of the Punjab, Pakistan, in 2011, the M.S. degree leading to the Ph.D. degree from the Department of Computer Science and Engineering, Hanyang University ERICA Campus, Ansan, South Korea, funded by the Higher Education Commission, Pakistan, in 2014, and the Ph.D. degree in computer science and engineering from Hanyang University, in August 2018. He served as an Assistant Professor for the Department of Software Engineering, The University of Lahore, Pakistan. He is currently serving as an Assistant Professor for the Faculty of Information and Technology, University of Central Punjab (UCP), Lahore, Pakistan. His research interests include software product line, the IoT systems and applications and embedded software applications, big data analytics, and machine learning.



ALI KASHIF BASHIR (Senior Member, IEEE) received the B.Sc. degree (Hons.) in computer forensics and security from the Department of Computing and Mathematics, Manchester Metropolitan University, U.K., and the Ph.D. degree in computer science and engineering from Korea University, South Korea.

His past assignments include an Associate Professor of ICT with the University of the Faroe Islands, Denmark; Osaka University, Japan; Nara College, National Institute of Technology, Japan; the National Fusion Research Institute, South Korea; Southern Power Company Ltd., South Korea, and Seoul Metropolitan Government, South Korea. He is currently a Senior Lecturer/an Associate Professor and the Program Leader of the Department of Computing and Mathematics, Manchester Metropolitan University. He is also with the School of Electrical Engineering and Computer Science, National University of Science and Technology (NUST), Islamabad, as an Adjunct Professor, and the School of Information and Communication Engineering, University of Electronics Science and Technology of China (UESTC) as an Affiliated Professor and the Chief Advisor of the Visual Intelligence Research Center. He has worked on several research and industrial projects of South Korean, Japanese and European agencies and Government ministries. In his career, he has received over 2.5 Million USD funding. He has authored over 180 research articles, received funding as the PI and the Co-PI from research bodies of South Korea, Japan, EU, U.K., and Middle East, and supervising/co-supervising several graduate (M.S. and Ph.D.) students. His research interests include the Internet of Things, wireless networks, distributed systems, network/cyber security, network function virtualization, and machine learning.

Dr. Bashir is a member of the IEEE Industrial Electronic Society and ACM. He is leading many conferences as the chair (program, publicity, and track) and had organized workshops in flagship conferences, like IEEE Infocom, IEEE Globecom, and IEEE Mobicom. He is serving as the Editor-in-Chief for the IEEE Future Directions Newsletter. He is also serving as an Area Editor for *KSII Transactions on Internet and Information Systems*, and an Associate Editor for *IEEE Internet of Things Magazine*, IEEE Access, *PeerJ Computer Science*, *IET Quantum Communication*, and *Journal of Plant Diseases and Protection*. He is a Distinguished Speaker of ACM.