# Etat de l'Art des Techniques d'Unpacking pour les Applications Android

Pierre Graux, Jean-François Lalande, Valérie Viet Triem Tong

# État de l'Art des Techniques d'Unpacking pour les Applications Android

Pierre Graux, Jean-François Lalande, Valérie Viet Triem Tong
CentraleSupélec, Univ Rennes, Inria, CNRS, IRISA
F-35000 Rennes
{prenom.nom}@inria.fr

***Résumé*—Le nombre d'applications malveillantes Android grandissant, de nombreux systèmes de détection et d'analyse ont été développés. Pour contrer ces méthodes, les malware se sont tournés vers des techniques de dissimulation. L'une d'entre-elles, le "packing", a pour but d'empêcher l'analyse statique du bytecode de l'application en le chiffrant. Cet article présente un état de l'art des différentes techniques existantes pour détecter l'utilisation d'une telle protection ainsi que des méthodes permettant d'obtenir le code original de l'application.**

## I. Introduction

Avec 84% de part de marché en 2016, Android est le système d'exploitation le plus utilisé dans le domaine des mobiles [1]. Cependant, le parc de téléphones Android n'est globalement pas à jour puisqu'en 2018 plus de 20% des téléphones actifs fonctionnent encore sous une version datant de 2015 (Marshmallow) [2]. Il est donc privilégié par les malware : chaque jours environ 8400 nouveaux malware sont recensés [3].

C'est pourquoi de nombreux systèmes de détection et d'analyse ont vu le jour [4]. Afin d'empêcher le bon fonctionnement de ces outils, les malware se sont donc tournés vers des techniques de dissimulation de comportements malveillants. Parmi-elles, l'emploi de packers tend à se renforcer [5]. Ce type d'outil, initialement prévu pour empêcher le repackaging d'application, est également utilisé pour empêcher l'analyse statique du bytecode [6]. En effet, le packing consiste à chiffrer le bytecode de l'application qui n'est alors déchiffré qu'au moment de l'exécution.

Afin de permettre aux systèmes de détection et d'analyse de fonctionner correctement, il est donc nécessaire de pouvoir détecter l'utilisation de packers et d'extraire automatiquement le bytecode original de l'application. Cet article constitue un court état de l'art des techniques spécifiques à Android permettant de réaliser ces deux tâches. Il met notamment en avant les problématiques qui ne sont pas encore complètement résolues, à savoir l'automatisation complète de ce processus ainsi que la suppression du bytecode du packer au sein du bytecode extrait.

Dans la suite de l'article, nous rappelons les particularités d'Android dans la section II. Nous définissons les packers dans la section III. Nous présentons un état de l'art des méthodes de détection dans la section IV et des méthodes d'unpacking dans la section V. Leurs limitations sont présentées dans la section VI.

## II. Structure d'une application Android

Android est un système d'exploitation utilisé dans le domaine mobile. Il est basé sur le noyau Linux auquel il ajoute un ensemble de piles logiciels nommé Android Open Source Project (AOSP). AOSP comprend notamment un environnement d'exécution d'applications et de services dédiés du système d'exploitation.

Une application est un fichier `APK` ("`.apk`") qui est une archive `ZIP` contenant :

- un manifest : fichier lu lors de l'installation de l'application qui permet de définir les permissions ainsi que les différentes activités et services de l'`APK`.
- un dossier de ressources ("`res/`") : dossier contenant les images, les textes, les polices de caractère, etc. de l'application.
- un fichier `DEX` ("`.dex`") : fichier résultant de la compilation d'un code Java en bytecode Dalvik.
- un dossier de bibliothèques natives ("`lib/`") : dossier contenant des bibliothèques compilées pour une architecture spécifique.

Cet environnement propose notamment deux fonctionnalités largement utilisées par les packers Android. Il permet d'une part d'exécuter du code natif présent dans une bibliothèque partagée. Le code natif possède les même droits que le code Dalvik et peut donc interférer sur ce dernier. De plus il permet de charger dynamiquement du code. Il est par exemple possible de charger, au cours de l'exécution, un fichier `DEX` secondaire ou une nouvelle classe.

## III. Packers Android

Un packer est un outil qui a pour but de rendre plus complexe la rétro-analyse d'un programme tout en conservant son comportement global, en chiffrant tout ou une partie du code du programme.

Les packers ne sont pas spécifiques à Android et ont donc déjà été largement étudiés, notamment pour l'environnement `x86` [7]. Ces travaux étudient les effets des packers au niveau du système d'exploitation. Cependant, ils ne sont pas applicables à Android [5, 6, 8]. En effet, AOSP introduit un

niveau supplémentaire entre le noyau et l'application qui doit nécessairement être pris en compte.

Le fonctionnement d'un packer Android consiste à modifier une application, appellée application originale, en une nouvelle application, appellée application packée. Ce processus de modification peut être découpé en deux phases. Dans un premier temps, il chiffre le code original c'est à dire le fichier `DEX` de l'application originale qui est dès lors appelé `DEX` packé. Puis, il ajoute une routine de déchiffrement c'est à dire un nouveau fichier `DEX` qui se charge de déchiffrer le fichier `DEX` packé puis de le charger dynamiquement lors de l'exécution. Cette routine est généralement implémentée au sein d'une bibliothèque native.

Le fichier `DEX` d'une application packée ne contient pas le bytecode original mais celui de la routine de déchiffrement. Analyser statiquement ce fichier `DEX` conduirait à rétro-analyser la routine de déchiffrement ce qui serait couteux et difficile.

C'est pourquoi la suite de l'article détaille les techniques spécifiques à Android qui permettent de détecter les applications packées dans la section IV et de les analyser dans la section V.

## IV. Détection d'Applications Packées

De nombreux services de packers existent en ligne [1]. Ils proposent à leurs utilisateurs de déposer une application sur leur plate-forme afin de leur renvoyer l'application packée correspondante. Chaque service possède son propre packer et il est donc possible de lister pour chacun d'entre eux un ensemble d'artefacts permettant de détecter leur présence. Par exemple, il est possible de repérer les noms des classes ou des fichiers qui sont insérés dans l'application packée [6, 9]. Cette méthode est très précise puisqu'elle permet de déterminer quel packer a été utilisé, mais elle ne permet pas de détecter de nouveaux packers.

Afin de détecter les packers non-connus, une première méthode consiste à détecter le comportement de l'unpacker [6]. Classiquement il utilise l'appel système `mprotect` afin de rendre inscriptible la zone mémoire du fichier `DEX` packé avant d'utiliser des appels systèmes de manipulation de la mémoire pour le déchiffrer. Ce comportement n'est pas un comportement classique d'application Android. Il est donc possible d'intercepter, au niveau du système d'exploitation, l'utilisation de tels appels systèmes afin de détecter, lors de l'exécution d'une application, l'utilisation d'un tel schéma.

Une seconde méthode utilise les propriétés du fichier `Manifest` [10], cf. section II. En effet, celui-ci permet de définir les activités et les services de l'application. Or il n'est lu par Android qu'au moment de l'installation de l'application. Il n'est donc pas possible, pour un packer, de supprimer les références aux classes packées qui y sont présentes. Si une application est packée, il est donc probable que certaines classes présentes dans le Manifest ne soit pas détectées lors d'une analyse statique. C'est ce marqueur qui est utilisé pour détecter une application packée.

Une dernière méthode généralise l'approche précédente [11]. En effet elle propose de comparer le résultat d'une analyse statique à celle d'une analyse dynamique d'une application. Lors de l'analyse statique, le fichier DEX est décompilé pour obtenir l'ensemble des noms des classes. L'analyse dynamique instrumente la fonction `Dalvik_dalvik_system_DexFile_defineClassNative` afin de récupérer le nom de chaque classe chargée à l'exécution. Si une classe non détectée lors de l'analyse statique, est détectée lors de l'exécution de l'application, c'est que cette application est packée.

## V. Récupération automatique de code d'applications packées

La récupération automatique de code d'applications packées est appelée unpacking. Cette section retrace l'évolution des packers ainsi que des techniques d'unpacking automatiques associées. Les techniques manuelles c'est-à-dire consistant à comprendre le fonctionnement complet de la routine de déchiffrement pour appliquer la fonction inverse à tout le fichier `DEX` packé ne sont pas traitées puisqu'elles varient pour chaque packer et qu'elles présentent donc un problème de passage à l'échelle. Le tableau I présente une liste des différents unpackers existants.

Les routines de déchiffrement des premiers packers déchiffrent totalement le fichier `DEX` packé puis utilisent le framework Android pour charger dynamiquement l'ensemble du fichier `DEX` dépacké [12, 13]. Le fichier `DEX` original est donc présent en mémoire lors de l'exécution de l'application. Ainsi, les premières techniques d'unpacking (type I) consistent à exécuter l'application packée. Une fois qu'elle est lancée, l'unpacker recherche en mémoire une signature du fichier `DEX` [12, 13]. Par exemple, il est possible de rechercher son `magic number` c'est à dire les octets caractéristiques du début d'un fichier DEX (`64 65 78 0A 30 33 35 00`). Lorsque le fichier `DEX` original est trouvé, il est récupéré.

Certaines parties d'un fichier `DEX`, comme par exemple le `magic number`, ne sont pas utilisées lors de son chargement par le framework Android. Il est donc possible de les modifier sans changer le comportement de l'application. En altérant ces points spécifiques, certains packers parviennent à empêcher la localisation du fichier `DEX` dépacké rendant ainsi inopérants les unpackers précédemment décrits. Néanmoins, le framework Android connaît la localisation du fichier `DEX` dépacké puisqu'elle lui est fournie lors du chargement dynamique du `DEX`. Les unpackers ont donc choisi de surcharger les fonctions du framework Android responsables du chargement des fichiers `DEX` (type II) [14–16]. Grâce à cette méthode, l'emplacement mémoire du fichier dépacké leur est disponible ce qui leur permet de récupérer le bytecode original de l'application.

Afin d'empêcher que le fichier `DEX` ne soit dépacké entièrement, le comportement des routines de déchiffrement

1. Alibaba Inc. : http://jaq.alibaba.com/, Baidu Inc. : http://app.baidu.com/, Bancle Inc. : https://www.bangcle.com/, Ijiami Inc. : http://www.ijiami.cn/, Qihoo360 : http://dev.360.cn/, Tencent Inc. : https://intl.cloud.tencent.com/product/ms

TABLE I
LISTE DES DIFFÉRENTS UNPACKERS.

| Type | Nom | Code | Remarques |
|---|---|---|---|
| I | *NC* [12] | ✓ | Plugin pour Volatility |
| | *NC* [13] | ✓ | Application Standalone et script GDB |
| II | *NC* [15] | ✗ | Hook de fonctions particulières |
| | DWroidDump [14] | ? | Hook `dvmDexFileOpenFromFd` |
| | *NC* [16] | ✓ | Hook de `OpenAndReadMagic` et `DexFile::DexFile` |
| III | DexHunter [6] | ✓ | Hook `defineClassNative` |
| | AppSpear [5] | ? | Monitoring de la représentation du DEX au sein du framework |
| | CrackDex [10] | ✓ | Appels à `dvmDefineClass` |
| IV | PackerGrind [8] | ✓* | Hooks non prédéfinis |

? : auteur à contacter directement.
✗ : code source non disponible.
✓ : code source disponible.
✓* : code source partiellement disponible.

a par la suite été modifié pour ne plus laisser le fichier DEX complètement dépacké en mémoire. Le DEX est déchiffré par partie. Ainsi une routine de déchiffrement ne déchiffre qu'une fonction ou qu'une classe avant son utilisation puis la re-chiffre ensuite. Les unpackers ont donc également évolué afin d'être capable de récupérer les différentes parties du fichier DEX avant de les assembler (type III) [5, 6, 10]. Ils surchargent les fonctions du framework Android de chargement de classe, de fonction, d'ouverture de fichier DEX, afin de récupérer chaque partie correspondante. À la fin de l'exécution de l'application toutes les parties sont assemblées dans un seul DEX final. Ces techniques, bien qu'automatiques, présentent le désavantage d'obtenir uniquement le code orginal des parties qui sont effectivement exécutées. Cette limitation est un problème bien connu de l'exécution dynamique et fait l'objet d'études spécifiques. Par exemple, CrackDex [10] résout ce problème en simulant le chargement des classes en appelant directement la fonction du framework Android qui en est responsable. Il suppose que le déchiffrement des méthodes est réalisé au moment du chargement de la classe.

Le dernier type de packer apparu sont les packers qui embarquent leur propre copie du framework Android [8]. Ainsi en utilisant leur propre fonction pour charger les différents éléments du fichier DEX ils rendent inopérantes les modifications du framework réalisées par les unpackers. Un seul unpacker est capable de gérer ce type de packer (type IV) [8]. Cependant cet unpacker n'est pas complètement automatique. En effet il propose de réaliser une trace de l'exécution de l'application packée tout en monitorant les modifications mémoires des fichiers DEX présents. En lisant une telle trace, il est alors possible pour un analyste de définir quels sont les "points de collection" à utiliser, c'est-à-dire quelles sont les fonctions à hooker afin de pouvoir récupérer une partie du DEX.

## VI. CONCLUSION

Les packers Android font donc l'objet de nombreuses études [5, 6, 8, 10, 11]. Néanmoins les solutions proposées ne sont pas encore pafaites. La solution à l'état de l'art, PackerGrind [8], n'est pas complètement automatique et requiert l'intervention d'un analyste pour définir les points de collection cohérents avec le packer utilisé. Elle ne peut donc pas être utilisée pour filtrer les applications packées avant leur analyse dans un système automatique.

De plus, aucune solution ne traite du problème d'exclusion du code de la routine de déchiffrement. En effet, le bytecode qui lui correspond n'est pas traité différemment, par le framework Android, du bytecode dépacké. Les solutions d'unpacking récupèrent donc également ce bytecode et l'ajoute au bytecode original. Ce comportement est problématique puisqu'il peut induire des erreurs lors de l'exécution du fichier DEX ainsi reconstitué ce qui n'est pas encore évalué à ce jour.

Ces deux problèmes ne sont pas encore résolus et nécessitent donc une étude plus approfondie.

## RÉFÉRENCES


[1] Statista, "Global market share held by smartphone operating systems from 2009 to 2016," 2018. [Online]. Available : https://www.statista.com/statistics/263453/global-market-share-held-by-smartphone-operating-systems/

[2] Android, "Dashboards," Janvier 2018. [Online]. Available : https://developer.android.com/about/dashboards/index.html

[3] C. Lueg, "8,400 new android malware samples every day," Avril 2017. [Online]. Available : https://www.gdatasoftware.com/blog/2017/04/29712-8-400-new-android-malware-samples-every-day

[4] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, p. 76, 2017.

[5] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu, "Appspear : Bytecode decrypting and dex reassembling for packed android malware," *International Workshop on Recent Advances in Intrusion Detection (RAID)*, Septembre 2015.

[6] Y. Zhang, X. Luo, and H. Yin, "Dexhunter : toward extracting hidden code from packed android applications," *European Symposium on Research in Computer Security (ESORICS)*, Septembre 2015.

[7] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "Sok : Deep packer inspection : A longitudinal study of the complexity of run-time packers," *36th IEEE Symposium on Security and Privacy (SP)*, Mai 2015.

[8] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of android apps," *39th International Conference on Software Engineering (ICSE)*, Mai 2017.

[9] R. Nigam, "Android packers : Separating from the pack," *Hacktivity*, Octobre 2014.

[10] Z. Jiang, A. Zhou, L. Liu, P. Jia, L. Liu, and Z. Zuo, "Crackdex : Universal and automatic dex extraction method," *7th IEEE International Conference on Electronics Information and Emergency Communication (ICEIEC)*, Juillet 2017.

[11] Y. Liao, J. Li, B. Li, G. Zhu, Y. Yin, and R. Cai, "Automated detection and classification for packed android applications," *5th IEEE International Conference on Mobile Services (MS)*, Juillet 2016.

[12] R. Yu, "Android packers : facing the challenges, building solutions," *24th Virus Bulletin International Conference (VB2014)*, 2014.

[13] T. Strazzere and J. Sawyer, "Android hacker protection level 0," *DEF CON 22*, Août 2014.

[14] D. Kim, J. Kwak, and J. Ryou, "Dwroiddump : Executable code extraction from android applications for malware analysis," *International Journal of Distributed Sensor Networks (IJDSN)*, vol. 11, no. 9, p. 379682, 2015.

[15] Y. Park, "We can still crack you ! general unpacking method for android packer (no root)," *Black Hat Asia*, Mars 2015.

[16] A. Bashan and S. Makkaveev, "Unboxing android : Everything you wanted to know about android packers," *DEF CON 25*, Juillet 2017.