IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 5, 2020, accepted November 25, 2020, date of publication November 30, 2020,
date of current version December 14, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3041432

A Comprehensive Analysis of the Android Permissions System

IMAN M. ALMOMANI[1,2], (Senior Member, IEEE), AND AALA AL KHAYER[1]
[1]Security Engineering Laboratory, Department of Computer Science, Prince Sultan University, Riyadh 11586, Saudi Arabia
[2]Computer Science Department, King Abdullah II School for Information Technology, The University of Jordan, Amman 11942, Jordan

Corresponding author: Iman M. Almomani (imomani@psu.edu.sa)

This work was supported by the Security Engineering Laboratory, Prince Sultan University, Saudi Arabia, through the ARO: Android Ransomware Ontology Research Project, under Grant SEED-CCIS-2020-64.

ABSTRACT Android is one of the most essential and highly used operating systems. Android permissions system is a core security component that offers an access-control mechanism to protect system resources and users' privacy. As such, it has experienced continuous change over each Android release. However, previous research on the permissions system has employed static analysis techniques. Furthermore, most of these studies are outdated, covering older versions of Android. This paper aims to discuss the permissions system intensively to provide a nutshell overview of the Android platform's access-control mechanism. The paper presents a comprehensive analysis of the Android permissions system since it was introduced in 2008 until now, accompanied by a formal model of its components. The results of the analysis reveal a continuous growth in the number of permissions since the original release—a growth of seven times in some permission categories. A case study has been conducted for the last five years' versions of the top Android apps to examine the permissions system's evolution and its attendant security issues from the applications' perspective. Some apps showed an increase in permissions usage of 73.33% by the 2020 release. Additionally, the results of the case study contribute to the understanding of permissions deployment by both vendors and developers. Finally, a discussion of the permission-based security enhancements discloses that the Android permissions system faces various security issues. In general, this paper provides researchers and academics an up-to-date, comprehensive, self-contained reference study of the Android permissions system.

INDEX TERMS Access control, analysis, android, android application, android security, API level, APK, formal model, permission evolution, permission system, user privacy, security attack, survey.

I. INTRODUCTION

Android Operating System (OS) is the most popular platform for mobile devices since it owns 74.5% of the market place.[1] As a result, there is a significant increase in developing third-party applications by individual developers and companies to respond to this market shift. The most attractive features of Android are its open-source and unrestricted application market distribution. Consequently, allowing independent developers to develop their own applications and distribute them. Furthermore, the Android platform provides third-party development with a large-scale application programming interface (API) which enables applications to access the system resources and device hardware.

Android applications are installed as compressed Android packages (APK) including all the required files, libraries, and metadata for the apps to be executed. These APKs can be installed from the market place by calling the Package Manager system service [1]. For example, an application can be installed from the Google market store through the Google play application, an elevated permission app that downloads the required application. Another method of calling the Package Manager is copying the application's APK to the device and requesting Android OS to launch it. Additionally, an advanced tool, Android Debug Bridge (ADB), can be used to install applications through the Android Software development kit (SDK) [2].

To protect user's privacy,the Android platform enforces application to request permissions to access sensitive information [3], [4]. Therefore, to proceed with the app installation process, Android OS asks the user to grant the app's

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

[1]https://gs.statcounter.com/os-market-share/mobile/worldwide

required permissions explicitly. If the user rejects to grant the permissions, the installation process will be canceled. However, most users accept the permissions prompts without fully understanding the listed permissions. As a result, some applications might require extra permissions only to collect user data, which can invade user privacy and increase malicious acts [5]. Thus, Android security strongly depends on the efficiency of its permission system mechanism.

Previous studies on the Android permissions system have mainly investigated static analysis techniques to study specific permissions. However, these studies lack covering up-to-date Android versions [6], [7]. Section II provides detailed comparisons among related works illustrating their shortage to fully present, explain, discuss, and model the Android permissions system with all its components and versions. Moreover, addressing new apps' permissions usage and security attacks threatening Android OS; were also missing.

This paper provides a comprehensive, well-structured study and analysis of the Android platform's permissions system since it was introduced until now. Moreover, the paper discusses the user-permissions model of Android, which outlines how applications handle high-risk information and resources. In this regard, the main contributions of this work are as follows,

1) Present, discuss, and compare state-of-the-art in the field of Android permissions system.
2) Conduct an up-to-date study of the Android permissions system.
3) Make the first step towards an in-depth and comprehensive analysis of the entire permissions system.
4) Build a formal model of the Android permissions system. This model allows performing mathematical checking against security-related issues.
5) Demonstrate recent security flaws found in the Android permissions system.
6) Validate the Android permissions system by conducting a case study that includes a set of test apps. These apps were used for analyzing the permissions evolution and exposing the privacy and security issues.
7) Provide new datasets for Android OS. These datasets include (a) the permissions of all categories in all Android versions and API releases, (b) the permissions of top leading Android applications by downloading or/and revenue.

The rest of the paper is structured as follows. Section II provides a summary and comparisons among current related works analysing Android permissions system. An overview of Android security structure has been discussed in section III. Following that, section IV discusses the components of the Android permissions system and proposes its formal model. Section V analyzes the evolution of the Android permissions system. Then, section VI presents the case study and its results analysis. Section VII highlights the security issues and enhancements of the Android permissions system. Finally, the paper is concluded and possible future work is presented in section VIII.

II. LITERATURE REVIEW

The permissions system is considered a cornerstone component of the Android security model [8]. Consequently, the misuse of Android permissions is considered a major concern in the research community [9]–[11]. Several studies have been conducted on the evolution of Android permissions system [5]–[7], [12], [13]. Table 1 illustrates a summary and comparisons among existing works analyzing the Android permissions system.

The authors of [6] have performed an early investigation on the permissions system to determine how the permissions have grown. The study covers API levels 3 to 15 on a set of 237 Android applications. They concluded that the permissions system increased in complexity for both the Android platform and its applications until 2012. Following that, the authors of [7] investigated the evolution of the permissions system until the Android platform version 23 in 2015. The rising consideration was about accessing sensitive information by privileged third-party applications without the user awareness, drove them to investigate the new run-time permissions deeply. They discovered various security concerns regarding the complexity of the run-time Android permissions system that the community has to address.

However, relatively little work has provided formal and abstract modeling of the Android permissions system. Among the earliest studies that provided formal modeling of the Android permissions system were [17] and [18]. In [17], the authors built the formal model by specifying the essential components of the permissions scheme, including the interaction between its components. Furthermore, they deployed the suggested model to verify the security threats in the Android system. The work of [18] developed the Sorbet framework, an enforcement model that enables permissions to specific security policies. Other recent modeling approaches focused on abstracting the permission system for specific purposes, such as the permission-based model for attack defense [19] and language-based security analysis [20]. However, most papers that implemented abstract models lack a comprehensive analysis covering the permissions system's changes throughout its published releases.

Other studies focused on a specific aspect of the Android permissions system by studying a special type of permissions [1], [14], [15] or analyzing the permissions for a particular type of applications [16]. For example, the authors of [1] investigated the security issues of the run-time permissions. They conducted a deep analysis of the new permissions model introduced in API 23. However, the authors of [14] focused on the dangerous permissions to detect malicious applications. They defined a new algorithm called fine-grained dangerous permission (FDP), which uses dangerous permissions to delineate the differences between benign applications and malware applications. Furthermore, the analysis of the permissions system was conducted for a special kind of appli-

TABLE 1. Summary and comparison among current works analysing Android permissions system.

related work	purpose	API Level	Studied Dataset	Used Tools	Protection Level	Application Type	Other Studied Components
[1]	investigate the security issues of the run-time permissions to specifically describe how Android restricts the access control of the sensitive data.	22-23	-	-	all	-	flags, groups
[5]	reduce the over-privileged apps by applying collaborative filtering and static analysis to determine the least required permissions for an app.	-	16,343 apps	Apktool	dangerous and normal	third-party apps	-
[6]	analyze the Android permissions evolution, investigate on the least privileged applications, and highlight the issues of the studied applications.	3- 15	237 apps with 1,703 versions	Aapt tool	all	pre-installed, and third-party apps	groups
[7]	empirically investigate the evolution of permission system and provide an analysis of the new permission model focusing, mainly, on the run-time permissions.	4-23	-	-	all	-	flags
[13]	apply Kernel Density Estimation to build patterns of permissions used by benign and malicious apps.	-	120,000 applications	-	all	third-party apps	-
[8]	analyze crowd-sourced user reviews in order to understand the reason of permissions requesting by an application.	1-28	20K apps	Stanford Parser	dangerous	third-party apps	-
[9]	propose a permissions-based system to detect ransomware by identifying the Android permissions used by the ransomware apps.	28	500 benign apps and 500 ransomware	Apktool, Weka	all	third-party apps	-
[10]	propose an approach to automatically define the malicious behaviours of insatll-time requested permissions.	23-28	140 applications	Soot	dangerous and normal	third-party apps	groups
[11]	conduct an empirical study to enhance the understanding of the issues related to Android permission during the development-cycle.	21-22	574 GitHub repositories	M-Perm, P-Lint	dangerous and normal	third-party app	-
[12]	create a dataset of the changes and issues of permissions in Android applications to help researchers and Android developers enhance their understanding of the best practices of the permission system.	23	2,002 apps	F-Droid, M-Perm, P-Lint	all	third-party app	-
[14]	create a new algorithm called fine-grained dangerous permission (FDP), which uses dangerous permissions to detect malware applications.	28	1600 malicious apps and 1700 benign apps	Apktool, WEKA	dangerous	third-party apps	groups
[15]	analyze the patterns of permission explaining behaviors to identify action performed by the application.	23	83,244 apps	WHYPER	dangerous (run-time permissions)	third-party apps	-
[16]	analyze the pre-installed applications on Android devices to uncover relationships between different vendors.	14-23	1,742 device models	Lumen	dangerous	pre-installed	Custom Permissions

cations. The authors of [16] mainly focused on analyzing the permissions system of the pre-installed application, applications provided by default from the device manufacturers. They applied crowd-sourcing methods on applications collected from approximately 200 vendors. After that, they extracted a large set of custom defined permissions created by the third-party developers and hardware vendors. They concluded that such permissions could be misused to bypass the Android permissions system to access sensitive system resources and private data.

Some work focused on providing solutions and suggestions to enhance the security of the Android permissions system were presented in [5], [21]–[23]. A recent paper by [5] discussed the risk framework's evaluation based on the identification of a minimum set of permissions to find a solution for the over-privileged applications' security issues. They developed a framework, MPDroid, which performed a static analysis to determine the minimum required permissions by an application. Then, MPDroid identified the unnecessary permissions called by the application to evaluate its unprivileged risk level. Olukoya *et al.* investigated the matching between the requested permissions by an application with the textual description provided to the user the usage of these permissions by the application [23]. The results showed that their approach improved the accuracy of detecting mismatches between the declared and the used application permissions.

In reference to the aforementioned work, it can be concluded that there is a lack of a comprehensive and up-to-date analysis of the Android permissions system. Furthermore, the previous studies that proposed the Android permissions system's abstract modeling were either outdated or incomplete modeling of its entire components. Additionally, the previous research work mainly aimed to either analyze a specific type of permissions; or analyze the permissions system for particular applications. In this work, a nutshell analysis of the recent Android permissions system was carried out, along with a discussion on the security enhancements and issues. Moreover, a formal model of the whole permissions system is proposed. Finally, a case study that includes both pre-installed apps and third-party apps was conducted to provide a deep analysis of the Android permissions system from application perspectives.

III. ANDROID SECURITY STRUCTURE

Android platform is a Linux-based open-source operating system for mobile applications developed by Open Handset Alliance (OHA). As Figure 1 illustrates, Android platform architecture consists of three main layers: application, middle, and kernel layer [21]. The Linux kernel is the core foundation of Android platform, enabling other Android platform components to perform basic functionalities such as device drivers and low memory management [24]. On top of the kernel layer, the middle layer consists of the Android runtime and Java API framework. Applications and services use Android Runtime (ART) to manage the runtime processes on the Android platform. ART is an optimized Java virtual machine

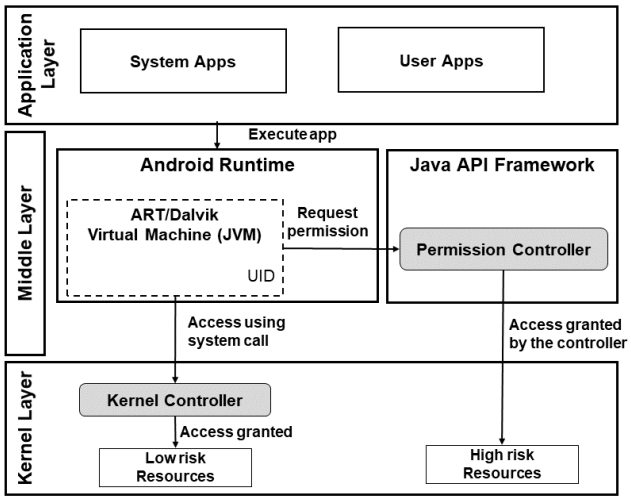

FIGURE 1. Android Platform Architecture.

for mobile devices. In Android 5.0 and higher, each application runs within its own instance of the ART virtual machine [25]. However, for older Android versions, prior to Android 5.0, Applications ran on Dalvik Java Virtual Machine (JVM). Beside the ART, the middle layer also comprises of Java API Framework which includes all Android APIs. At the top of Android architecture, there is the application layer which includes the system applications; set of Android core apps and user & third-party applications.

A. ANDROID SANDBOX

Android OS takes advantage of using a Linux kernel to apply security features such as process isolation and user-based permission system. The process of isolation is enforced by applying the application sandbox. As shown in Figure 1, each Android platform application is separated from the system and other apps since it runs on an isolated JVM. Upon executing the application, Android assigns each application a unique Linux User ID (UID) and a new ART/DVM virtual machine is forked [26].

For an interaction to occur, applications communicate with each other are required to interact through an inter-component communication (ICC) by passing *Intent* massages. An *Intent* message is a trigger event for an activity or service to be performed along with the required data which supports that requested action. However, studies have shown that Android ICC includes various security threats [27]. For example, the exchange of *Intent* messages can be used to escalate the privileges of malicious apps [28].

B. ANDROID APPLICATION COMPONENTS

Android applications are written in Java and compiled to Dalvik byte-code. There are four types of Android application components, including activities, services, content providers, and broadcast receivers. *Activities* run in the foreground, defining the interacting interface with the user. Typically,

an Android application defines the main activity, which includes sub-activities. The user interacts with these sub-activities to perform a specific task. The second component is the *service*, which does not have a screen user interface since it runs in the background. Services remain active in the background even if the application is not activated in the foreground screen.

The *content provider* supplies the required data storage. In addition to the required data storage, these providers are used to share data between different applications. Each content provider uniquely identifies its data set via disclosing a public Uniform Resource Identifier (URI). This data set can be accessed or updated by other components or applications by using SQL queries. The *broadcast receiver* manage the inter-communication between the application components. Furthermore, they are responsible for the communication between the system and the components. The receivers inform the application components of the received wide-system broadcasts.

Each component of the application can be executed separately and interact with other components, and it can even be instantiated by other applications as required. To accomplish the communication between these components, Android platform uses *Intents*. These intents deliver data through asynchronous messages. Furthermore, the intents are used to create new instances at the recipient component. When a component calls another component through an intent, it has to explicitly specify the package and class name of the recipient component or implicitly by specifying the action that the intent is attempting to initiate at the recipient component.

C. API CALLS

Android Platform provides large-scale application programming interfaces (APIs) to support the third-party application. These APIs enable the developer to access the system's features and resources such as user data, settings and hardware. The structure of Android API is divided into two parts, a library and an implementation of that API. The API library is located in each application's virtual machine, while the implementation runs as a process in the system. Those APIs reside in the Android Software Development Kits (SDKs). As a result, they frequently change as the Android OS evolves [29]. The main functionalities provided by the API and changes can be found in the API documentation.

As depicted in Figure 2, Android handles the API calls in three main steps [25]. First, the API is called by an application located in the library. Then, a private interface is invoked by the library. Finally, the private interface initiates a remote procedure call (RPC) request with the system. The system process assigns a service to run the API.

D. ANDROID APPLICATIONS

Mainly, there are two categories of Android applications system and user applications. The system applications are the pre-installed apps in the system, and the device vendors provide them. The design and configurations of these application

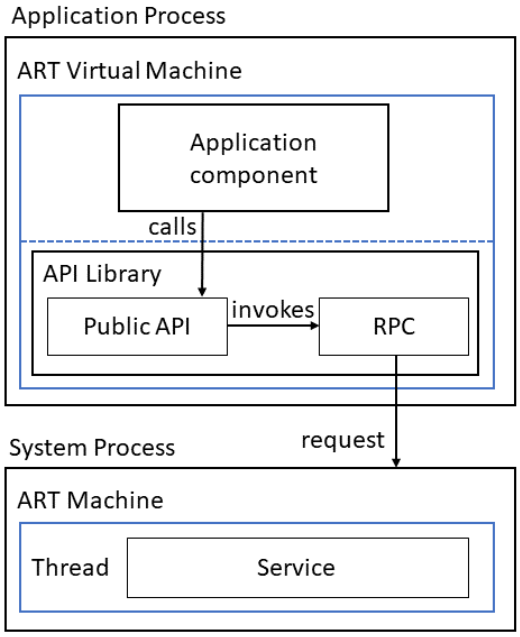

FIGURE 2. API call handling by the Android system.

can be customized for a particular device model by the manufacturers based on the vendors requirements. Whereas the user applications, the third-party apps, can be downloaded from various market stores such as Google Play, Anzhi, and AppChina [30], [31]. The third-party applications are developed by individual developers, that can include benign and malicious applications.

IV. ANDROID PERMISSION SYSTEM COMPONENTS AND FORMAL MODELING

A core design of the security architecture of Android is that no application can perform any operations that might adversely affect the user, other applications, or the operating system itself. Therefore, the application's requests to access components, sensitive data, and specific system features are regulated by the Android permissions system. Table 2 shows the evolution of the Android permissions system. For each release, Table 2 displays the Android release code name, version ranges, the API level ranges, the number of permissions per protection level, and the total number of permissions included in each release.

Permissions are uniquely defined as strings of characters (e.g., `android.permission.READ_SMS`). If a permission is bounded to an object, the object can only be accessed by another application after the permission is granted to the application.

To formally analyze the architecture of the Android permission system, an abstract model has been proposed. Formalizing the Android system provides a more precise level of evolution [32]. Furthermore, it underlines the differences between the features of the Android versions. Additionally,

TABLE 2. Android platform official releases.

Release Code Name	Android Platform Version	API Level	Dangerous	Normal	Signature	Signature orSystem	Total Permissions
Android 11 Beta	11	30	30	47	48	42	167
Q	10	29	30	45	42	41	158
PI	9	28	27	42	38	41	148
Oreo	8.0–8.1	26 - 27	26	39	38	41	144
Nougat	7.0 – 7.1	24 - 25	24	35	35	41	135
Marshmallow	6	23	24	35	32	40	131
Lollipop	5.0 – 5.1	21 - 22	24	32	24	40	120
KitKat Watch	4.4W	20	24	32	18	40	113
KitKat	4.4	19	23	32	18	40	112
Jelly Bean	4.1 – 4.3.1	16 - 18	23	29	16	36	104
Ice Cream Sandwich	4.0.1- 4.0.4	14-15	20	29	14	35	98
Honeycomb	3.0.x – 3.2	11-13	19	29	12	35	95
Gingerbread	2.3- 2.3.5	9-10	19	29	11	35	94
Froyo	2.2.x	8	18	27	11	35	87
Eclair	2.0 -2.1	5-7	18	26	9	33	86
Donut	1.6	4	18	26	9	32	85
Cupcake	1.5	3	17	25	8	31	81
Base	1 -1.1	1-2	17	24	7	27	73

TABLE 3. Equation notations and their meanings of the formal model.

Notation	Meaning
C_{acv}	Active component of Android application
C_{pas}	Passive component of Android application
P_{decl}	Permission declaration
P_{req}	Permission requesting
P_{grnt}	Permission granting
N_{prm}	Normal permission level
D_{prm}	Dangerous permission level
S_{prm}	Signature permission level
Ss_{prm}	SignatureOrSystem permission level
G_{prm}	Group of permissions
InT	Install-time permission request
RnT	Run-time permission request
f	flag
p	permission

the formal model can be used to expose the implementation of predefined security properties [17], [18]. Table 3 displays equation notations and their meanings of the proposed formal model. Each component of the Android application composes of Active components (C_{acv}) and Passive components (C_{pas}). The Active components (C_{acv}) interact with the system or other application's components such as services, activities, and broadcast receivers. Whereas, the Passive components (C_{pas}), such as the application content providers, only receive requests. Thus, an Android application can be defined as a combination of Active and Passive components, as in (1).

$$Android_{app} = C_{acv} | \, C_{pas} \tag{1}$$

A. MANIFEST FILE

Each application includes one Manifest file. The *AndroidManifest.xml* file is located at the root folder of an Android application. It contains the permissions used by the app and metadata about the app. The application permissions expected to be granted in order to access system or other apps resources/data are labeled with `<uses-permission>` tag in the *AndroidManifest.xml* file. Furthermore, the *AndroidManifest.xml* file contains permissions reacquired by the app to protect its own components. These permissions are declared with the `<permission>` tag.

Moreover, the Manifest file contains the components of the application. Each component of the application must declare its basic properties, including the permissions that the app uses. Since each application runs separately in the Android platform, it needs to protect its components by declaring specific permissions (P_{decl}). Furthermore, the application must define the requested permissions (P_{req}) to access external resources. Upon each API call, the application component has to check the permission granting status (P_{grnt}).

$$\begin{aligned} &\exists\, interaction(C_{acv_x} \longrightarrow Target(C_{acv}|C_{pas})) \iff \\ &\quad P_{decl} \in Target(C_{acv}|C_{pas}) \& \; P_{req} \in C_{acv_x} \; \& P_{grnt} = true \end{aligned} \tag{2}$$

The permission control process is described in (2). There is an interaction between a specific active component C_{acv_x} and its targeted component, whether it's another active component C_{acv} or a passive one C_{pas}, if and only if, three conditions are met. The targeted component declares the required permission to access its resources P_{decl}. Furthermore, the active component C_{acv_x} defines the requested permission in its *AndroidManifest.xml* file as P_{req}. Finally, to accomplish the interaction, the permission must be granted by the targeted component.

B. PROTECTION LEVELS

The levels refer to the intended use of a permission, as well as the consequences of using the permission. The standard permissions have a predefined protection level. Currently, Android platform supports three protection levels as

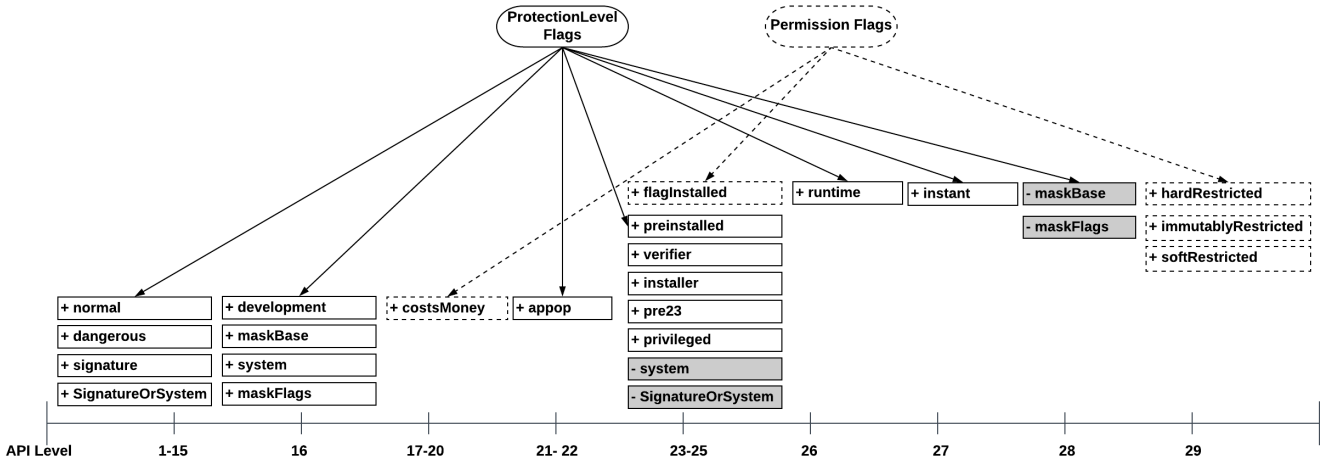

FIGURE 3. Protection level flags and permission flags of Android Permission System.

described in (3). These protection levels are called the *Base Permission Level* which can be defined as one of the following types Normal (N_{prm}), Dangerous (D_{prm}), and Signature protection level (S_{prm}).

$$BasePermissionType = N_{prm} \mid D_{prm} \mid S_{prm} \tag{3}$$

1) Normal protection level (N_{prm}) presents a low risk to Android applications. This type of permission does not require explicit approval by the user since the system automatically grants the access.
2) Dangerous protection level (D_{prm}) provides access to the user's private information, other applications operations, and device features. Applications requesting a dangerous permission can only run the functionality of that permission if the user explicitly permits it.
3) Signature protection level (S_{prm}) requires a comparison of the certification between the requesting application and the declaring application. These permissions can only be granted if the same certification signs both parties.

A fourth protection level is SignatureOrSystem (Ss_{prm}), which is granted to the requesting application if it is signed by the same certificate of the system image. These permissions are used by vendors who have multiple applications that require sharing certain features of a common system image. However, in Android 6.0, the SignatureOrSystem (Ss_{prm}) protection level was deprecated. Nevertheless, the permission analysis of [33] shows that this type of permissions is still used.

C. PERMISSION FLAGS

The behavior of a permission can be further controlled by using protection level flags and permission flags by assigning values to the fields *protectionLevel* and *permissionFlags*, respectively, in the *PermissionInfo* class. The *PermissionInfo* class is used to retrieve information regarding a specific permission in the system. Each protection level consists of a base type and can be followed by zero or more flags, as shown in (4). If the protection level is not defined in the *Manifest* file, it will be assigned as a *normal* protection level by the system.

$$\begin{aligned} PermissionDeclaration &= BasePermissionType \\ &\mid (protectionLevelFlag \mid permissionFlag)^{*} \\ &Where * \text{ is zero or more occurrence.} \end{aligned} \tag{4}$$

The *protectionLevel* attribute was deprecated in API level 28 and replaced with *getProtection()* and *getProtectionFlags()*. However, protection level flags can still be used, as shown in Figure 3. On the other hand, permission flags were added in API level 17. The protection level flags and the permission flags can only be used with the *signature* protection level. Correspondingly, (5) declares that a flag f can be assigned to a permission p, if and only if, that permission is defined as S_{prm} type. Using flags with other protection levels will cause a parsing error at manifest file.

$$\exists f \in p \longleftrightarrow p \in S_{prm} \tag{5}$$

D. PERMISSION GROUPS

Permission group (G_{prm}) is a logical categorization of the app's permissions. It is defined in the Manifest file as `<permission-group>` tag. Then, permissions can be added to this group by declaring an *android:permissionGroup* element inside the `<permission>` tag. Figure 4 shows an example of creating a permission group in the Manifest file. Two permissions, SEND_SMS and RECEIVE_SMS, have been added to the group SMS.

Android permissions system categorizes all dangerous permissions to permission groups, as described in (6). Accordingly, for all the permissions of type dangerous (D_{prm}), a permission group exists where that permission (p) belongs. Figure 5 shows that Android declares 11 *Dangerous*

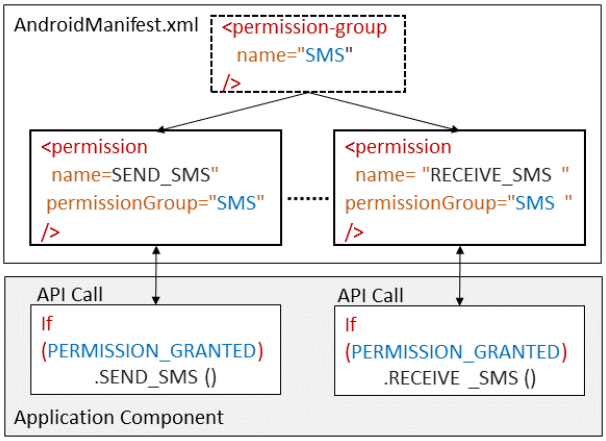

FIGURE 4. An illustrative example of Android grouping permissions.

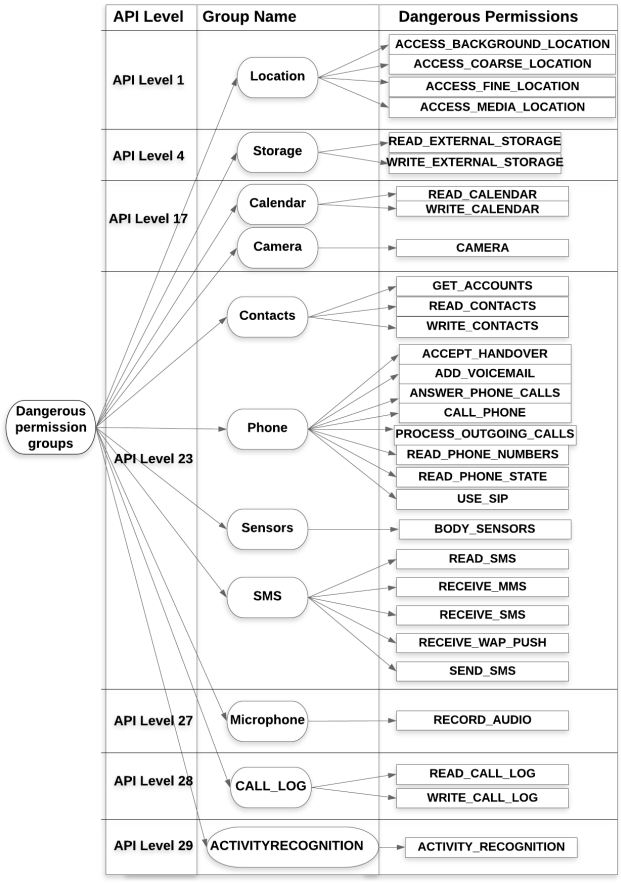

FIGURE 5. Android dangerous permission groups.

permission groups. However, regardless of the protection level, any permission can be added to a permission group.

$$\forall\, p \in D_{prm} \longrightarrow \exists\, G_{prm}; \quad p \in G_{prm} \tag{6}$$

E. PERMISSION TREES

Even though the Android platform does not support the hierarchical permissions, it provides permission trees encoding mechanism. A permission tree is a name-space that defines a family of permissions. This name-space represents a shared prefix name of the permission's family. The application that defines the permission tree's base-name owns all the permission names declared in this hierarchy. This prevents any other app from using permission with the same name. In case of conflict, the other application's protection levels will be automatically changed to *signature* protection level. Furthermore, the permission tree provide the application with the ability to dynamically add new custom permissions at runtime. For instance, google's APIs declare a name-space for each service to dynamically add individual permissions at runtime.

F. PERMISSION ENFORCEMENT

When an application requests a permission, the system checks the application's API level and the target SDK version, as illustrated in Figure 6. If the Application targeted SDK version lower than 23, all permissions requested by the application are granted at install-time. However, in Android 6.0, the permissions granting process is divided into two levels, install-time level (InT) and run-time level (RnT). The install-time permissions granting includes Normal (N_{prm}), Signature (S_{prm}) and SignatureorSystem (Ss_{prm}) permissions, as described in (7). If the user declines the install-time permission requests, the app installation will be abandoned. On the other hand, the run-time requests used to explicitly ask the user's approval for the dangerous permissions (D_{prm}).

$$PrmGrnt = \begin{cases} \forall\, p \in \{N_{prm}, S_{prm}, Ss_{prm}\} \rightarrow InT(p) \\ \forall\, p \in \{D_{prm}\} \rightarrow RnT(p) \end{cases} \tag{7}$$

Thus, if the application is running on an Android 6.0 device or higher, the system handles the permission granting as follows,

- The *Normal* permissions are granted automatically at install-time.
- For the *Signature* permissions, the system immediately grants the permissions if both applications are signed by the same certificate, without the user approval. Otherwise, the *Signature* permissions are granted at install-time.
- In order for the system to grant a *Dangerous* permission, it first checks the permission group of the requested permission. If the application has previously granted a dangerous permission of that group, the system's permission is immediately granted without any interaction with the user. On the other hand, if the permission group has not granted a permission, the permission is requested at the run-time by displaying a dialog to the user. If the group permission approval is granted, only the requested permission is given access by the system.

Permission Controller controls the run-time permission-related handling. In Android 9 and lower, the functionalities of the *Permission Controller* were embedded in the *Package-Installer* package. However, in Android 10, This module was separated from the Package Installer as an independent entity.

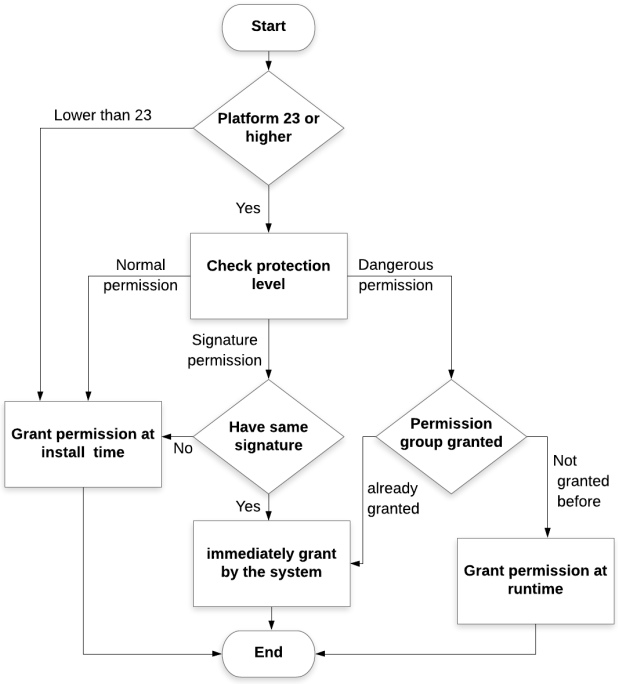

FIGURE 6. Permission granting based on protection level.

Name	Formula
Android Application	$Android_{app} = C_{acv} \mid C_{pas}$
Components Interaction	$\exists\, interaction\,(C_{acvx} \rightarrow Target\,(C_{acv} \mid C_{pas})) \leftrightarrow P_{decl} \in Target\,(C_{acv} \mid C_{pas}) \,\&\, P_{req} \in C_{acvx} \,\&\, P_{grant} = true$
Permission Declaration	$Base\ Permission\ Type = N_{prm} \mid D_{prm} \mid S_{prm}$ $Permission\ Decleration = BasePermissionType \mid (protectionLevelFlag \mid permissionFlag)^*$
Flags Condition	$\exists f \in p \leftrightarrow p \in S_{prm}$
Dangerous Groups	$\forall p \in D_{prm} \rightarrow \exists G_{prm}; p \in G_{prm}$
Permission Granting	$PrmGrnt = \begin{cases} \forall p \in \{N_{prm}, S_{prm}, Ss_{prm}\} \rightarrow InT(p) \\ \forall p \in \{D_{prm}\} \rightarrow RnT(p) \end{cases}$

FIGURE 7. Android Formal Model.

The *Permission Controller* controls the granting of run-time permissions, permissions grouping, and permissions usage tracking and roles.

The overall formal model of the Android permissions system is summarized in Figure 7.

V. ANDROID PERMISSIONS SYSTEM EVOLUTION AND ANALYSIS

This section presents the evolution of the permissions system in Android OS since it was launched till now. The permissions defined in each Android release for each API level will be listed and compared based on their protection levels. Moreover, the list of permissions added or deprecated in each release will also be highlighted.

A. ANDROID PERMISSIONS DATASET

To analyze each API level's permissions, a python script was developed to scan the *AndroidManifest* file and extract the permissions for each release from the developer website.[2] The script uses the <*permission*> tag to get all the defined permissions in the *AndroidManifest* file. Subsequently, the protection level and the permission status of all the fetched permissions for each corresponding API level are stored in the dataset records. In this paper, all the API levels, from API level 1 (2008) to API level 30 (2020), have been studied. With each release upgrade, the previously replaced permissions were not removed. Rather they were deprecated to enable using them by existing applications.

The dataset resulted from the permissions extraction process includes all added and deprecated permissions. This dataset will be publicly available on the Security Engineering laboratory (SEL) website.[3]

B. PERMISSIONS SYSTEM ANALYSIS

Table 2 and Figure 8 show that the total number of Android permissions increases over the successive releases.

In API level 1, the total number of permissions was 73. The net gain of the current Android version (API level 30) of 84 permissions resulted from adding 94 permissions and deprecating 10 permissions. Moreover, the permissions of types Dangerous, Normal, and SignatureOrSystem were almost doubled since API level 1. In contrast, Signature-based permissions were increased by around 7 times.

Generally, as depicted in Figure 8, the descending order of protection levels in terms of the number of permissions per level from API level 1 to API level 30 is: SignatureOrSystem, Normal, Dangerous, and then Signature permissions. Except for Signature permissions, that started to increase and exceed other types of permissions from API level 22. The percentage of increase in the case of Signature permissions reached 585.7% in API level 30 compared to API level 1.

Table 6 and Table 7 give a deep insight into the Android permissions system by providing the (a) Exact list of permissions; Android developers started with, and (b) permissions that were added or deprecated in each successive release throughout the Android permissions system lifetime. Table 6 lists all Dangerous, Normal, Signature, and SignatureOrSystem permissions defined in Android API level 1. Table 7 studies the permissions that were added or deprecated in each protection category at each API level. For example, API level 2 added one Normal permission to the list of permissions defined in API level 1 named `<CHANGE_WIFI_MULTICAST_STATE>`. Whereas, `<UNINSTALL_SHORTCUTE>` was the first

[2]https://developer.android.com/reference/android/Manifest.permission
[3]https://sel.psu.edu.sa/Research/datasets/2020_Permissions_API-30.php

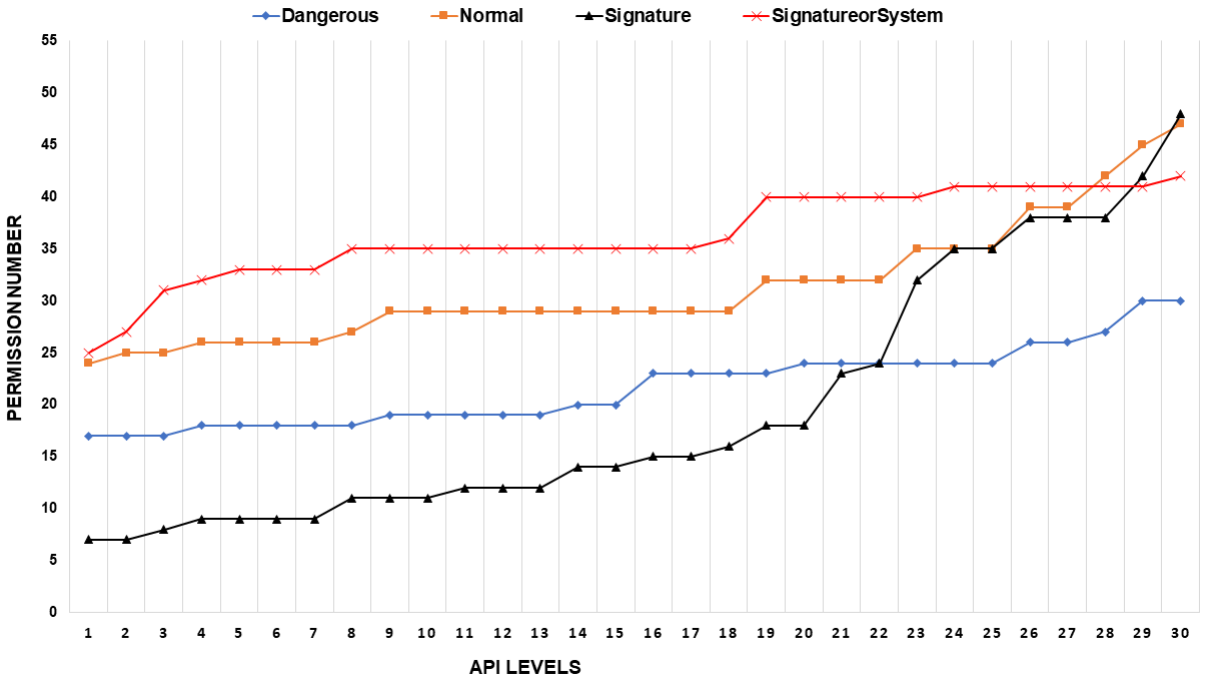

FIGURE 8. Permissions of protection levels over Android API evolution.

Normal permission that was deprecated by API level 19. The last three Dangerous permissions added to the Android system in API level 29 were:

<ACTIVITY_RECOGNITION>,
<ACCESS_BACKGROUND_LOCATION>,
<ACCESS_MEDIA_LOCATION>.

The rest of all permissions' updates in all API levels are detailed in Table 7. The *+number* indicates the total permissions were added by this level, whereas, *-number* indicates the total permissions were deprecated by this level.

VI. CASE STUDY: APPLICATIONS-BASED PERMISSIONS ANALYSIS

To investigate the evolution of Android permissions from applications perspective, a case study has been conducted. This study has chosen top applications used or installed by mobile users to be analyzed and studied. These apps were selected according to the Google Play website and the leading app by revenue according to Statista report.[4] Two types of applications were considered; the pre-installed and third-party applications. This section discusses the usage of Android applications permissions across 100 versions of 50 third-party apps and 50 pre-installed apps in the last five years (between 2016 and 2020). The list of apps is shown in Figure 9. The permissions requested by these apps were deeply analyzed from different viewpoints.

Figure 10 presents the analysis results of the third-party applications in the last five releases. The figure shows how

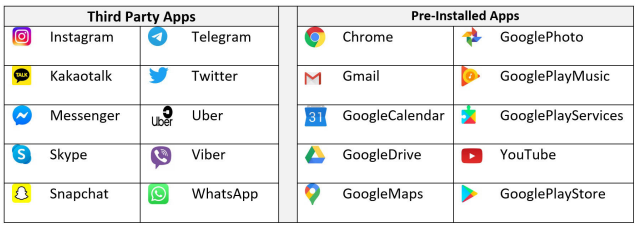

FIGURE 9. Case Study Apps.

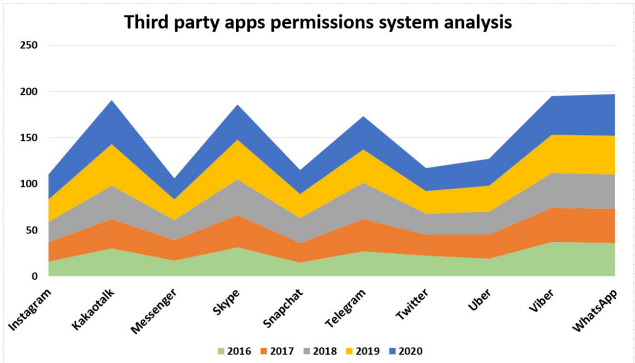

FIGURE 10. Third party apps permissions analysis.

the permissions were increased in all apps during the last five years. Although the amount of increase among these apps within the same year has varied, but the overall growth in the number of permissions can be observed from one year to the next.

[4]https://www.statista.com/statistics/693959/leading-google-play-communication-apps-worldwide-by-revenue/

TABLE 4. Third party apps permission increase analysis.

App Name	Siglsys	Signature	Dangerous	Normal	Increase Percentage
Instagram	-1	1	1	10	68.75%
Kakaotalk	5	1	5	7	60%
Messenger	0	0	1	5	35.29%
Skype	1	2	0	4	22.58%
Snapchat	0	3	1	7	73.33%
Telegram	0	4	1	4	33.33%
Twitter	0	1	-1	3	13.64%
Uber	-1	1	0	10	52.63%
Viber	1	3	-3	4	13.51%
WhatsApp	0	1	4	4	25%

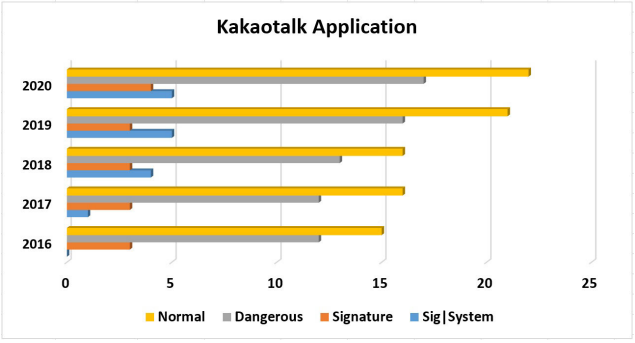

FIGURE 11. Kakaotalk permissions analysis from 2016 to 2020.

Table 4 shows the difference in each permissions category per app between the year 2016 and now. For example, *Instagram* app: (a) reduced the number of SignatureOrSystem permission by one, (b) increased the Signature and Dangerous permissions by one, and the Normal permissions by ten. So, the overall increase percentage in comparison to 2016 release is 68.75%. Another remarkable permissions increase can be observed by *Snapchat* app, which reached 73.33%.

Figure 11 highlights the changes in the permissions list of the *Kakaotalk* application, one of the leading communication apps in the world by revenue in reference to 2020 statista's statistics[4]. The figure illustrates the updates in the four protection levels from year 2016 to 2020. It can be observed how the number of permissions in all of them has been increased throughout these 5 years. The percentage of increase (from 2016 to 2020) of Normal, Dangerous, Signature, SigOrSystem permissions was 46.7%, 41.7%, 33.3% and 500%; respectively.

On the other hand, Figure 12 shows the analysis of the permissions updates in the pre-installed apps in the last five years. Again, it can be observed that the permissions requested by these apps were also increased in all categories in each successive year.

In comparison to year 2016, the main increase in *Chrome* app was for Dangerous and Normal permissions with percentages 33.3% and 35.7%; respectively. In *Gmail* app, the Dangerous permission was increased by 100%. No significant changes for *GoogleClanedar* app can be noticed. *GoogelDrive* increased the the Normal permissions by 40%.

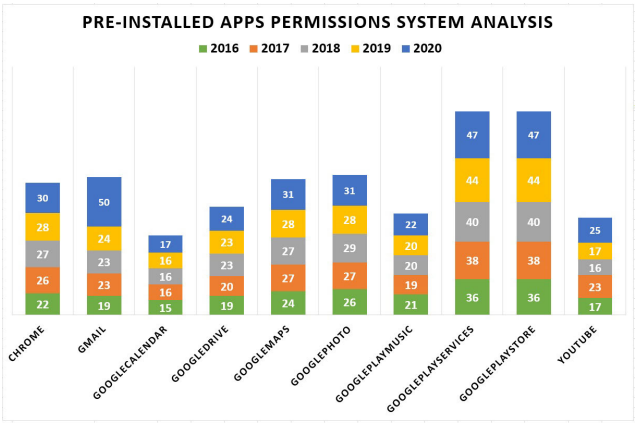

FIGURE 12. Pre-Installed apps permissions analysis.

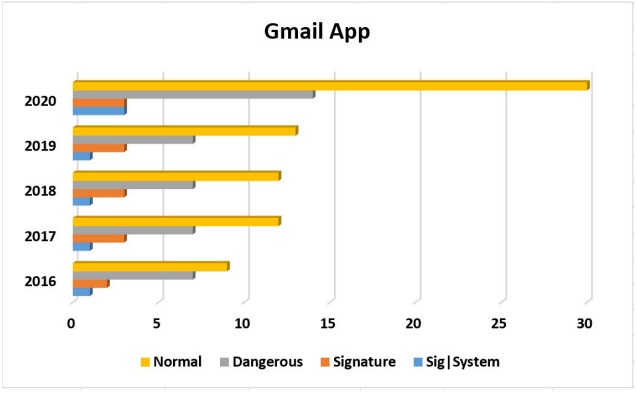

FIGURE 13. Gmail permissions analysis.

GoogelMaps has reached 200% and 57.1 % increase in case of SigOrSystem and Dangerous permissions, respectively. *GooglePhoto*, *GooglePlayMusic*, *GooglePlayServices* and *GooglePlayStore* have manily increased the SignatureOrSystem permissions. *YouTube* increased all types of permissions and reached 60% increase in case of Dangerous permissions.

Figure 13 presents the changes in the permissions list of the *Gmail* application. *Gmail* increased all types of permissions. The total increase in comparison to 2016 release reached 163.2%. Highest increase was observed in the Normal permissions with 233.33% and also the Dangerous permissions were doubled.

Figure 14 provides a comparative analysis among all apps under study, whether third-party or pre-installed apps. The results stress the fact that not only Android permissions system has introduced more permissions under all categories since year 2008, but also the apps themselves are utilizing more of these permissions in comparison with their previous editions.

Table 5 lists more detailed analyses in the case of Normal and Dangerous permissions in all apps. The table provides (a) the percentage of app's Normal permissions to the total permissions requested by the app itself (b) the percentage of app's Normal permissions to the total Normal permissions

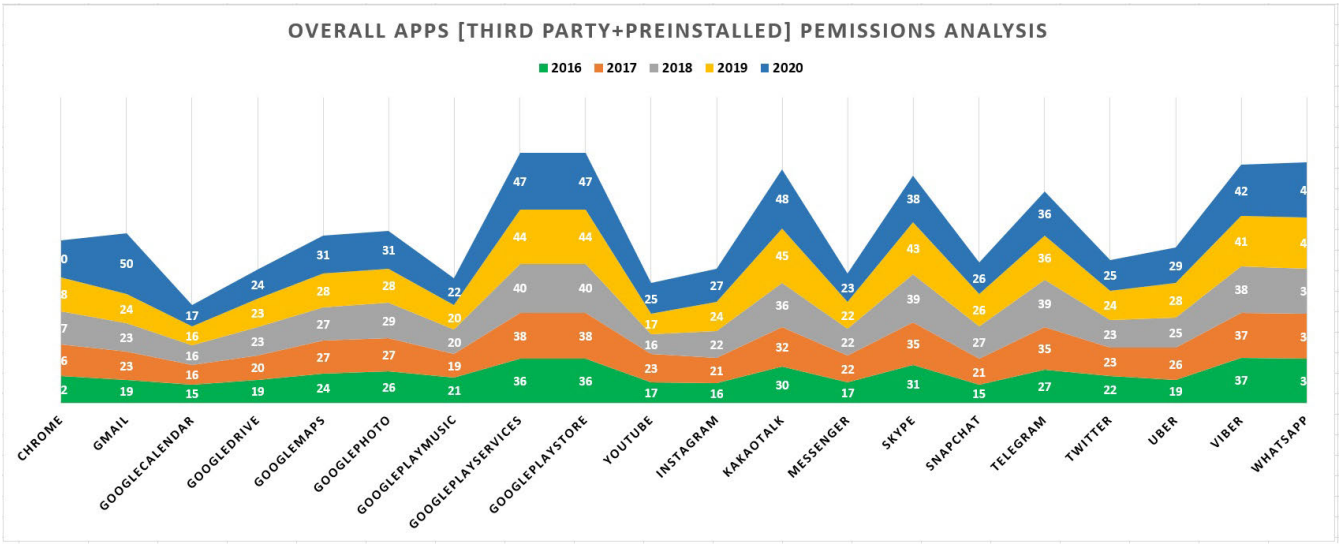

FIGURE 14. All Apps permissions analysis.

TABLE 5. Normal and Dangerous permissions analysis of all apps.

App Name	%of Normal permissions to the app's permission	%of Normal permissions to overall Android Normal permissions	%of Dangerous permissions to the app's permission	%of Dangerous permissions to the overall Android Dangerous permissions
Chrome	63.3	40.4	26.7	26.7
Gmail	60.0	63.8	28.0	46.7
GoogleCalendar	58.8	21.3	35.3	20.0
GoogleDrive	58.3	29.8	20.8	16.7
GoogleMaps	58.1	38.3	35.5	36.7
GooglePhoto	58.1	38.3	25.8	26.7
GooglePlayMusic	50.0	23.4	18.2	13.3
GooglePlayServices	34.0	34.0	23.4	36.7
GooglePlayStore	34.0	34.0	23.4	36.7
Instagram	48.1	27.7	33.3	30.0
Kakaotalk	35.4	36.2	35.4	56.7
Messenger	95.7	46.8	34.8	26.7
Skype	34.2	27.7	31.6	40.0
Snapchat	73.1	40.4	38.5	33.3
Telegram	36.1	27.7	33.3	40.0
Twitter	68.0	36.2	32.0	26.7
Uber	41.4	25.5	44.8	43.3
Viber	35.7	31.9	28.6	40.0
WhatsApp	48.9	46.8	35.6	53.3
YouTube	96.0	51.1	32.0	26.7

defined by Android OS (c) the percentage of app's Dangerous permissions to the total permissions requested by the app itself (d) the percentage of app's Dangerous permissions to the total Dangerous permissions defined by Android OS.

Many interesting observations can be seen in the table for some of the apps. For example, in terms of Normal permissions, YouTube allocated 96.0% of its permissions of type Normal. Also, YouTube used 51.1% of the Normal permissions defined by Android OS. In terms of Dangerous permissions, WhatsApp used 32% Dangerous permissions to its overall permissions. Also, it used 53.3% of the Dangerous permissions defined by Android OS in its latest release. The rest of the percentages for all the apps are shown in Table 5.

VII. PERMISSION BASED SECURITY DISCUSSION

With the rapid rise of the Android OS functionality, correspondingly, the vulnerabilities have been increased. To reduce security issues, Google has restricted third-party applications from invoking API's system-level [34]. This section highlights the permission-based security enhancements, issues and implications.

A. PERMISSION-BASED SECURITY ENHANCEMENTS

Android OS enhances the protection level by limiting the access of the third-party apps to the system resources. Consequently, the new versions of Android explicitly require placing permissions checks throughout the application [35]. Furthermore, as can be seen from the previous analysis of the Android permissions system, there is a continuous deprecation in the permissions. As a result, the out-dated applications that depend on system permissions cannot work on the recent versions of Android [34].

In Android 4, a new permission view has been introduced by categorizing the permissions into *groups*. The *groups* organize the permission display for the user. The user can still view a brief description of each permission in the group. However, one of the biggest enhancements in the security of the permission system was introducing the run-time permissions in Android 6.0. Run-time permissions provide the users with the ability to control access for both Marshmallow and pre-Marshmallow applications [34]. Furthermore, new access control has been implemented in Android 8.0, where users

TABLE 6. List of API level 1 permissions.

Dangerous Permissions	
READ_CALENDAR	RECORD_AUDIO
WRITE_CALENDAR	CALL_PHONE
CAMERA	PROCESS_OUTGOING_CALLS
GET_ACCOUNTS	READ_PHONE_STATE
READ_CONTACTS	READ_SMS
WRITE_CONTACTS	RECEIVE_MMS
ACCESS_COARSE_LOCATION	RECEIVE_SMS
ACCESS_FINE_LOCATION	RECEIVE_WAP_PUSH
SEND_SMS	
Normal Permissions	
PERSISTENT_ACTIVITY	GET_PACKAGE_SIZE
RESTART_PACKAGES	INTERNET
GET_TASKS	MODIFY_AUDIO_SETTINGS
ACCESS_LOCATION_EXTRA_COMMANDS	READ_SYNC_SETTINGS
ACCESS_NETWORK_STATE	READ_SYNC_STATS
ACCESS_WIFI_STATE	RECEIVE_BOOT_COMPLETED
BLUETOOTH	REORDER_TASKS
BLUETOOTH_ADMIN	SET_WALLPAPER
BROADCAST_STICKY	SET_WALLPAPER_HINTS
CHANGE_NETWORK_STATE	VIBRATE
EXPAND_STATUS_BAR	WAKE_LOCK
WRITE_SYNC_SETTINGS	CHANGE_WIFI_STATE
Signature Permissions	
SET_PREFERRED_APPLICATIONS	DELETE_CACHE_FILES
WRITE_SETTINGS	BATTERY_STATS
SYSTEM_ALERT_WINDOW	CHANGE_CONFIGURATION
CLEAR_APP_CACHE	
SignatureorSystem Permissions	
ACCESS_CHECKIN_PROPERTIES	MODIFY_PHONE_STATE
BROADCAST_PACKAGE_REMOVED	MOUNT_UNMOUNT_FILESYSTEMS
CALL_PRIVILEGED	READ_INPUT_STATE
CHANGE_COMPONENT_ENABLED_STATE	READ_LOGS
CONTROL_LOCATION_UPDATES	REBOOT
DELETE_PACKAGES	SET_ALWAYS_FINISH
DIAGNOSTIC	SET_ANIMATION_SCALE
DUMP	SET_DEBUG_APP
FACTORY_TEST	SET_PROCESS_LIMIT
INSTALL_PACKAGES	SET_TIME_ZONE
MASTER_CLEAR	SIGNAL_PERSISTENT_PROCESSES
WRITE_APN_SETTINGS	STATUS_BAR
WRITE_GSERVICES	

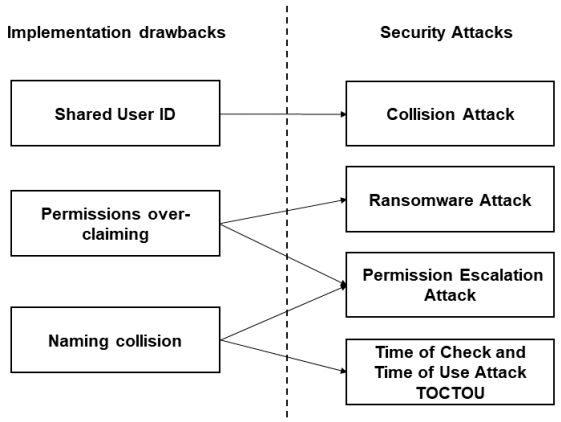

FIGURE 15. Implementation drawbacks and security attacks relation.

were implicitly required to grant permission to install applications from a non-first-party market store [36]. Recently, in Android 10, the PermissionController module has been created as a separate module that handles the permissions updates.

B. PERMISSION-BASED SECURITY ISSUES

Even though the security of Android permission system seems protected, the continues enhancements introduce various security issues and drawbacks [10], [11], [14]. Regardless of the Android version, the access enforcement in Android platform is subject to the API targeted level. Consequently, the application may abuse this system feature to boost its privileges [1]. For example, if an application targets an API level before Android 6.0, it will be granted all its permissions, including the dangerous ones, at install-time. Another issue that threatens Android security is the over-privileged apps [28]. An application may over claim permission even though the application does not use this particular requested permission.

However, due to these security weaknesses, the Android OS is vulnerable to various security attacks including, but not

TABLE 7. List of API level 2-30 permissions.

API level	Dangerous Permissions	Normal Permission	Signature Permissions	SignatureOrSystem Permissions
1	17	24	7	25
2	0	+ **1** (DISABLE_KEYGUARD)	0	+**2**(BROADCAST_SMS, BROADCAST_WAP_PUSH)
3	0	0	+**1** (BIND_INPUT_METHOD)	+**4**(BIND_APPWIDGET MOUNT_FORMAT_FILESYSTEMS UPDATE_DEVICE_STATS WRITE_SECURE_SETTINGS)
4	+**1** (WRITE_EXTERNAL_STORAGE)	+**1**(CHANGE_WIFI_MULTICAST_STATE)	+**1** (GLOBAL_SEARCH)	+**1**(INSTALL_LOCATION_PROVIDER)
5	0	0	0	+**1** (ACCOUNT_MANAGER)
6	0	0	0	0
7	0	0	0	0
8	0	+**1**(KILL_BACKGROUND_PROCESSES)	+**2**(BIND_DEVICE_ADMIN BIND_WALLPAPER)	+**2**(LOCATION_HARDWARE SET_TIME)
9	+**1**(USE_SIP)	+**2** (NFC SET_ALARM)	0	0
10	0	0	0	0
11	0	0	+**1** (BIND_REMOTEVIEWS)	0
12	0	0	0	0
13	0	0	0	0
14	+**1** (ADD_VOICEMAIL)	0	+**2** (BIND_TEXT_SERVICE BIND_VPN_SERVICE)	0
15	0	0	0	0
16	+**3** (READ_CALL_LOG WRITE_CALL_LOG READ_EXTERNAL_STORAGE)	0	+**1** (BIND_ACCESSIBILITY_SERVICE)	0
17	0	0	0	0
18	0	0	+**1**(BIND_NOTIFICATION _LISTENER_SERVICE)	+**1** (SEND_RESPOND_VIA_MESSAGE)
19	0	+**2**(INSTALL_SHORTCUT TRANSMIT_IR) **-1** (UNINSTALL_SHORTCUT)	+**2**(BIND_NFC_SERVICE BIND_PRINT_SERVICE)	+**4**(BLUETOOTH_PRIVILEGED CAPTURE_AUDIO_OUTPUT MEDIA_CONTENT_CONTROL MANAGE_DOCUMENTS)
20	+**1**(BODY_SENSORS)	0	0	0
21	0	0	+**5**(BIND_DREAM_SERVICE BIND_VOICE_INTERACTION BIND_TV_INPUT READ_VOICEMAIL WRITE_VOICEMAIL)	0
22	0	0	**-1**(BIND_CARRIER_ MESSAGING_SERVICE)	0

TABLE 7. *(Continued.)* **List of API level 2-30 permissions.**

API level	Dangerous Permissions	Normal Permission	Signature Permissions	SignatureOrSystem Permissions
23	0	+**2** (ACCESS_NOTIFICATION_POLICY REQUEST_IGNORE_ BATTERY_OPTIMIZATIONS) **-1** (USE_FINGERPRINT)	+**7** (REQUEST_INSTALL_PACKAGES BIND_CARRIER_SERVICES BIND_INCALL_SERVICE BIND_TELECOM_CONNECTION_SERVICE GET_ACCOUNTS_PRIVILEGED PACKAGE_USAGE_STATS BIND_MIDI_DEVICE_SERVICE) **-1**(BIND_CHOOSER_TARGET_SERVICE)	0
24	0	0	+**3** (BIND_SCREENING_SERVICE BIND_CONDITION_PROVIDER_SERVICE BIND_VR_LISTENER_SERVICE)	+**1** (BIND_QUICK_SETTINGS_TILE)
25	0	0	0	0
26	+**2** (ANSWER_PHONE_CALLS READ_PHONE_NUMBERS)	+**4** (READ_PHONE_NUMBERS MANAGE_OWN_CALLS REQUEST_COMPANION _RUN_IN_BACKGROUND REQUEST_COMPANION_USE_ DATA_IN_BACKGROUND REQUEST_DELETE_PACKAGES)	+**3** (INSTANT_APP_ FOREGROUND_SERVICE BIND_VISUAL_VOICEMAIL_SERVICE BIND_AUTOFILL_SERVICE)	0
27	0	0	0	0
28	+**1** (ACCEPT_HANDOVER)	+**3** (FOREGROUND_SERVICE NFC_TRANSACTION_EVENT USE_BIOMETRIC)	0	0
29	+**3** (ACTIVITY_RECOGNITION ACCESS_BACKGROUND_ LOCATION ACCESS_MEDIA_LOCATION)	+**3** (CALL_COMPANION_APP REQUEST_PASSWORD_COMPLEXITY USE_FULL_SCREEN_INTENT)	+**4** (SMS_FINANCIAL_TRANSACTIONS BIND_CARRIER_MESSAGING _CLIENT_SERVICE START_VIEW_PERMISSION_USAGE BIND_CALL_REDIRECTION_SERVICE)	0
30	0	+**2** (QUERY_ALL_PACKAGES NFC_PREFERRED_PAYMENT_INFO)	+**6** (INTERACT_ACROSS_PROFILES READ_PRECISE_PHONE_STATE BIND_QUICK_ACCESS_WALLET_SERVICE ACCESS_CALL_AUDIO MANAGE_EXTERNAL_STORAGE LOADER_USAGE_STATS)	+**1** (BIND_CONTROLS)

limited to, collision attack, permission escalation attack, Time of Check and Time of Use (TOCTOU) attack, and Ransomware attack. Figure 15 shows the relationship between the implementation drawbacks and the security attacks.

a) **Collision attack** occurs when two applications are assigned with the same User ID. The User ID attribute is the unique package name, and it is located in the *AndroidManifest.xml* file. Two or more applications can access each others' granted permission if they are assigned with the same User ID and signed by the same certificate [37]. For example, application *A* and *B* share the same User ID, *Shared_User_ID*. Application *A* is granted a permission to *READ_SMS* and *CALL_PHONE* while application *B* has the permission to use *CAMERA*. Consequently, both apps can use all the three aforementioned permissions since they are sharing the same User ID, *Shared_User_ID* [38].

b) **Escalation attack**. When two applications collaborate to access sensitive data or system resources without explicitly requesting permissions [39]. For instance, the component (cmp_B) of application *B* is granted permission to access the resources of application *A*. Hence, application *C* can collaborate with application *B* to transitively access the resources of application *A*. Once application *B* grants application *C* access to one of its component, application *C* can access resources of application *A* via component (cmp_B).

c) **TOCTOU attack**. Due to the absence of the Android permission system's naming restriction, any two permissions that share the same name are considered equivalent albeit they belong to different apps [40]. Suppose an application *A* is granted a permission p_1 which access critical system resources. However, a malicious app *B* might declare a permission p_2 with same name of p_1. As a result, the malicious app *B* can use the permission p_2 to access the critical system resources.

d) **Ransomware attack**. The misuse of Android permissions and API packages makes Android applications and their users vulnerable to ransomware attacks. Ransomware attacks can lock or encrypt Android users' devices or/and data and ask for ransom to release the data/control back to the users [41], [42]. Mainly, the permissions requested to implement such attack are related to the victim device that include *SYSTEM_ALERT_WINDOW*, *WAKE_LOCK*, and *KILL_BACKGROUND_PROCESS* [43].

C. SECURITY IMPLICATIONS

The security implications of the Android permissions system can be discussed from two perspectives; developers and end-users [44]. The developer's responsibility in protecting the Android application against the risk of security issues arises in two circumstances (1) when the application requires access to the system resources and (2) when it communicates with third-party apps. In both cases, the developer is required to define the corresponding permissions manually in the Manifest file. However, the developer may accidentally leave the application's components exposed to be used by third-party applications unprotected by any permission or assign it to improper permission. Consequently, leaving the application vulnerable to harmful malicious attacks and put the application's reputation at risk. A large number of research studies discussed the development mistakes that put the applications at risks and how they can be avoided [12].

On the other hand, the end-user is handled a major role in the Android permissions system's implementation mechanism. Involving the end-user in making decision regarding filtering the applications needs to be carefully investigated. Currently, end-users have a minor understanding of permission warnings [45]. Subsequently, to improve the user-based permission system's efficiency, there is an urgent need to educate the end-user of the security implications of the permissions system [44]. Nonetheless, selectively allowing or denying permissions by the end-user is significantly affected by the user experience [45]. In general, the display system of the permissions dialog is not very user-friendly which makes it quite difficult for the user to understand the risks associated with these permissions. The permissions granting process can be improved by using a simpler visualization approach that a normal user can understand to help him/her make proper decisions. In this context, Google has carefully designed the permissions dialog to prevent the user's confusion [46]. Nevertheless, more efforts need to put in this regard.

Furthermore, the run-time permissions introduced by Android OS enables the end-user to control permission access in the real context of the application. For instance, a *LOCATION* permission will be requested only when the application needs to access the device location. As a result, this approach improves the user's understanding of the requested permission.

VIII. CONCLUSION

To control access to sensitive data and system resources, the Android platform enforces a permission-based mechanism. However, in an effort to enhance efficiency and improve flexibility, the Android permissions system has been thoroughly updated in each release. This paper presents a detailed overview of the Android permissions system along with a formal model of system components and relationships.

This study began with a review of all permissions from the first release of Android OS until the current one, including all permissions at different protection levels (Normal, Dangerous, Signature, or SignatureOrSystem) that have been added or deprecated in each release at each API level. Recent related works were presented, discussed, and compared in this research. Additionally, a formal model to describe the components of the Android permissions system and their functionality was also proposed.

After the analysis of all releases of the Android permissions system, a case study was conducted to analyze the top applications in various versions. This analysis was aimed at examining changes in the permissions system and its security flaws during the last five years. Additionally, the investigation of permission-based security enhancement reveals several security threats that may open the OS to various attacks, such as collision attack, escalation attack, TOCTOU attack and ransomware attack.

In conclusion, this research provides a comprehensive, self-contained reference study of the Android permissions system. It provides a history and an update on the status of Android permissions. In addition to a formal definition of its components, a comparative analysis of the permissions systems across top Android apps will be shared with the research community, along with the results and datasets generated in this paper.

In future research, the impact on the security of Android users and vendors of using Android permissions at different protection levels might be studied more thoroughly. Other permission systems on other mobile OSes could be investigated, such as that of iOS. Moreover, further studies could investigate more if the defined permissions of the existing apps are fully utilized by them; or they are classified as over-privileged apps. These studies can consider different categories and analyze the percentages and types of permissions used by these over-privileged apps and their impact on the security of the users' devices and data.

ACKNOWLEDGMENT

This work was supported by the Security Engineering Laboratory, Prince Sultan University, Saudi Arabia, through the ARO: Android Ransomware Ontology Research Project, under Grant SEED-CCIS-2020-64.

REFERENCES

[1] E. Alepis and C. Patsakis, "Unravelling security issues of runtime permissions in android," *J. Hardw. Syst. Secur.*, vol. 3, no. 1, pp. 45–63, Mar. 2019.

[2] K. S. Noori and A. A. Fahad, "Factors affecting application launch time with Android OS," *Iraqi J. Sci.*, pp. 1791–1797, Jul. 2020.

[3] M. Hatamian, "Engineering privacy in smartphone apps: A technical guideline catalog for app developers," *IEEE Access*, vol. 8, pp. 35429–35445, 2020.

[4] M. Alenezi and I. Almomani, "Abusing Android permissions: A security perspective," in *Proc. IEEE Jordan Conf. Appl. Electr. Eng. Comput. Technol. (AEECT)*, Oct. 2017, pp. 1–6.

[5] 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.

[6] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Permission evolution in the Android ecosystem," in *Proc. 28th Annu. Comput. Secur. Appl. Conf.*, 2012, pp. 31–40.

[7] Y. Zhauniarovich and O. Gadyatskaya, "Small changes, big changes: an updated view on the Android permission system," in *Proc. Int. Symp. Res. Attacks, Intrusions, Defenses*. Cham, Switzerland: Springer, 2016, pp. 346–367.

[8] R. Wang, Z. Wang, B. Tang, L. Zhao, and L. Wang, "SmartPI: Understanding permission implications of Android apps from user reviews," *IEEE Trans. Mobile Comput.*, vol. 19, no. 12, pp. 2933–2945, Dec. 2019.

[9] S. Alsoghyer and I. Almomani, "On the effectiveness of application permissions for Android ransomware detection," in *Proc. 6th Conf. Data Sci. Mach. Learn. Appl. (CDMA)*, Mar. 2020, pp. 94–99.

[10] J. Huang, W. Huang, F. Miao, and Y. Xiong, "Detecting improper behaviors of stubbornly requesting permissions in Android applications," *IJ Netw. Secur.*, vol. 22, no. 3, pp. 381–391, 2020.

[11] G. L. Scoccia, A. Peruma, V. Pujols, I. Malavolta, and D. E. Krutz, "Permission issues in open-source Android apps: An exploratory study," in *Proc. 19th Int. Work. Conf. Source Code Anal. Manipulation*, Sep. 2019, pp. 238–249.

[12] G. L. Scoccia, A. Peruma, V. Pujols, B. Christians, and D. Krutz, "An empirical history of permission requests and mistakes in open source Android apps," in *Proc. IEEE/ACM 16th Int. Conf. Mining Softw. Repositories (MSR)*, May 2019, pp. 597–601.

[13] M. S. Saleem, J. Misic, and V. B. Misic, "Examining permission patterns in Android apps using kernel density estimation," in *Proc. Int. Conf. Comput., Netw. Commun. (ICNC)*, Feb. 2020, pp. 719–724.

[14] X. Jiang, B. Mao, J. Guan, and X. Huang, "Android malware detection using fine-grained features," *Sci. Program.*, vol. 2020, pp. 1–13, Jan. 2020.

[15] X. Liu, Y. Leng, W. Yang, W. Wang, C. Zhai, and T. Xie, "A large-scale empirical study on Android runtime-permission rationale messages," in *Proc. IEEE Symp. Vis. Lang. Hum.-Centric Comput. (VL/HCC)*, Oct. 2018, pp. 137–146.

[16] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "An analysis of pre-installed Android software," 2019, *arXiv:1905.02713*. [Online]. Available: http://arxiv.org/abs/1905.02713

[17] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka, "A formal model to analyze the permission authorization and enforcement in the Android framework," in *Proc. IEEE 2nd Int. Conf. Social Comput.*, Aug. 2010, pp. 944–951.

[18] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey, "Modeling and enhancing Android's permission system," in *Proc. Eur. Symp. Res. Comput. Secur.* Berlin, Germany: Springer, 2012, pp. 1–18.

[19] K. Javed and M. Tariq, "Formal modeling of security concerns in android," *Lgurjcsit*, vol. 4, no. 1, pp. 33–37, 2020.

[20] W. Khan, M. Kamran, A. Ahmad, F. A. Khan, and A. Derhab, "Formal analysis of language-based Android security using theorem proving approach," *IEEE Access*, vol. 7, pp. 16550–16560, 2019.

[21] S. Kumar, R. Shanker, and S. Verma, "Context aware dynamic permission model: A retrospect of privacy and security in Android system," in *Proc. Int. Conf. Intell. Circuits Syst. (ICICS)*, Apr. 2018, pp. 324–329.

[22] F.-H. Hsu, N.-C. Liu, Y.-L. Hwang, C.-H. Liu, C. S. Wang, and C.-Y. Chen, "DPC: A dynamic permission control mechanism for Android third-party libraries," *IEEE Trans. Dependable Secure Comput.*, early access, Aug. 27, 2020, doi: 10.1109/TDSC.2019.2937925.

[23] O. Olukoya, L. Mackenzie, and I. Omoronyia, "Security-oriented view of app behaviour using textual descriptions and user-granted permission requests," *Comput. Secur.*, vol. 89, Feb. 2020, Art. no. 101685.

[24] H. Shukla, "A survey paper on Android operating system," *J. Gujarat Res. Soc.*, vol. 21, no. 5, pp. 299–305, 2019.

[25] H. Cai and B. G. Ryder, "A longitudinal study of application structure and behaviors in Android," *IEEE Trans. Softw. Eng.*, early access, Feb. 19, 2020, doi: 10.1109/TSE.2020.2975176.

[26] S. Kumar and S. K. Shukla, "The state of Android security," in *Cyber Security in India*. Singapore: Springer, 2020, pp. 17–22.

[27] M. A. El-Zawawy, E. Losiouk, and M. Conti, "Do not let next-intent vulnerability be your next nightmare: Type system-based approach to detect it in Android apps," *Int. J. Inf. Secur.*, vol. 6, pp. 1–20, Mar. 2020.

[28] A. K. H. Hussain, M. Kakavand, M. Silval, and L. Arulsamy, "A novel Android security framework to prevent privilege escalation attacks," *Int. J. Comput. Netw. Inf. Secur.*, vol. 12, no. 1, pp. 20–26, Feb. 2020.

[29] M. Fazzini, Q. Xin, and A. Orso, "Automated API-usage update for Android apps," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, 2019, pp. 204–215.

[30] I. Almomani and A. Khayer, "Android applications scanning: The guide," in *Proc. Int. Conf. Comput. Inf. Sci. (ICCIS)*, Apr. 2019, pp. 1–5.

[31] I. Almomani and M. Alenezi, "Android application security scanning process," in *Telecommunication Systems*, I. A. Alimi, P. P. Monteiro, and A. L. Teixeira, Eds. Rijeka, Croatia: IntechOpen, 2019, p. 3, doi: 10.5772/intechopen.86661.

[32] J. He, T. Chen, P. Wang, Z. Wu, and J. Yan, "Android multitasking mechanism: Formal semantics and static analysis of apps," in *Proc. Asian Symp. Program. Lang. Syst.* Cham, Switzerland: Springer, 2019, pp. 291–312.

[33] J. Gajrani, M. Tripathi, V. Laxmi, G. Somani, A. Zemmari, and M. S. Gaur, "Vulvet: Vetting of vulnerabilities in Android apps to thwart exploitation," *Digit. Threats: Res. Pract.*, vol. 1, no. 2, pp. 1–25, Jul. 2020.

[34] R. Sikder, S. Khan, S. Hossain, and W. Z. Khan, "A survey on Android security: Development and deployment hindrance and best practices," *Telkomnika*, vol. 18, no. 1, pp. 485–499, 2020.
[35] W. Enck, "Analysis of access control enforcement in android," in *Proc. 25th ACM Symp. Access Control Models Technol.*, Jun. 2020, pp. 117–118.
[36] L. Verderame, D. Caputo, A. Romdhana, and A. Merlo, "On the (Un)Reliability of privacy policies in Android apps," 2020, *arXiv:2004.08559*. [Online]. Available: http://arxiv.org/abs/2004.08559
[37] S. S. Joshi and R. Sharma, "A review of Android security system," *Int. J. Sci. Res. Eng. Trends*, vol. 1, pp. 615–619, Dec. 2019.
[38] Y. Amirgaliev, B. Sayazhan, and Y. Bakbergen, "Android security issues," Suleyman Demirel Univ., Kaskelen, Kazakhstan, Tech. Rep. 20.51, 2018, p. 125.
[39] L. Shen, H. Li, H. Wang, and Y. Wang, "Multifeature-based behavior of privilege escalation attack detection method for Android applications," *Mobile Inf. Syst.*, vol. 2020, pp. 1–16, Jun. 2020.
[40] M. Deypir, "Entropy-based security risk measurement for Android mobile applications," *Soft Comput.*, vol. 23, no. 16, pp. 7303–7319, Aug. 2019.
[41] S. Alsoghyer and I. Almomani, "Ransomware detection system for Android applications," *Electronics*, vol. 8, no. 8, p. 868, Aug. 2019, doi: 10.3390/electronics8080868.
[42] H. Faris, M. Habib, I. Almomani, M. Eshtay, and I. Aljarah, "Optimizing extreme learning machines using chains of salps for efficient Android ransomware detection," *Appl. Sci.*, vol. 10, no. 11, p. 3706, May 2020, doi: 10.3390/app10113706.
[43] J. W. Hu, Y. Zhang, and Y. P. Cui, "Research on Android ransomware protection technology," in *J. Phys., Conf. Ser.*, vol. 1584, no. 1, 2020, Art. no. 012004.
[44] A. K. Jha and W. J. Lee, "Analysis of permission-based security in Android through policy expert, developer, and end user perspectives," *J. UCS*, vol. 22, no. 4, pp. 459–474, 2016.
[45] N. Momen, S. Bock, and L. Fritsch, "Accept–Maybe–decline: Introducing partial consent for the permission-based access control model of android," in *Proc. 25th ACM Symp. Access Control Models Technol.*, Jun. 2020, pp. 71–80.
[46] D. Wermke, N. Huaman, Y. Acar, B. Reaves, P. Traynor, and S. Fahl, "A large scale investigation of obfuscation use in Google play," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, Dec. 2018, pp. 222–235.

IMAN M. ALMOMANI (Senior Member, IEEE) received the bachelor's degree in computer science from United Arab Emirates in 2000, master's degree in computer science from Jordan in 2002, and the Ph.D. degree in wireless network security from De Montfort University, U.K., in 2007. She was an Associate Professor and the Head of the Department of Computer Science Department, The University of Jordan, Jordan. She is currently an Associate Professor in Cybersecurity, the Associate Director of the Research and Initiatives Centre, and the Leader of the Security Engineering Laboratory, Prince Sultan University, Riyadh, Saudi Arabia. Her research interests include wireless networks and security, mainly wireless mobile ad hoc networks, wireless sensor networks, multimedia networking, security issues in wireless networks, electronic learning, and mobile learning. She has several publications in the above-mentioned areas in a number of reputable international and local journals and conferences. She is also a Senior Member of the IEEE WIE. She is in the organizing and technical committees for a number of local and international conferences. She also serves as a reviewer and a member of the editorial board in a number of international journals.

AALA AL KHAYER received the B.E. degree in information technology engineering from SVU University, Damascus, in 2017, and the bachelor's degree in software engineering from Prince Sultan University (PSU), Riyadh, Saudi Arabia, in 2018. She is currently a Research Engineer with the Security Engineering Laboratory, PSU. Her research interests include software engineering, networks security, malware analysis, multimedia networking, and computer vision.