Rétro-Ingénierie d'Applications Android

BASTIEN SECHER, ENSIBS, 1ère année cyberdéfense

Abstract - Le nombre de smartphones Android vendus dans le monde atteint des records. La sécurité de ces appareils est devenu un enjeu majeur, pour les particuliers comme pour les entreprises. Le format standardisé des applications Android rend la rétro-ingénierie particulièrement simple à mettre en oeuvre, d'où le fait qu'elles soient le vecteur d'attaque privilégié des pirates. Les grandes notions applicables à la rétro-ingénierie, ainsi que les enjeux et problématiques qui en découlent, permettent de montrer l'inertie qui s'applique dans ce domaine. En quelques années, les méthodes de rétro-ingénierie, à l'instar des contre-mesures, ont évolué très rapidement, mettant hors de course des stratégies qui jusqu'ici avaient fait leurs preuves.

Mots clés : Rétro-Ingénierie, Android, Java, Kotlin, Dalvik, Protection d'applications, Evasion

1 Introduction

En 2018, on recensaient plus d'un milliard de smartphones vendus à travers le monde [4]. Parmi eux, 85% utilisent le système d'exploitation Android [3]. A l'heure où chacun possède un smartphone, y stocke des données à caractère personnel ou professionnel, la sécurité de ces appareils a atteint un niveau d'importance sans précédent. L'un des vecteurs d'attaque principaux sur ces appareils sont les applications. Ces dernières pouvant être codées en Java, Kotlin ou même en C++ si on parle de code natif, les pirates vont chercher à appliquer la rétro-ingénierie pour retrouver le code source à partir de leur exécutable (les DEX files pour Java, Kotlin ou les SO files pour le code natif). Une fois l'accès au code source obtenu, il est possible de chercher des vulnérabilités, des secrets codés en dur, ou encore de modifier le comportement de l'application pour, par exemple, y ajouter un malware.

Afin de présenter l'état actuel de la rétro-ingénierie sous Android, nous commencerons par présenter au lecteur le processus qui a permis d'identifier la littérature de référence de ce domaine. Ensuite, à partir de ces références, il s'agira d'explorer les différentes méthodes de rétro-ingénierie, ainsi que l'efficacité des contre-mesures existantes, en critiquant et en mettant en relief les propos de chacune des références. Nous finirons par exposer les verrous, les problèmes actuels auxquels les acteurs de la rétro-ingénierie - attaquants et développeurs - doivent faire face.

2 Identification des travaux de référence

La méthode de recherche qui a été utilisée pour rédiger cet état de l'art constitue les fondations de ce travail. En effet, toute information exposée dans ce document a fait l'objet de traitements et d'analyses afin d'en vérifier la véracité. Ce traitement a été effectué à partir d'une méthode de recherche scientifique, explicitée ci-après.

La première étape de la recherche de travaux consiste à se construire une base de connaissance, en récoltant un grand nombre d'articles en lien direct avec le domaine étudié. Internet étant une source d'information sans limites, il a d'abord s'agit de filtrer les résultats obtenus afin de ne garder que la documentation qui traite du sujet recherché. Ce processus de récolte est détaillé en figure 1.

Author's address: Bastien Secher, baxytra@protonmail.com, ENSIBS, 1ère année cyberdéfense, Rue André Lwoff, Vannes, France, 56 000.

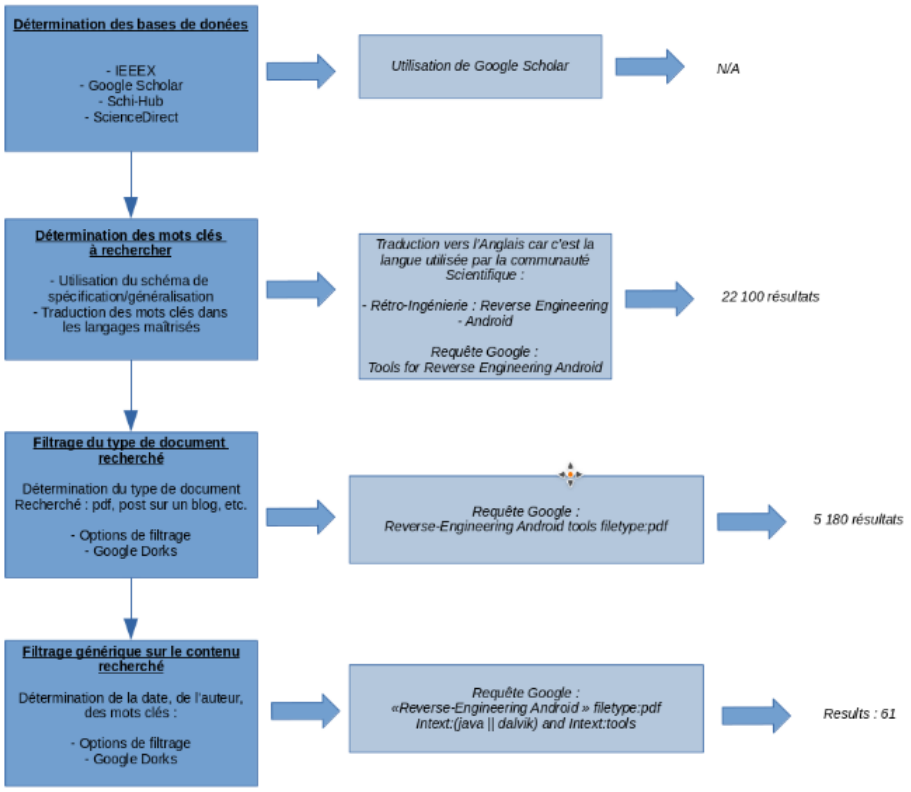

Fig. 1. Processus d'identification des travaux traitant le sujet

Une fois qu'un ensemble de documents traitant du sujet a pu être récupéré, la deuxième étape consiste à filtrer le contenu. Ce filtrage a été effectué en fonction de la confiance que l'on peut porter sur les propos tenus par l'article. Autrement dit, nous avons du évaluer la fiabilité de chacun des articles.

La pertinence d'un article s'évalue d'abord par le respect ou non d'un protocole de démonstration scientifique. En effet, tout article scientifique, pour être considéré comme tel, doit respecter le protocole présenté en figure 2. Aussi, l'article analysé doit connaître la notion d'impartialité. Il ne doit pas se limiter à une seule opinion, sauf si celle-ci est dûment justifiée et argumentée par l'auteur. On interprétera ainsi avec précaution les articles sponsorisés, ou tirés de magazines engagés, notamment lorsque nous présenterons des solutions commerciales.

Enfin, la troisième et dernière étape pour identifier les travaux de référence est de vérifier leur pertinence. La pertinence d'un article repose principalement sur ces deux questions :

- Qui a rédigé l'article ?
- Qui a publié l'article ?

En effet, il est cohérent d'évaluer la pertinence d'un article à partir de celle de l'auteur et/ou de l'éditeur. La renommée de l'auteur pourra notamment être évaluée à partir de son statut : chercheur universitaire ; scientifique d'institut ; amateur reconnu ; mais aussi par la confiance qui lui est accordée au sein de la communauté scientifique. Il s'agit là d'estimer le nombre de fois où l'auteur a été cité dans d'autres articles scientifiques, si tant est qu'ils soient fiables eux aussi. On utilisera alors le h-index, indice permettant de quantifier l'impact d'un auteur. Pour savoir si l'éditeur d'un article est fiable, on peut aussi chercher à savoir si le processus de validation mis en œuvre répond à la fiabilité recherchée : si l'éditeur requiert

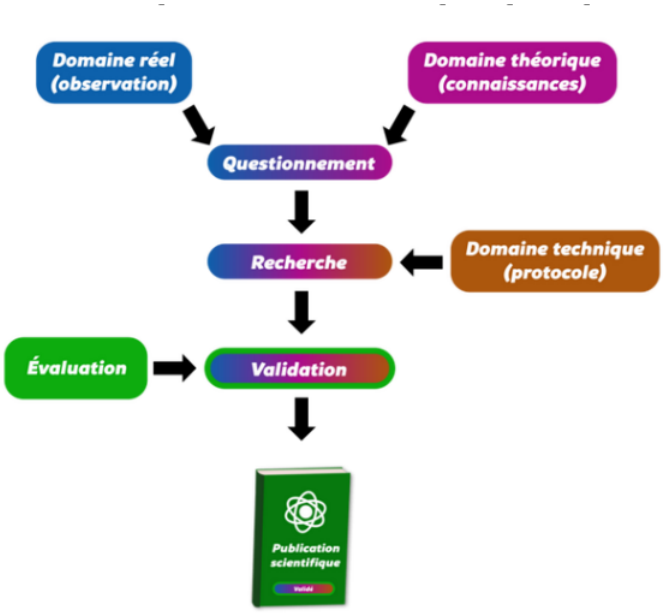

Fig. 2. Processus de filtrage des travaux pertinents

une validation par des pairs avant publication, cela peut révéler une volonté de respecter les protocoles scientifiques. Il existe un indicateur assigné à chaque journal scientifique, pour déterminer quel niveau de confiance on peut lui accorder : le facteur d'impact.

Si tous ces éléments sont respectés, l'article aura pu être analysé plus en profondeur. Notamment, seules les informations présentant une référence bibliographique explicite et vérifiée aura été retenue. Ainsi, la bibliographie doit être exhaustive et suffisante.

3 Présentation des travaux de référence

Cette deuxième partie présente au lecteur de manière structurée et critiquée les travaux de référence du domaine, concernant trois thèmes centraux de la rétro-ingénierie sous Android : les méthodes de rétro-ingénierie, les protections et enfin leur efficacité.

3.1 Les méthodes de rétro-ingénierie

Nous allons donc commencer par présenter les méthodes actuellement utilisées par des attaquants pour exploiter une application. Ces dernières étant nombreuses et parfois promues à tort, la littérature de référence dans ce domaine servira à énumérer et discuter chacune des techniques et outils présentés, pour donner au lecteur un avis critique et nuancé sur le sujet.

3.1.1 Les techniques existantes

L'OWASP, dans son introduction détaillée à la rétro-ingénierie sous Android[25], présente deux grandes familles de méthodes : la rétro-ingénierie statique et la rétro-ingénierie dynamique.

Premièrement la rétro-ingénierie statique consiste à retrouver le code source par analyse des binaires, sans nécessiter l'exécution de l'application. Concrètement, ceci est possible par deux moyens distincs. La première méthode de rétro-ingénierie statique est le disassembling. Il s'agit de retrouver, à partir des fichiers DEX, le code assembleur de l'application, au format SMALI [27]. La seconde est le decompiling.

Il s'agit cette fois non plus de trouver le code assembleur, mais d'obtenir du code Java, plus simple de compréhension. Le decompiling peut se faire en utilisant un format intermédiaire : on décompile les DEX files en JAR puis en Java, en utilisant un Java Decompiler ; ou bien sans format intermédiaire : on décompile les DEX files directement en Java, en utilisant un Dalvik Decompiler. Pau Oliva Fora ajoute, lors de la RSA CONFERENCE en 2014 [15], que le disassembling est la méthode la plus fiable pour comprendre et modifier une application, puisqu'elle engendre peu de perte d'informations lors du processus de reverse. En revanche, le decompiling direct, c'est-à-dire le decompiling n'utilisant pas de format intermédiaire, est un bon compromis entre facilité de compréhension du code et exhaustivité des informations. L'OWASP ne mentionne pas l'aspect "qualité" du code obtenu, et ne nous permet donc pas de comprendre comment faire le choix entre les différentes méthodes de rétro-ingénierie statique existantes.

En ce qui concerne la rétro-ingénierie dynamique, il s'agit cette fois d'exécuter l'application sur un appareil physique ou émulé, pour étudier son flow d'exécution, son comportement avec le système ou encore sa réaction face à des inputs malicieux. Là-encore, on retrouve de nombreuses techniques dans cette grande famille qu'est la rétro-ingénierie dynamique : analyse des appels systèmes, analyse des connexions réseau, listage des librairies chargées, debugging. L'OWASP décrit le debugging comme étant la méthode la plus efficace pour appliquer la rétro-ingénierie en condition réelle, sur des applications complexes. C'est d'ailleurs l'approche qui a été choisie dans un papier de F. Tchakounté et de P. Dayang [33], dans lequel ils cherchent à comprendre le fonctionnement d'un malware Android. Même si le choix de l'approche dynamique plutôt que statique n'est pas explicitement mentionné dans l'article, les avantages à utiliser la rétro-ingénierie dynamique sont clairement démontrés. Comme le rappelle l'OWASP dans son introduction, le debugging peut être effectué à deux niveaux : au niveau du Java Runtime, avec le Java Debug Wire Protocol (JDWP) ; ou au niveau de couches plus basses, à la manière d'un ptrace sous les systèmes Unix. Quel que soit le type de debugging choisi, l'intérêt principal réside dans l'efficacité à analyser le code. Cette technique permet en effet de mettre des points d'arrêts (breakpoints) sur certaines fonctions du code, pour suivre le flow d'exécution du programme tout en modifiant des valeurs de variables à la volée, et ainsi contourner de possibles verrous mis en place, comme une authentification.

Si l'on pouvait faire un reproche à la très pertinente conférence de Pau Oliva Fora, ce serait de ne pas évoquer les techniques de reverse dynamique dans une introduction à la rétro-ingénierie d'applications sous Android. De nos jours, il serait impensable de tenter de reverse une application sans utiliser les deux grandes familles de méthodes de rétro-ingénierie. Les deux ont un intérêt particulier à être utilisées, et d'autant plus lorsqu'elles sont utilisées chacune en complément de l'autre. La tendance actuelle serait même de considérer que la rétro-ingénierie dynamique est la technique la plus efficace en condition réelle, pour des raisons que nous détaillerons plus tard dans cet état de l'art.

3.1.2 Evaluation qualitative des outils de reverse-engineering

Le choix des outils de reverse-engineering d'applications Android est presque aussi important que le choix de la technique de reverse à appliquer. Les outils disponibles dans ce domaine sont nombreux, et ont souvent des fonctionnalités variées, parfois communes à d'autres outils, parfois spécifiques. Pour choisir l'outil le plus adapté à notre besoin, c'est-à-dire correspondant à la technique que l'on souhaite mettre en œuvre, et au résultat que l'on souhaite obtenir, il est pertinent d'avoir une vue d'ensemble sur ce qui existe aujourd'hui, et de pouvoir évaluer l'efficacité des différents outils.

En ce qui concerne la rétro-ingénierie statique, nous l'avons dit, l'objectif est d'obtenir du code compréhensible et exhaustif en terme d'informations, à partir de fichier exécutables, les DEX files. Les disassembler permettent d'obtenir ce résultat en passant par un langage intermédiaire (LI), que l'on peut

apparenter à une sorte de langage assembleur. Il existe différents LI, implémentés par différents outils : SMALI, implémenté sur apktool [18]; JASMIN, implémenté sur dex2jar [24] ; JIMPLE, implémenté sur soot [29]. La question à se poser est alors : "comment choisir quel LI utiliser ?". L'article "A Comparison of Android Reverse Engineering Tools via Program Behaviors Validation Based on Intermediate Languages Transformation" [38] propose de tester ces différents langages. L'un des points forts de cet article est sa capacité de démonstration. En effet, le procédé est le suivant :

- on transforme les DEX files en un langage intermédiaire défini
- on recompile le résultat de l'opération précédente
- on compare le comportement de l'application originale avec l'application recompilée, via des tests statistiques sur les évènements.

Le procédé est appliqué sur un très grand nombre d'applications, avant de faire une évaluation empirique des résultats.

L'étude a permis de montrer que SMALI permettait de préserver à 97 % le comportement initial de l'application, tandis que JASMIN et JIMPLE permettaient de préserver respectivement 69 et 73 % du comportement initial.

Il faut cependant nuancer les résultats de cette étude. En effet, malgré le fait que l'article se veuille être un comparateur des outils de reverse-engineering, un paramètre non-négligeable n'est pas pris en compte : la lisibilité du code. D'ailleurs, ce paramètre est totalement subjectif : certains attaquants seront bien plus à l'aise pour lire du JASMIN, tandis que d'autres préfèreront lire du SMALI. Ce qu'il faut comprendre ici, c'est simplement qu'un reverseur qui souhaitera modifier une application, préfèrera passer par du SMALI pour garantir un comportement similaire à celui qu'elle avait avant d'avoir été reverse. Ni plus, ni moins.

L'article "Qualitative Evaluation of Security Tools for Android" [32], écrit par les mêmes auteurs que l'article précédent, permet de combler les lacunes évoquées ci-avant. En effet, l'article ne présente plus seulement la qualité des outils sur le plan du résultat technique, mais aussi sur le plan de l'expérience utilisateur, de ses fonctionnalités et de sa modulabilité. Il en ressort alors que le choix de l'outil dépend d'abord logiquement des fonctionnalités qu'il propose, de la plateforme sur laquelle le reverseur travaille (la majorité des outils sont fait pour fonctionner avec OpenJDK, donc sous des systèmes Unix), mais aussi et surtout du niveau technique de l'utilisateur. Effectivement, il apparaît que rares sont les outils qui proposent une interface graphique : la majorité s'utilisent en ligne de commande, ce qui nécessite de connaître la syntaxe et les commandes. Aussi, certains outils proposent des fonctionnalités complexes, ou des possibilités d'extensions pertinentes, mais qui sont malheureusement trop peu documentées. L'utilisateur qui choisira ces outils devra donc avoir des prérequis techniques solides. Le tableau 1 permet d'avoir une vue d'ensemble sur les outils de rétro-ingénierie, en comparant leur facilité d'utilisation, déterminée à partir du format (GUI ou CLI) et de la qualité de la documentation ; ainsi que la modulabilité, déterminée à partir des fonctionnalités existantes et/ou ajoutables.

Enfin, l'article met en lumière l'un des enjeux majeurs lorsque l'on souhaite tester des outils comme ceux-ci : être vigilent vis-à-vis de la source de téléchargement. La plupart de ces outils sont utilisés par des chercheurs en sécurité, des professionnels, pour contrer les attaquants. Il semble donc logique que ces outils soient une cible de choix pour les pirates, qui n'hésitent pas à redistribuer ces outils sur des blogs, des plateformes après y avoir ajouté du code malveillant.

3.1.3 Les vues d'abstraction

Les vues d'abstraction sont un outil de visualisation de l'application, une fois avoir été reverse. Plus concrètement, c'est une vue qui sert à montrer quelque chose de spécifique, comme les relations entre

	Facilité d'utilisation	Modulabilité
Smali [6]	+	-
Dedexer [7]	-	+
Radare [8]	+	+
Dex2Jar [9]	+	-
Soot [10]	-	-
Jd-gui [11]	+	+
Jad [12]	-	-
Apktool [13]	+	-
AXMLprinter [14]	-	-

Table 1. Comparaison des outils de rétro-ingénierie

objets ou le comportement du programme sur une base de temps. En effet, il peut s'avérer difficile de comprendre une application simplement à partir de son code, d'autant plus si cette dernière est complexe et contient plusieurs milliers de lignes de code.

Les vues d'abstractions se divisent en trois familles : les vues statiques, les vues dynamiques et les vues merged. La vue statique a pour objectif de montrer toutes les informations pour lesquelles le temps n'a pas d'influence : les types de variables, leurs relations, les relations entres objets, etc. La vue dynamique montre quant à elle les transitions d'états des objets, les appels à d'autres ressources à des instants t, etc. Enfin, la vu merged est une vue qui regroupe des informations de la vue statique et des informations de la vue dynamique. D'après le papier "Static and Dynamic Reverse Engineering Techniques for Java Software Systems" [31], cette vue a l'avantage de faire apparaître les relations entre les résultats de la rétro-ingénierie statique et les résultats de la rétro-ingénierie dynamique. Aussi, sous Java, le polymorphisme fait qu'un objet en vue statique peut se trouver sous différentes formes dans la vue dynamique. La vue merged permet alors de faire apparaître l'objet statique sous ses différentes formes dynamique, ce qu'il n'est pas possible de faire dans une vue simple.

Cependant, l'article nuance ses propres propos en ajoutant qu'une vue merged reste bien souvent difficile à exploiter, dans la mesure où sa structure est de nature complexe, car non-intuitive. Les vues statiques et dynamiques sont en effet totalement différentes, et les rassembler dans une même vue n'a pas de logique à proprement parler, même si cela peut aider à la compréhension. Chaque type de vue est formalisé en fonction des informations que l'on souhaite faire apparaître. On parle alors de diagrammes. Le papier "Reverse Engineering of Object Oriented Code" [34] permet d'avoir une bonne représentation de ce à quoi correspond chaque diagramme. Les diagrammes les plus pertinents sont, de manière non-exhaustive :

- Class diagram : représentation globale de la vue statique
- Sequence diagram : décrit les interactions entre objets en fonction du temps
- Statechart diagrams : expose les états et transitions d'états des objets

On regrettera que cet article, qui comme son nom l'indique traite de la rétro-ingénierie de code orienté objet, ne mentionne pas le Scenario Editor (SCED), alors que ce dernier est particulièrement utilisé pour du code orienté objet. Le SCED est un outil pour modéliser le comportement dynamique d'un programme. Il offre deux diagrammes : le scenario diagram (similaire au sequence diagram) ainsi que le statechart diagram.

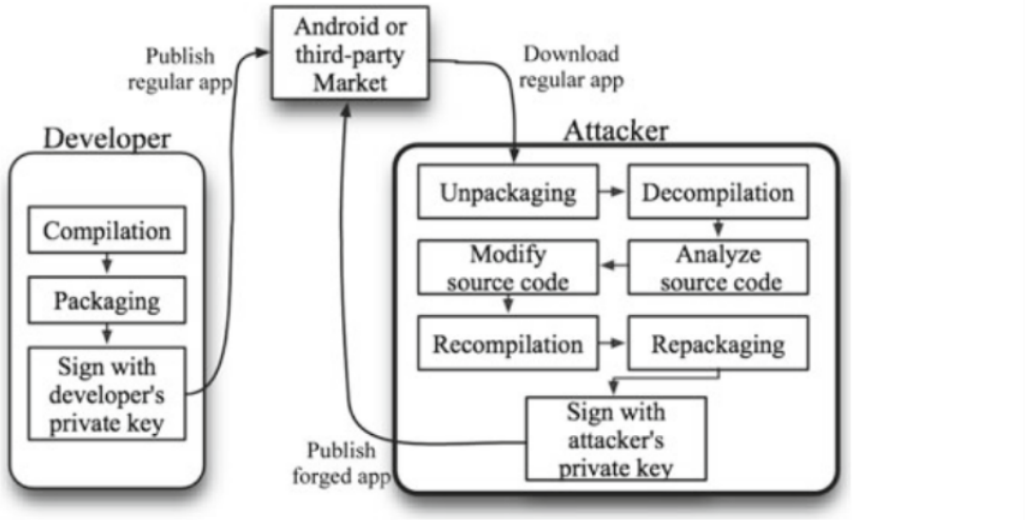

Fig. 3. Processus de repacking d'une application Android

3.1.4 Repacking d'applications modifiées

Comme nous l'avons vu, il peut arriver que l'on souhaite reverse une application afin de modifier son comportement. C'est ce qu'on appelle une attaque par forgeage, si l'on souhaite remplacer l'application déjà existente sur le store, ou une attaque par repackaging si l'objectif est de copier l'application pour l'ajouter sur le store sous un autre nom. Une fois le code modifié, il faut donc être en mesure de pouvoir repacker l'application, avec la nouvelle version du code source. (voir figure 3)

L'article "Repackaging Attack on Android Banking Applications and Its Countermeasures" [20] présente la vulnérabilité comme étant la conséquence directe d'une mauvaise signature-policy. Les chercheurs ayant rédigé cet article parviennent en effet à modifier une application bancaire pour détourner de l'argent, en suivant le process de l'attaque par forgeage. Une fois l'application modifiée, ils parviennent simplement à téléverser l'application sur un store, alors que celle-ci est auto-signée avec leur propre clé.

Un exemple typique de repacking d'application est présenté dans l'article "Android Applications Reversing 101" [11]. Cet article a la particularité de contenir un grand nombre d'informations essentielles pour débuter dans la rétro-ingénierie d'applications Android, tout en restant dans des notions simples et accessibles. C'est d'ailleurs pour cela que l'on peut le retrouver dans les sources de plusieurs articles de recherche. Dans cet exemple, le repacking de l'application n'est cette fois pas utilisé pour la re-téléverser sur un store et piéger ses utilisateurs, mais plutôt pour faciliter l'application de la rétro-ingénierie.

En effet, on peut aussi utiliser le repacking pour modifier certains attributs de l'application, et notamment contourner des dispositifs de sécurité, des authentifications, etc. Pour revenir à l'article "Android Applications Reversing 101", l'auteur va modifier le fichier manifest.xml, sorte de sommaire des différentes parties de l'application, afin d'y ajouter un attribut *android:debuggable* avec la valeur True. Il suffit alors de repacker l'application, de la signer avec sa propre clé, et la rétro-ingénierie dynamique s'en trouvera alors grandement simplifiée.

Bien des mesures auraient permit d'éviter ce type d'attaque, certaines étant particulièrement simple à mettre en œuvre. Cela nous amène alors à notre deuxième partie : les stratégies utilisées pour contrer la rétro-ingénierie d'applications sous Android.

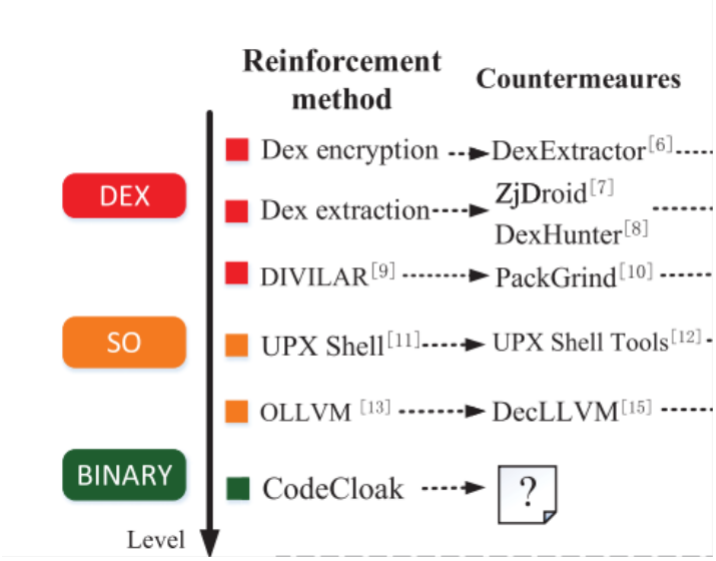

Fig. 4. Contournement de dispositifs d'obfuscation

3.2 Protections Anti-Rétro-Ingénierie

Dans cette partie, nous allons chercher à comprendre les techniques actuellement mises en œuvres par les développeurs d'applications, pour mettre à l'échec les techniques présentées dans la partie précédente.

3.2.1 L'obfuscation du code

Comme nous l'avons vu jusqu'ici, quel que soit l'objectif du reverseur, la compréhension du programme reste la clé. Limiter la compréhension du code par un attaquant semble donc être la meilleure solution pour éviter la rétro-ingénierie statique, et ce à juste titre. Aujourd'hui, un très grand nombre d'applications sont compilées à partir de code obfuscé. C'est-à-dire que le code garde sa logique initiale, mais est complexifié du point de vue d'un lecteur.

Pour Z. Tang, auteur d'un article présentant l'outil CodeCloak [16], permettant l'obfuscation au niveau du Dalvik Bytecode, cette solution est l'une des rares barrières efficaces contre la rétro-ingénierie. La structure des applications Android étant imposée d'une telle manière, le reverse en devient relativement simple, et les contre-mesures de sécurité facilement contournables. Pourtant, comme l'auteur le dit lui-même dans l'article, la robustesse de la méthode tient seulement dans le fait qu'aucun outil à ce jour ne permet de faire le processus inverse : la désobfuscation. Ceci est dû au fait que l'obfuscation au niveau du bytecode est plus difficile à contourner, pour l'instant, qu'une obfuscation au niveau du code source. Malheureusement, les techniques d'obfuscations deviennent obsolètes très rapidement. Aujourd'hui, nombreuses sont les techniques d'obfuscation qui sont obsolètes. Chaque fois qu'une nouvelle technique d'obfuscation est déployée, de nouveaux outils de désobfuscation font leur apparition. (voir figure 4)

Le document "Android Application Protection against Static Reverse Engineering based on Multidexing" [23] part d'ailleurs de ce fait pour justifier le choix d'une autre méthode de protection. L'idée ici est d'utiliser des packers comme liapp [8], ijiami [2]. Ces packers refaçonnent l'architecture de l'application, pour que certaines sections de code soient obfsucées (voir figure 5). Cela permet d'éviter les attaques utilisant la rétro-ingénierie statique. Cependant, l'article expose un problème de taille : les packers, qui semblaient jusqu'ici avoir fait leurs preuves, ne sont pas compatibles avec les nouvelles architectures des

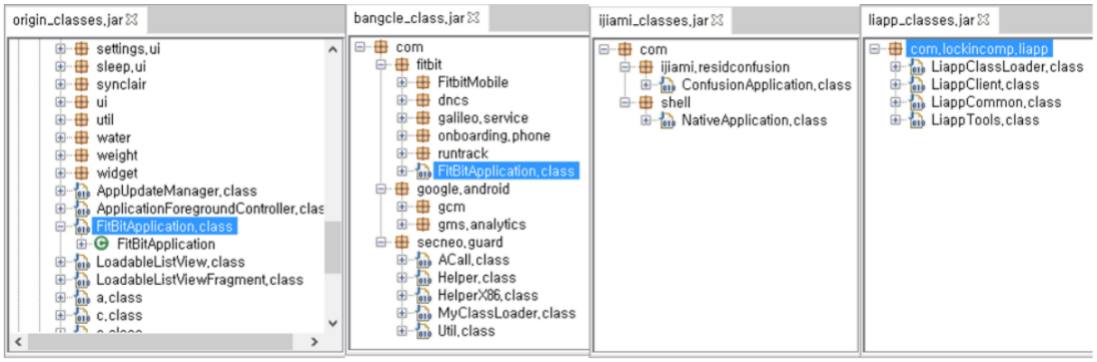

Fig. 5. Obfuscation de la première classe par différents outils

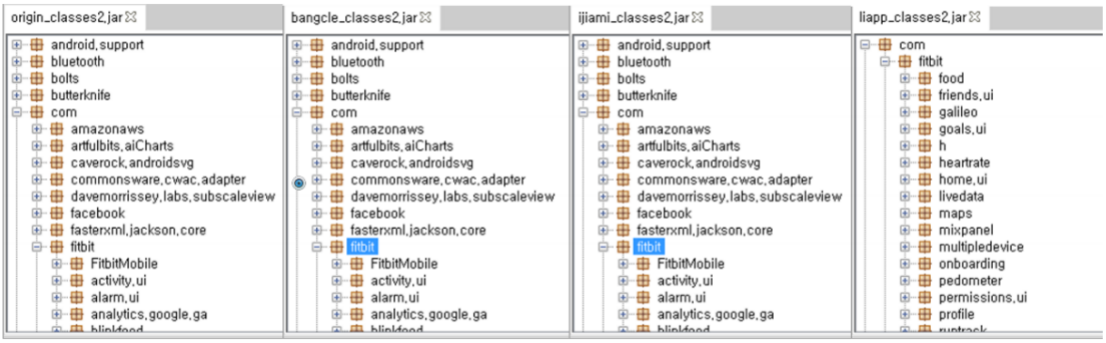

Fig. 6. Obfuscation d'une seconde classe par différents outils

applications. Depuis quelques années, il est en effet possible de construire une application à partir de plusieurs DEX files distincs, ce qui n'était pas possible avant. Des tests menés dans l'article permettent de montrer que sur ces nouvelles architectures, les packers perdent en efficacité puisque les sections de code à charger dynamiquement peuvent provenir de différentes sources (voir figure 6).

La solution alors proposée par l'équipe de chercheurs est d'utiliser la technique de chiffrement des DEX files, qui sont ensuite déchiffrés et chargés dynamiquement depuis une classe principale. Les tests réalisés dans l'article permettent de démontrer la robustesse d'une telle solution. Malgré tout, cette dernière n'est pas viable à l'heure actuelle dans la mesure où le temps d'exécution de l'application se voit démultiplié, allant jusqu'à 19 secondes pour lancer une application simple.

Ici encore, on se retrouve face à des contre-mesures de premier abord efficaces, mais dont les limites sont rapidement atteintes. Aussi, nous avons vu que de nombreuses méthodes permettaient de limiter relativement simplement les techniques de rétro-ingénierie statique. Mais qu'en est-il des méthodes de rétro-ingénierie dynamique ?

3.2.2 Contrôler la plateforme d'exécution du code

A l'heure actuelle, la stratégie la plus répandue pour contrer les attaques par rétro-ingénierie dynamique consiste à vérifier différents paramètres quant à l'environnement d'exécution de l'application. Les paramètres recherchés servent à déterminer principalement deux choses : le smartphone est-il rooté ? ; et le debugging est-il activé ? ; c'est du moins ce qui est présenté dans "An Android Application Protection Scheme against

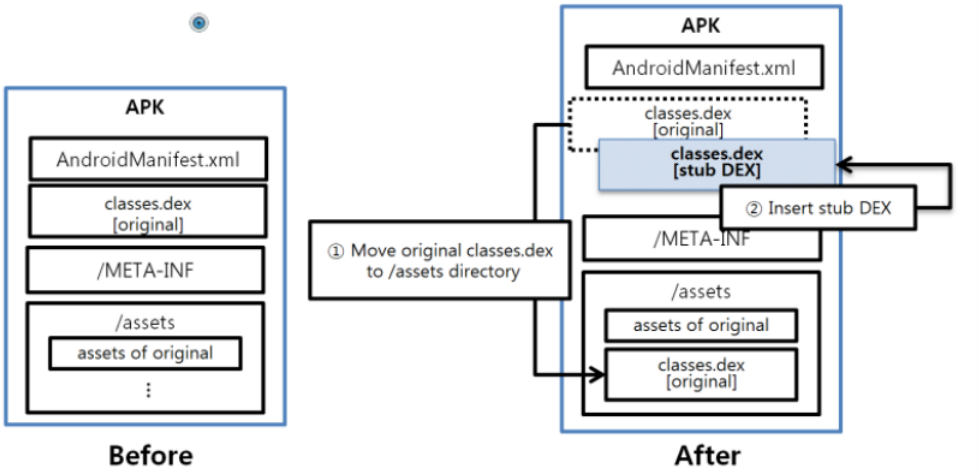

Fig. 7. Remplacement du DEX principal de l'application

Dynamic Reverse Engineering Attacks" [21]. On y apprend alors que quels que soient les outils mis en place, les techniques restent sensiblement les mêmes [13], et sont facilement contournable. Pour cette raison, les auteurs de cette publication présentent une solution robuste, permettant de résister à des outils d'évasion comme Xposed.

Le fonctionnement de cet outil est complexe, et ne pourrait être détaillé ici. Il faut simplement retenir que l'outil modifie l'architecture de l'APK (voir figure 7), en remplacement le DEX file principal par un jeu d'instruction différent. Ce nouveau DEX file va alors commencer par vérifier si le debugging est activé, et si le smartphone est rooté. A ce niveau, les techniques mises en œuvres sont similaire à ce qui se fait ailleurs : l'application vérifie les valeurs de certaines variables d'environnement pour déterminer si le smartphone est rooté ou en mode debug. Cependant, pour éviter tout contournement, le programme va vérifier que les instructions exécutées correspondent bien à celles qui étaient initialement prévues. En cas d'erreur à ce niveau, l'exécution se stoppe. Si tout a fonctionné, le fichier DEX falsifié va charger dynamiquement le DEX principal de l'application.

Les tests effectués dans l'article prouvent l'efficacité de cette stratégie, notamment contre les techniques d'évasion ou de contournement. On regrettera cependant que le test d'évasion n'est effectué qu'avec un seul outil, Xposed, alors que d'autres outils utilisent des méthodes différentes pour échapper aux détecteurs. Aussi, si un attaquant exécute l'application sur un smartphone émulé, il sera en mesure de debugger cette dernière à une couche plus basse : celle de l'émulateur. Pour cette raison, on retrouve d'autres stratégies ajoutant cette fois la détection d'environnement émulés. "Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware" [26] explique que ces détecteurs reposent sur le fait que les smartphones d'aujourd'hui possèdent un grand nombre de capteurs (accéléromètre, gyroscope, GPS, etc.) fournissant en continu un nombre impressionnant de données. Certains de ces capteurs ne peuvent être émulés efficacement à cause de leur complexité. Une analyse des capteurs sur le smartphone peut donc permettre de savoir si un environnement est possiblement émulé.

3.2.3 Packing d'applications

Jusqu'ici, nous avons trouvé des solutions pour faire face à la rétro-ingénierie statique, avec l'obfuscation ; ainsi qu'à la rétro-ingénierie dynamique, avec la détection d'environnement de debugging. Les packers, dont on a déjà parlé dans la partie 3.2.1 L'obfuscation du code, sont en fait une solution packagée pour répondre à plusieurs problématiques, et pas seulement l'obfuscation comme on a pu le dire jusqu'ici.

L'article "Adaptive Unpacking of Android Apps" [36] présente trois fonctionnalités communes à l'ensemble des packers du marché :

- dispositifs anti-rétro-ingénierie statique, implémentant des fonctions d'obfuscations (cf partie 3.2.1)
- dispositifs anti-dump des DEX files, implémentant des fonctions de détection d'environnement de debugging (cf partie 3.2.2)
- et enfin le cachement de contenu des DEX files, mettant en œuvre de la modification dynamique de code.

Cette modification dynamique de code consiste à rendre impossible toute technique de rétro-ingénierie, en exécutant un code qui se trouve modifié une fois chargé en mémoire. D'une manière générale, les fonctions responsables de cette modification du code se trouvent dans des méthodes JNI, dans du Native Code, ce qui rend difficile des exploitations de ces dernières.

L'article "Are mobile banking apps secure ?" [14], se propose de tester la sécurité d'un panel d'application bancaires, afin de déterminer les stratégies de protection les plus efficaces. Les auteurs de cet article arrivent à la conclusion que les packers sont le moyen idéal pour sensiblement augmenter la sécurité de son application, et ce de manière fiable.

3.2.4 Détecter le repacking

Comme nous l'avons expliqué dans la partie 3.1.4 Repacking d'applications modifiées, les attaques utilisant le repacking sont de deux natures :

- Attaque par forgeage : on modifie l'application et on re-upload en se faisant passer pour son développeur. L'objectif peut être d'injecter un malware dans une application légitime
- Attaque par repackaging : on copie le code d'une application pour l'upload sur un store sous un autre nom. L'objectif peut être de copier le code d'un concurrent.

Les attaques par forgeage sont relativement simples à contrer [20]. Une simple policy restreignant l'upload des applications auto-signées permet de résoudre le problème, même si en pratique, cela peut s'avérer compliqué, comme expliqué dans l'article "Multi-signature based integrity checking scheme for detecting modified applications on android" [19].

Les attaques par repackaging sont quant à elles bien plus compliquées à détecter, puisque la deuxième application, celle créée par l'attaquant, n'a à première vue rien à voir avec l'application originale. Les solutions mises en places se trouvent au niveau des stores d'applications. On retrouve alors des détecteurs d'applications repackagées qui utilisent l'analyse statique, tandis que d'autres utilisent l'analyse dynamique.

Il existe de très nombreux types de détecteurs par analyse statique, puisque ces derniers sont massivement utilisés à l'heure actuelle. On y retrouve la comparaison sémantique, visant à chercher si le flow d'exécution est similaire, en analysant le Dalvik Bytecode ; l'analyse par Fuzzy Hashing, consistant à comparer les hashs des instructions par localité ; et enfin l'analyse par PDG, consistant à comparer les graphes de dépendance des applications. Malheureusement, comme l'expose l'article "A Framework for Evaluating Mobile App Repackaging Detection Algorithms" [17], l'efficacité de ces détecteurs est vite limitée, notamment si l'attaquant décide d'obfuscer l'application repackée. Les auteurs de l'article proposent alors

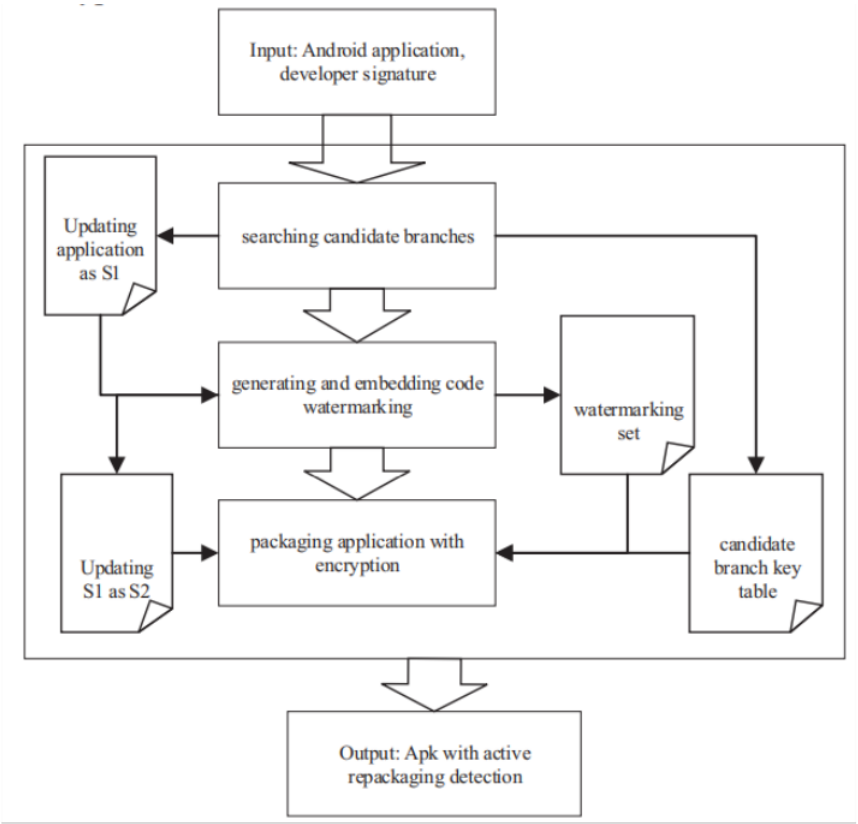

Fig. 8. Processus de Watermarking [30]

un framework, permettant de tester l'efficacité de chacune des solutions face à certains types d'obfuscation. Il en ressort alors que chaque type de détecteur peut-être contourné par au moins un type d'obfuscation.

Pour répondre à cette nouvelle problématique, de nouveaux détecteurs ont fait leur apparition, cette fois-ci en utilisant l'analyse dynamique. La technique de watermarking, présentée dans "An Active Android Application Repacking Detection Approach" [30] semble relativement prometteuse. Le principe du processus est expliqué en figure 8.

Comme on peut le voir, le code watermarking est appliqué juste avant la signature de l'application par le développeur. Ainsi, si un attaquant copie le code, le store n'aura qu'à vérifier la présence d'un watermarking pour savoir s'il s'agit d'une application copiée. L'expérience menée dans l'article montre l'efficacité de cette stratégie, que l'on devrait rencontrer de plus en plus souvent.

3.3 Contournement des dispositifs de sécurisation

La deuxième partie de cet état de l'art nous a permis de mettre en lumière les dispositifs actuellement utilisés pour limiter la rétro-ingénierie des applications Android. Dans la mesure où un certain nombre de solution de sécurité apparaissent comme des présentations marketing dans la littérature de référence, cette troisième partie a pour objectif d'exposer les limites de ces dispositifs, en présentant des techniques de contournement actuellement exploitées par les attaquants.

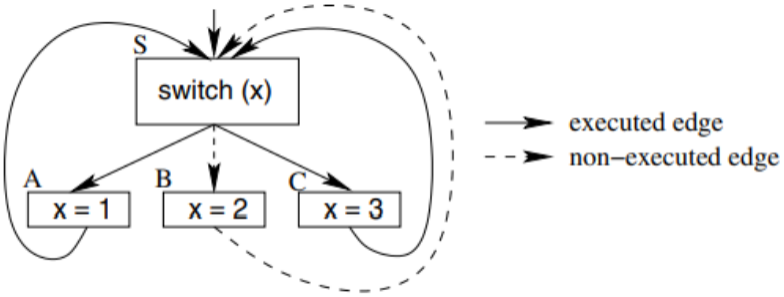

Fig. 9. Ecartement des chemins non-exécutés [35]

3.3.1 Désobfuscation de code

L'obfuscation, comme nous avons pu le dire, a pour objectif de limiter les résultats apportés par une analyse statique. Une première réflexion serait de dire qu'il suffit d'utiliser l'analyse dynamique. Malheureusement, si on se réfère à la partie 3.1 Les méthodes de rétro-ingénierie, l'analyse dynamique ne peut remplacer l'analyse statique, puisqu'elle ne permet pas d'obtenir les mêmes informations. Il faut en réalité agir avec plus de subtilité : mettre en œuvre de la rétro-inégnierie dynamique pour permettre une analyse statique. L'article "A Generic Approach to Automatic Deobfuscation of Executable Code" [37] présente les lignes directrices pour mettre en œuvre une désobfuscation efficace. On commencera par noter que, comme expliqué dans l'article, une désobfuscation efficace ne consiste pas à remonter au code source initial, avant obfuscation, mais bien de simplifier le code de manière à pouvoir analyser son comportement. La subtilité est importante : la désobfuscation ne va pas consister à appliquer une méthode d'obfuscation inverse, mais simplement de déterminer à quoi ressemble l'application une fois simplifiée. C'est ce qui permet d'ailleurs de rendre cette technique efficace face à tout type d'obfuscation.

Dans un premier temps, les auteurs de l'article proposent d'utiliser la rétro-ingénierie dynamique , en injectant différents inputs, pour déterminer quelles sont les instructions qui sont impactées par les données en entrée. Cela permet d'exclure le code mort de l'application, présent simplement pour complexifier l'architecture du code. "Deobfuscation: reverse engineering obfuscated code" [35], référence incontestée dans le domaine de la désobfuscation, cité plus de 220 fois dans des articles scientifiques, offre un exemple visuel, permettant de mieux se rendre compte de ce qui se passe à cette étape (voir figure 9).

Dans cet exemple, le bloc B n'est jamais exécuté. Les instructions permettant de passer de S à B, et de B à S, ne sont donc jamais marquées. Les flèches en pointillées n'apparaitront alors plus dans les prochaines étapes.

Ensuite, une fois que tous les flow d'exécution valides sont identifiés, on applique une transformation du code, à partir de la sémantique identifiée à l'étape précédente.

La dernière étape consiste à générer un Control Flow Graph Construction (CFG), présentant une abstraction haut-niveau du code de l'application : boucles, conditions, etc.

Cette méthodologie est appliquée de façon itérative (voir figure 10) afin d'affiner le CFG, en supprimant tout flow d'exécution n'ayant pas d'intérêt propre à l'application. Une évaluation de cette technique a permis de montrer qu'il était possible de retrouver le CFG initial de l'application avec une similarité d'en moyenne 80 %, et ce quels que soient les outils utilisés pour l'obfuscation.

Les techniques de désobfuscation présentées jusqu'ici ont l'avantage de pouvoir être implémentées quelle que soit la plateforme, et quelle que soit la nature du code. Ce sont des techniques qui sont cependant lourdes à implémenter [35]. Il serait regrettable de ne pas évoquer des techniques de désobfuscation

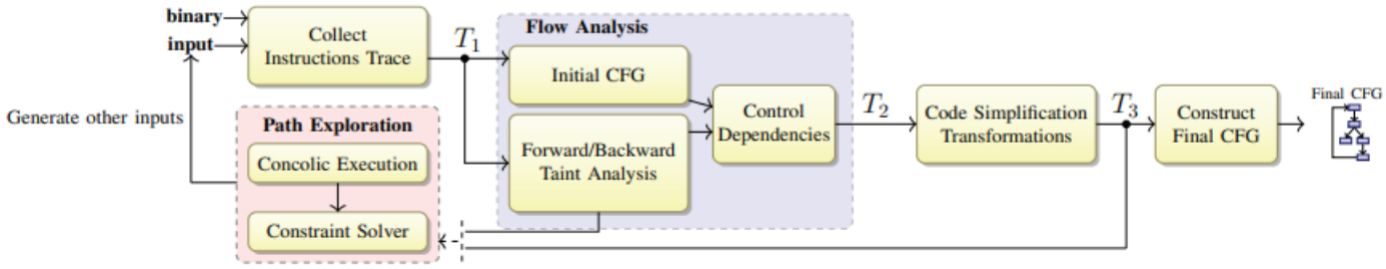

Fig. 10. Processus d'affinage d'un CFG [35]

exploitant les particularités des applications android, et se révélant donc plus efficaces en terme de coût-résultat. Le document "Statistical Deobfuscation of Android Applications" [12] répond exactement à cette problématique. Ici, on va exploiter la particularité des applications Android qui nous a aidé tout au long de cet état de l'art : le fait que les applications ont toutes la même structure. L'outil présenté, DeGuard, propose de désobfuscer du code ayant subit le traitement de ProGuard [10], en utilisant l'analyse statistique.

Concrètement, DeGuard va comparer le code obfuscé à une grand quantité de codes non-obfuscés, pour prédire le nom réel des variables, des classes, ainsi que déterminer quelles parties du codes ont la plus grande probabilité d'avoir été ajoutées par un obfuscateur. Les résultats de l'évaluation sont impressionnants : 80 % des prédictions faites par l'outil sont exactes. On peut cependant nuancer ce résultat puisque l'étude ne permet pas de savoir quel est le taux de similarité entre l'application désobfuscée et l'application initiale, avant obfuscation.

3.3.2 Evasion contre des détecteurs d'environnement de debugging

La partie 3.2.2 Contrôler la plateforme d'exécution du code nous a permis d'apprendre qu'une grande part des détecteurs d'environnement de debugging se contentaient de vérifier des valeurs de variables d'environnement, ou l'existence de fichiers, pour déterminer si le smartphone est rooté ou en mode debug.

L'un des travaux les plus importants en ce qui concerne l'évasion face aux détecteurs d'environnement rooté est nommé RootCloak [35]. Ce framewrok utilise à la fois Xposed [21] et Cydia Subsrate[9] pour contourner la majorité des méthodes de détection. Comme montré en figure 11, Xposed se place comme interface entre l'API Android et les applications exécutées, ce qui lui permet de prendre le contrôle du Java ClassLoader object [5]. Xposed peut intercepter l'appel à l'API, ce qui permet à l'outil RootCloak de tout simplement modifier les valeurs retournées par l'API Android lors de root-checks.

Par exemple, si un détecteur cherche des applications en lien avec le rooting, RootCloack modifiera le résultat de l'API avec des données faussées.

L'article "Android Rooting: An Arms Race between Evasion and Detection" [22], qui présente différentes techniques d'évasion de rooting, nous alerte cependant sur un point : l'évasion par CodeCloak ne fonctionne que sur les détecteurs qui agissent au niveau du code Java. Les détecteurs implémentés en Native Code, c'est-à-dire dans les fichiers .so de l'application, ne pourront être évadés par cette technique. Cependant, l'article précise aussi que sur 92 applications implémentant un root-checker, 89 ont pu être évadées par cette technique.

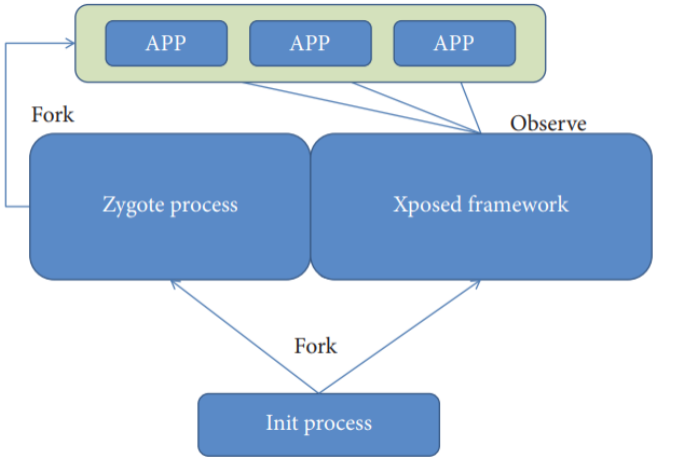

Fig. 11. Implémentation de XPosed[22]

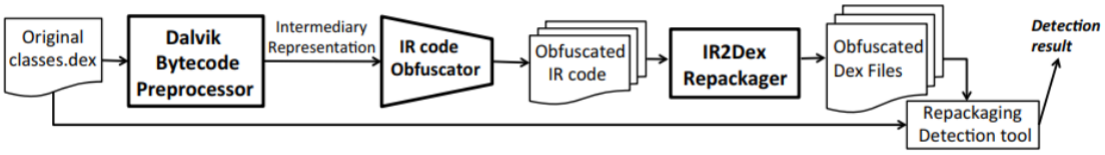

Fig. 12. Obfuscation d'une application repackée [17]

En ce qui concerne des détecteurs plus élaborés, implémentés dans des applications bancaires, les chercheurs ont déterminé que parmi 18 applications bancaires tirées au hasard, il était possible d'évader les détecteurs simplement en renommant les fichiers d'applications utilisés pour rooter le smartphone. Pour ainsi dire, à l'heure actuelle, de nombreuses applications massivement utilisées sont encore vulnérables aux techniques d'évasion simples. Des outils comme celui présenté en partie 3.2.2 tendent à réduire ces risques d'évasion, en cherchant à contrer directement les méthodes utilisées par les attaquants. On peut donc s'attendre à les rencontrer de plus en plus souvent.

3.3.3 Evasion contre des détecteurs d'applications repackées

Nous l'avons déjà évoqué, les auteurs de l'article "A Framework for Evaluating Mobile App Repackaging Detection Algorithms" [17] estiment que la majorité des détecteurs d'application repackées sont facilement contournables en utilisant l'obfuscation, pour faire croire au détecteur que l'application n'a rien à voir avec une autre. La force de cet article est qu'il présente à la fois les stratégies actuellement utilisées par les détecteurs d'applications repackées, mais aussi les techniques d'obfuscation qui permettent de les outre-passer. On peut notamment parler du process exposé ci-dessous en figure 12.

Afin d'échapper aux détecteurs, la première étape consiste à désassembler (et non décompiler, cf partie 3.1) les DEX Files, afin d'en obtenir une représentation intermédiaire (IR). Avec cette IR, il est en effet beaucoup plus simple de manipuler le code de manière efficace et précise, à l'instar des langages intermédiaires (LI) utilisés pour modifier des applications. Par ailleurs, on choisit d'obtenir un IR plutôt

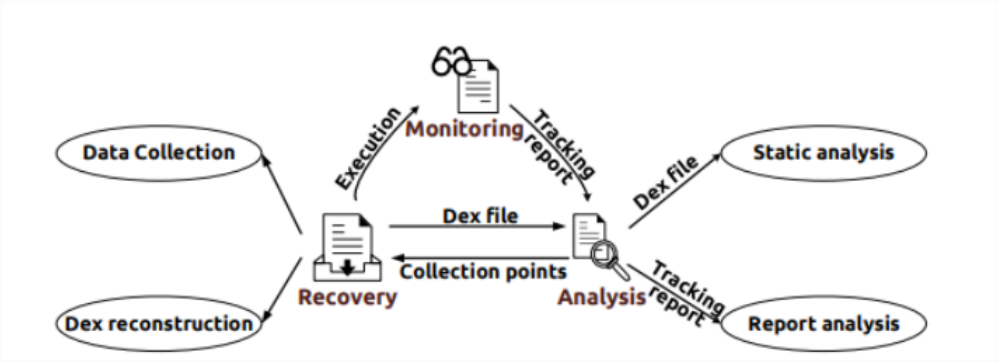

Fig. 13. Processus itératif d'unpacking [36]

qu'un LI de plus haut-niveau comme SMALI ou JASMIN, dans la mesure où l'IR peut plus facilement être re-transformé en bytecode, sans perte d'informations.

Une fois l'IR obtenu, on arrive à la phase d'obfuscation. Ici, on utilise principalement les mêmes techniques que celles initialement conçues pour protéger les applications de la rétro-ingénierie (cf 3.1 L'obfuscation du code).

Vient enfin la phase de repackaging, qui consiste à recompiler l'IR obfuscé en DEX files. A ce niveau, le choix de l'outil utilisé pour recompiler l'IR est d'une importance cruciale : il faut choisir un compileur qui n'effectue que très peu d'optimisations, afin de ne pas réduire l'effet de l'obfuscation. Les auteurs de l'article sus-mentionné ont déterminé un bon candidat pour cette tâche : dx tool [7], puisqu'après avoir analysé son code source, il apparaît que cet outil ne performe que des optimisations mineures, et garde la sémantique du code lors de la compilation.

3.3.4 Rétro-Ingénierie sur application packée

Dans la partie 3.2.3, l'article "Android Application Protection against Static Reverse Engineering based on Multidexing" [23] nous introduisait la notion de packers, en la présentant comme une solution particulièrement performante. Des articles comme "Are mobile banking apps secure ?" [14] vont même jusqu'à dire qu'il s'agit à l'heure actuelle de la solution la plus efficace pour protéger une application de la rétro-ingénierie. Cette fois encore, nous nous devons de nuancer ces propos en présentant une nouvelle méthode d'unpacking d'applications, présentée dans "Adaptive Unpacking Of Android Apps" [36]. Pour les auteurs de cet article, il est important d'identifier ce qui amène un unpacker à échouer dans sa tâche : sa non-évolutivité. En effet, comme les packers implémentent de nombreuses fonctions d'obfuscations et les font évoluer régulièrement, il est important qu'un unpacker soit en mesure de s'adapter.

Pour ce faire, l'outil PackerGrind, présenté dans l'article, utilise un processus itératif, présenté en figure 13.

Pour retrouver le DEX file original, Packergrind va dans un premier temps exécuter l'application et analyser son comportement à partir de trois éléments : l'Android Run Time (ART) [28], les évènements système et les instructions utilisées. Ces informations sont remontées sous forme d'un Tracking Report. En parallèle, l'outil va chercher à collecter des données dans les DEX files, notamment pour connaître l'obfuscation utilisée ou l'architecture globale de l'application. Ces données sont alors utilisées pour

reconstruire les DEX files à la fin de chaque exécution. Le processus est répété X fois, jusqu'à ce que l'utilisateur détermine si le code obtenu lui convient.

Les auteurs de l'article "DexHunter: Toward Extracting Hidden Code from Packed Android Applications" [39] proposent une approche sensiblement différente, et qui a l'avantage de s'adapter à des versions Android plus anciennes utilisant DVM, et non ART [28]. Pour expliquer simplement le process, l'outil DexHunter va utiliser le DEX file optimisé de l'application, généré par ART ou DVM, chercher les magic numbers dans le header du fichier et dumper la mémoire correspondante.

De nouveaux problèmes se posent alors. Premièrement, les packers ne mettent pas en œuvre seulement de l'obfuscation (cf. partie 3.2.3) mais aussi des méthodes d'anti-debugging. Aussi, rien n'est mis en œuvre pour contourner la modification dynamique de code utilisée par certains packers.

Pour résoudre ces problèmes, DexHunter propose une approche en deux étapes :

- Localisation de la mémoire
- Exploiter le chargement proactif des classes

Pour localiser les adresses mémoires qui devront être analysées, l'outil va modifier le code source de l'ART ou du DVM. De cette manière, lorsqu'un DEX file optimisé est exécuté, sa localisation exacte pourra être retrouvée. Il n'y aura alors plus qu'à parser le header pour retrouver l'adresse mémoire sur laquel il s'exécute.

Une fois que l'adresse mémoire est retrouvée, DexHunter ne va pas directement debugger les instructions, mais va exploiter le fait que sous Android, toute classe est chargée en amont, avant d'être exécutée. Cela permet alors à l'outil de savoir quelle instruction est réellement appelée, pour contourner la modification dynamique de code.

Malheureusement, comme le précisent les auteurs de l'article sur PackerGrind, DexHunter est un unpacker extrêment efficace, mais sa stratégie a des limites importantes. Par exemple, il ne permet pas de retrouver le code implémenté dans les fonctions OnCreate(), exécutée lors du lancement de l'application, puisque cette dernière n'est appelée qu'après le pré-chargement de toutes les classes.

4 Synthèse des problématiques

Nous avons présenté dans la partie précédente les notions apparaissant dans la documentation de référence de la rétro-ingénierie sous Android. L'objectif de cette partie est désormais de mettre à disposition du lecteur les problématiques actuellement rencontrées dans ce domaine, ainsi que les potentielles perspectives qui se dessinent, à travers une synthèse et une interprêtation de tout ce qui a été évoqué jusqu'ici.

4.1 La course à l'obfuscation

L'article "Exploiting Binary-Level Code Virtualization to Protect Android Applications Against App Repackaging" [16] présente un bon point de départ pour évoquer la problématique de la stratégie d'obfuscation. Le nombre de recherches sur les moyens d'obfuscer du code est extrêment important. Les principales techniques avaient d'ailleurs été résumées en figure 4. Aussi, un grand nombre de travaux consistent à obfuscer le code au niveau des DEX files, c'est le cas d'outils comme ProGuard [10] ou DexGuard [1].

D'un autre côté, la réponse des attaquants se caractérise par une augmentation des techniques de désobfuscation, parfois en utilisant l'analyse dynamique, parfois en exploitant de mauvaises implémentations de fonctions d'obfuscation [35].

La figure 14 permet d'avoir une vue d'ensemble sur l'état actuel des techniques d'obfuscations et leurs contournements.

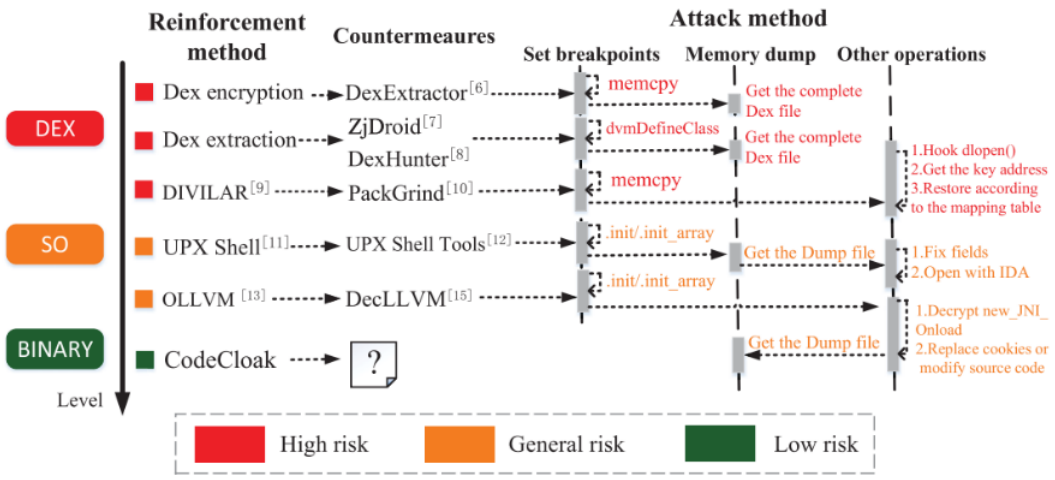

Fig. 14. Obfuscation et contre-mesures

Il apparaît alors que la grande majorité des techniques d'obfuscations massivement utilisées peuvent être déjouées, notamment si l'obfuscation se fait à un plus haut niveau que celui du binaire. On peut donc, dans un premier temps, logiquement s'attendre à ce que de plus en plus de techniques d'obfuscations agissent au niveau binaire, les techniques de contournement précédemment utilisées n'étant plus efficaces.

Cependant, la réalité est tout autre. La stratégie d'obfuscation n'a de toute façon pas de réel intérêt si elle est implémentée seule : en utilisant la rétro-ingénierie dynamique, il est possible de désobfuscer quasiment n'importe quel code [37], même si ce dernier s'effectue au niveau du binaire. La force de l'obfuscation réside donc dans le choix des mesures complémentaires, comme, par exemple, la mise en place de détecteur d'environnement de debugging. Si ces détecteurs sont efficaces, l'utilisation de la rétro-ingénierie dynamique par les pirates ne sera plus possible pour déjouer l'obfuscation, ce qui en fera une solution particulièrement robuste.

Par ailleurs, certains considèrent que l'obfuscation n'est plus une solution anti-rétro-ingénierie viable, et tendent à utiliser des techniques de chiffrement des DEX files [23] afin de remplacer l'obfuscation. Cette solution semble quant à elle peu exploitable, du moins pour l'instant, dans la mesure où lorsque les DEX files sont nombreux, les performances de l'application peuvent se trouver grandement impactées.

4.2 La fragilité des détecteurs d'environnement de debugging

Dans un premier temps, les développeurs sensibilisés à la sécurité ont cherché à mettre en place des contre-mesures à la rétro-ingénierie statique. Cette dernière reposant sur des principes définis et stable (évoluant peu dans le temps), les solutions semblaient relativement simple à mettre en oeuvre et particulièrement efficaces. Pour déjouer ces dispositifs anti-rétro-ingénierie statique, les pirates ont accentué leurs efforts sur la rétro-ingénierie dynamique, afin de proposer des méthodes toujours plus complexes, et des possibilités toujours plus variées. L'analyse dynamique est alors devenue la technique de rétro-ingénierie la plus compliquée à contrer pour cette raison. Cela explique notamment pourquoi elle est utilisée à plusieurs reprises pour déjouer des stratégies anti-rétro-ingénierie statique dans différentes parties de cet état de l'art.

Comme on pouvait l'apprendre dans "An Android Application Protection Scheme against Dynamic Reverse Engineering Attacks" [21], les techniques existantes pour limiter l'analyse dynamique reposent sur des principes plutôt simples et similaires. Les possibilités de déjouer l'analyse dynamique sont donc particulièrement limitées. Les techniques plus élaborées qui ont été évoquées dans cet état de l'art reposent exclusivement sur de la contre-mesure appliquée à une technique de rétro-ingénierie particulière. Cela rend donc ces dispositifs de sécurisation sensibles à l'évolution des attaques, qui, comme nous le savons depuis la partie 3.3.2, sont de plus en plus élaborées. Les outils de contournement, pour s'adapter, changent simplement de stratégie et de vecteur d'attaque pour rendre les détecteurs de debugging obsolètes. On se souviendra par exemple de l'étude menée par l'article "Android Rooting: An Arms Race between Evasion and Detection" [22], expliquant alors qu'une grande part des applications bancaires implémentant des détecteurs à l'état de l'art, avaient pu être évadés par des techniques très peu coûteuses pour l'attaquant.

D'après tous les éléments cités auparavant, on peut imaginer que seules des modifications dans l'architecture d'Android pourraient un jour limiter considérablement les possibilités d'analyse dynamique. En effet, à l'instar de XPosed qui s'insère en interface entre l'API Android et l'application, la quasi-totalité des outils d'évasion connus utilisent directement l'architecture d'Android pour faire croire à l'application que l'appareil n'est pas rooté. A aucun moment ce type d'outil n'intervient sur l'application de façon directe. Cela explique notamment pourquoi il est si difficile d'avoir une détection fiable, même avec des détecteurs à l'état de l'art.

4.3 Le repacking d'applications, l'espoir d'une solution efficace

Pour rappel, nous avons vu en partie 3.2.4 que le repacking est implémenté dans deux types d'attaques : attaques par forgeage et attaques par repackaging. L'article "Repackaging Attack on Android Banking Applications and Its Countermeasures" [20] nous montrait alors que les attaques par forgeage pouvaient très facilement être contrées.

Les attaques par repackaging ont quant à elles recourt à l'obfuscation, dans la mesure où nombre de détecteurs d'applications repackées reposent sur une analyse statique, et que les techniques d'obfuscation évoluent rapidement (cf partie 3.2.1). On entre alors dans une configuration du "serpent qui se mord la queue" [6] : les chercheurs en sécurité implémentent de nouvelles méthodes d'obfuscation efficaces, et ces dernières sont utilisées pour déjouer d'autres dispositifs de sécurité.

Pour sortir de cette configuration, un nouveau type de détecteurs d'applications repackeés semble tout à fait prometteur : le code watermarking [30]. On peut assez aisément s'attendre à ce que ce type de solutions soit implémenté sur des applications sensibles, de type applications bancaires ou administratives. En effet, par nature, le code Watermarking répond à l'ensemble des problématiques quant à la détection d'applications repackées. Qu'une application soit obfuscée ou compilée différemment, le watermark du code restera tel qu'il est. Mais cette solution a l'inconvénient d'être particulièrement lourde à l'implémentation : il faut que l'outil de détection et l'outil de watermarking utilisé sur l'application correspondent. En effet, l'outil de détection doit mettre en oeuvre une analyse dynamique, nécessitant une exécution de l'application. Aussi, il est nécessaire que l'outil de watermarking suive un standard particulier pour implémenter le watermarking sur l'application, de manière à ce que l'outil de détection soit en mesure de le lire par la suite. Tant que cette solution ne sera pas standardisée et adoptée par les plus grands stores d'applications, on ne peut attendre grand-chose de cette solution.

4.4 Unpacking d'application, vers des techniques de plus en plus complexes

Les packers sont le résultat d'une mise en commun de toutes les techniques les plus efficaces pour limiter la rétro-ingénierie d'applications. A chacune des mises à jour des packers, les fonctionnalités évoluent pour adopter ce qui se fait de mieux en matière de protection contre la rétro-ingénierie.

Le fait de combiner un panel de techniques de protection permet de limiter un grand nombre d'attaques possibles : nous avions en effet vu que la rétro-ingénierie dynamique peut-être utilisée pour faciliter l'analyse statique, et inversement. C'est ici que réside toute la force de ces packers : ils sont évolutifs. Les auteurs de "Adaptive Unpacking Of Android Apps" [36] expliquent d'ailleurs que c'est pour cette raison que les techniques de contournement sont si peu efficaces.

On se retrouve alors avec des techniques de contournement itératives, comme avec PackerGrind. Les résultats obtenus sont particulièrement pertinents, quoique déjà très limités par rapport à une application qui n'aurait pas été packée. En effet, l'article sus-mentionné montre que les résultats obtenus sont souvent partiels, puisque l'implémentation de l'outil ne permet pas de récupérer l'ensemble du code de l'application. Aussi, le coût pour l'attaquant est particulièrement important, dans la mesure où la technique mise en œuvre est complexe à comprendre et difficile à mettre en place, tandis que le coût pour les équipes de développement est faible, les packers étant généralement simples à mettre en oeuvre, voire directement automatisés dans les processus de développement.

Contrairement aux détecteurs de debugging, qui eux sont rythmés par les techniques d'attaques, ce sont ici les unpackers qui sont rythmés par les techniques de protection : les unpackers n'ont d'autres moyens que de contourner ces stratégies de sécurié une à une, spécifiquement. Une conséquence à cela serait de voir les packers évoluer, tandis que les unpackers devraient mettre en place des techniques de plus en plus complexes pour arriver à des résultats de plus en plus incomplets, comme cela peut être le cas pour l'outil DexHunter [36]. Comme le prédisent un certain nombre de chercheurs dans le domaine [14], les packers sont l'avenir de la sécurité des applications Android. Ils offrent un niveau de sécurité impressionnant tout en garantissant un coût de mise en oeuvre faible pour les développeurs d'applications.

5 Conclusion

Aujourd'hui, les smartphones contiennent un grand nombre de données à caractère privé ou professionnel. Par ailleurs, Android représente 85 % des ventes de smartphone dans le monde. La sécurité de ces appareils étant devenu une nécessité, la sécurité des applications sous Android est devenue un enjeu majeur de l'ère du smartphone. De part la structure standardisée de ces applications, ainsi que le manque de sensibilisation de certaines équipes de développeurs, ces dernières sont devenues le vecteur d'attaque principal des pirates. Ainsi, le nombre de techniques d'attaque et d'outils a explosé ces dernières années, poussant les équipes de sécurité à redoubler d'efforts pour atteindre le niveau de protection attendu.

Cet état de l'art a permis de montrer qu'aujourd'hui encore, une course technologique s'opère entre d'un côté les chercheurs en sécurité qui mettent en oeuvre des stratégies de défense telles que l'obfuscation, la détection de debugging ; et d'un autre côté les pirates mettant en oeuvre des attaques toujours plus complexes. Cette course technologique, encore loin d'être terminée, permet déjà d'entrevoir des éléments de réponse sur l'avenir de la sécurité d'Android.

Notamment, il ressort de cette étude que l'utilisation de packers est et restera très probablement la meilleure technique de défense, dans la mesure où les stratégies implémentées sont complémentaires et combinées, l'obfuscation et la détection de debugging présentant toutes deux des fragilités importantes lorsqu'elles sont implémentées seules. Il reste néanmoins des possibilités de contournement de ces

protections, qui résident dans l'architecture-même du système d'exploitation Android. On peut, pour cette raison, s'attendre à voir des techniques de rétro-ingénierie plus axées sur l'analyse dynamique, exploitant des fonctionnalités particulières d'Android plutôt que des mauvaises configurations dans l'application.

References

[1] [n. d.]. android application securityDexGuard: Android App Security. Retrieved 5 mai, 2020 from https://www.guardsquare.com/en/products/dexguard?utm_medium=ppc&utm_term=&utm_campaign=SYN%20-%20PPC%20-%20DSA

[2] [n. d.]. Android/iOS/H5 Application Security Reinforcement. Retrieved 5 mai, 2020 from https://www.ijiami.cn/enappProtect

[3] [n. d.]. Chiffres clés : les OS pour smartphones. Retrieved 5 mai, 2020 from https://www.zdnet.fr/actualites/chiffres-cles-les-os-pour-smartphones-39790245.htm

[4] [n. d.]. Chiffres clés : les ventes de mobiles et de smartphones. Retrieved 5 mai, 2020 from https://www.zdnet.fr/actualites/chiffres-cles-les-ventes-de-mobiles-et-de-smartphones-39789928.htm

[5] [n. d.]. Class Loader - Android Developers. Retrieved 5 mai, 2020 from https://developer.android.com/reference/java/lang/ClassLoader

[6] [n. d.]. c'est le serpent qui se mord la queue. Retrieved 5 mai, 2020 from https://fr.wiktionary.org/wiki/c%E2%80%99est_le_serpent_qui_se_mord_la_queue

[7] [n. d.]. dx tool - what is it ? Retrieved 5 mai, 2020 from https://stackoverflow.com/a/23936884

[8] [n. d.]. The easiest and powerful mobile app security solution. Retrieved 5 mai, 2020 from https://liapp.lockincomp.com/

[9] [n. d.]. The powerful code modification platform behind Cydia. Retrieved 5 mai, 2020 from http://www.cydiasubstrate.com/

[10] [n. d.]. ProGuard: JavaScript and Android App Optimizer. Retrieved 5 mai, 2020 from https://www.guardsquare.com/en/products/proguard

[11] 2017. Android Applications Reversing 101. Retrieved 5 mai, 2020 from https://www.evilsocket.net/2017/04/27/Android-Applications-Reversing-101/

[12] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. 2016. Statistical Deobfuscation of Android Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, USA, 343–355. https://doi.org/10.1145/2976749.2978422

[13] Andrea-Susana Cuadros Casta. 2015. *ANDROID ROOTING:METHODS, DETECTION,AND EVASION*. Ph.D. Dissertation. Barcelone, Espagne. AAT 8506171.

[14] Sen Chen, Ting Su, Lingling Fan, Guozhu Meng, Minhui Xue, Yang Liu, and Lihua Xu. 2018. Are Mobile Banking Apps Secure? What Can Be Improved?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 797–802. https://doi.org/10.1145/3236024.3275523

[15] Pau Oliva Fora. 2014. Beginners Guide to Reverse Engineering Android Apps. RSA CONFERENCE. https://www.youtube.com/watch?v=7SRfk321I5o

[16] Z. He, G. Ye, L. Yuan, Z. Tang, X. Wang, J. Ren, W. Wang, J. Yang, D. Fang, and Z. Wang. 2019. Exploiting Binary-Level Code Virtualization to Protect Android Applications Against App Repackaging. *IEEE Access* 7 (2019), 115062–115074. https://doi.org/10.1109/ACCESS.2019.2921417

[17] Liu Peng Wu Dinghao Huang Heqing, Zhu Sencun. 2013. A Framework for Evaluating Mobile App Repackaging Detection Algorithms. In *Trust and Trustworthy Computing*. Springer Berlin Heidelberg, 169–186. https://doi.org/10.1007/978-3-642-38908-5

[18] ibotpeaches. [n. d.]. Apktool. Retrieved 5 mai, 2020 from https://ibotpeaches.github.io/Apktool/

[19] Jin-Hyuk Jung, Ju Young Kim, Hyeong-Chan Lee, and Jeong Hyun Yi. 2013. Multi-signature based integrity checking scheme for detecting modified applications on android. *SENSOR LETTERS* 11, 9 (2013), 1820 – 1827. https://doi.org/10.1166/sl.2013.3004

[20] Jin-Hyuk Jung, Ju Young Kim, Hyeong-Chan Lee, and Jeong Hyun Yi. 2013. Repackaging Attack on Android Banking Applications and Its Countermeasures. *Wireless Personal Communications* 73, 4 (Dec. 2013), 1421–1437. https://doi.org/10.1007/s11277-013-1258-x

[21] Kyeonghwan Lim, Younsik Jeong, Seong je Cho, Minkyu Park, and Sangchul Han. 2016. An Android Application Protection Scheme against Dynamic Reverse Engineering Attacks. *JoWUA* 7 (2016), 40–52.

[22] Seongeun Kang Souhwan Jung Long Nguyen-Vu, Ngoc-Tu Chau. 2017. Android Rooting: An Arms Race between Evasion and Detection. (2017). https://doi.org/10.1155/2017.4121765

[23] Seong-je Cho Minkyu Park Nak Young Kim, Jaewoo Shim and Sanghcul Han. 2016. Android Application Protection against Static Reverse Engineering based on Multidexing. *Journal of Internet Services and Information Security (JISIS)* 6, 4 (2016), 54–64. https://doi.org/10.1109/ACCESS.2019.2921417

[24] Kali Org. [n. d.]. dex2jar Package Description. Retrieved 5 mai, 2020 from https://tools.kali.org/reverse-engineering/dex2jar

[25] owasp-mstg contributors. 2018. 0x05c-Reverse-Engineering-and-Tampering. Retrieved 5 mai, 2020 from https://github.com/OWASP/owasp-mstg/blob/master/Document/0x05c-Reverse-Engineering-and-Tampering.md

[26] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Rage against the Virtual Machine: Hindering Dynamic Analysis of Android Malware. In *Proceedings of the Seventh European Workshop on System Security*. Association for Computing Machinery, New York, NY, USA, Article 5, 6 pages. https://doi.org/10.1145/2592791.2592796

[27] sebsauvage. 2020. Hacking d'applications Android. Retrieved 5 mai, 2019 from https://sebsauvage.net/wiki/doku.php?id=apk-hacking

[28] Ankit Sinhal. 2017. Closer Look At Android Runtime: DVM vs ART. Retrieved 5 mai, 2019 from https://android.jlelse.eu/closer-look-at-android-runtime-dvm-vs-art-1dc5240c3924

[29] Do Son. 2019. Soot v4.1 releases: A framework for analyzing and transforming Java and Android applications. Retrieved 5 mai, 2020 from https://securityonline.info/soot-java-optimization-framework/

[30] X. Sun, J. Han, H. Dai, and Q. Li. 2018. An Active Android Application Repacking Detection Approach. In *2018 10th International Conference on Communication Software and Networks (ICCSN)*. 493–496.

[31] TARJA SYSTA. 2000. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. University of Tampere, Depar tment of Computer and Information Sciences, Finland.

[32] Franklin Tchakounte and Paul Dayang. 2013. Qualitative Evaluation of Security Tools for Android. *Maejo international journal of science and technology* 2 (2013).

[33] Franklin Tchakounte and Paul Dayang. 2013. System Calls Analysis of Malwares on Android. *Maejo international journal of science and technology* (2013).

[34] P. Tonella. 2005. Reverse engineering of object oriented code. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* 724–725. https://doi.org/10.1109/ICSE.

2005.1553682

[35] S. K. Udupa, S. K. Debray, and M. Madou. 2005. Deobfuscation: reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering (WCRE'05)*. 10 pp.–54.

[36] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu. 2017. Adaptive Unpacking of Android Apps. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 358–369.

[37] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *2015 IEEE Symposium on Security and Privacy*. 674–691.

[38] NGOC MINH NGO YAUHEN LEANIDAVICH ARNATOVICH1, LIPO WANG and CHARLIE SOH. 2018. A Comparison of Android Reverse EngineeringTools via Program Behaviors Validation Based on Intermediate Languages Transformation. *Maejo international journal of science and technology* (2018). https://doi.org/10.1109/ACCESS.2018.2808340

[39] Yin Haoyang Zhang Yueqian, Luo Xiapu. 2015. DexHunter: Toward Extracting Hidden Code from Packed Android Applications. In *Computer Security – ESORICS 2015*. Springer International Publishing, 293–311. https://doi.org/10.1007/978-3-319-24177-7