

Android App Copy Protection Mechanism with Semi-trusted Loader

Kuo-Yu Tsai*

* Department of Management Information Systems, Hwa Hsia University of Technology,
No.111, Gongzhuang Rd., Zhonghe Dist., New Taipei 235, Taiwan
kytsai@cc.hwh.edu.tw

Abstract— In 2014, Tsai *et al.* propose an Android App copy protection mechanism based on dynamic loading, in which an Android App is composed of an incomplete main App (IMA for short, such as an APK file) file and a separate essential class (SES for short, such as a Jar file)file. After purchasing an App, a mobile user firstly downloads the APK file from the Android market. Then, the mobile user installs the APK file in his/her mobile device. As the mobile user attempts to run the App, the embedded dynamic loading function will download the Jar file from the market and execute dynamic loading for all functionalities. The dynamic loading function will delete the SES file as all functionalities are loaded. However, the mobile user has to download the Jar file every time as he/she wants to execute all functionalities of the Android App. In addition, the dynamic loading function might be replaced with another malicious loading function. In this paper, we propose an Android App Copy protection mechanism with a semi-trusted loader. In our proposed mechanism, an Android App is also composed of an APK file and a Jar file. At the first execution time, the embedded semi-trusted loader will download the encrypted Jar file from the market and the corresponding decryption key for the Jar file. Then, the semi-trusted loader decrypts the Jar file by using the decryption key and executes the loading for all functionalities. After the loading, the semi-trusted loader will delete the decryption key and store the encrypted Jar file in the mobile device. After that, the semi-trusted loader only download the decryption key from the market as the mobile user wants to execute all functionalities of the App.

Keywords— Android, App, Copy Protection, Dynamic Loading, Semi-trusted Loader

I. INTRODUCTION

Gartner's report points out that the downloaded times of mobile applications will be more than 268 billion, and the App market scale will be more than \$77 billion by 2017. However, piracy is a serious issue for Apps. Google recently announced two security mechanisms to solve various piracy problems for Android applications, including the file permissions control mechanism and the license verification library mechanism. In the file permissions control mechanism, an App's APK file will be stored in the /data/app folder after the Android App is installed in a mobile user's device. The folder can only be accessible by the application [1, 3]. In the license verification library mechanism, a mobile user attempts to run his/her Android App, and it will submit a query to the Google license server for obtaining the license status. If the license status is

available, it allows the mobile user to run the App [1, 3]. However, the above mechanism cannot resist against the root attack on Android. Suppose that a malicious mobile user attempts to root his/her device to get the folder permission. After getting the permission, he/she can disassemble the APK and modify the APK file to disable the licensing service.

By using the asymmetric encryption technique, Moon *et al.* [7] proposed a copyright protection system for the Android platform. Suppose that a mobile user purchases an Android App and downloads the APK file from the Android market. The APK files was encrypted by using the mobile user's public key. When the mobile user obtains the encrypted APK file, he/she can decrypt the file by using his/her private key. Then, the mobile user can install the APK file, and runs the application. Moon *et al.* claimed that this mechanism can prevent the App form be used by an illegal mobile user. However, the APK file is stored in the /data/app folder after installation, and hence a malicious mobile user may illegally replicate the APK file. In the same year, Jeong *et al.* [6] proposed an anti-privacy mechanism with class separation and dynamic loading, in which an Android App consists of an incomplete main App file (IMA for short) and a separate essential class file (SES for short). Suppose that a mobile user purchases an App, and downloads the IMA file from the Android market. The mobile user will obtain the encrypted SES file from the Android market when he/she runs the Android application for the first time. The mobile user can decrypt the encrypted SES file, and then SEC files is dynamic loaded for execution. In addition, the SES file will be stored in the secure space. However, Jeong *et al.*'s proposed mechanism cannot resist against the root attack on the Android platform. A malicious mobile user may root his/her mobile device, and hence access the secure space for replicating the decrypted SES file. Inspired from Moon *et al.*'s mechanism [7] and Jeong *et al.*'s mechanism [6], Tsai *et al.* [8] proposed an Android App copy protection mechanism with the dynamic loading function. In Tsai *et al.*'s proposed mechanism, a complete Android App also includes a SES file and an IMA file. Suppose that a mobile user purchases an Android App, and then he/she can download the IMA file from the Android market. If the mobile user wants to runs the Android App, he/she first downloads the SES file from the Android market. Then, the dynamic loading function will load the SES file for enabling all functionalities. Finally, the

dynamic loading function will delete the SES file after the loading. However, Tsai *et al.*'s proposed mechanism cannot also resist against the root attack on Android. Suppose that a malicious mobile user attempts root his/her device to replicate the APK. He/she can disassemble the APK and modify the dynamic loading function of the APK file to disable the deletion capability. Hence, the malicious user can replicate the SES file, and illegally distribute the SES file. In addition, the mobile user has to download the encrypted SES file as he/she wants to run the Android App. Other related Android App protection works are proposed [2, 4, 5].

In this paper, we propose an Android App Copy protection mechanism with a semi-trusted loader. In our proposed mechanism, we adopt the signature scheme to protect the embedded semi-trusted loader. At the first running time, the embedded semi-trusted loader will download the encrypted SES file from the market and the corresponding decryption key for the encrypted SES file. Then, the semi-trusted loader decrypts the encrypted SES file by using the decryption key and executes the loading for all functionalities. After the loading, the semi-trusted loader will delete the decryption key and the SES file and store the encrypted SES file in the mobile device. After that, the semi-trusted loader only download the decryption key from the market as the mobile user wants to execute all functionalities of the App.

II. OUR PROPOSED MECHANISM

Our proposed consists four phases: **Initialization and User Registration Phase**, **Android App Uploading Phase**, **Purchase Phase**, **App First Running Phase**, and **App Second Running Phase**. There are four kinds of roles in our proposed mechanism: **Mobile User (MU for short)**, **App Market (AM for short)**, **App Security Enhancer (ASE for short)**, and **App Developer (AD for short)**. Detailed descriptions are as follows.

A. Initialization and User Registration Phase:

In this phase, a mobile user MU registers with the AM by submitting his/her identity ID_{MU} , password PW_{MU} , and international mobile equipment identity ($IMEI$ for short) via a secure channel. Upon receiving the registration information, the AM generates a license UL_{MU} for the mobile user MU and two session keys $\{K_{AM}, K_{ASE}\}$ for the MU. The AM sends $\{UL_{MU}, K_{AM}, K_{ASE}\}$ to the user via a secure channel and stores them.

B. Android App Uploading Phase:

An AD develops a complete App composed of a separate essential class (SES for short) and an incomplete main App (IMA for short), where the SES is a Jar file, and the IMA is an APK file. The APK file includes the semi-trusted loader (STD for short) and the characteristics $Char_{STD}$ for the STD. The STD is responsible for loading a SES and some basic functionalities and deleting secure keys and the decrypted SES. The SES includes the other additional functionalities. The AD performs the following steps.

Step 1. Generate a signature $Sig_{AD} = S_{AD}(H(APK || AID || Char_{STD}))$ for the App, where $S_{AD}()$ is a signing algorithm with AD's private key, and AID is the App

identity, and $Char_{STD}$ is the characteristics for the STD.

Step 2. Upload the IMA and the App identity AID into the AM and the SES and Sig_{AD} into the ASE, respectively.

C. Purchase Phase:

Suppose that a mobile user MU wants to purchase an App. The detailed steps are as follows.

Step 1. The MU has to encrypt the AID and his/her own UL_{MU} with his/her secret key K_{AM} and sends $\{E_{K_{AM}}(UL_{MU} || AID || PW_{MU} || IMEI || N_{MU}), ID_{MU}\}$ to the AM, where $E_{K_{AM}}()$ is a symmetric encryption with K_{AM} , the K_{AM} is shared between the MU and the AM, and N_{MU} is a nonce chosen by the MU.

Step 2. Upon receiving the information, the AM performs the decryption $D_{K_{AM}}(E_{K_{AM}}(UL_{MU} || AID || PW_{MU} || IMEI || N_{MU}))$ to obtain UL_{MU} , AID , PW_{MU} , and N_{MU} , where $D_{K_{AM}}()$ is a symmetric decryption with K_{AM} . After that, the AM performs the following verification processes.

Step 2-1 Verify if the nonce N_{MU} is used. If it is new, then continue to perform Step 2-2; otherwise, reject to perform the following step and return the failed information.

Step 2-2 Verify if the decrypted UL_{MU} and the corresponding PW_{MU} are valid. If both are valid, the AM generates a signature $Sig_{AM} = S_{AM}(H(APK || AID || UL_{MU} || IMEI || N_{AM}))$, where $H()$ is a hash function, and APK stands for the APK file for the App; otherwise, reject the request and return the failed information.

Step 2-3 Encrypt the APK and the signature Sig_{AM} with K_{AM} , and then sends $E_{K_{AM}}(APK || Sig_{AM} || IMEI || N_{AM})$ to the MU.

Step 3. After reviving the data, the MU performs the following steps.

Step 3-1 Decrypt $E_{K_{AM}}(APK || Sig_{AM} || IMEI || N_{AM})$ to the obtain APK , Sig_{AM} , $IMEI$, and N_{AM}

Step 3-2 Verify if the nonce N_{AM} is used. If N_{AM} is new, then continue to perform Step 3-3; otherwise, reject the received messages.

Step 3-3 Perform $V_{AM}(S_{AM}(H(APK || AID || UL_{MU} || IMEI || N_{AM})))$ with the AM's public key, where $V_{AM}()$ is a signature verification algorithm. If the signature is valid, then continue to the next step; otherwise, reject the received messages.

Step 3-3 Install the APK in his/her mobile device.

D. App First Running Phase:

At the first time, the MU attempts to execute all functionalities of his/her purchased App, the STD embedded in the App will perform the authentication process.

Step1. The STD retrieves characteristics $Char_{STD}$ and generates the encryption $E_{K_{ASE}}(UL_{MU} || Sig_{AM} || AID || IMEI || Sig_{AD} || Char_{STD} || N'_{MU})$, and sends $\{E_{K_{ASE}}(UL_{MU} || Sig_{AM} || AID || IMEI || Sig_{AD} || Char_{STD} || N'_{MU})\}$ to the ASE, where N'_{MU} is a nonce for the transmission performed by the MU, and K_{ASE} are shared between the MU and the ASE.

- Step 2. Upon receiving the information, the ASE decrypts $E_{K_{ASE}}(UL_{MU} || Sig_{AM} || AID || IMEI || Sig_{AD} || Char_{STD} || N'_{MU})$ and performs the following steps.
- Step 2-1 Verify if the nonce N'_{MU} is used. If it is new, then continue to perform Step 2-2; otherwise, reject to perform the following step and return the failed information.
- Step 2-2 Perform $V_{AM}(S_{AM}(H(APK || AID || UL_{MU} || IMEI || N_{AM})))$ with the AM's public key. If the signature is valid, then continue to the next step; otherwise, reject the received messages.
- Step 2-3 Retrieve the $Char_{STD}$ from the STD and verifies Sig_{AD} . If it is the same as the $Char_{STD}$ uploaded by the AD, the ASE generates the signature $Sig_{ASE} = S_{ASE}(H(UL_{MU} || AID || SC || IMEI || K_i || N_{ASE}))$ and the encryption $E_{K_{ASE}}(SC || Sig_{ASE} || K_i || N_{ASE})$, and sends $\{E_{K_{ASE}}(SC || Sig_{ASE} || K_i || N_{ASE})\}$ to the user, where N_{ASE} is a nonce for the transmission performed by the ASE, and SC is a separated class; otherwise, reject the request.
- Step3. After receiving $\{E_{K_{ASE}}(SC || Sig_{ASE} || K_i || N_{ASE})\}$, the STD decrypts $E_{K_{ASE}}(SC || Sig_{ASE} || K_i || N_{ASE})$ and performs the following steps.
- Step 3-1 Verify if the nonce N_{ASE} is used. If it is new, then continue to perform Step 3-2; otherwise, reject to perform the following step and return the failed information.
- Step 3-2 Perform $V_{ASE}(S_{ASE}(H(UL_{MU} || AID || SC || IMEI || K_i || N_{ASE})))$ with the ASE's public key. If the signature is valid, then continue to the next step; otherwise, reject the received messages.
- Step 3-3 As the functionalities are loaded, the STD encrypts the SC and Sig_{ASE} with the key K_i , and further deletes the SC and K_i in the mobile device. The STD only stores $E_{K_i}(SC || Sig_{ASE})$ in the secure space. A malicious mobile user only replicate $E_{K_i}(SC || Sig_{ASE})$ even though he/she has rooted his/her device. However, the malicious mobile cannot obtain SC because he/she does not have the description key K_i .

E. App Second Running Phase:

After the first time for running the Android App, MU wants to execute all functionalities of his/her purchased App, the STD embedded in the App will perform the authentication process.

- Step1. the STD generates $E_{K_{ASE}}(UL_{MU} || Sig_{AM} || AID || IMEI || Sig_{AD} || Char_{STD} || N''_{MU})$, and sends $\{E_{K_{ASE}}(UL_{MU} || Sig_{AM} || AID || IMEI || Sig_{AD} || Char_{STD} || N''_{MU})\}$ to the ASE, where N''_{MU} is a nonce for the transmission performed by the MU, and K_{ASE} are shared between the MU and the ASE.
- Step2. Upon receiving the information, the ASE performs the same verification processes (Steps 2-1~2-3) in **App First Running Phase**.
- Step 3. If all are valid, it means that the App execution request is new, the MU is a legal user with his/her specified mobile device, and the STD is valid. Hence,

the ASE generates the encryption $E_{K_{ASE}}(T_{ASE} || K_i)$, and sends $E_{K_{ASE}}(T_{ASE} || K_i)$ to the user, where T_{ASE} is a timestamp for the transmission performed by the ASE; otherwise, the ASE rejects the App execution request.

- Step3: After receiving $E_{K_{ASE}}(T_{ASE} || K_i)$, the STD decrypts $E_{K_{ASE}}(T_{ASE} || K_i)$ to obtain K_i . Then, the STD uses K_i to decrypt $E_{K_i}(SC || Sig_{ASE})$ and allows the MU to execute all functionalities of the App; otherwise, the dynamic loading function stops the execution.

III. DISCUSSIONS AND ANALYSES

In comparison of Jeong *et al.*'s mechanism [6], Moon *et al.*'s mechanism [7] and Tsai *et al.*'s mechanism [8], our proposed mechanism can resist the root attack on the Android platform. That is, any malicious user cannot execute all functionalities of the App even though he/she copy any the APK or encrypted class file. In additional, the proposed mechanism achieves the following security properties.

- 1) Resistance to illegal copy: In our experiment, the STD can dynamically load and delete the SC and decryption key K_i . Only the encrypted SC is stored in the mobile device. Hence, a malicious user cannot illegally obtain the SC even though he/she has rooted his/her mobile device.
- 2) Unforgeability: There are two cases to analyze (1) Suppose that an adversary attempts to cheat the ASE by sending a forged signature $Sig_{AM} = S_{AM}(H(APK || AID || UL_{MU} || IMEI || N_{AM}))$. However, the adversary cannot successfully forge Sig_{AM} without the AM's private key. (2) Consider that an adversary may try to cheat the STD by using a forged $Sig_{ASE} = S_{ASE}(H(UL_{MU} || AID || SC || IMEI || K_i || N_{ASE}))$. However, the adversary cannot succeed because he/she does not have the ASE's private key.
- 3) Resistance to the man-in-the middle attack: Consider that an adversary attempts to intercept the transmitted messages and decrypt the encrypted messages in **App Purchase** and **App Execution Phases**. However, he/she cannot obtain the decrypted data without K_{AM} and K_{ASE} .
- 4) Resistance to the replay Attack: Suppose that an adversary want replay the intercepted message. However, he/she cannot success due to each transmitted message includes a nonce.

IV. CONCLUSIONS

In this paper, we propose an Android App Copy protection mechanism with a semi-trusted loader. In our experiment, the semi-trusted loader can be regarded as the semi-trusted server embedded the mobile devices. If an adversary attempts to modify the loader, the ASE can detect the modification attack by verifying the $Char_{STD}$.

ACKNOWLEDGMENT

The authors gratefully acknowledge the support from Taiwan Information Security Center (TWISC) and National Science Council under the grants 103-2221-E-146 -005 -MY2 and 103-2221-E-011 -090 -MY2 in Taiwan.

REFERENCES

- [1] C.Y. Chuang, Y.C. Wang, and Y.B. Lin, "Digital right management and software protection on Android phones," IEEE 71st Vehicular Technology Conference, pp. 1-5, 2010.
- [2] S. Choi, J. Jang, and E. Jae, "Android application's copyright protection technology based on forensic mark," ACM Research in Applied Computation Symposium, pp. 338-339, 2012.
- [3] Google, "Application Licensing," <http://developer.android.com/google/play/licensing/index.html> , 2014.
- [4] J. Jang, J. Jung, H. Ji, J. Hong, D. Kim, and S. K. Jung, "Protecting android applications with steganography based software watermarking," ACM Symposium on Applied Computing, pp. 106-120, 2013.
- [5] H. Ji and W. Kim, "Design of a mobile inspector for detecting illegal Android applications using fingerprinting," 2013 Research in Adaptive and Convergent Systems, 2013.
- [6] Y. S. Jeong, J. C. Moon, D. Kim, Y. U. Park, S. J. Cho, and M. Park, "An anti-piracy mechanism based on class separation and dynamic loading for Android applications," ACM Research in Applied Computation Symposium, pp. 328-332, 2012.
- [7] Y. C. Moon, J. H. Noh, A. R. Kim, and S. R. Kim, "Design of copy protection system for Android platform," International Conference on Information Technology, System and Management, 2012.
- [8] K. Y. Tsai, Y. H. Chu, and T. C. Wu, "Android App copy protection mechanism based on dynamic loading," IEEE International Symposium on Consumer Electronics, 2014.



Kuo-Yu Tsai was born in 1976 in Taiwan. He is an Assistant Professor at the Department of Management Information Systems, Hwa Hsia University of Technology, Taiwan. He received his Ph.D. Degree in the Department of Information Management from National Taiwan University of Science and Technology, Taiwan, in 2009. His recent research interests include information security, cryptography, network security, and cloud computing.