


Detection of Integrity Attacks on Permissions of Android-Based Mobile Apps: Security Evaluation on PayPal

Omar Hussein

Department of Management Information Systems, Faculty of Management Sciences, October University for Modern Sciences and Arts (MSA), 6th October, Egypt

ohusseins@gmail.com

 <https://orcid.org/0000-0002-0282-7541>

Abstract

The objective of this paper is to detect unauthorized modifications to genuine permissions of legitimate Android-based mobile apps in real-time, with demonstration on PayPal payment gateway mobile app. The scientific value of this work lies in finding a remedy for lack of binary protection vulnerability in Android-based mobile apps. The motivation behind conducting this research on PayPal is because of its widespread popularity, and the reported increase in the attacks targeting Android apps along with the sensitive nature of payment gateway mobile apps. This paper proposes an anti-circumvention security approach called Android Apps Permissions Integrity Verifier (AAPIV) to achieve the desired goal. AAPIV captures and computes the authentic unique 256-bit hash of the AndroidManifest.xml file of a legitimate Android-based mobile app. An app's permissions are registered in AndroidManifest.xml file in its Android Package Kit file. AAPIV stores the computed hash in its cloud-based database server. For every access request to the data stored in the database server of the mobile app service provider, the 256-bit hash of the AndroidManifest.xml file of the requesting app is captured, extracted, computed, and verified for authenticity against that stored in AAPIV's cloud-based database server. In case both hashes are identical, this denotes a legitimate access request from an authentic mobile app, and accordingly the access request is allowed, otherwise the access request is denied. An experimental security evaluation was applied on PayPal Android-based payment gateway mobile app. It demonstrated that AAPIV effectively achieved its intended objective.

Keywords: Android-Based Apps Security; Mobile Apps Permissions; Integrity Attacks; Android Package Kit

1. Introduction

Android is an open-source operating system based on Linux kernel and owned by Google [1]. It is the dominating operating system for mobile devices with a market share of 70.1% in the fourth quarter of 2023¹. Google Play App Store is the first-largest store for Android apps. In the third quarter of 2022 it hosted 3.55 million Android apps². Android-powered devices (e.g., smartphones, tables) are equipped with multiple sensors that capture personal data. This widens the attack surfaces of Android-based devices in front of adversaries. Android apps are classified as either system or user apps. System apps (pre-installed apps) are provided by vendors of mobile devices. Based on vendors requirements, mobile device manufacturers can tailor system apps' design and configuration settings for a particular device model. Examples of pre-installed apps include: Google Chrome, and Google Maps. User apps (third-party apps) are developed by individual

¹ STATISTA, Market Share of Mobile Operating Systems Worldwide 2009-2023.

<https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/>, 2024 (last accessed 25 January 2024)

² STATISTA, Number of Apps Available in leading App Stores Q3 2022.

<https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/>, 2024 (last accessed 25 January 2024)

developers. These apps can be benign or malicious, and are downloadable from various sources. Examples of benign third-party apps include: X, and WhatsApp. In the second quarter of 2022, 405,684 malicious Android Package Kit (APK) files were discovered by Kaspersky Security Network³. Examples of malicious third-party apps are “SafeGraph” that was recently banned by Google⁴, and “SafeChat”; its hidden malicious functionality was lately revealed by Singapore-based cybersecurity firm called “CYFIRMA”⁵. Reliance on mobile devices in carrying out online financial transactions has increased; especially as social distancing was rigidly required since COVID-19 pandemic. Unfortunately, this was accompanied by a spike in mobile-based cyber security breaches [2]. As reported in [3], more than 90% of mobile device malicious software (malware) targets the Android operating system. Vulnerabilities in Android source code are the primary causes of these attacks [4, 5]. For example, the *Next-Intent* security vulnerability is a known exploitable Android vulnerability that went unpatched for an extended time period [6-8]. Wang et al [9] illustrated the possibility of capturing a user’s password in real-time by exploiting the *Activity* component of Android. An integrity attack on permissions of an Android-based mobile app refers to attacks that tamper with the permissions of a mobile app to compromise the app’s security. This paper aims at maintaining the integrity of Android-based mobile apps permissions. The objective is to detect unauthorized modifications to an app’s permissions. The main contributions of this paper are as follows: (1) present a proposed user-transparent method to cover lack of binary protection vulnerability in Android-based mobile apps; and (2) propose a real-time security approach to detect unauthorized modifications to the permissions of Android-based mobile apps.

The remainder of this paper is organized as follows. Sections 2 and 3 are devoted to cover the conceptual background, and explore related work respectively. Section 4 details the different aspects of the proposed security approach including its applied experimental security evaluation on PayPal payment gateway mobile app. Section 5 discusses novelty of the proposed security approach and its merits. Finally, Section 6 concludes this paper and outlines the future work.

2. Conceptual Background

2.1. Android Apps Compilation and Decompilation Processes

Android apps are written in Java programming language. Android Studio is an integrated development environment to develop Android apps. Android Studio compiles Java code. It packages data, besides resource and configuration files into a single APK file [10]. Java bytecode is the resulting compilation of Java object code of an app. Java bytecode (.class) in turn is compiled by dex compiler (component of Android Software Development Kit (SDK)) into Dalvik Executable/DEX code (.dex). All (.class) files are integrated into a single *classes.dex* file. Dalvik Virtual Machine (DVM), which is a part of Android, executes the compiled DEX code [11]. A single APK file is an Android app file that contains *classes.dex*, *AndroidManifest.xml* files, plus resource files. It is used for installation on Android-powered devices [12]. Fig.1 depicts the compilation process of an APK file, starting from writing an app in Java until obtaining the APK file. The ZIP file format is used by APKs files. It is possible to unzip an APK file using any file archiver. However, the extracted files and folders from an unzipped APK file are illegible. Decompilation is the opposite of compilation. It means translation of machine-readable executable code back to human-readable source code [13]. Android Studio allows decompilation of APK files to access and modify apps’ functionalities and security settings [14]. Through Android Studio 4.0, an APK file can be decompiled by choosing “Analyze APK” menu option from the “Build” drop-down menu.

³ SECURELIST, IT threat evolution in Q2 2022. Mobile statistics.

<https://securelist.com/it-threat-evolution-in-q2-2022-mobile-statistics/107123/>, 2024 (last accessed 25 January 2024)

⁴ The Verge, Google bans tracking tool that sold users’ location data.

<https://www.theverge.com/2021/8/12/22621685/google-ban-safe-graph-android-user-data-location-tracking>, 2024 (last accessed 25 January 2024)

⁵ Cyfirma, APT Bahamut Targets Individuals with Android Malware Using Spear Messaging.

<https://www.cyfirma.com/outofband/apt-bahamut-targets-individuals-with-android-malware-using-spear-messaging/>, 2024 (last accessed 25 January 2024)

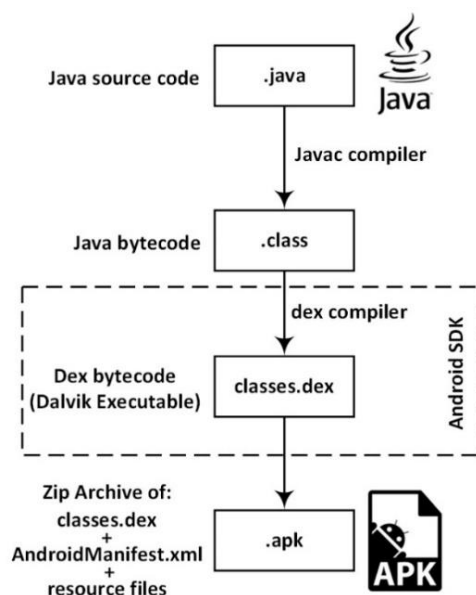


Fig.1. Android app compilation process

2.2. Permissions in Android-Based Apps and Potential Risks

Android's access control security mechanism mandates apps to request permissions at installation time (Fig.2), and individually at runtime (Fig.3), before accessing and using any system resource. At app installation time, Android requires the user to expressly accept the app's required access rights/permissions. In case the user refuses to grant access rights to a particular app, its installation is terminated. Apps that request excessive permissions (i.e., the problem of apps being overprivileged) generate security vulnerabilities that can be maliciously exploited [15, 16]. An app's permissions are registered in *AndroidManifest.xml* file in its APK file and located at the root directory of the app source set [14]. This XML file plays essential roles as it declares the following⁶: (1) app components; (2) app permissions to access other apps, or parts of the system; (3) permissions granted to other apps to access the app's content; and (4) hardware and software requirements that are needed to install the app on a device from Google Play Store. In Android, each permission has a protection level⁷. There are three permission protection-levels: (1) *normal*; (2) *dangerous*; and (3) *signature*. A permission is a constant value in *AndroidManifest.xml* file that begins with a prefix "android.permission.". For example, "android.permission.VIBRATE" is a *normal* protection-level android-based app permission, whereas "android.permission.GET_ACCOUNTS" is a *dangerous* protection-level android-based app permission. Additionally, "android.permission.MANAGE_Ongoing_CALLS" is an example of a *signature* protection-level permission. *Normal* protection-level permissions are automatically granted to an Android-based app without the user's consent. They are characterized as being with low-risk to the system and other apps. *Dangerous* protection-level permissions require user's consent before installing the app. They affect the user's privacy as they access his/her data and core device functionalities.

⁶ ANDROID FOR DEVELOPERS, App Manifest Overview.

<https://developer.android.com/guide/topics/manifest/manifest-intro>, 2024 (last accessed 25 January 2024)

⁷ ANDROID FOR DEVELOPERS, "<permission>".

<https://developer.android.com/guide/topics/manifest/permission-element>, 2024 (last accessed 25 January 2024)

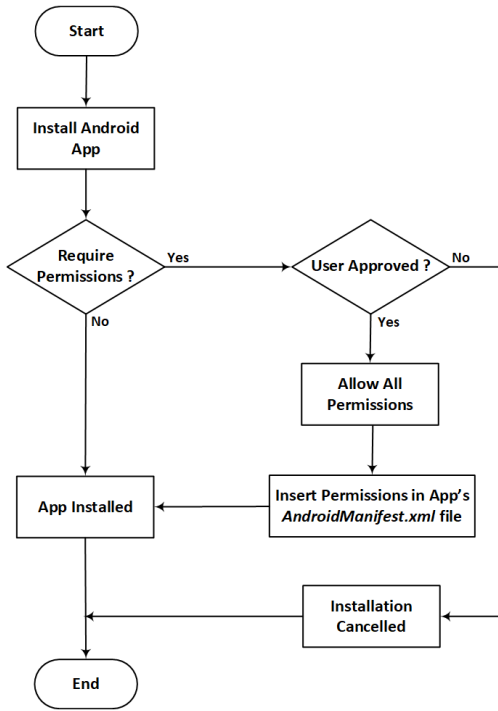


Fig.2. Android app permissions requests at installation time

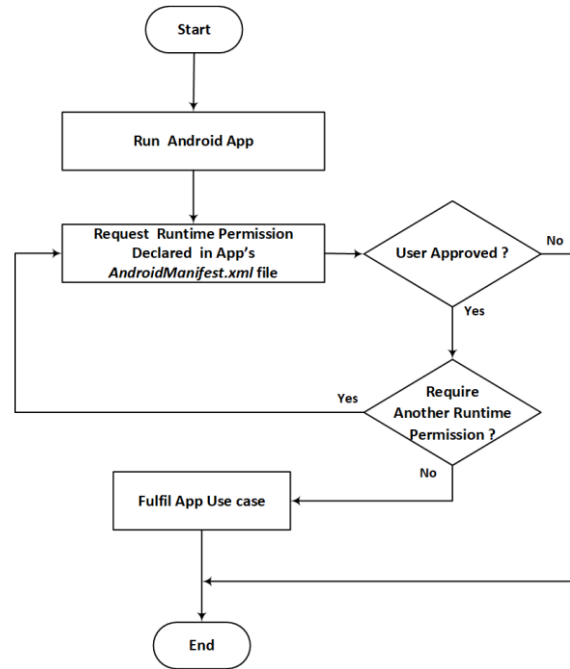


Fig.3. Android app permissions requests at runtime

Every Android app uploaded to Google Play Store should be signed with the app developer's signature, which is the developer's cryptographic private key. A private key is essential to identify and verify the owner of an Android app. During a new Android app installation, and in a *signature* protection-level permission, the app requesting the permission must be digitally signed with the same developer's signature as that of a previously installed app that defines the needed permission on the device. An Android app refers to the *AndroidManifest.xml* file to enforce the intended app's permissions during installation and execution. It tags each permission with `<uses-permission>`. Fig.4 depicts a portion of PayPal app's group of permissions stated in its *AndroidManifest.xml* file. In order to deliver its malicious payload, an infected Android app will request permissions irrelevant to its intended functionality. Table 1 lists examples of *dangerous* protection-level permissions⁸.

2.3. Functionality of Payment Gateway Apps

Payment gateway apps are specialized in managing online payments through debit/credit cards. As depicted in Fig.5, an online payment gateway app captures debit/credit card details from its users. These card details include card number, card type, expiration date, card verification value, card holder name, and payment value. The payment gateway app passes the card and payment details to the card issuing bank via the card payment network. The card issuing bank validates the card details and balance, then approves the transaction. Finally, the card issuing bank sends back payment confirmation to the card holder, and deposits the payment amount to the beneficiary's account.

⁸ ANDROID FOR DEVELOPERS, Manifest.permission.

<https://developer.android.com/reference/android/Manifest.permission>, 2024 (last accessed 25 January 2024)

```

24  <uses-permission
25      ="android.permission.ACCESS_FINE_LOCATION" />
26
27  <uses-permission
28      ="android.permission.ACCESS_NETWORK_STATE" />
29
30  <uses-permission
31      ="android.permission.READ_CONTACTS" />
32
33  <uses-permission
34      ="android.permission.READ_EXTERNAL_STORAGE" />
35
36  <uses-permission
37      ="android.permission.CAMERA" />
38
39  <uses-permission
40      ="android.permission.READ_PHONE_STATE" />

```

Fig.4. A portion of the permissions stated in PayPal's *Androidmanifest.xml* fileTable 1. Examples of *Dangerous* Protection-Level Android Permissions and Their Descriptions

Permission: A Constant Value in <i>AndroidManifest.xml</i> that Begins with a prefix "android.permission."	Description
"GET_ACCOUNTS"	Allows an app to access the list of accounts in the Accounts Service
"ACCESS_FINE_LOCATION"	Allows an app to access the precise location
"READ_EXTERNAL_STORAGE"	Allows an app to read from external storage
"WRITE_EXTERNAL_STORAGE"	Allows an app to write to external storage
"READ_CONTACTS"	Allows an app to read the user's contacts data
"WRITE_CONTACTS"	Allows an app to write the use's contacts data
"READ_SMS"	Allows an app to read Short Message Service (SMS) messages
"SEND_SMS"	Allows an app to send SMS messages
"RECEIVE_SMS"	Allows an app to receive SMS messages
"READ_PHONE_STATE"	Allows read only access to phone state, including the current cellular network information, the status of any ongoing calls, and a list of any phone numbers registered on the device

2.4. Why PayPal in Particular?

In this paper, PayPal Android-based online payment gateway mobile app is used because of its widespread popularity. As reported in [17], PayPal is the first payment gateway service provider for financial services worldwide with over 100 million download counts in January 2024⁹. PayPal allows online fund transfer amongst individuals and businesses. Its services are available in more than 200 countries. It is capable of

⁹ GOOGLE PLAY STORE. PayPal – Send, Shop, Manage

<https://play.google.com/store/apps/details?id=com.paypal.android.p2pmobile&hl=en&gl=US>, 2024 (last accessed 27 January 2024)

dealing with 25 currencies¹⁰. PayPal's Android-based latest app version 8.55.1 APK file can be downloaded from APKFlash¹¹. APK files can also be downloaded from other websites, such as APKPURE¹², APK-DL¹³, and APKCombo¹⁴. Additionally, apps' APK files can be downloaded from Google Play Store using a Google Chrome extension called "APK Downloader". This extension can be installed from Chrome Web Store¹⁵.

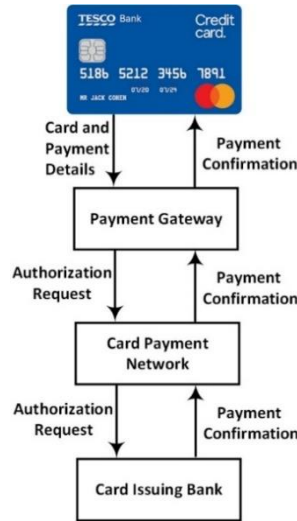


Fig.5. Parties involved in an online payment gateway

3. Related Work

3.1. Deep Learning

Deep learning analyzes features extracted from the app (e.g., code, permissions, and network traffic) using complex neural networks to identify malicious behavior patterns. Garg and Baliyan [3] attempted to match malicious software affecting Android with vulnerabilities with different severity levels. In order to detect malicious software attacks, features extracted from Android apps were mined with transformer models (XLNET and BERT). The generated features were employed to implement methods based on deep learning (TextCNN, RNN, and MLP). The goal was to gauge the severity of malicious software with regard to unexploited vulnerabilities at early stages of Android apps development. Alecakir and Sen [18] used attention mechanisms in deep neural architectures to model the discrepancies between an Android app's description in the Android marketplace, and the actual granted permissions when the app is installed. The objective was to identify suspect mobile apps. Rathore et al [19] carried out a feature analysis to determine the important Android permissions, and offer an effective deep learning and machine learning based Android malware detection engine. The proposed solution requires less time to train and test while maintaining a high level of model accuracy. However, it was noticed that deep neural networks achieve accuracy that is comparable to the baseline values, but at a significant computational cost. Kim et al. [20] presented a model to detect malicious software in Android-based execution environments. Seven attributes of an Android-based app were identified and correlated to feature types that were used to train the initial deep neural network. Thousands of

¹⁰ PAYPAL. About Us. https://www.paypal.com/eg/webapps/mpp/about?locale.x=en_EG, 2024 (last accessed 27 January 2024)

¹¹ APKFLASH. PayPal. <https://apkflash.com/apk/app/com.paypal.android.p2pmobile/paypal>, 2024 (last accessed 27 January 2024)

¹² APKPURE. <https://apkpure.net>, 2024 (last accessed 27 January 2024)

¹³ APK-DL. Android APK Store. <https://apk-dl.com>, 2024 (last accessed 27 January 2024)

¹⁴ APKCOMBO. Download APF – Latest Version. <https://apkcombo.com>, 2024 (last accessed 27 January 2024)

¹⁵ CHROME WEB STORE. APK Downloader. <https://chromewebstore.google.com/detail/apk-downloader/glngapejbnmnicccdcemghaoapdji?pli=1>, 2024 (last accessed 27 January 2024)

malicious and benign app samples were used to train the final network. Authors claim that their model achieved 98% in detecting malicious apps.

3.2. Static Analysis

Static analysis examines the app's code and resources without executing it. It identifies potential vulnerabilities based on predefined rules and patterns. In order to address the problem of being overprivileged, Xiao et al [15] suggests a method that combines collaborative filtering accompanied by static analysis to determine the minimal permissions for an Android app. This method is based on the app description and its Application Programming Interface (API) usage. APIs allow apps to access mobile devices' hardware and system resources. The proposed method first uses collaborative filtering to determine the app's initial minimum set of permissions. Eventually, the final set of minimal permissions that an app actually needs are then determined through static analysis. Darvish and Husain [21] analyzed the security posture of a collection of payment gateway apps, where it concluded that 80% of these apps were found vulnerable to different types of threats. The paper also developed a guide for checking Android apps security.

3.3. Dynamic Analysis

Dynamic analysis executes the app in a controlled environment, and monitors its behavior (e.g., network traffic, and file system access). It detects malicious actions the app might perform at runtime. Diamantaris et al [22] presents a dynamic analysis system that tracks permission requests made by an Android app in real-time as part of its core functionality, and separates those permission requests from requests made by third-party libraries linked with the Android app. The objective was to counter confidential information leakage attacks committed by third-party libraries linked to Android apps. The study found that 65% of the permissions requested by multiple Android apps were requested by third-party libraries linked to those apps rather than from the core functionality of those apps. Rubio-Medrano et al [23] aimed at preventing data leakage by detecting malicious permission-abusing mobile apps. They presented their security framework to restrict the behavior of such apps at run-time. Their proposed framework was built on top of Android Enterprise that allowed users and administrators to specify and enforce Counter-Policies without having previous technical security background.

3.4. Code Obfuscation

Code obfuscation in Android apps adds a layer of protection by making the code harder to understand and tamper with. It obstructs static analysis, and makes it more difficult to identify vulnerabilities. Several studies explore obfuscation's effectiveness in hindering reverse engineering, intellectual property theft, and malware analysis [24]. However research also acknowledges potential downsides like increased app size, performance impact, and debugging challenges. Other studies analyze and compare various obfuscation techniques, including name obfuscation, control flow obfuscation, and string encryption [25]. Additionally, studies explore newer approaches like using machine learning for dynamic obfuscation or leveraging hardware-based security features [26].

4. The Proposed Security Approach

This section presents the technical contribution of this paper. The objective is to detect unauthorized modifications to genuine permissions of legitimate Android-based mobile apps. This article presents an applied research on PayPal app to achieve the desired goal. The proposed security approach aims at maintaining Android-based mobile apps' integrity by detecting unauthorized modifications to the permissions declared in *AndroidManifest.xml* file embedded in these apps in real-time. It is called Android Apps Permissions Integrity Verifier (AAPIV). This section consists of four subsections that explain: (1) the attack vector; (2) identification of the security vulnerability that attackers could exploit; (3) the functionality of the proposed security approach; and (4) the integrity attack scenario on PayPal mobile app, and the accompanying experimental security evaluation of the proposed security approach.

4.1. Attack Vector

Android-based mobile apps are available for installation from Google Play Store. As mentioned earlier in subsection 2.4, apps' APK files can be downloaded from multiple sources. An adversary downloads a legitimate Android-based app APK file from Google Play Store (using "APK Downloader") to his/her PC/laptop. The adversary decompiles the downloaded APK file using Android Studio. He/she maliciously inserts extra *dangerous* protection-level permissions to *AndroidManifest.xml* to create a malicious fake mobile app, compiles it, and uploads the resulting APK file back to Google Play Store with the *same* legitimate app name, but with a *different* APK file name. An incautious customer installs the malicious and fake app on his Android-powered device. Accordingly, the victim is subject to numerous severe negative consequences. Accordingly, the situation ends up in adversaries gaining highly privileged *dangerous* permissions over victimized systems' resources, besides permissions needed to interact with other systems installed on mobile devices. Unfortunately, this attack vector can be carried out with no need for sophisticated methods or tools. For instance, GITHUB¹⁶ is a free tool; it can be used to bypass Android app signature and integrity checks. Fig.6 depicts the attack vector.

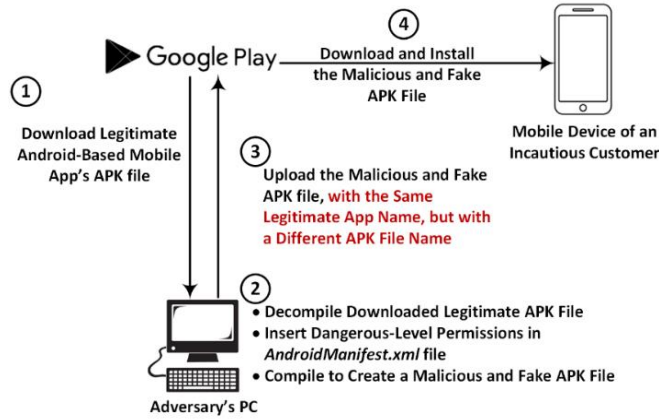


Fig.6. The attack vector

4.2. Identification of the Exploitable Security Vulnerability

Security vulnerabilities refer to defects or weaknesses in the design, implementation, operation, or management of a system that could be exploited to violate the system's security policy [5]. A vulnerability in a system could be exploited to obtain unauthorized access to, or compromise the system. An application without binary protection can be readily analyzed, altered, or back-engineered by an adversary [27]. The vulnerability that makes the mentioned attack vector applicable and viable is non-existence of binary protection in Android-based apps. An Android app can be easily decompiled to access its source code. Malicious source code and additional *dangerous* protection-level permissions can easily be inserted in contents of Android-based apps' APK files.

4.3. Functionality of the Proposed Security Approach

Fig.7 depicts AAPIV's process of capturing, computing, and storing the authentic unique 256-bit hash of the *AndroidManifest.xml* file of PayPal's legitimate Android-based payment gateway mobile app. The process begins by decompiling the legitimate PayPal app's APK file. The next step is to extract the *AndroidManifest.xml* file and apply the one-way irreversible Secure Hash Algorithm-256 (SHA-256). This algorithm is used to generate a unique constant 256-bit output message digest/hash that distinctly identifies

¹⁶ GITHUB, Android-Signature-And-Integrity-Check-Bypass.

<https://github.com/riyadmondol2006/Android-Signature-And-Integrity-Check-Bypass/releases/tag/V2>, 2024 (last accessed 30 January 2024)

the arbitrary-length *AndroidManifest.xml* file. Finally, the generated hash is inserted in AAPIV's cloud-based database server to be used to verify the genuineness of the *AndroidManifest.xml* file. A hash value is a distinct value that corresponds to a file's content. Altering any character in a file's contents changes the file's hash value. Hash values are used to assert that a file's contents were not subject to any modifications. Hash values are used to check whether two files have identical contents.

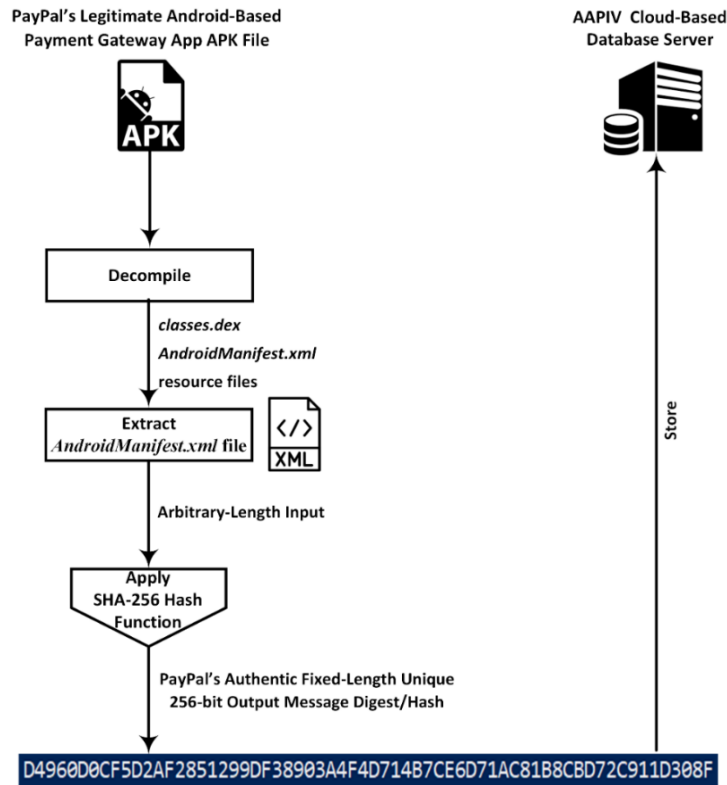


Fig.7. AAPIV capturing and storing the authentic unique 256-bit hash of the *Androidmanifest.xml* file of PayPal's legitimate Android-based payment gateway mobile app

As depicted in Fig.8, for every access request to the data stored in the database server of the payment gateway service provider (e.g., PayPal), a database-level trigger (stored procedure) is fired automatically to call AAPIV. A database-level trigger could be a BEFORE INSERT trigger, BEFORE UPDATE trigger, or BEFORE DELETE trigger. AAPIV captures, extracts, and computes the 256-bit hash of the *AndroidManifest.xml* file of the requesting app. It verifies the computed hash against that stored in AAPIV's cloud-based database server for authenticity. In case both hashes are identical, this denotes a legitimate access request from an authentic payment gateway mobile app, and accordingly the access request is allowed, otherwise the access request is denied. This proposed security approach guarantees to a high extent that sensitive financial data is only accessible by the legitimate payment gateway app. Due to the fact that SHA-256 has not yet been cracked [28], it is adopted in AAPIV. SHA-256 was published by the National Institute of Standards and Technology [29]. Through SHA-256, reconstruction of an input message that matches a specified output message digest/hash is computationally impossible. In order to determine whether an input message has changed after its digest was output, a message digest/hash is used. Additionally, SHA-256 is used to generate pseudo-random 256-bit hashes [30].

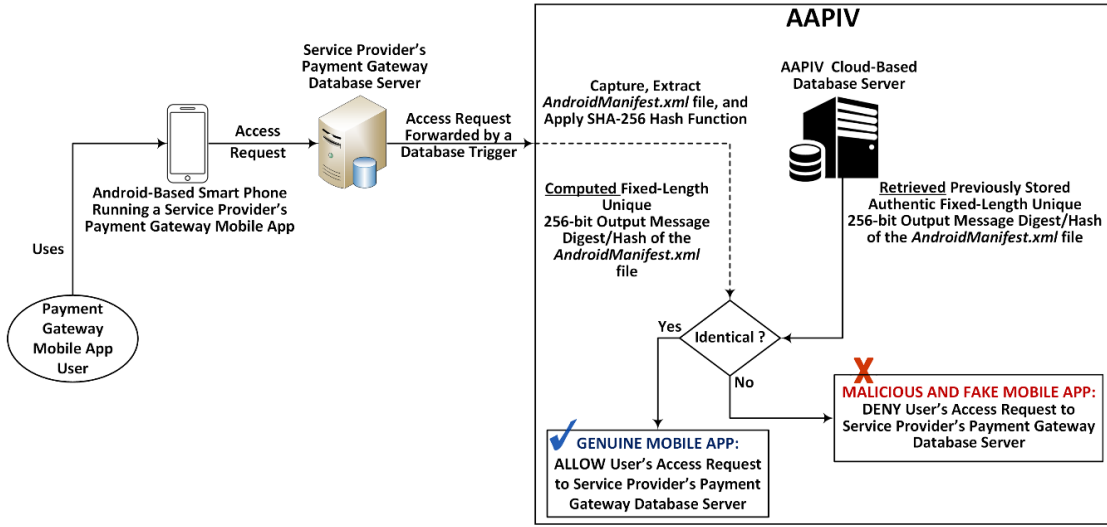


Fig.8. AAPIV's *AndroidManifest.xml* file authenticity verification process

4.4. Integrity Attack Scenario on PayPal and the Proposed Security Approach Experimental Security Evaluation

This subsection presents the implementation of an integrity attack scenario on permissions of Android-Based PayPal gateway mobile app. Additionally, it illustrates the experimental security evaluation of the proposed security approach on PayPal payment gateway Android-based mobile app.

4.4.1. Integrity Attack Scenario on PayPal

The attack scenario is implemented by an adversary carrying out the following sequence of steps:

- Download PayPal's APK file from Google Play Store (using "APK Downloader") to his/her PC/laptop.
- Decompile the downloaded APK file using Android Studio.
- Insert extra *dangerous* protection-level permissions to *AndroidManifest.xml* to create a malicious fake PayPal mobile app.
- Compile and upload the resulting APK file back to Google Play Store with the *same* legitimate app name (i.e., PayPal), but with a *different* APK file name.
- Eventually, an incautious app user installs the malicious and fake PayPal mobile app on his/her Android-powered device, thereby exposing him/herself to numerous severe negative financial consequences.

4.4.2. Experimental Security Evaluation of the Proposed Security Approach

AAPIV's experimental security evaluation is presented in a proof-of-concept illustration to demonstrate the core idea. It is applied on PayPal Android-based payment gateway mobile app. The integrity of a file can be verified using *Powershell* command shell. The *Get-FileHash* cmdlet¹⁷ from the *Powershell* computes the hash value/message digest of a given file by using a specified hash algorithm. This cmdlet supports computing the message digest using any of the following Secure Hash Algorithms (SHAs): SHA1, SHA256, SHA384, SHA512, and the Message-Digest algorithm (MD5).

¹⁷ MICROSOFT POWERSHELL UTILITY, Get-FileHash.

<https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/get-filehash?view=powershell-7.2>, 2024 (last accessed 30 January 2024)

For AAPIV to generate the authentic unique 256-bit output hash of PayPal's legitimate *AndroidManifest.xml* file, it downloads PayPal's APK file from Google Play Store, as this store is the most trusted source of legitimate mobile apps. This is accomplished using "APK Downloader" Google Chrome extension. This extension can be installed from Chrome Web Store. Through AAPIV, PayPal's APK file is decompiled. Next, AAPIV extracts PayPal's *AndroidManifest.xml* file, and hashes it using SHA-256. This is accomplished from within AAPIV using the *Powershell Get-FileHash* cmdlet. The authentic fixed-length unique 256-bit output message digest/hash of PayPal's legitimate *AndroidManifest.xml* file is then stored in AAPIV's cloud-based database server.

On PayPal's database server, a database-level trigger (stored procedure) is fired automatically to call AAPIV whenever it receives an access request; that is, whenever data is inserted (before INSERT trigger), modified (before UPDATE trigger), or deleted (before DELETE trigger). For every access request to the data stored in the PayPal's database server, the 256-bit hash of the *AndroidManifest.xml* file of the requesting app is captured, extracted, computed, and verified for authenticity against that stored in AAPIV's cloud-based database server. In case both hashes are identical, this denotes a legitimate access request from an authentic PayPal app, and accordingly the access request is allowed, otherwise the access request is denied. This approach guarantees to a high extent that sensitive financial data is only accessible by the genuine PayPal payment gateway mobile app. Fig.9 depicts a portion of PayPal's authentic *AndroidManifest.xml* file contents. Fig.10 depicts the hash/message digest of PayPal's authentic *AndroidManifest.xml* file using *Get-FileHash* cmdlet from the *Powershell* command shell. The hash is identical to that shown in Fig.7.

Fig.11 depicts a portion of PayPal's *Androidmanifest_Modified.xml* file contents, where an attacker inserted an additional permission (lines 24 and 25). This is an integrity attack on permissions of PayPal app. The inserted permission "android.permission.WRITE_CONTACTS" is categorized as a *dangerous* protection-level permission¹⁸. Fig.12 depicts the 256-bit hash of PayPal's *AndroidManifest_Modified.xml* file using *Get-FileHash* cmdlet from the *Powershell* command shell. From Fig.10, Fig.11, and Fig.12, it is apparent that inserting an additional permission in *AndroidManifest_Modified.xml* file of a fake and malicious PayPal app (Fig.11) resulted in a totally different 256-bit hash (Fig.12) than that of PayPal's authentic *Androidmanifest.xml* file 256-bit hash (Fig.10). As previously explained, AAPIV verifies the authenticity of a mobile app (e.g., PayPal) by comparing both 256-bit hashes (i.e., the authentic previously stored against that of the requesting-to-access app). Since they are different (Fig.10 and Fig.12), AAPIVT will deny the access request issued from the fake and malicious PayPal app containing *AndroidManifest_Modified.xml* file.

```

24      <uses-permission
25          ="android.permission.ACCESS_FINE_LOCATION" />
26
27      <uses-permission
28          ="android.permission.ACCESS_NETWORK_STATE" />
29
30      <uses-permission
31          ="android.permission.READ_CONTACTS" />
32
33      <uses-permission
34          ="android.permission.READ_EXTERNAL_STORAGE" />

```

Fig.9. A portion of the authentic PayPal's *Androidmanifest.xml* file contents

¹⁸ WRITE_CONTACTS.

https://developer.android.com/reference/android/Manifest.permission#WRITE_CONTACTS, 2024 last (accessed 30 January 2024)

```
PS C:\> Get-FileHash C:\PayPal\AndroidManifest.xml -Algorithm SHA256 | Format-List

Algorithm : SHA256
Hash      : D4960D0CF5D2AF2851299DF38903A4F4D714B7CE6D71AC81B8CBD72C911D308F
Path      : C:\PayPal\AndroidManifest.xml
```

Fig.10. 256- bit hash of the authentic PayPal's *Androidmanifest.xml* file

```

24 <uses-permission
25     ="android.permission.WRITE_CONTACTS" />
26
27 <uses-permission
28     ="android.permission.ACCESS_FINE_LOCATION" />
29
30     <uses-permission
31         ="android.permission.ACCESS_NETWORK_STATE" />
32
33     <uses-permission
34         ="android.permission.READ_CONTACTS" />
35
36     <uses-permission
37         ="android.permission.READ_EXTERNAL_STORAGE" />

```

} Unauthorized
Permission
Added (Integrity
Attack)

Fig.11. A portion of the fake and malicious PayPal's *Androidmanifest_Modified.xml* file contents

```
PS C:\> Get-FileHash C:\PayPal\AndroidManifest_Modified.xml -Algorithm SHA256 | Format-List

Algorithm : SHA256
Hash      : 18024D0EB2202C2E54214E61E552474F933F757F116A1EDDD8784B02BCAD12B7
Path      : C:\PayPal\AndroidManifest_Modified.xml
```

Fig.12. 256-bit hash of the fake and malicious PayPal's *Androidmanifest_Modified.xml* file

5. Novelty of the Proposed Security Approach and its Merits

With reference to Related Work (Section 3), the proposed security approach (AAPIV) presented in this paper is novel in the sense that no other research tackled the problem of safeguarding genuine permissions of legitimate Android-based mobile apps in real-time as AAPIV did. Table 2 illustrates the the novelty of AAPIV as compared to previous related work: (1) deep learning; (2) static analysis; (3) dynamic analysis; and (4) code obfuscation. As detailed in subsection 4.4.2., the proof-of-concept illustration of AAPIV's experimental security evaluation demonstrated that through its file integrity verification capability, it is capable of achieving 100% detection accuracy of integrity attacks on permissions of Android-based mobile apps. As depicted in Fig.13, deep learning models achieve 85% malware detection accuracy, but require significant training data and computational resources. Static analysis achieves 70% malware detection accuracy; it is faster but can be fooled by code obfuscation. Dynamic analysis achieves 80% malware detection accuracy; it offers a good balance between deep learning and static analysis. However, it requires a secure sandbox environment. Fig.13 illustrates the accuracy of AAPIV compared to other Android-based malware detection techniques.

Table 2. AAPIV as Opposed to Deep Learning, Static Analysis, Dynamic Analysis, and Code Obfuscation

Technique	Description	Advantages	Disadvantages	Use Case in Mobile App Scanning
Deep Learning [3][18][19][20]	Analyzes features extracted from the app using complex neural networks to identify malicious behavior patterns	Effective at detecting novel malware	Requires large datasets for training. It can be computationally expensive. It may produce opaque results	Flags previously unknown attacks. Identifies complex malware behavior
Static Analysis [15][21]	Examines the app's code and resources without executing it. Identifies potential vulnerabilities based on predefined rules and patterns	Fast, lightweight, and identifies common issues early in development	Limited to detecting known vulnerabilities; however, it may miss complex malware that relies on runtime behavior	Identifies insecure coding practices. Detects usage of malicious permissions
Dynamic Analysis [22][23]	Executes the app in a controlled environment, and monitors its behavior. Detects malicious actions the app might perform at runtime	Can uncover vulnerabilities missed by static analysis. Provides insights into app behavior	Time-consuming, resource-intensive, and may miss well-hidden malware that avoids suspicious actions during analysis	Identifies malware that downloads malicious payloads. Detects apps that exhibit suspicious network behavior
Code Obfuscation [24][25][26]	Technique used by developers to intentionally obscure the app's code, and making it harder to understand and analyze	Protects intellectual property, and hinders reverse engineering	Hinders static analysis. Can be bypassed by sophisticated malware analysis tools	Makes static analysis less effective, and may be used by malware authors to hinder detection
File Integrity Verification Through AAPIV	AAPIV is this paper's novel proposed security approach to detect unauthorized modifications to genuine permissions of legitimate Android-based mobile apps	Detailed in the following subsections	Compared to older and no more secure hashing algorithms like MD5, SHA-256 takes slightly more processing power (20%-30%) and time to calculate the hash ¹⁹	Flags previously unknown attacks. Detects Android-based mobile app malicious permissions

¹⁹ FREECODECAMP. MD5 vs SHA-1 vs SHA-2 - Which is the Most Secure Encryption Hash and How to Check Them

<https://www.freecodecamp.org/news/md5-vs-sha-1-vs-sha-2-which-is-the-most-secure-encryption-hash-and-how-to-check-them/>, 2024 (last accessed 23 April 2024)

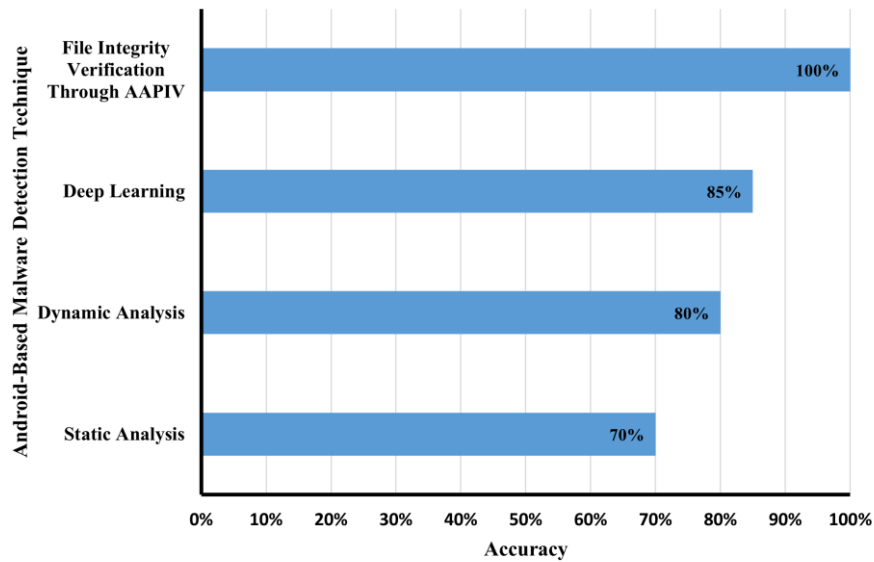


Fig.13. Accuracy of AAPIV as compared to other Android-based malware detection techniques

The following subsections explain the advantages of applying AAPIV.

5.1. Cover Non-Existence of Binary Protection Vulnerability

Android-based mobile apps suffer from non-existence of binary protection. This vulnerability opens the door wide open to adversaries. As explained earlier, it is always possible to tamper with contents of APK files as these files lack binary protection. With the adoption of AAPIV, this vulnerability is covered. Through AAPIV’s integrity verification process on *AndroidManifest.xml*, it is possible to detect unauthorized modifications to its contents. AAPIV heavily contributes in preventing violation of Android-based apps’ authentic permissions, and usage of tampered-with malicious mobile apps that may lead to financial fraud.

AAPIV can be applied on any Android-based mobile app, especially apps that manage financial transactions, such as InstaPay. InstaPay²⁰ is an Egyptian mobile app that allows instant money transfer between bank accounts or mobile phone numbers, as long as the involved banks are part of the InstaPay’s network. It links a user’s bank accounts from participating banks into one app, and allows transferring money instantly between linked bank accounts. The similarities between both apps, PayPal and InstaPay, lay in that they require similar core permissions like Internet access for online transactions and communication. However, there are several differences between PayPal and InstaPay as shown in Table 3.

5.2. Provide Anti-Circumvention Security Approach

AAPIV provides anti-circumvention capability. A mobile app service provider’s database-level trigger (i.e., stored procedure) is fired automatically to call AAPIV whenever data is inserted, modified, or deleted using the mobile app. For every access request to the data stored in the database server of the mobile app service provider, the 256-bit hash of the *AndroidManifest.xml* file of the requesting app is captured, extracted, computed, and verified for authenticity against that stored in AAPIV’s cloud-based database server. By no means a mobile app user would be able to circumvent or bypass such hash verification requirement. This

²⁰ INSTAPAY
<https://www.instapay.eg/?lang=en>, 2024 (last accessed 21 April 2024)

guarantees to a high extent that sensitive data is only accessible by legitimate mobile apps. AAPIV adopted Secure Hash Algorithm-256 (SHA-256) rather than SHA-512 for a number of reasons: (1) SHA-256 is superior over SHA-512 in its processing speed; (2) SHA-256 is considered secure for most current applications due to the fact that SHA-256 has not yet been cracked [28]; and (3) National Institute of Standards and Technology (NIST)²¹ encourages usage of SHA-256 especially for applications that require file integrity verification using hash values generated from hash functions. Table 4 summarizes the differences between SHA-256 and SHA-512.

Table 3. PayPal versus InstaPay

Feature	PayPal	InstaPay
Region	Global	Egypt
Account Funding	Can be linked to bank accounts, credit cards, debit cards	Requires linked bank accounts from participating Egyptian banks
Money Transfer	International transfers possible	Between Egyptian bank accounts, and transfer money to other InstaPay users using their mobile phone number
Bill Payment	Wide variety of billers worldwide	Limited to Egyptian utilities and telecommunication companies
Availability	Widely available	Requires Egyptian banks to be part of the InstaPay network
Location Permission	Request location permission for features like finding nearby stores or Automatic Teller Machines (ATMs)	Does not require location permission for its functionalities
Telephony	Does not require phone numbers, as the primary focus is on emails	Require access to phone numbers for sending money using mobile contacts

Table 4. SHA-256 versus SHA 512

Feature	SHA-256	SHA-512
Hash Output Size	256 bits	512 bits
Security Level	Offers collision resistance up to 128 bits. Considered secure for most current applications	Offers collision resistance up to 256 bits. More secure for cryptanalysis
Processing Speed	Faster due to smaller hash output size	Slower due to larger hash output size, and more complex internal operations
Suitable Applications	Widely used for file integrity verification, digital signatures, password hashing, , and other scenarios where a strong and compact hash is needed.	Ideal for applications such as digital certificates, and blockchain transactions

²¹ NIST. Hash Functions. NIST Policy on Hash Functions

<https://csrc.nist.gov/projects/hash-functions/nist-policy-on-hash-functions>, 2024 (last accessed 21 April 2024)

5.3. Provide User-Transparent Functionality

Through AAPIV's *AndroidManifest.xml* file integrity verification process, user transparency is provided. That is, users of Android-based mobile apps (both benign users and attackers) would not notice that the apps that they are currently using to access the mobile app service providers databases are being verified for legality and authenticity.

5.4. Adoption of Defense-in-Depth Security Principle

The defense-in-depth security principle is based on layering of security measures [31]. Layering aims at mitigating potential security risks. In order to safeguard information assets, a number of overlapping security defenses are placed accumulatively. So that if one security safeguard was breached, still there exists another security barrier to defend against attackers. AAPIV is a security countermeasure that adopts the defense-in-depth security principle. Even if a mobile app user enters the correct credentials (the first line/layer of defense), AAPIV will additionally verify the genuineness of permissions granted to the requesting-to-access mobile app (the second line/layer of defense). This ensures that integrity attacks using malicious and fake mobile apps that were compromised and maliciously modified are caught by AAPIV as a second security countermeasure. AAPIV's verification function compares the authentic unique 256-bit hash of the *AndroidManifest.xml* file of a legitimate Android-based mobile app hash against that of the possibly fake malicious mobile app that is currently being used. Accordingly, it allows or denies the access request.

5.5. Real-Time Identification of Malicious Mobile Apps

AAPIV can identify attackers in real-time through its effective and precise verification functionality. AAPIV captures and computes the authentic unique 256-bit hash of the *AndroidManifest.xml* file of a legitimate Android-based mobile app. An app's permissions are registered in *AndroidManifest.xml* file in its Android Package Kit file. AAPIV stores the computed hash in its cloud-based database server. For every access request to the data stored in the database server of the mobile app service provider, the 256-bit hash of the *AndroidManifest.xml* file of the requesting app is captured, extracted, computed, and verified for authenticity against that stored in AAPIV's cloud-based database server. In case both hashes are identical, this denotes a legitimate access request from an authentic mobile app, and accordingly the access request is allowed, otherwise the access request is denied.

5.6. Reinforcement of Mobile Apps Users' Trust

Through AAPIV's orientation of layered-based security defenses, an Android-based mobile app is more immune against integrity attacks on its embedded permissions. Eventually, AAPIV boosts the users' trust in the effectiveness of the adopted security defenses integrated in the Android-based mobile app.

6. Conclusion

The objective of this paper is to detect unauthorized modifications to genuine permissions of legitimate Android-based mobile apps. This objective was met through a proposed approach called Android Apps Permissions Integrity Verifier (AAPIV). The main idea behind AAPIV is capturing, computing, and storing the authentic unique 256-bit hash of *AndroidManifest.xml* file that contains all the permissions of an Android-based mobile app. This authentic hash is used to verify the genuineness of permissions granted to a requesting-to-access mobile app in real-time. It is computed by applying the one-way irreversible Secure Hash Algorithm-256 (SHA-256). In a proof-of-concept illustration applied on Android-based PayPal payment gateway mobile app, the experimental security evaluation demonstrated that AAPIV achieved its intended objective. The proposed security approach presented in this paper is novel in the sense that no other research tackled the problem of safeguarding genuine permissions of legitimate Android-based mobile apps in real-time as AAPIV did. The scientific value of this work lies in finding a remedy for lack of binary protection vulnerability in Android-based mobile apps. AAPIV provides several merits: (1) cover non-existence of binary protection vulnerability; (2) provide anti-circumvention security approach; (3) provide user-transparent functionality; (4) adoption of defense-in-depth security principle; (5) real-time identification of malicious

mobile apps; and (6) reinforcement of mobile apps users' trust. Compared to older and no more secure hashing algorithms like Message Digest Method 5, SHA-256 takes slightly more processing power (20%-30%) and time to calculate the hash. However, this might be a minor consideration for most tasks. As for future work, it is intended to focus on testing mobile apps running in Apple's iPhone Operating System (iOS) execution environment.

References

- [1] D. O. Sahin, S. Akleylek, and E. Kilic, "LinRegDroid: detection of android malware using multiple linear regression models-based classifiers," *IEEE Access*, vol. 10, pp. 14246–14259, Jan. 2022.
- [2] O. Hussein, "A proposed anti-fraud authentication approach for mobile banking apps," in *Proc. 4th Novel Intelligent and Leading Emerging Sciences Conf. (NILES)*, Giza, Egypt, 2022, pp. 56-61.
- [3] S. Garg and N. Baliyan, "M2VMapper: malware-to-vulnerability mapping for android using text processing," *Expert Syst. with Applications*, vol. 191, Article 116360, Apr. 2022.
- [4] G. Renjith, and S. Aji, "Unveiling the security vulnerabilities in android operating system," in *Proc. Second Int. Conf. on Sustainable Expert Syst. (ICSSES)*, Singapore, 2021, pp. 89-100.
- [5] A. Girma, A., A. Guo, and J. Irungu, J., "Identifying shared security vulnerabilities and mitigation strategies at the intersection of application programming interfaces (APIs), application-level and operating system (OS) of mobile devices," in *Proc. The Future Technologies Conf. (FTC)*, Cham, ,2022, pp. 499-513.
- [6] J. Tang et al., "NIVAnalyzer: a tool for automatically detecting and verifying next-intent vulnerabilities in android apps," in *Proc. 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Tokyo, Japan, 2017, pp. 492-499.
- [7] 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. 11-20, Mar. 2020.
- [8] Z. Alshara, A. Shatnawi and Y. Jararweh, "NIV-Detector: an automated approach for detecting next-intent security vulnerability in android applications," in *Proc. 2022 Ninth Int. Conf. on Softw. Defined Syst. (SDS)*, Paris, France, 2022, pp. 1-7.
- [9] Z. Wang, C. Li, Y. Guan, Y. Xue and Y. Dong, "ActivityHijacker: hijacking the android activity component for sensitive data," in *Proc. 2016 25th Int. Conf. on Comput. Commun. and Networks (ICCCN)*, Waikoloa, HI, USA, 2016, pp. 1-9.
- [10] M. Al-Fawa'reh, A. Saif, M. T. Jafar, and A. Elhassan, "Malware detection by eating a whole APK," in *Proc. 32nd Int. Conf. for Internet 2124 Technol. Secured Trans. (ICITST)*, Dec. 2020, pp. 1-7.
- [11] Cho I. Cho, D. Towey and P. Kar, "Using obfuscators to test compilers: a metamorphic experience," *2023 IEEE 47th Annu.Computers, Softw., and Applicat. Conf. (COMPSAC)*, Torino, Italy, 2023, pp. 1786-1791.
- [12] Y. L. Arnatovich, L. Wang, N. M. Ngo, and C. Soh, "A comparison of android reverse engineering tools via program behaviors validation based on intermediate languages transformation," *IEEE Access*, vol. 6, pp. 12382–12394, Feb. 2018.
- [13] G. Nolan, *Decompiling Android*. Apress, 2012.
- [14] H. H. R. Manzil and M. S. Naik, "COVID-Themed Android Malware Analysis and Detection Framework Based on Permissions," *2022 Int. Conf. for Advancement in Technology (ICONAT)*, Goa, India, 2022, pp. 1-5.
- [15] J. Xiao, S. Chen, Q. He, Z. Feng, and X. Xue, "An Android application risk evaluation framework based on minimum permission set identification," *J. of Syst.and Softw*, Vol. 163, pp. 110533, May 2020.
- [16] 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.
- [17] FINDEXABLE LIMITED. (2020) "The Global Fintech Index 2020" [Online]. Available: https://fintechznz.org.nz/wp-content/uploads/sites/5/2019/12/Findexable_Global-Fintech-Rankings-2020.pdf.
- [18] H. Alecakir, B. Can, and S. Sen, "Attention: there is an inconsistency between Android permissions and application metadata!," *Int. J. Inform. Security*, vol. 20, no. 6, pp. 797–815, Jan. 2021.
- [19] H. Rathore, S. K. Sahay, R. Rajvanshi, and M. Sewak, "Identification of significant permissions for efficient android malware detection," in *Proc. Int. Conf. on Broadband Commun., Networks and Syst.*, Cham, Switzerland, 2020, pp. 33–52.
- [20] T. Kim, B. Kang, M. Rho, S. Sezer, E. Im, "A multimodal deep learning method for Android malware detection using various features," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 773-788, Mar. 2019.
- [21] H. Darvish and M. Husain, "Security analysis of mobile money applications on android," in *Proc. 2018 IEEE Int. Conf. on Big Data (Big Data)*, Seattle, WA, USA, 2018, pp. 3072-3078.
- [22] M. Diamantaris, E. P. Papadopoulos, E. P. Markatos, S. Ioannidis and J. Polakis, "Reaper: Real-time app analysis for augmenting the android permission system," in *Proc. the Ninth ACM Conf. on Data and Applicat. Security and Privacy*, Richardson, TX, USA, 2019, pp. 37-48.
- [23] C. Rubio-Medrano, P. Soundrapandian, M. Hill, L. Claramunt, J. Beak, G. Ahn, "DyPolDroid: Protecting against permission-abuse attacks in android," *Information Systems Frontiers*, vol. 25, pp. 529-548, Oct. 2023.

- [24] X. Zhang, F. Breitering, E. Luechinger, S. O'Shaughnessy, "Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations," *Forensic Science International: Digital Investigation*, vol. 39, article 301285, Dec. 2021.
- [25] V. Balachandran, D. Tan, V. Thing, "Control flow obfuscation for Android applications," *Computers & Security*, vol. 61, pp. 72-93, Aug. 2016.
- [26] C. Yadav, S. Gupta, "A review on malware analysis for IoT and Android system," *SN Computer Science*, vol. 4, no. 118, Dec. 2022.
- [27] S. Bojjagani and V. N. Sastry, "VAPTAI: A threat model for vulnerability assessment and penetration testing of Android and iOS mobile banking apps," in *Proc. IEEE 3rd Int. Conf. Collaboration Internet Comput. (CIC)*, Oct. 2017, pp. 77-86.
- [28] S. Rawal, L. Maganti, and V. Godha, "Comparative study of SHA-256 optimization techniques," in *Proc. 2022 IEEE World AI IoT Congress (AIoT)*, Seattle, Washington, USA, 2022, pp. 387-392.
- [29] National Institute of Standards and Technology - *Federal Information Processing Standards Publication - Secure Hash Standard (SHS)*, FIPS PUB 180-4, 2015.
- [30] O. Hussein, "A proposed impregnable 256-bit hash producer," in *Proc. 15th Int. Comput. Eng. Conf. (ICENCO)*, Cairo, Egypt, 2019, pp. 50-55.
- [31] R. Savold, N. Dagher, P. Frazier, and D. McCallam, "Architecting cyber defense: a survey of the leading cyber reference architectures and frameworks," in *Proc. 2017 IEEE 4th Int. Conf. on Cyber Security and Cloud Computing (CSCloud)*, New York, NY, USA, 2017, pp. 127-138.


الكشف عن الهجمات على سلامة أذونات التطبيقات المحمولة المستندة إلى أندرويد: تقييم أمني على باي بال

عمر حسين

قسم نظم المعلومات الإدارية – كلية علوم الإدارة – جامعة أكتوبر للعلوم الحديثة والآداب (MSA) –

مدينة ٦ أكتوبر – جمهورية مصر العربية

ohusseins@gmail.com

 <https://orcid.org/0000-0002-0282-7541>

الهدف من هذه الورقة البحثية هو عرض نهج أمني مقترح للكشف في الوقت الحقيقي عن التعديلات الغير مصرح بها على أذونات التطبيقات المحمولة المستندة إلى نظام التشغيل أندرويد، مع تقييم النهج الأمني المقترح على تطبيق بوابة الدفع باي بال. تكمن القيمة العلمية لهذه الورقة البحثية في إيجاد علاج للثغرة الأمنية بتطبيقات الأجهزة المحمولة المستندة إلى نظام التشغيل أندرويد المتمثلة في عدم وجود حماية ثنائية لكود المصدر بتلك التطبيقات. الدافع وراء إجراء هذا البحث على باي بال تحديداً هو شعبيته الواسعة، وتزايد الهجمات التي تستهدف تطبيقات أندرويد، إلى جانب الطبيعة الحساسة لتطبيقات بوابات الدفع من خلال الأجهزة المحمولة. تقترح هذه الورقة نهجاً أمنياً باسم "مدقق سلامة أذونات تطبيقات أندرويد" لمكافحة التحاليل وتحقيق الهدف المنشود.

الفكرة الرئيسية وراء النهج الأمني المقترح هي التقاط وحوسبة وتخزين التجزئة الأصلية الفريدة للملف الذي يحوى جميع أذونات تطبيق الجهاز المحمول المستند إلى نظام التشغيل أندرويد. يتم استخدام هذا التجزئة الأصلية للتحقق من صحة الأذونات الممنوحة للتطبيق محل الاستخدام في الوقت الفعلي. تم تقييم النهج المقترح على تطبيق بوابة الدفع باي بال. أظهر التقييم الأمني التجريبي أن النهج المقترح حقق هدفه المنشود في الكشف في الوقت الحقيقي عن التعديلات الغير مصرح بها على أذونات التطبيقات المحمولة المستندة إلى نظام التشغيل أندرويد. بعد النهج الأمني المقترح المقدم في هذه الورقة جديداً، حيث أنه لم يتم من قبل حل مشكلة حماية الأذونات الأصلية الشرعية للتطبيقات المحمولة المستندة إلى نظام التشغيل أندرويد في الوقت الفعلي كما حققه النهج الأمني المقترح في هذا البحث. تكمن القيمة العلمية لهذا البحث في إيجاد حل لمشكلة عدم وجود حماية ثنائية لكود المصدر بتطبيقات الأجهزة المحمولة لمستندة إلى نظام التشغيل أندرويد.

القسم الأول بهذا البحث يعطى مقدمة عن نظام التشغيل أندرويد مع ذكر أمثلة لبعض التطبيقات المحمولة الحميدة والخبيثة المستندة إلى نظام التشغيل أندرويد. يغطي القسم الثانى بهذا البحث الخلفية المفاهيمية من خلال الأقسام الفرعية التالية: (١) عمليات تجميع وتفكيك تطبيقات أندرويد، (٢) الأذونات في التطبيقات المستندة إلى نظم التشغيل أندرويد والمخاطر المحتملة، (٣) وظائف تطبيقات بوابة الدفع، و(٤) لماذا باي بال على وجه الخصوص؟. القسم الثالث بهذا البحث يغطي البحوث السابقة ذات الصلة، والتي تم تجميعها وفقاً إلى: (١) التعلم العميق، (٢) التحليل الساكن، (٣) التحليل الديناميكي، و(٤) تشويش الكود. القسم الرابع يعرض تفاصيل الجوانب المختلفة للنهج الأمني المقترح بما في ذلك التقييم الأمني التجريبي المطبق على تطبيق بوابة الدفع باي بال. يناقش القسم الخامس حداثة النهج الأمني المقترح ومزاياه. أخيراً، يختتم القسم السادس هذه الورقة البحثية ويحدد العمل المستقبلي.

يوفر النهج الأمني المقترح العديد من المزايا: (١) تغطية ثغرة أمنية المتمثلة في عدم وجود حماية ثنائية لكود المصدر بتطبيقات الأجهزة المحمولة المستندة إلى نظام التشغيل أندرويد، (٢) توفير نهج أمني لمكافحة التحاليل، (٣) توفير الخواص الأمنية بطريقة شفافة للمستخدم النهائي، (٤) اعتماد مبدأ الأمن الدفاعي العميق، (٥) الكشف في الوقت الحقيقي عن تطبيقات الأجهزة المحمولة الضارة، و(٦) تعزيز ثقة مستخدمي تطبيقات الأجهزة المحمولة المستندة إلى نظام التشغيل أندرويد. بالنسبة للعمل المستقبلي، فهو يهدف إلى التركيز على إختبار تطبيقات الأجهزة المحمولة التي تعمل بنظام التشغيل أى فون الخاص بشركة أبل.