IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received December 8, 2021, accepted January 7, 2022, date of publication January 12, 2022, date of current version January 21, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3142345

Privacy Protection Framework for Android

BHARAVI MISHRA[1], AASTHA AGARWAL[1], AYUSH GOEL[1], AMAN AHMAD ANSARI[1], PRAMOD GAUR[2], DILBAG SINGH[3], (Member, IEEE), AND HEUNG-NO LEE[3], (Senior Member, IEEE)

[1]The LNM Institute of Information Technology, Rajasthan, Jaipur 302031, India
[2]Birla Institute of Technology and Science Pilani, Dubai, United Arab Emirates
[3]School of Electrical Engineering and Computer Science, Gwangju Institute of Science and Technology, Gwangju 61005, South Korea

Corresponding author: Heung-No Lee (heungno@gist.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government (MSIP) under Grant NRF-2021R1A2B5B03002118; and in part by the Ministry of Science and ICT (MSIT), South Korea, under the Information Technology Research Center (ITRC) Support Program under Grant IITP-2021-0-01835 supervised by the Institute of Information & Communications Technology Planning & Evaluation (IITP).

ABSTRACT The increase in popularity and users of the Android platform in recent years has led to a lot of innovative and smart Android applications (apps). Many of these apps are highly interactive, customizable, and require user data to provide services. While being convenient, user privacy is the primary concern. It is not guaranteed that these apps are not storing user data for their need or scrapping algorithms through them. Android uses the system of permissions to provide security and protect user data. The user can grant permission for requested resources either at runtime or during the installation process. However, this system is often misused in practice by demanding extra permissions that are not required to provide services. These kinds of apps stop functioning if all permissions are not granted to them. Therefore, in this paper, a privacy preserved secure framework is proposed to prevent an app from stealing user data by restricting all unnecessary permissions. Unnecessary permissions are recognized by predicting the permissions required by a given app by using collaborative filtering and frequent permission set mining algorithms. Thus, the proposed model interacts with the target application and modifies the permission data inside. Experimental results reveal that the proposed model not only protects the user data but also ensures the proper functioning of the given application.

INDEX TERMS Android, instrumentation, permission model, security, privacy.

I. INTRODUCTION

Android is the most popular operating system (OS) when it comes to mobile platforms. According to Global Stats [1], Android OS enjoys almost 75% of the market share in the Mobile OS Industry, followed by iOS with a 25% share in June 2020. Users prefer Android because of its free and open-source nature with support for many apps. Developers also select Android over the competitive iOS as it is open-source in nature. Applications for Android are written mainly in Java and are commonly referred to as 'apps'.

Security is a crucial aspect of apps. The nature of Android apps makes it difficult to rely on standard, traditional, and dynamic malware analysis systems [2]. Google launched a Google play app security improvement program for providing security services to Google Play app developers to improve the security of their apps [3]. Apps are scanned for potential malware before uploading on Play Store. In 2017, Google worked on detecting malware and potentially harmful apps for improving security on devices and Play Store using Google Play Protect [4]. In Android 9, Google put a restriction on the access of sensors in the background so that apps running in the background cannot access the camera, microphone, and sensors [5].

To protect user data and passwords, Google has provided a feature for hardware-backed keys. Safe Browsing application programming interface (API) is also present for protection against deceptive websites. While there have been significant developments towards platform security, application development security, and secure Android OS, the apps taking user data can sometimes be malicious. With over 2.7 million apps already present in the Google play store [6], it is hard to determine which app is malicious or which may take data to analyze behavior or sell it to any third party.

To prevent the issue of data security and malicious usage of the applications, Android works on the principle of

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

permissions, i.e., the apps must ask for the permissions required for their applications from the user, and Android provides access to the APIs of that very permission only if the user grants it [7]–[9]. The Android system has permission categories based on security levels - normal, dangerous, and signature permissions. The developer just needs to mention the required permissions in the manifest file. The Android system does not prompt users for such permissions. The dangerous permissions could lead to some severe data breach problems. Hence, these permissions need to be mentioned in the manifest file and should prompt the user to allow the app to use these permissions inside the app. It allowed users to use the app without really permitting access to any permissions. However, this still did not solve the issue of data breaching as some apps started to crash when the user declined the requested permission.

As discussed, the Android OS platform works on permission mechanisms. Certain apps take many permissions. The intent and nature behind these permissions motivated us to study data trade, user privacy leaks, and current security policies in Android. After analyzing many of the applications, it is found that the application developer misuses permissions to steal the user's private and personal data. Android 6.0 (Marshmallow) Android has provided support to allow/decline permissions, i.e., users could decide whether an app requires specific permissions and could decline whatever permissions they find unnecessary. However, this often leads to apps being crashed on purpose by the developer if specific malicious permission(s) are rejected.

Due to a lack of protection in dangerous permissions associated with sensitive APIs, user privacy is exploited by malicious apps by manipulating users and application services. This situation motivated us to make a system that would improve the app's functioning while securing sensitive data. In this paper, a framework of data protection is proposed that will ensure data security and try to stop on-purpose crashes by making the app believe it has access to the requested data.

This paper addresses the malfunctioning of applications when permissions are denied and protects data privacy. Considering all these issues, service is enforced to identify the malicious permissions inside the installed applications on the user's device. The proposed framework provides services to predict the permissions required by an app and instructs the app to prevent malfunction dynamically. The instrumented Android Package (APK) file is installed on the target device. It communicates with the server service at runtime whenever a potentially dangerous API is triggered. The mobile client communicates over the insecure public channel (the Internet). The communicated messages can be read or modified over the network, or the mobile client's identity can be known to an adversary. Therefore, an anonymous authentication and key agreement scheme are also proposed to protect communication without revealing the client's identity. Thus, the proposed framework protects user data without affecting the functionality of the application.

The paper is organized into the following sections. Section II describes security measures that are available in Android. Sections III and IV describe the motivation and previous work done in this regard, respectively. The proposed framework and the communication scheme are discussed in section V. The application of the proposed model, through a case study, is discussed in section VI. Conclusions and future work are shown in section VII.

II. SECURITY IN ANDROID

A. ANDROID ARCHITECTURE

Android is a Linux-based open-source software program or rather, an Operating System (OS) [2]. The platform security is based on the OS's architecture, Fig. 1, which is achieved by separating resources and accessibility in subsequent layers.

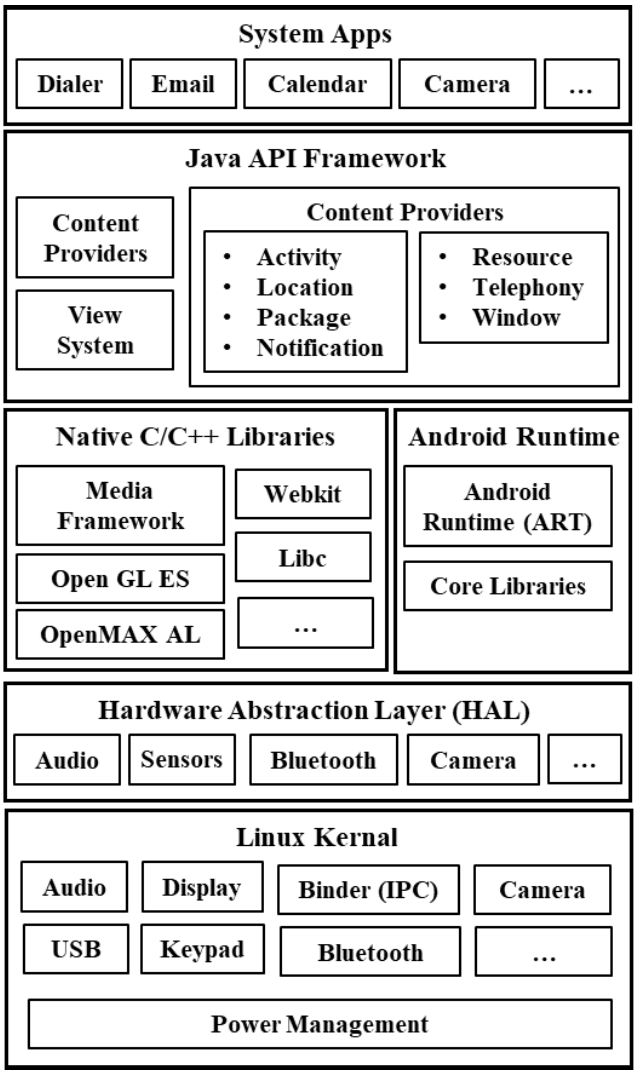

FIGURE 1. Android platform architecture.

Each layer assumes that the proceeding layer is secure. Subsequent layers become less accessible. The Linux kernel runs on the lowest level and is responsible for performing the basic OS operations such as process management, memory

management, and managing other device-related operations. Above the Linux, kernel layer runs the Hardware Abstraction Layer (HAL) which is responsible for providing a standard interface to hardware so that hardware vendors can build their hardware without affecting higher-level layers. Now, the OS can be updated without changing or re-configuring hardware implementations. It also provides an advantage in the HAL. It promotes the principle of least privileges, as HALs in a process do not have access to an identical set of permissions compared to the rest of the process [10].

HAL lies on the services layer of the Android system. The service layer consists of Android runtime libraries and native C libraries. These can be easily accessed by the application layer to perform various Android related functions such as accessing the camera module to record, media module to play something, notification service for sending notifications, and various other Android provided services.

The Java API layer is placed on the top of the service layer. It is an intermediary layer that makes it easy for the application layer to access Android services using Java-based APIs corresponding to them. The topmost layer is the application layer where all the applications installed on the device remain and make use of the Java API framework to access various required Android services.

The Android low-level security model is based on application sandboxing [4], [5]. Android sandboxing is the process of isolating an application in the system. It prevents outside influences on the layers mentioned in the architecture. All applications are assigned a user ID while they are running. They have access rights to their own files only. It prevents outside malware and security threats; if an application experiences a security breach, other applications' operations will not be affected.

Android provides hardware-backed key protection for cryptographic services. The stored keys provide a safe, secure channel for the authentication of user data. Verified Boot is used to check the state of the system when it starts [4]. It verifies whether the system is in a good state. In Android 8.0 (Oreo), Google introduced Project Treble to increase low-level security [4], [11]. Project Treble separates the open-source Android OS framework from the hardware code implementations at the vendor level. It has had a positive impact on device security and the speed of updates.

B. APPLICATION SECURITY

Android uses the permission model to prevent an app from using sensitive data and resources that are not required during runtime. Apps need corresponding permissions to use APIs to interact with the underlying system [12]–[14]. All permissions taken by an app must be specified in the application's manifest file [15].

Permissions are grouped into three categories corresponding to the risk and security level associated with resources and APIs: normal, dangerous, and signature permissions. Normal permissions include the permissions where the application must interact with resources out of the sandbox and do not pose any threat to user privacy. Normal permissions include Bluetooth, KILL_BACKGROUND _PROCESS, and Internet. Signature permissions are granted at the install time and allow an application to use the permissions signed by the identical certificate. VPN_SERVICE is included in the signature permissions. Dangerous permissions are the permissions that could pose a potential security threat or a threat to user privacy. The user is required to approve each of these permissions needed by the application after installation. SMS, storage, and camera permissions are some of the permissions included in this category.

In earlier versions (Until Android 5.0), users were not allowed to choose a subset of permissions. They needed to accept all the permissions mentioned by the application in their manifest file to install the software application into their devices. In Android 6.0 (Android Marshmallow), Google introduced a new mechanism for permission, called runtime permission [16]. Where users are notified of any dangerous permissions at runtime and choose not to give specific permission, it gives a choice to the user to understand the usage of the app and determine whether the requested permission is required for the proper functioning of the app. In some cases, if permissions are not given, the app may not work correctly. Ultimately, the user is forced to accept all the permissions to use the app.

Android's System Alert Window API was modified in Android 8.0. It does not allow apps to draw special windows used to notify the user of the critical messages. It has resulted in the prevention of clickjacking that was used by malicious apps creating overlays on the screen. Users are now allowed to tap the notification to hide overlays [16].

For protecting user phone data, Android provides strict policies for sensitive APIs. In Android 8.0 and above, the GET_ACCOUNTS permission is no longer sufficient to gain complete access to the list of accounts active on the device. For example, the user is now required to grant permission to the Gmail app to access the Google account on the device even though Google owns Gmail. As concrete examples, Settings.Secure.ANDROID_ID or SSAID is an ID provided to all apps. To prevent misuse of the ANDROID_ID value, Android 8.0 provides a mechanism that does not allow the change in ANDROID_ID when the application is re-installed until the package name and key are identical. Another feature, Build.getSerial() returns the actual serial number of the device till the caller holds the PHONE permission. Android 8.0 has deprecated this API's use, and it protects the serial number of the device from being misused by the applications.

Android has seen advancements in hardening security policies. However, it can be noted that as of October 2020, only 40.35% of devices are running Android 10.0, and 22.59% of devices are running Android 9.0 (Pie). More than 35 percent of users are using older versions of Android on their phones [17]. Due to a lack of knowledge in users and lack of security in sensitive APIs, users are often manipulated into using over-privileged applications.

III. RELATED WORK

With the growth of Android in the market, malicious applications have also surfaced, which has driven many studies and research works towards it. Iman and Aala [25] proposed a comprehensive analysis of Android permission systems. They provided important insight into the permission system evolution over the years and how permission usage has increased up to 73.33% in top applications by 2020. Sanz *et al.* [18] proposed a method to recognize malicious Android applications with the help of machine learning (ML) techniques, which extracts the Android permissions from the application. Permissions extracted from the Android Manifest file of an app were utilized to categorize an application as malware using the machine learning model for Android permissions. Karim *et al.* [15] suggested permissions of an Android app using the collaborative filtering method, associative rule mining, and Bayesian text mining. This approach tried to predict the permissions that must be used by the application after making an association with similar applications. This was developed to help developers know what type of permissions their app might require; it does not include the feedback from the end-users.

Mathur *et al.* [27] presented a malware detection framework for Android called NATICUSdroid, which investigated and classified benign and malware using statistically selected native and custom Android permissions as features for various ML classifiers. However, these approaches were limited to only mobile resources for its processing and classification. Furthermore, these approaches lacked learning abilities, dynamic processing, and they did nothing to stop these malicious activities in the application, which kept them from being overly successful. Azim and Neamtiu [19] used static dataflow analysis on the apps bytecode for systematic testing of Android apps, which as a result, constructed a high-level control flow graph among various activities inside the apps. They deduced a method of depth-first flow among these activities, which mimicked the user actions. This approach showed good potential and was the basis of dynamic analysis, though it was still unable to make it learn for itself. It used mobile processing power to do the analysis, which has some shortcomings.

Ricardo *et al.* [20] worked on a framework for Android apps, which instrumented the app with injections to keep track of any malicious activity an app performs. This approach used dynamic analysis and is the basis of the proposed work. However, this approach was also unable to use the previous results and did not learn from the apps. Shahriar *et al.* [21] proposed an approach to reduce the number of apps needed to be sandboxed to determine if they are malicious. They used Latent Semantic Indexing (LSI) to identify malware apps though this was limited to the identification of malware applications.

Sadeghi *et al.* [22] presented a Terminator framework which can provide an effective yet non-disruptive defense against permission-induced attacks by identifying the system's safe state and controlling the permission based on this. It provided access to the permissions and revoked identified unsafe permissions without modifying the app's implementation logic. Zhang *et al.* [23] presented VetDroid to analyze fine-grained causes of information leaks by capturing the app's sensitive behaviors with permission to use graphs. Security analysts were utilized to analyze the internal sensitive behaviors of the app by reconstructing these behaviors after they have been allowed dangerous permissions. Although, it has a lot of potential but lacks a way to inform and educate users about security threats and does nothing to protect it.

Wu *et al.* [24] proposed a system that achieved the robust and interpretable classification of Android malware. Their work demonstrated state-of-the-art obfuscation-resilient malware analysis which can work on obfuscated Android apps hiding their functionality. Mill *et al.* [26] proposed a way to classify both obfuscated and unobfuscated apps as malicious or benign. Qu *et al.* proposed Permizer, an automatic permission optimization to recommend app permission configuration to users. Permizer builds a mapping between permissions and functionalities for each app then regulates the relationship between permission and functionality based on the user's privacy preferences [28]. Xiao *et al.* proposed an approach to identify minimum required permissions for an android app. They used collaborative filtering to determine the initial minimum permissions of the app. Then, they find the actual requirements that the app really required for proper functioning using static analysis and evaluate the risk by inspecting extra permissions requested by the app, thereafter, generate a permission recommendation [29].

Gao *et al.* suggested an autonomous permission recommendation system, AutoPer+. It automatically recommends permission decisions to users at runtime. They proposed a deep semi-supervised machine to identify similar apps and explore the privacy permission usage in a cluster of apps that help in determining the correlation between permission and app, which is used in generating permission recommendations [30]. Li *et al.* built an automatic fuzzing tool, CUPERFUZZER+, to detect vulnerabilities related to custom permissions in existing Android OS and given general design guidelines to secure custom permissions [31].

From literature, it is found that the existing methods do very little to determine how to stop the applications from being malicious and still use it; almost all of them used mobile processing power for the framework and did not learn with time. One of the works in which permissions were being revoked after use, also does not prevent the application from using user data at runtime. This study addresses all these issues by providing a system that analyses an application's functionality permission by permission and prevents them from using user data potentially for malicious purposes. The results that are obtained from the server during analysis are processed and stored for all new applications.

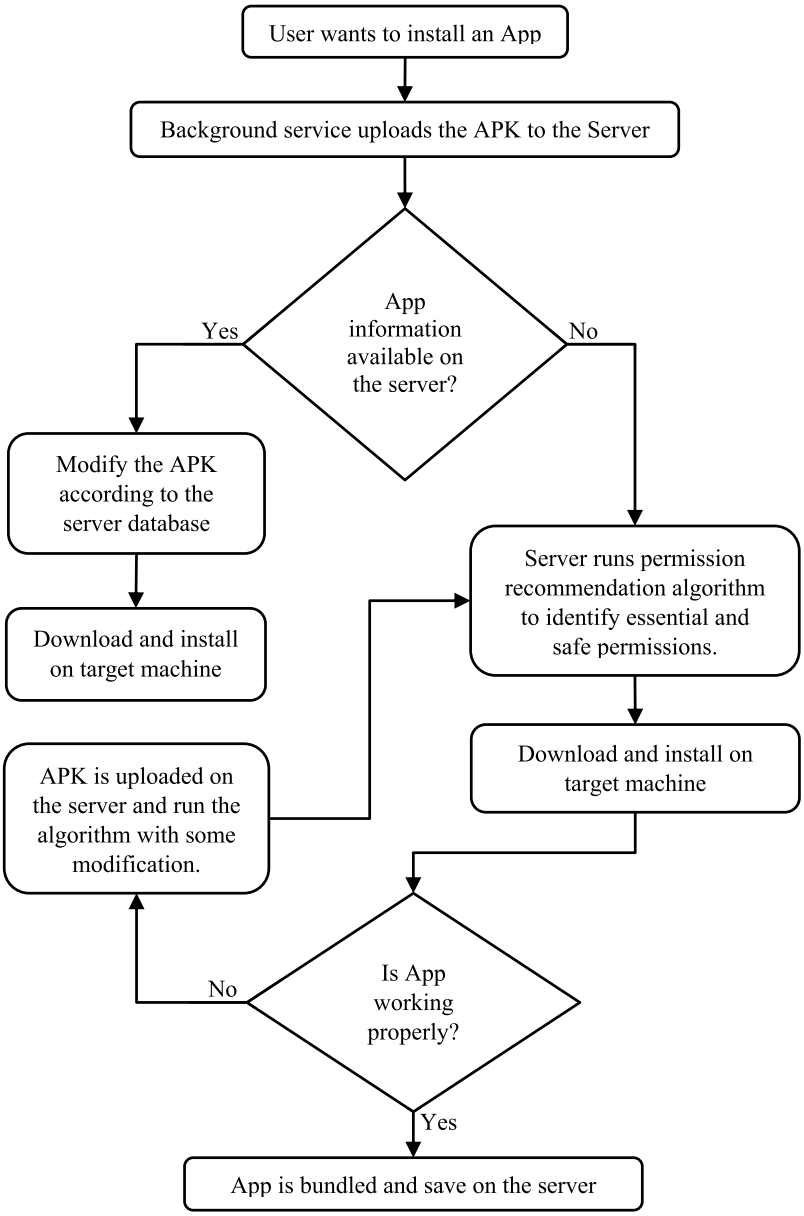

FIGURE 2. Flowchart of the proposed framework.

IV. PROPOSED APPROACH

We propose an end-to-end framework to ensure the proper functioning of the app along with user privacy protection. A background data protection service is installed on the user's phone to capture the dangerous API calls on the runtime. It returns garbage data to be sent back to the malicious app. Dedicated server analyze and instrument apps that the user is using for making it compatible with the proposed service. Table 1 shows various keywords used in the proposed model.

The proposed approach uses two algorithms to determine malicious permissions asked by the apps that needs some initial dataset to work on. The control flow of the proposed work is illustrated in Fig. 2. The following two components are presented in the proposed framework:

A. ANALYSIS AND INSTRUMENTATION OF THE APK

The analysis of the app is done using the resources and permissions the app requires. Analysis of permissions against the API calls and the utility of the application is done. The permission recommendations are used to predict whether the application is demanding extra permissions for stealing user data. For that purpose, the algorithms collaborative filtering and frequent permission set mining are used on permissions. The training set is based on the data collected for various applications in various categories (as described in the later section). The result formed by the intersection of the results from both the algorithms individually containing unsafe and extra permissions is sent to the instrumentation engine, which instruments the APK to support functionality to call the Data

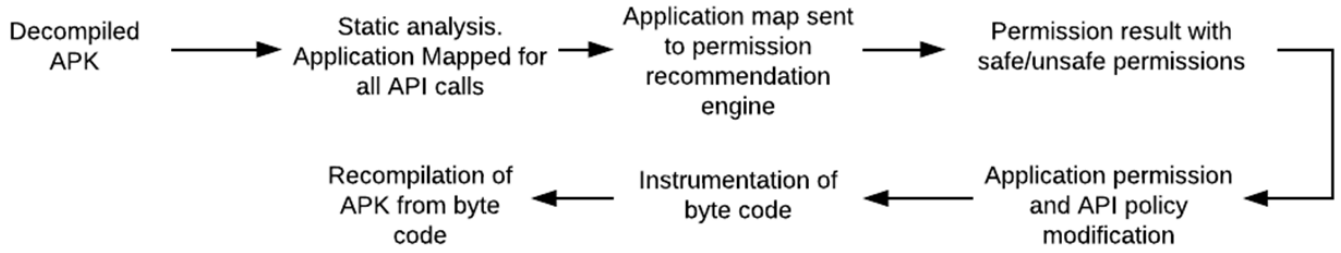

FIGURE 3. The flow of instrumentation engine.

TABLE 1. Keywords used in role specification in HLPSL.

Keywords	Description
agent	Data-type for agents
channel(dy)	Data-type for channels. *dy* means Dolev-Yao channels.
composition	Marks beginning of composition section of a composed role.
cons	Add an element to set.
def	Indicates beginning of body of a role.
delete	Delete an element from set.
end role	Indicates end of role.
exp	Exponentiation operator.
i	Intruder's identity.
in	Check if element is in list or set.
init	Indicates initialisation of local variables.
inv	Return inverse of a key: given a public key returns private key.
intruder_info	Defines knowledge of the intruder.
local	Indicates local variable section.
message	General type of message contents.
nat	Data-type for natural numbers.
not	Logical negation.
owns	Ownership of a variable: if a role owns a variable, only this role may change the value of the variable.
played_by	For basic roles: specifies which agent is playing this role.
public_key	Data-type for public keys.
request	Used to check strong authentication (together with witness).
secret	Used to check secrecy.
set	Data-type for unordered collection of typed values.
symmetric_key	Data-type for symmetric keys.
text	Data-type for uninterpreted bit-strings (like nonces).
transition	Marks beginning of transitions section of basic role.
witness	Used to check authentication (together with (w)request).
wrequest	Used to check weak authentication (together with witness).
xor	Prefix xor operator.

Protection Service. Both algorithms work by classifying the permissions of app based on the category that lies in in the play store, and the acquired dataset.

B. DATA PROTECTION SERVICE

To prevent user privacy, the proposed service runs in the background on the user's phone andsends the APK file of the installed app to the server for analysis and instrumentation. It facilitates the installation of the instrumented APK for the user. Finally, when the application starts running, it provides garbage data to the app whenever an identified malicious call to API is made. Garbage data is produced using the broadcast receiver. The Android framework has a facility to allow users to register for events using a broadcast receiver according to the lifetime i.e., statistically and dynamically. In the case of dynamic, the lifetime depends upon Context.registerReceiver() and Context.unregisterReceiver() on the app component. In the case of static, a receiver is specified in the AndroidManifest.xml and has an identical lifetime to the app. The receiver utilizes a callback approach i.e., BroadcastReceiver.onReceive(), to override SDK calls [27].

The following approaches on which the proposed framework works are described in detail:

1) DATA COLLECTION

The data collection approach is divided into two parts. The initial data collection is done by developing and circulating one data collection app. This app is downloaded by roughly 300 users through which information about 1000 unique applications is collected. Thereafter, the data of asked permissions and the permissions provided by the user are extracted, and the half probability rule is used to determine whether the permission is necessary or malicious. Afterward, whenever the algorithms run on the server, unique app data is added to the database, which would help to increase the dataset and help the proposed framework learn with time.

2) ANALYSIS AND INSTRUMENTATION

The engine on the server runs to analyze and instrument APKs (see Fig. 3). It begins with the decompilation of the app using Apktool which is used to reverse engineer Android apps. It decompiles app into *Smali* code, i.e., the assembly code that runs on the Dalvik Virtual Machine (Android's Java Virtual Machine). The decompiled code goes through the following stages in the parsing and instrumentation engine.

The application is then repackaged using Apktool. It is installed on the user's Android phone by the Data Protection Service.

a: STATIC ANALYSIS

The original *Smali* code is given as input to the proposed engine. Static analysis is performed by parsing *Smali* code files. A map of app is created, which shows the file name and the methods included in the file, class names, and API calls. A clear map is created for reference in instrumentation in the later stages. The manifest file is parsed separately. All permissions are extracted from the file, and dangerous permissions from the superset are kept for analysis. A python-based parser handles the manifest file and *Smali* code; it traverses the directory of the decompiled APK and maintains a record of class names, method names, and API Calls for each file. A map is created for all method calls. During enforcement of redefined permission policies, the map is used to locate the functions and files to be instrumented.

b: PERMISSION ANALYSIS

As discussed earlier, two approaches were used to identify the permissions required by an app: collaborative filtering and frequent permission set mining. The training set consists of apps of all categories as available in the Google play store.

COLLABORATIVE FILTERING

Collaborative filtering is one of the commonly used techniques in recommender systems. It utilizes the information contained in a group to recommend information on a new entity related to the group. It is based on the idea that entities that share certain evaluation criteria of certain items in the past are likely to agree again in the future. Feature vectors are used in this method to represent the items in the entity that goes through the evaluation process of finding a similarity score.

(i) Finding the feature vector in the proposed engine

The app permissions are used as feature vectors for the collaborative filtering engine. Permissions of an app are extracted in a vector as $V =< P_1, P_2, \ldots, P_n >$, where P_i can take values from the set [0, 1] depending upon whether the app takes that permission. Feature vectors from the apps in the training data set are taken. The engine first extracts the apps in the same category as that of the test app. It then extracts all the permissions in the feature vectors for filtering and recommendation. $A_i = \{AP_1, AP_2, \ldots, AP_n\}$, where AP_i takes the value from the set [0, 1].

(ii) Evaluation of similarity

A similarity score is a measure of how closely related two entities are. The similarity is calculated for the app with all the apps in the same category using the Jaccard similarity score as

$$S(A_i, A_t) = \frac{F_{11}}{(F_{01} + F_{10} + F_{11})} \tag{1}$$

A_i is the app from the data set in the same category. A_t is the test app. F_{11} is the frequency of matches in the permissions between A_i and A_t. F_{01} is the frequency of permissions 1 in the case of A_i and 0 in the case of A_t. F_{10} is the frequency of permissions 1 in the case of A_t and 0 in the case of A_i.

(iii) Recommendation of Permissions

A recommendation score is generated for each permission request by the app A_t. It can be calculated using

$$RScoreCF\ (P_i) = \sum S\ (A_i, A_t) \tag{2}$$

Here, majority voting is considered, with the voting's weight proportional to the similarity score generated above. The generated *RScore* is normalized. Depending on the score, the permission is marked as safe if recommended for the app to be used; otherwise, it is marked unsafe. The step-by-step flow of the permission segregator is presented in Algo. 1.

Algorithm 1 Permission Segregator

```
JaccardSim ()
for i in range(len(app)):do
    check the vector scores 11, 10 and 01
            if "11" then
            → num++, denum++
            elif "01" or "10" t hen
            → denum++
    return num / denum
FindSupportForPermissions () {
→ every asked permission is analyzed based on category present a score is calculated and provided using Jaccard similarity for permission set
return permissionScore[ ] }
perms[ ] = FindSupportForPermissions ( )
for i in range (len (perms)):do
    if (perms [i] > threshold) then
    self.safePermissions.append (self.permsAt[i])
    else
    self.unsafePermissions.append (self.permsAt[i])
```

FREQUENT PERMISSION SET MINING

The second recommendation algorithm is based on predicting permission pair values that occur together. Relationships and patterns of the permissions requested simultaneously are studied. Based on the relationship that two permissions share, permissions are recommended for an application. Preliminary support calculation in predicting an event is based on frequency. Support is used to discover relationships among entities. Suppose an event (event B) taken from a dataset of N events occurs f times (frequency of event B). The support of event B is

$$Sup\ (B) = \frac{Frequency\ (B)}{N} \tag{3}$$

The permissions of the proposed test app A_t are extracted in $V =< P_1, P_2, \ldots, P_n >$ where P_i can take values from the set [0, 1] depending upon whether the app takes the permission. Vectors of training data apps containing their permissions as $A_i = \{AP_1, AP_2, \ldots, AP_n\}$, where AP_i takes the values from the set [0, 1].

(i) Training of the Proposed Model

Applications belonging to the same category follow some pattern of frequently co-occurring permissions as:

$$A_i = < P_1, P_3, P_5, P_6 > \quad (4)$$
$$A_{i+1} = < P_1, P_2, P_4, P_3, P_7 > \quad (5)$$
$$A_{i+2} = < P_1, P_3, P_8, P_{10} > \quad (6)$$
$$A_{i+3} = < P_5, P_4, P_7 > \quad (7)$$

Taking in pairs $< P_i, P_j >$, the support of the co-occurring permission pair is calculated as:

$$Sup\left(< P_i, P_j >\right) = \frac{Freq\left(< P_i, P_j >\right)}{N} \quad (8)$$

where N is the total number of applications in the category $Freq\left(< P_i, P_j >\right)$ is the frequency of the pair $< P_i, P_j >$ when it is requested together.

(ii) Recommendation of permissions

The support calculated for all pairs is analyzed.

$$if\ Sup\left(< P_i, P_j >\right) > t\ then$$
$$Recommend\left(< P_i, P_j >\right) = 1 \quad (9)$$

Here, t defines the threshold value. Permission pairs having a support value higher than threshold are marked safe and recommended. Rest is marked unsafe $Recommend\left(< P_i, P_j >\right) = 0$. After calculating safe permissions from each recommender, the intersection of the resulting permission sets is the final permission set that will be used for further processing. Algo. 2 presents the various steps of the permission miner.

C. INSTRUMENTATION

In this study, our primary focus is on dangerous permissions. The permissions suggested by the permission recommendation engine are fed into the instrumentation engine. The policies suggested by the permission recommendation engine are marked safe. The instrumentation engine modifies the policies marked as unsafe. *Smali* code is instrumented to facilitate communication with the background service at runtime. Hence, through instrumentation, the communication between the malicious detected app and the background service through broadcast receivers is enabled. Background processes are utilized as services on Android. These processes do not provide graphical components and are implemented for background activities for a given program. All services utilized by an app must be added in the manifest [23]. Permissions that are marked unsafe are injected with the piece of code invoked by the required services using broadcast receivers. All the unsafe policies are instrumented within the app then repackaged using Apktool.

1) DATA PROTECTION SERVICE

The Data Protection Service is installed on the user's phone andserves two purposes:

Algorithm 2 Permission Miner

1. Permissions are numbered from 0 to $n-1$.
2. Data is then read to identify patterns.
3. for $i = 0$ to $n-1$ do
4. Identify and store the dangerous permissions
5. let there be k permissions
6. for $i = 1$ to k do
7. for $j = i+1$ to k do
8. Support between permissions i and j is calculated and stored {
   Support is calculated using:
   for i, row in Rows do
   if (values in two col same) then
   count ++
   }
9. Max, Min, and Avg support are calculated for the k permission set.
   for i in 1 to k do
   Max = Max > support[i] ? Max: support[i]
   Min = Min < support[i] ? Min: support[i]
   sum = sum + support[i]
   Avg = sum / k
10. for key, value in the stored list do
11. if (key > Avg) then
12. Permission required
13. else
14. Permission not required

a: INTERFACE FOR INSTALLATION FOR AN INSTRUMENTED APP

It is the job of the service installed on the device to communicate with the server using a secure communication channel when the user asks the service app to secure the malicious app. The background service uploads the APK file of the app to be sent to the server for analysis and instrumentation. After the instrumentation is completed, the server sends the instrumented APK file back to the service using the same secure channel. The Data Protection Service receives the instrumented app to be installed back to the user's phone. On receiving app, the background service prompts the user to ''uninstall and install'' i.e., uninstall the previous build and install the new modified build. The service runs a check on that app for its working. If the app is found to work without issue, the modified APK configuration is approved and sent to the server for future use. Whenever an app is sent to the server for analysis, the server checks the database for pre-existing records of the corresponding app. If found, it instruments the APK file using pre-processed values else, the algorithm determines the required permission set, and instrumentation is done accordingly.

b: BACKGROUND SERVICE

Instrumented apps that get installed on the phone are now allowed to communicate to the pre-installed background service. Garbage values are returned to the app when the call

to unsafe policies is made. The call to the API triggers an intent to the background service running on the user's phone. The service returns a garbage value, and hence the user's privacy is protected. Thus, the data received by the proposed app is carefully monitored, and the broadcast receiver sends the garbage data corresponding to that permission to the malicious app. The data received by the app is treated as real data, while it is garbage. Hence, the proposed framework does not hamper the proper functioning of the app, solving the problem of malfunction of the apps if the user declines some unwanted permissions.

2) ANONYMOUS AUTHENTICATION AND KEY AGREEMENT SCHEME

To provide secure communication between mobile clients and the server, an anonymous authentication and key agreement scheme is also proposed. The notations used in this section are presented in Table 2.

TABLE 2. Notations used in this section.

Notation	Explanation
C_i	i^{th} Client
S	Server
EID_i	e-mail id of the i^{th} client
ID_i	Identity of i^{th} client
AID_i	Anonymous Identity of i^{th} client
s_m	The private key of the server
P_{pub}	The public key of the server
$E_k(m)/D_k(m)$	Encryption/Decryption of m using k
$h_1(.), h_2(.), h_3(.)$	Secure one-way hash functions
SK	Session key
$\oplus$	Bitwise XOR
$\|\|$	Concatenation

a: PROPOSED SCHEME

In the proposed scheme, each mobile is treated as a client. The proposed scheme provides client authentication without revealing the real identity of the client and session key agreement for secure communication between client and server. The scheme consists of four phases: system setup phase, client registration phase, authentication phase, and client's secret parameter updating phase. The working of the scheme is as follows:

SYSTEM SETUP PHASE

This phase is to generate initial parameters for the client registration and authentication phase of the scheme. The system generates initial parameters as follows

(i) Choose two large prime numbers p and q, and an elliptic curve E over a prime field F_p ($y^2 = x^3 + ax + b \; mod \; p$, where $a, b \in F_p$, and $4a^3 + 27b \neq 0$). Define O at infinity, P is the generator point of E with order q (where $P \neq O$).

(ii) The server S chooses a random number s_m as the private key and computes $P_{pub} = s_m.P$, and selects three one-way hash functions h_1, h_2, and h_3.

(iii) Server S publishes as

$$\left\{E_q \backslash F_p, P, P_{pub}, h_1(.), h_2(.), h_3(.)\right\}. \quad (10)$$

CLIENT REGISTRATION PHASE

When a client C_i wants to register with server S, client and server perform the following steps:

(i) The client C_i generates a random number u_i and computes the ID_i using the user's email id EID_i and u_i, $ID_i = h_1(EID_i \| u_i)$. C_i Encrypts ID_i with the server's public key P_{pub}, $C_1 = E_{P_{pub}}(ID_i \| T_1)$, where T_1 is a timestamp. C_i sends $< C_1 >$ through a public channel. While ID_i *and* T_1 are encrypted with the server's public key, only the server's private key can decrypt C_1.

(ii) On receiving a registration request $< C_1 >$, the server S decrypts the message to read the identity of the client C_i and timestamp, $(ID_i, T_1) = D_{s_m}(C_1)$. Server S checks the freshness of the timestamp. If it is not fresh, S drops the registration process; otherwise, S checks for a collision in the verifier table. If the collision happens, the server S informs the client C_i to restart the registration process; else, the server S chooses a random number v_i and computes the client's secret key $K_i = ((v_i)/(s_m.ID_i')).P$, client's anonymous identity $AID_i = ID_i \oplus h_2(v_i \| s_m)$, and symmetric key $KT = h_2(ID_i \| T_1)$ to communicate K_i and AID_i securely. After that, server S encrypts K_i *and* AID_i using KT, $C_2 = E_{KT}(K_i \| AID_i \| T_2)$ and sent $< C_2 >$ to client C_i over a public channel. Server S stores $\{ID_i, v_i, AID_i\}$ in a table.

(iii) After receiving C_2 from the server S, the client C_i computes $KT = h_2(ID_i \| T_1)$ and decrypts the message $(K_i \| AID_i \| T_2) = D_{KT}(C_2)$. Client C_i checks the freshness of T_2, C_i will abort the current registration attempt and start the registration process from the start if T_2 fails the freshness test; else, C_i stores $\{ID_i, K_i, AID_i\}$ into device memory.

AUTHENTICATION PHASE

In this phase, mutual authentication shall be accomplished between client C_i and server S, and a session key will be generated. To achieve this, server and client perform the following steps. The details are illustrated in Fig. 4.

(i) The client C_i chooses a random number r_i and computes $R_i = r_i.P$, $K' = r_i.K_i$, $M_1 = h_3(ID_i \| R_i \| K' \| T_3)$ and send $< AID_i, R_i, M_1, T_3 >$ to the server S.

(ii) On receiving the message, server S checks the freshness of T_3. If T_3 fails the freshness test, the session is terminated by the S; otherwise, the server searches the verifier table for AID_i. If AID_i is not in the verifier table, S sends an error message and terminates the session; else, server

Client (C_i)	Server (S)
Generate a random number r_i and computes $R_i = r_i.P$ $K^{'} = r_i.K_i$ $M_1 = h_3(ID_i \| R_i \| K^{'} \| T_3)$ $< AID_i, R_i, M_1, T_3 >$ $\longrightarrow$	
	Check $T_3^* - T_3 \leq \Delta T$ Check if $M_1 \stackrel{?}{=} h_3(ID_i \| R_i \| \left(\frac{v_i}{s_m.ID_i}\right).R_i \| T_3)$ Generate a random number r_s and computes $R_s = r_s.P$ $R_{sc} = r_s.R_i (= r_s.r_i.P)$ $AID_{inew} = AID_i \oplus h_3(R_{sc} \| ID_i)$ Check for collision in verifier table $SK = h_3\left(ID_i \| R_{sc} \| \left(\frac{v_i}{s_m.ID_i}\right).P \| T_3 \| T_4\right)$ $M_2 = h_3(ID_i \| R_s \| T_4 \| SK)$ $< AID_i, R_s, M_2, T_4 >$ $\longleftarrow$
Check $T_4^* - T_4 \leq \Delta T$ $R_{cs} = r_i.R_s (= r_i.r_s.P = R_{sc})$ $SK = h_3(ID_i \| R_{cs} \| K_i \| T_3 \| T_4)$ Check $M_2 \stackrel{?}{=} h_3(ID_i \| R_s \| T_4 \| SK)$ $AID_{inew} = AID_i \oplus h_3(R_{cs} \| ID_i)$ Replace AID_i with AID_{inew}	Update AID_i with AID_{inew} after receiving the first message encrypted with SK.

FIGURE 4. The authentication phase.

S checks if $M_1 = ?h_3(ID_i \| R_i \| ((v_i)/(s_m.ID_i)).R_i \| T_3)$. If M_1 fails, the session is terminated by the S; otherwise, the server S chooses a random number r_s and computes $R_s = r_s.P$, $R_{sc} = r_s.R_i (= r_s.r_i.P)$, $AID_{inew} = AID_i \oplus h_3(R_{sc} \| ID_i)$ and check for collisions in the verifier table. If the collision happened, S chooses new r_s and computes again. Now, the server S computes session key $SK = h_3(ID_i \| R_{sc} \| ((v_i)/(s_m.ID_i)).P \| T_3 \| T_4)$ and $M_2 = h_3(ID_i \| R_s \| T_4 \| SK)$ and sends $< AID_i, R_s, M_2, T_4 >$ to the client C_i through a public channel. Session key agreement suggests that the C_i wants to communicate securely with server S, subsequent session work as an acknowledgment, and server S updates the old AID_i with AID_{inew} in the verifier table. If server S does not receive any message encrypted with SK from C_i after mutual authentication and key agreement, the server S will know the client C_i may have lost the message and stops AID_i.

(iii) After receiving the message $< AID_i, R_s, M_2, T_4 >$ from server S, client C_i checks the freshness of T_4. If not, C_i drops the session; else, C_i computes $R_{cs} = r_i.R_s (= r_i.r_s.P = R_{sc})$, session key $SK = h_3(ID_i \| R_{cs} \| K_i \| T_3 \| T_4)$ and checks if $M_2 =^? h_3(ID_i \| R_s \| T_4 \| SK)$. If not, the C_i terminates the session; otherwise, the client C_i accepts SK and computes the anonymous identity $AID_{inew} = AID_i \oplus h_3(R_{cs} \| ID_i)$ and replaces the old AID_i with AID_{inew}.

CLIENT'S SECRET PARAMETER UPDATING PHASE

After authentication and agreeing on the session key, the client C_i sends an update request to the server S, encrypted using session key SK. On receiving an update request, S generates v_{inew} for C_i and computes $K_{inew} = \left(\frac{v_{inew}}{s_m.ID_i}\right).P$ and $AID_{inew} = ID_i \oplus h2(v_{inew} \| s_m)$. After that, S sends K_{inew} and AID_{inew} with a timestamp to C_i encrypted using session key SK. Client C_i updates the K_i and AID_i with received parameters K_{inew} and AID_{inew} and send acknowledgement to the server S. After receiving an acknowledgment from C_i, S updates v_i with v_{inew} and AID_i with AID_{inew} in the verifier table.

b: SECURITY ANALYSIS

This section provides formal security verification using AVISPA and informal security analysis to prove that this scheme provides mutual authentication, client anonymity, session key agreement, and the scheme is secure against known attacks.

FORMAL SECURITY VERIFICATION USING AVISPA

In the scheme execution, Client Ci receives the start signal and sends the identity IDi with timestamp $T1$ encrypted with the server's public key as the registration request. Afterward, client Ci receives the security parameter Ki and anonymous identity $AIDi$ with timestamp $T2$ encrypted with symmetric

```
role client (C,S:agent,
Ppub:public_key,
H1,H2,H3,MUL:hash_func,
Snd,Rcv: channel (dy) )
played_by C
def=
local State : nat,
EIDi,IDi,Ki,AIDi,AIDinew,RPi,K,
RPs,Rsc,SK,P,T1,T2,T3,T4,Ui,Ri,
Rs,M1,M2:text,
KT:symmetric_key
const ctos,stoc,sec1,sec2,sec3,
sec4,sec5,sec6,sec7,sec8:protocol_id
init State:=0
transition
1.State=0/\Rcv(start)=|>
State':=1/\Ui':=new()
/\IDi':=H1(EIDi.Ui')
/\T1':=new()
/\KT':=H2(IDi'.T1')
/\Snd({IDi.T1}_Ppub)
/\secret({Ui},sec1,{C})
/\secret({IDi},sec2,{C,S})
2.State=1/\Rcv({Ki.AIDi.T2}_KT)=|>
State':=2/\Ri':=new()
/\RPi':=MUL(Ri'.P)
/\K':=MUL(Ri'.Ki)
/\T3':=new()
/\M1':=H3(IDi.RPi'.K'.T3)
/\Snd(AIDi,RPi',M1',T3')
/\secret({Ri'},sec3,{C})
/\secret({Ki},sec4,{C,S})
/\witness(C,S,ctos,Ri')
3.State=2/\Rcv(AIDi,RPs,M2,T4)=|>
State':=3/\Rsc':=MUL(Ri.RPs)
/\SK':=H3(IDi.Rsc'.Ki.T3.T4)
/\M2':=H3(IDi.RPs.T4.SK')
/\AIDinew':=xor(AIDi,H3(Rsc'.IDi))
/\request(C,S,stoc,Rs)
end role
```

FIGURE 5. Role specification of client *Ci* in HLPSL.

key KT from server S, and store IDi, Ki, and $AIDi$ into memory. Server S stores IDi, vi, and $AIDi$ in the verifier table.

During the authentication phase, client Ci sends $< AIDi, Ri, M1, T3 >$ to the server S. On receiving the message from client Ci, the server S computes session key SK and new anonymous identity $AIDinew$ for Ci and sends a message $< AIDi, Rs, M2, T4 >$. Afterward, Ci computes session key SK using IDi, Rsc, Ki, $T3$, and $T4$ where Rsc is a session-specific shared secret between Ci and S. Role of Ci and S are given in Figs. 5 and 6, respectively.

The constants $sec1$, $sec2$, $sec3$, $sec4$, $sec5$, $sec6$, $sec7$, $sec8$, $ctos$, and $stoc$ are used to identify the goal of secrecy and authentication in the goal section (see Fig. 9). The communication channel (dy) used in the implementation of this scheme belongs to the Dolev-Yao threat model in which intruders (i) can intercept, analyze, reroute, and modify the message. The HLPSL code has been simulated using the SPAN (Security Protocol ANimator) to examine the results. HLPSL code of environment and session is given in Figs. 7 and 8, respectively. The simulation results are shown in Fig. 10 and Fig. 11 for the OFMC and CL-AtSe model. Results show that the proposed scheme is safe.

INFORMAL SECURITY ANALYSIS

(i) Mutual authentication: In this scheme, the server S authenticates the client C_i by checking M_1. If M_1 is valid, S authenticates C_i. On the other hand, C_i verifies the legitimacy of S by checking M_2. If M_2 is valid,

```
role server (C, S: agent,
Ppub: public_key ,
H1,H2,H3,MUL: hash_func,
Snd, Rcv: channel (dy) )
played_by S
def=
local State : nat,
IDi,Vi,Ki,AIDi,AIDinew,Ri,Rs,
RPi,RPs,Rsc,SK,P,T1,T2,T3,T4,
M1,M2:text,
KT:symmetric_key
const ctos,stoc,sec1,sec2,sec3,
sec4,sec5,sec6,sec7,sec8:protocol_id
init State:=0
transition
1.State=0/\Rcv({IDi.T1}_Ppub) =|>
State':=1/\Vi':=new()
/\Ki':=MUL((Vi'.inv(Ppub).IDi).P)
/\AIDi':=xor(IDi,H2(Vi'.inv(Ppub)))
/\KT':=H2(IDi.T1)
/\T2':=new()
/\Snd({Ki'.AIDi'.T2'}_KT')
/\secret({IDi,Ki'}, sec5, {S,C})
/\secret({Vi'}, sec6, {S})
2.State=1/\Rcv(AIDi,RPi,M1,T3)=|>
State':=2/\Rs':=new()
/\T4':=new()
/\RPs':=MUL(Rs'.P)
/\Rsc':=MUL(Rs'.RPi)
/\AIDinew':=xor(AIDi,H3(Rsc'.IDi))
/\SK':=H3(IDi.Rsc'.Ki.T3.T4')
/\M2':=H3(IDi.Rs'.Rsc'.T4'.SK')
/\Snd(AIDi,Rs'.M2'.T4')
/\secret({Rs'},sec7,{S})
/\secret({Rsc'},sec8,{S,C})
/\witness(S,C,stoc,Rs')
/\request(S,C,ctos,Ri)
end role
```

FIGURE 6. Role specification of server *S* in HLPSL.

```
role environment()
def=
const c,s,i:agent,
ppub:public_key,
h1,h2,h3,mul:hash_func,
p,ipub,aidi,rpi,rps,m1,
m2,t3,t4:text
%Represents Intruder knowledge
intruder_knowledge={c,s,p,h1,h2,
h3,mul,ppub,ipub,inv(ipub),aidi,
rpi,rps,m1,m2,t3,t4}
composition
session(c,s,ppub,h1,h2,h3,mul)
/\session(i,s,ppub,h1,h2,h3,mul)
/\session(c,i,ppub,h1,h2,h3,mul)
end role
```

FIGURE 7. Role specification of the environment in HLPSL.

```
role session( C,S:agent,
Ppub:public_key,
H1,H2,H3,MUL:hash_func)
def=
local S1,R1,S2,R2:channel(dy)
composition
client(C,S,Ppub,H1,H2,H3,MUL,S1,R1)
/\server(C,S,Ppub,H1,H2,H3,MUL,S2,R2)
end role
```

FIGURE 8. Role specification of the session in HLPSL.

```
goal
%Verifies secrecy of the
%confidential information
secrecy_of sec1
secrecy_of sec2
secrecy_of sec3
secrecy_of sec4
secrecy_of sec5
secrecy_of sec6
secrecy_of sec7
secrecy_of sec8
%Verifies authenticity
authentication_on ctos
authentication_on stoc
end goal
```

FIGURE 9. Role specification of goal in HLPSL.

```
% OFMC
% Version of 2006/02/13
SUMMARY
 SAFE
DETAILS
 BOUNDED_NUMBER_OF_SESSIONS
PROTOCOL
 /home/span/span/testsuite/results/Android_Protection_scheme.if
GOAL
 as_specified
BACKEND
 OFMC
COMMENTS
STATISTICS
 parseTime: 0.00s
 searchTime: 0.02s
 visitedNodes: 12 nodes
 depth: 4 plies
```

FIGURE 10. Simulation result for the OFMC back-end.

C_i authenticates S. Thus, the proposed scheme attains mutual authentication.

(ii) Client anonymity and privacy: In the proposed scheme, only identity-related information used in communication is anonymous identity AID_i. Initially, it is computed using the client's identity ID_i, servers master key s_m and client-specific random number v_i, secured using a one-way hash function h_2. s_m and v_i are only known to S, and ID_i is not used in unsecured communication. So, an attacker cannot relate any communication to the client id. Anonymous id AID_i is updated in every session using shared computed session value R_{sc} and the client's identity ID_i. Where R_{sc} is computed using two random values r_i and r_s ($R_{sc} = r_i.r_s.P$), which guarantees the randomness of anonymity so it is computationally hard to distinguish whether the messages belong to the same client.

(iii) Resilience to replay attack: An attacker can capture an authentication message communicated through a public channel, then replay it as a new request, so it is necessary to check the validity of the message. In the proposed scheme, timestamps T_3 and T_4 are attached to the messages and used in the computation of M_1 and M_2. The server S will check for the validity of timestamp if valid then check the validity of M_1. If both are valid, the server S will authenticate the client C_i. The attacker cannot change M_1 without knowledge of ID_i. As a result, the attacker will not be able to launch a successful replay attack.

```
SUMMARY
 SAFE

DETAILS
 BOUNDED_NUMBER_OF_SESSIONS
 TYPED_MODEL

PROTOCOL
 /home/span/span/testsuite/results/Android_Protection_scheme.if

GOAL
 As Specified

BACKEND
 CL-AtSe

STATISTICS

 Analysed  : 0 states
 Reachable  : 0 states
 Translation: 0.01 seconds
 Computation: 0.00 seconds
```

FIGURE 11. Simulation result for the CL-AtSe back-end.

(iv) Malicious insider attack: if an attacker is also a registered client, they know their secret information ID_e, K_e and AID_e where $AID_e = ID_e \oplus h_2(v_e \| s_m)$ and $K_e = ((v_e)/(s_m.ID_e)).P$ as well as messages communicated between server S and other clients on the unsecured channel, e.g., $< AID_i, R_i, M_1, T_3 >$ and $< AID_i, R_s, M_2, T_4 >$ where AID_i is an anonymous identity of the client, $R_i = r_i.P$, $M_1 = h3(ID_i \| R_i \| K' \| T_3)$, $K' = r_i.K_i$, $R_s = r_s.P$, $M_2 = h3(ID_i \| R_s \| T_4 \| SK)$ and $SK = h_3(ID_i \| R_{cs} \| ((v_i)/(s_m.ID_i)).P \| T_3 \| T_4)$. It is computationally hard to extract s_m from k_e and D_e, r_i from R_i, and r_s from R_s. ID_i, K', and SK are secured with a one-way hash function. So, using these parameters, the attacker will not be able to extract any secret parameters of the server or other clients.

(v) Impersonation attack: To impersonate as the client C_i, the attacker needs ID_i, K_i, and current AID_i. These parameters are secured on the client device or can be computed using the server's master key and client-specific information stored on the server. Let us assume that the attacker knows the real and anonymous identity of the client C_i. To generate the request message, it can compute $R_{ie} = r_e.P$, but computing M_1 required the client's secret parameter K_i. To impersonate server S, an attacker needs the server's master key s_m, as it does not have s_m and client's identity ID_i. Hence, the attacker cannot impersonate a client or server.

V. RESULTS AND CASE STUDY

In this study, the behavior of nine permissions is analyzed. These permissions (listed in Table 3) are the permissions

TABLE 3. Permission set for study.

Permission Name	Permission ID
AP.READ_CALENDAR	0
AP.READ_CONTACTS	1
AP.GET_ACCOUNTS	2
AP.ACC_FINE_LOC	3
AP.ACC_COARSE_LOC	4
AP.READ_PHONE_STATE	5
AP.RECEIVE_SMS	6
AP.READ_SMS	7
AP.READ_EXTERNAL_STORAGE	8

```
All Permissions:
android.permission.CAMERA
android.permission.FLASHLIGHT
android.permission.STATUS_BAR
android.permission.WAKE_LOCK
android.permission.ACCESS_COARSE_LOCATION
android.permission.ACCESS_FINE_LOCATION
android.permission.ACCESS_NETWORK_STATE
android.permission.INTERNET
android.permission.READ_PHONE_STATE
```

FIGURE 12. Snapshot of a part of all permissions of brightest flashlight obtained from the proposed framework.

```
Dangerous Permissions:
android.permission.CAMERA
android.permission.ACCESS_FINE_LOCATION
android.permission.ACCESS_COARSE_LOCATION
android.permission.READ_PHONE_STATE
android.permission.WRITE_EXTERNAL_STORAGE
```

FIGURE 13. Potentially dangerous permissions of brightest flashlight identified in static analysis obtained from the proposed framework.

that 'read' user data on the phone. The proposed framework begins the analysis of the manifest file and Smali code concerning these permissions. To check the working of the proposed framework, the three apps such as Brightest Flashlight (golden-shores-technologies. Brightest-flashlight.free), Peacock Flashlight (com.peacock. flashlight), and Flashlight (com. spend apps. torch) were studied.

Each of the apps required a different set of permissions to run. The Brightest Flashlight app asked for various permissions, the peacock flashlight asked for location (loc) and storage permissions, and the spend apps flashlight asked for none. To get a minimum set of permissions and instrumentation, these apps were sent to the proposed framework. The apps were decompiled and Smali code was generated for each APK. Here, AP and ACC represent Android.permission and access, respectively.

```
--------------------------------------------------
Class Name: ['Lcom/flurry/sdk/in$a']
Method Name: constructor <init>
 Ljava/lang/Object;-><init>()
 Lcom/flurry/sdk/ij$a;-><init>()
 Lcom/flurry/sdk/la;-><init>(Lcom/flurry/sdk/lb;)
--------------------------------------------------
Method Name: a
 Lcom/flurry/sdk/in$a$2;-><init>(Lcom/flurry/sdk/in$a;Ljava
 /io/InputStream;)
 Ljava/io/DataInputStream;->readLong()
 Ljava/io/DataInputStream;->readLong()
 Ljava/io/DataInputStream;->readLong()
 Ljava/io/DataInputStream;->readInt()
 Lcom/flurry/sdk/ir;->a(I)
```

FIGURE 14. Class and corresponding method in brightest flashlight APK shown in the map obtained from the Smali code parser.

A. STATIC ANALYSIS

The first step in the process is static analysis; it yields the permissions declared in the AndroidManifest.xml and classes and their respective methods from the Smali code using Smali parsers of the engine. A list of all permissions that are processed by our engine that is required by Brightest flashlight is shown in Fig. 12, and all the dangerous permissions are shown in Fig. 13. As discussed in Section V in analysis and instrumentation, a map is obtained from the Python parser which shows method traces and data flow. The class names and method calls of these dangerous permissions of Brightest Flashlight APK are shown in Fig. 14.

The dangerous permissions declared in the Brightest Flashlight app are AP.CAMERA, AP.ACC_FINE_LOC, AP.ACC_COARSE_LOC, AP.READ_PHONE_STATE, and AP.WRITE_EXTERNAL_STORAGE, as shown in Figure 13. Similar code analysis on the two apps shows that the dangerous permissions in the Peacock Flashlight are AP.CAMERA, AP.WRITE_EXTERNAL_STORAGE, AP.ACC_FINE_LOC, and AP.ACC_COARSE_LOC. Meanwhile, the Splendapps flashlight takes AP.CAMERA permission only which is justified as per the requirement.

B. PERMISSION ANALYSIS

The permissions parsed as described in the previous subsection are provided as input to permission recommendation algorithms. The algorithms evaluate each permission and yield result vectors. Each result vector contains 9 elements which can be 0 or 1. Each value in the vector corresponds to permission as given in Table 2. If the permission is marked safe to use and is required by the application, the corresponding value is 1 else the value is 0.

Running the permission recommender for Brightest Flashlight: for collaborative filtering, using a threshold value as 0.1. The resultant vector can be obtained as:

$$r_p = [0, 0, 0, 0, 0, 0, 0, 0, 0] \tag{11}$$

Here, r_p shows the *resultPermissions*. The *RScoreCF* for each permission was found below the threshold value. It signifies that this app required none of the permissions and all the three permissions 'AP.ACC_FINE_LOC',

'AP.ACC_COARSE_LOC', and 'AP.READ_PHONE_STATE' are classified as unsafe.

From frequent permission set mining, each permission's support was computed and evaluated against the average value as mentioned in Section V. The result vector can be obtained as:

$$r_p = [0, 0, 0, 1, 1, 0, 0, 0, 0] \tag{12}$$

It signifies that permissions AP.ACC_FINE_LOC', 'AP.ACC_COARSE_LOC', are safe whereas 'AP. READ_PHONE_STATE' is unsafe. It is found that, on the final recommendation, all three permissions are marked unsafe. Thus, the Brightest Flashlight takes three extra permissions for which it is instrumented.

Running the permission recommender for Peacock Flashlight, it is found that it takes dangerous permissions 'AP.ACC_FINE_LOC' and 'AP.ACC_COARSE_LOC'.

Since Splendid Torch took no extra permissions, therefore, we did not perform any permission analysis for this app.

C. INSTRUMENTATION AND FINAL RESULTS

The permission analysis phase identified LOC and READ_PHONE_STATE permissions as unsafe for the flashlight applications. Brightest Flashlight and Peacock Flashlight were instrumented and installed on the target device. The instrumented apps interacted with the background service at runtime. Garbage location data was sent to the apps, and it was seen that the apps functioned properly after instrumentation.

After completing the whole process, the proposed framework gave the following results:

1) A flashlight app requires CAMERA permission for its operation.

2) The rest of the permissions that the two applications requested are classified as unsafe. The results of the applications were added to the dataset for use in the future.

3) Instrumentation and re-packaging the application restored the application's true use while protecting user data that could have been used for malicious activities.

We took three apps in the same category and of the same utility to study their patterns of operation. The three apps although, of the same nature, behaved differently as they were taking different permissions which were not directly related to the actual functionality that they have been listed for. The results obtained from the permission recommender show that two of the applications are taking extra permissions. After instrumentation, it is seen that their operation was unaltered, which shows that their functioning had not been impacted. At the same time, the user location was protected from a potentially malicious Android application.

The above case study shows that the proposed framework can be used to analyze and instrument Android applications to prevent user data from being used maliciously.

The existing related works discussed in Section IV is compared to the proposed framework as depicted in Table 4. For comparison, four features such as use of dynamic learning (DL), instrumentation (IN), permission recommendation (PR), and app based dangerous permission detection (ADPD) are considered. It is found that the proposed framework is able to support the analysis of an Android application as well as the prevention of user data theft. Existing works focused on detecting whether the app is benign/malware. Terminator prevents the application from using extra permissions by revoking access to those permissions identified as dangerous. But it is failed in the scenarios where an application fails to start without access to the permissions it requires. The proposed solution addresses this issue with instrumentation and ensures that the app is functioning as expected.

TABLE 4. Comparison of the proposed privacy protection framework with existing works.

Features / Method	DL	IN	PR	ADPD
Terminator [22]	✗	✗	✓	✓
NATICUSdroid [27]	✗	✗	✓	✓
VetDroid [23]	✗	✗	✓	✓
Permizer [28]	✗	✗	✓	✓
MPDroid [29]	✗	✗	✓	✓
AutoPer+ [30]	✗	✗	✓	✓
CUPERFUZZER+ [31]	✗	✗	✗	✓
Proposed Framework	✓	✓	✓	✓

VI. CONCLUSION AND FUTURE DIRECTION

The smartphone market has grown extensively in recent years and has become a repository for users' private data making the security of the device a big challenge. As technology advances, the risk of data breaches and invasion of privacy increases. Various research approaches were presented to identify the malicious behavior of Android applications. A privacy-preserving secure framework was proposed to prevent the applications from stealing user data by restricting all unnecessary permissions using instrumentation and re-packaging of the application. These permissions were recognized by predicting the permissions required by a given Android app by using collaborative filtering and frequent permission set mining algorithms. Thus, the proposed model interacts with the target app and modifies the permission data inside. A layer of security was added in proposed framework to prevent attackers from intercepting communications. Therefore, the proposed framework is more secure and efficient than the competitive models. Experimental results have shown that the proposed model not only protects the user data but also ensures the proper functioning of the given application.

However, this approach may achieve poor results for sealed protected applications that generally come under the category of finance/ payments as these applications come with additional security. Hence, these apps cannot be installed after they have been instrumented. In the future,

the framework can be modified to make it resilient to the additional securities/ protections in the applications.

REFERENCES

[1] *Mobile Operating System Market Share Worldwide | StatCounter Global Stats*. Accessed: Oct. 29, 2020. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide

[2] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic reconstruction of Android malware behaviors," in *Proc. NDSS Symp.*, San Diego, CA, USA, 2015, pp. 1–15, doi: 10.14722/ndss.2015.23145.

[3] S. Hassan, C. Tantithamthavorn, C.-P. Bezemer, and A. E. Hassan, "Studying the dialogue between users and developers of free apps in the Google play store," *Empirical Softw. Eng.*, vol. 23, no. 3, pp. 1275–1312, Jun. 2018.

[4] *Android Security 2017 Year in Review 2*. Accessed: Oct. 17, 2020. [Online]. Available: https://source.android.com/security/reports/Google_Android_Security_2017_Report_Final.pdf

[5] *Android Security 2018 Year in Review 2*. Accessed: Oct. 19, 2020. [Online]. Available: https://source.android.com/security/reports/Google_Android_Security_2018_Report_Final.pdf

[6] *Biggest App Stores in the World 2020 | Statista*. Accessed: Nov. 22, 2020. [Online]. Available: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

[7] K. W. Y. Au, Y. F. Zhou, Z. Huang, P. Gill, and D. Lie, "Short paper: A look at smartphone permission models," in *Proc. 1st ACM Workshop Secur. Privacy Smartphones Mobile Devices*, 2011, pp. 63–67, doi: 10.1145/2046614.2046626.

[8] M. Kaur, D. Singh, V. Kumar, B. B. Gupta, and A. A. A. El-Latif, "Secure and energy efficient-based e-health care framework for green Internet of Things," *IEEE Trans. Green Commun. Netw.*, vol. 5, no. 3, pp. 1223–1231, Sep. 2021, doi: 10.1109/TGCN.2021.3081616.

[9] M. Kaur and D. Singh, "Multiobjective evolutionary optimization techniques based hyperchaotic map and their applications in image encryption," *Multidimensional Syst. Signal Process.*, vol. 32, no. 1, pp. 281–301, Jan. 2021.

[10] *Android Architecture | Android Open Source Project*. Accessed: Nov. 22, 2020. [Online]. Available: https://source.android.com/devices/architecture

[11] M. Kaur and V. Kumar, "Parallel non-dominated sorting genetic algorithm-II-based image encryption technique," *Imag. Sci. J.*, vol. 66, no. 8, pp. 453–462, Nov. 2018.

[12] M. Kaur and V. Kumar, "Beta chaotic map based image encryption using genetic algorithm," *Int. J. Bifurcation Chaos*, vol. 28, no. 11, Oct. 2018, Art. no. 1850132.

[13] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky, "Android security framework: Extensible multi-layered access control on Android," in *Proc. 30th Annu. Comput. Secur. Appl. Conf.*, Dec. 2014, pp. 46–55, doi: 10.1145/2664243.2664265.

[14] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi, "ASM: A programmable interface for extending Android security," in *Proc. 23rd USENIX Secur. Symp. (USENIX Security)*, San Diego, CA, USA, 2014, pp. 1005–1019.

[15] M. Y. Karim, H. Kagdi, and M. Di Penta, "Mining Android apps to recommend permissions," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reeng.*, Mar. 2016, pp. 427–437, doi: 10.1109/SANER.2016.74.

[16] *Android Platform | Android Developers*. Accessed: Nov. 24, 2020. [Online]. Available: https://developer.android.com/about

[17] *Mobile & Tablet Android Version Market Share Worldwide | StatCounter Global Stats*. Accessed: Dec. 10, 2020. [Online]. Available: https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide

[18] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, and G. Álvarez, "PUMA: Permission usage to detect malware in Android," in *International Joint Conference CISIS'12-ICEUTE'12-SOCO'12 Special Sessions*. Berlin, Germany: Springer, 2013, pp. 289–298, doi: 10.1007/978-3-642-33018-6_30.

[19] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 641–660, 2013, doi: 10.1145/2544173.2509549.

[20] R. Neisse, G. Steri, D. Geneiatakis, and I. N. Fovino, "A privacy enforcing framework for Android applications," *Comput. Secur.*, vol. 62, pp. 257–277, Sep. 2016, doi: 10.1016/j.cose.2016.07.005.

[21] H. Shahriar, M. Islam, and V. Clincy, "Android malware detection using permission analysis," in *Proc. IEEE SOUTHEASTCON*, Mar. 2017, pp. 1–6, doi: 10.1109/SECON.2017.7925347.

[22] A. Sadeghi, R. Jabbarvand, N. Ghorbani, H. Bagheri, and S. Malek, "A temporal permission analysis and enforcement framework for Android," in *Proc. 40th Int. Conf. Softw. Eng.*, May 2018, pp. 846–857, doi: 10.1145/3180155.3180172.

[23] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in Android apps with permission use analysis," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 611–622.

[24] Y. Wu, S. Dou, D. Zou, W. Yang, W. Qiang, and H. Jin, "Obfuscation-resilient Android malware analysis based on contrastive learning," Jul. 2021, *arXiv:2107.03799*.

[25] I. M. Almomani and A. A. Khayer, "A comprehensive analysis of the Android permissions system," in *IEEE Access*, vol. 8, pp. 216671–216688, 2020, doi: 10.1109/ACCESS.2020.3041432.

[26] S. Millar, N. McLaughlin, J. Martinez del Rincon, P. Miller, and Z. Zhao, "DANdroid: A multi-view discriminative adversarial network for obfuscated Android malware detection," in *Proc. 10th ACM Conf. Data Appl. Secur. Privacy (CODASPY)*, Mar. 2020, pp. 353–364, doi: 10.1145/3374664.3375746.

[27] A. Mathur, L. M. Podila, K. Kulkarni, Q. Niyaz, and A. Y. Javaid, "NATICUSdroid: A malware detection framework for Android using native and custom permissions," *J. Inf. Secur. Appl.*, vol. 58, May 2021, Art. no. 102696, doi: 10.1016/j.jisa.2020.102696.

[28] Y. Qu, S. Du, S. Li, Y. Meng, L. Zhang, and H. Zhu, "Automatic permission optimization framework for privacy enhancement of mobile applications," *IEEE Internet Things J.*, vol. 8, no. 9, pp. 7394–7406, May 2021.

[29] J. Xiao, S. Chen, Q. He, Z. Feng, and X. Xue, "An Android application risk evaluation framework based on minimum permission set identification," *J. Syst. Softw.*, vol. 163, May 2020, Art. no. 110533.

[30] H. Gao, C. Guo, D. Huang, X. Hou, Y. Wu, J. Xu, Z. He, and G. Bai, "Autonomous permission recommendation," *IEEE Access*, vol. 8, pp. 76580–76594, 2020.

[31] R. Li, W. Diao, Z. Li, S. Yang, S. Li, and S. Guo, "Android custom permissions demystified: A comprehensive security evaluation," *IEEE Trans. Softw. Eng.*, early access, Oct. 14, 2021, doi: 10.1109/TSE.2021.3119980.

[32] AVISPA Team, *HLPSL Tutorial: A Beginner's Guide to Modelling and Analysing Internet Security Protocols*, AVISPA, 2006.

BHARAVI MISHRA received the master's degree from the Indian Institute of Information Technology, Allahabad, India, and the Ph.D. degree from the Indian Institute of Technology (BHU), Varanasi. He is working as the Assistant Professor with the Department of Computer Science and Engineering, The LNM Institute of Information Technology, Jaipur, India. He published more than 15 research articles in reputed journals and conferences. He also published three book chapters. His research interests include machine learning and its applications, security, and privacy.

AASTHA AGARWAL received the B.Tech. degree from The LNM Institute of Information Technology. Currently, she is working at VMware, India, as a Software Development Engineer. Her research interests include brain–computer interface with psychology, android security, and machine learning.

AYUSH GOEL received the B.Tech. degree from The LNM Institute of Information Technology. Currently, he is working at Zeta Suite (Directi), India, as a Software Development Engineer. His research interests include brain–computer interface, analysis of non-stationary signals, and machine learning.

AMAN AHMAD ANSARI received the M.Tech. degree from the Indian Institute of Information Technology, Allahabad, India. He is currently pursuing the Ph.D. degree with The LNM Institute of Information Technology, Jaipur, India. His current research interests include security and privacy.

PRAMOD GAUR received the B.E. degree (Hons.) in computer science and engineering from the University of Rajasthan, Jaipur, India, in 2004, the PGDIT degree from the Indian Institute of Technology Kharagpur, Kharagpur, India, in 2006, the M.E. degree in software engineering from the Birla Institute of Technology, Ranchi, India, in 2008, and the Ph.D. degree from Ulster University, U.K., in 2018. He is currently working as an Assistant Professor at the Birla Institute of Technology and Science, Pilani, Dubai. Previously, he worked as an Assistant Professor at The LNM Institute of Information Technology (LNMIIT), Jaipur, and a Postdoctoral RA in neuro-imaging technology at the Intelligent Systems Research Centre, Ulster University. His research interests include brain–computer interface, analysis of non-stationary signals, and machine learning.

DILBAG SINGH (Member, IEEE) received the M.Tech. degree from the Computer Science and Engineering Department, Guru Nanak Dev University, India, in 2012, and the Ph.D. degree in computer science and engineering from Thapar University, India, in 2019. He is currently working as a Research Professor at the School of Electrical Engineering and Computer Science, Gwangju Institute of Science and Technology (GIST), South Korea. He is the author and coauthor of more than 70 SCI/SCIE indexed journals, including refereed IEEE/ACM/Springer/Elsevier journals. He has also obtained three patents, three books, and two book chapters. His research interests include computer vision, medical image processing, machine learning, deep learning, information security, and meta-heuristic techniques. He was in the top 2% list issues by ''World Ranking of Top 2% Scientists,'' in 2021. He was part of the 11 Web of Science/Scopus indexed conferences. He has helped many under-graduated students to successfully implement their project work. He is a reviewer of more than 60 well-reputed journals, such as IEEE, Elsevier, Springer, SPIE, and Taylor & Francis. He is also acting as a Lead Guest Editor of *Mathematical Problems in Engineering* (Hindawi) (SCI and Scopus Indexed), an Executive Guest Editor of *Current Medical Imaging* (Bentham Science) (SCIE and Scopus Indexed), and an Associate Editor of *The Open Transportation Journal* (Scopus).

HEUNG-NO LEE (Senior Member, IEEE) received the B.S., M.S., and Ph.D. degrees in electrical engineering from the University of California at Los Angeles, Los Angeles, CA, USA, in 1993, 1994, and 1999, respectively. He was with HRL Laboratories, LLC, Malibu, CA, USA, as a Research Staff Member, from 1999 to 2002. From 2002 to 2008, he was an Assistant Professor with the University of Pittsburgh, PA, USA. In 2009, he moved to the School of Electrical Engineering and Computer Science, Gwangju Institute of Science and Technology, Gwangju, South Korea, where he is currently affiliated. His research interests include information theory, signal processing theory, blockchain, communications/networking theory, and their application to wireless communications and networking, compressive sensing, future Internet, and brain–computer interface. He has received several prestigious national awards, including the Top 100 National Research and Development Award, in 2012, the Top 50 Achievements of Fundamental Researches Award, in 2013, and the Science/Engineer of the Month (January 2014).