Analyse de maliciels sur Android par l'analyse de la mémoire vive

Mémoire

Bernard Lebel

Maîtrise en informatique
Maître ès sciences (M. Sc.)

Québec, Canada

Analyse de maliciels sur Android par l'analyse de la mémoire vive

Mémoire

Bernard Lebel

Sous la direction de:

Mohamed Mejri, directeur de recherche

Résumé

Les plateformes mobiles font partie intégrante du quotidien. Leur flexibilité a permis aux développeurs d'applications d'y proposer des applications de toutes sortes : productivité, jeux, messageries, etc. Devenues des outils connectés d'agrégation d'informations personnelles et professionnelles, ces plateformes sont perçues comme un écosystème lucratif par les concepteurs de maliciels. Android est un système d'exploitation libre de Google visant le marché des appareils mobiles et est l'une des cibles de ces attaques, en partie grâce à la popularité de celui-ci. Dans la mesure où les maliciels Android constituent une menace pour les consommateurs, il est essentiel que la recherche visant l'analyse de maliciels s'intéresse spécifiquement à cette plateforme mobile. Le travail réalisé dans le cadre de cette maîtrise s'est intéressé à cette problématique, et plus spécifiquement par l'analyse de la mémoire vive.

À cette fin, il a fallu s'intéresser aux tendances actuelles en matière de maliciels sur Android et les approches d'analyses statiques et dynamiques présentes dans la littérature. Il a été, par la suite, proposé d'explorer l'analyse de la mémoire vive appliquée à l'analyse de maliciels comme un complément aux approches actuelles. Afin de démontrer l'intérêt de l'approche pour la plateforme Android, une étude de cas a été réalisée où un maliciel expérimental a été conçu pour exprimer les comportements malicieux problématiques pour la plupart des approches relevées dans la littérature. Une approche appelée l'analyse différentielle de la mémoire vive a été présentée afin de faciliter l'analyse. Cette approche utilise le résultat de la différence entre les éléments présents après et avant le déploiement du maliciel pour réduire la quantité d'éléments à analyser. Les résultats de cette étude ont permis de démontrer que l'approche est prometteuse en tant que complément aux approches actuelles. Il est recommandé qu'elle soit le sujet d'études subséquentes afin de mieux détecter les maliciels sur Android et d'en automatiser son application.

Abstract

Mobile devices are at the core of modern society. Their versatility has allowed third-party developers to generate a rich experience for the user through mobile apps of all types (e.g. productivity, games, communications). As mobile platforms have become connected devices that gather nearly all of our personal and professional information, they are seen as a lucrative market by malware developers. Android is an open-sourced operating system from Google targeting specifically the mobile market and has been targeted by malicious activity due the widespread adoption of the latter by the consumers. As Android malwares threaten many consumers, it is essential that research in malware analysis address specifically this mobile platform. The work conducted during this Master's focuses on the analysis of malwares on the Android platform.

This was achieved through a literature review of the current malware trends and the approaches in static and dynamic analysis that exists to mitigate them. It was also proposed to explore live memory forensics applied to the analysis of malwares as a complement to existing methods. To demonstrate the applicability of the approach and its relevance to the Android malwares, a case study was proposed where an experimental malware has been designed to express malicious behaviours difficult to detect through current methods. The approach explored is called differential live memory analysis. It consists of analyzing the difference in the content of the live memory before and after the deployment of the malware. The results of the study have shown that this approach is promising and should be explored in future studies as a complement to current approaches.

Table des matières

Liste des tableaux

Table des figures

Liste d'abréviations

AD analyse dynamique

ADB Android Debug Bridge

ADMV analyse différentielle de la mémoire vive

AFC analyse de flux de contrôle

AFD analyse du flux de données

AM analyse par marqueurs (Taint Analysis)

AMV analyse de la mémoire vive

AOSP Android Open Source Project

AOT compilation anticipée (Ahead-Of-Time Compilation)

AP analyse des permissions

API Application Programming Interface

APK Android Package

ARP Address Routing Protocol

ART Android Runtime

AS analyse statique

ASLR distribution aléatoire de l'espace d'adressage (Address Space Layout Randomization)

CIC communication intercomposante

DAMM Differential Analysis of Malware in Memory

DMA Direct Memory Access

GND Gestion Numérique des Droits (Digital Rights Management)

MVD machine virtuelle Dalvik (Dalvik Virtual Machine)

ELF fichier Linux exécutable (Executable Linux File)

EUID identifiant d'utilisateur effectif (Effective User Id)

GID identifiant de groupe (Group Id)

CAM couche d'abstraction matérielle (Hardware Abstraction Layer)

IDT IDT Interrupt Descriptor Table

JIT compilation à la volée (Just-In-Time Compilation)

JNI Java Native Interface

JTAG Joint Test Action Group

LiME Linux Memory Extraction

LKM Linux loadable kernel module

PCI Peripheral Component Interconnect

PID identifiant de processus (Process Id)

PLT table des liens des procédures (Procedure Linkage Table)

PPID identifiant du processus parent (Parent Process Id)

SDK kit de développement logiciel (Software Development Kit)

SE système d'exploitation

SELinux Security Enhanced Linux

UID identifiant d'utilisateur (User Id)

MV machine virtuelle (Virtual Machine)

VPN Virtual Private Network

XML eXtensible Markup Language

À mon épouse, Marie-Anne

Remerciements

Je tiens à remercier sincèrement M. Mohamed Mejri pour sa supervision et son temps tout au long de ma maîtrise. J'ai apprécié la liberté laissée pendant la rédaction et la patience qu'il a démontrées tout au long de mon projet lorsque l'avancement est devenu plus lent due à des obligations professionnelles.

Je souhaite remercier le Fonds de recherche Nature et technologies (FRQNT) et le conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) pour leur contribution financière à cette maîtrise.

Je tiens également à remercier Thales Canada Inc., et plus spécifiquement les gens du bureau de Québec, pour le soutien financier, matériel et moral au courant de ma maîtrise.

J'envoie mes remerciements également à ma belle-famille pour leur encouragement et l'intérêt porté à mes travaux.

Un merci du plus profond de mon coeur à ma famille pour leur support et, tout particulièrement, mes parents Claude et Colette. Vous nous avez toujours soutenus dans nos projets. Votre détermination et votre énergie ont toujours été une source d'inspiration pour moi et ont su me donner la résolution pour terminer cette étape de ma vie. Merci de ne pas avoir perdu espoir au courant de ces longues années d'étude et merci de vos encouragements. Je vous serai toujours reconnaissant de m'avoir permis d'entreprendre ces longues études en vous souciant d'en adoucir les rigueurs de la vie d'étudiant.

Marie-Anne, mon amour, ma plus chère amie et ma tendre épouse, je te remercie infiniment pour ton soutien inconditionnel tout au long de mes aventures académiques. Toujours de bon conseil, tu as su m'inspirer par ton amour sans borne, ta compréhension, ta vivacité d'esprit et ton propre parcours académique. Tu as su m'aider à garder mon équilibre mental par toutes nos folies qui animent notre quotidien. Merci infiniment d'être celle que tu es, de m'avoir permis de terminer ce chapitre de ma vie et d'entamer la suite avec moi. Je t'aime à la folie.

En terminant, je dédie mes derniers remerciements à mes deux félins, Arthur et Gustave. Vous avez su me tenir compagnie les longues nuits isolées de rédaction et me rappeler la simplicité de la vie, une sieste de clavier à la fois.

Introduction

Les téléphones intelligents occupent une place centrale dans le quotidien. Ils sont devenus des outils de divertissement, de travail, de planification, de notes, etc. Les appareils intelligents se distinguent notamment par le système d'exploitation (SE) sur lequel ils reposent pour fonctionner. Android est un SE au code ouvert (*open-source*) qui vise ce marché. Il est maintenu par Google, mais adopté par grands nombres de joueurs de l'industrie des plateformes mobiles. Kaspersky Lab et INTERPOL [68] rapportent qu'en 2014 les appareils utilisant Android représentaient plus de 84% du marché mondial. Cette plateforme commune permet aux utilisateurs d'avoir accès à une expérience unifiée en limitant les disparités entre les appareils de différents fabricants. Notamment, il est possible pour un utilisateur d'avoir non seulement le même type d'interface graphique d'appareil en appareil, mais également d'avoir accès aux mêmes banques d'applications.

Dans les dernières années, cette popularité a suscité l'intérêt des concepteurs de maliciels qui y ont vu un nouveau vecteur d'attaques permettant de mettre la main sur les données personnelles abondantes des utilisateurs qui résident sur ces appareils[86]. Selon le modèle CIA (Confidentiality, Integrity, Availability) décrit par Bishop [14], les maliciels sont définis comme étant tout code visant à porter atteinte à :

— la confidentialité, c.-à-d. la protection des données protégées sur le système ;

— l'intégrité du système, c.-à-d. l'origine des données et le niveau de protection de ces données et/ou

— la disponibilité, c.-à-d. la capacité d'utiliser un système lorsque désiré.

Les approches traditionnelles de détection par signature c.-à-d. la détection de patrons connus (comparaison de somme de contrôle de l'application, reconnaissance de chaînes de caractères connus, etc.) ont rapidement été contournées par ces acteurs. Des techniques comme de la recompilation, du renommage de fichiers, l'encodage de chaîne de caractères et bien d'autres [99] ont vu le jour afin de brouiller les patrons recherchés par les antivirus. Cette réalité a mis de l'avant un besoin d'identifier des approches d'analyses de maliciels plus efficaces et adaptées à la plateforme que l'approche par détection de signature.

Les courants de recherches peuvent être regroupés en deux principales familles, soit les approches d'analyse statique et dynamique.

Moser *et al.* [85] définissent l'analyse statique (AS) comme l'analyse des composantes d'une application qui sont accessibles sans nécessiter d'exécuter cette dernière pour les obtenir. Les éléments ciblés par cette analyse peuvent être les fichiers binaires, les bibliothèques logicielles, les images, les fichiers de chaînes de caractères et autre contenu disponible sans requérir à l'exécution de l'application.

L'analyse dynamique (AD) est l'acquisition d'événements et de marqueurs survenant durant l'exécution d'une application afin de déterminer si cette dernière est malicieuse. L'analyse des actions effectuées par la plateforme et/ou l'application analysée permet notamment de capturer les éléments qui ne peuvent être résolus qu'au moment de l'exécution et qui peuvent échapper à l'analyse statique. Par exemple, la réponse d'un serveur à une requête donnée.

La plupart des approches peuvent être classées en tout ou en partie dans l'une de ces familles. Les approches diffèrent entre elles par le type d'éléments utilisés pour l'analyse, par le traitement fait sur ces éléments afin d'établir s'il y a présence d'un comportement malicieux et par leur résilience aux tentatives d'évasion de la détection communément retrouvées dans les maliciels Android.

L'analyse de la mémoire vive (AMV) est une approche d'analyse dynamique ayant reçu peu d'attention sur Android bien qu'elle soit bien établie sur d'autres SE comme Windows ou Linux. La mémoire vive est une composante essentielle au fonctionnement d'un système informatisé. Elle permet de stocker les données devant être manipulées par le processeur lors du fonctionnement d'un processus. Les temps d'accès à cette mémoire sont plus rapides que pour les périphériques de stockage (p. ex. disque dur, mémoire Flash, etc.), mais, contrairement à ces derniers, son contenu est perdu lorsqu'elle est mise hors tension. Puisqu'elle permet de stocker les résultats intermédiaires des différentes opérations du microprocesseur, il a été démontré que les données qui y transitent peuvent être extraites par des techniques d'analyse de la mémoire vive (AMV). Par exemple, les travaux de Saltaformaggio *et al.* [101] ont permis de reconstruire l'interface graphique d'une application Android à partir des parcelles de données extraites de la mémoire vive. Ces données peuvent également demeurer récupérables même après que leur espace de stockage soit libéré pour une allocation future [109]. Or, cette approche, qui tire ses racines de l'informatique d'enquête (ou *cyberforensic*), a été démontrée réalisable par des travaux précédents sur Android (p. ex. [80, 101, 113]), mais son applicabilité et ses forces demeurent mal comprises.

L'objectif de ce mémoire est d'évaluer si l'AMV est une approche d'intérêt pour la plateforme Android en tant qu'outil d'analyse de maliciels. Plus précisément, une technique adaptée à l'analyse de maliciels est proposée afin d'aider les analystes à isoler les éléments d'intérêt adaptés à ce contexte : l'analyse différentielle de la mémoire vive.

Pour ce faire, il incombe de comprendre les mécanismes internes de la plateforme. À cette fin, le chapitre 1 présente les principales composantes d'Android et son fonctionnement global.

Par la suite, le chapitre 2 présente l'état de l'art sur les maliciels Android et les méthodes d'analyses statiques et dynamiques présentes dans la littérature pour les détecter. Le chapitre 3 s'intéresse plus spécifiquement aux travaux précédents sur l'analyse de la mémoire vive et situe les principaux concepts que l'on retrouve dans ce domaine. Le chapitre 4 s'appuie sur les concepts présentés au chapitre 3 pour présenter une étude de cas menée dans le cadre de ces travaux afin de démontrer la pertinence de l'analyse différentielle appliquée à l'AMV sur Android. La description du scénario, l'expérimentation, les résultats obtenus et la conclusion de l'étude et ses impacts y sont rapportés. Finalement, une conclusion reprend les principales contributions des présents travaux et propose des pistes de recherche futures.

Chapitre 1

Android

Afin de comprendre le comportement des maliciels et l'analyse de ceux-ci sur Android, il est nécessaire de s'intéresser au fonctionnement du SE lui-même. Ce chapitre documente les principales composantes de l'écosystème d'Android requises pour la compréhension de l'analyse de maliciels sur la plateforme. Plus précisément, la structure interne du SE, le fonctionnement des applications et les principales mesures de sécurité d'Android sont décrits ci-après.

1.1 Structure interne du SE

Android est construit en suivant une structure en couche telle que représentée au tableau 1.1 (tirée de [50]), et est construit sur la base d'un noyau Linux. Ce type d'architecture permet d'abstraire les actions élémentaires (accès mémoire, gestion de l'alimentation, etc.) pour les développeurs d'applications et d'isoler les composantes les unes des autres pour des questions de sécurité.

Chaque couche est responsable d'exécuter un ensemble spécifique de tâches et communique aux autres couches via des interfaces clairement définies. Elles sont présentées ci-après.

Couche applicative La couche applicative est le domaine d'exécution de l'ensemble des applications Android déployées sur un appareil. Le SE Android expose un kit de développement logiciel (Software Development Kit, SDK) en Java permettant la création d'applications capables d'interagir avec le système via l'Application Programming Interface (API) d'Android. Le SDK et l'API sont mis à jour régulièrement par Google tant pour corriger des problèmes de sécurité que pour l'ajout de nouvelles fonctionnalités. Pour qu'une application s'exécute sur un appareil Android, celui-ci doit avoir une version du SE suffisamment à jour pour supporter les fonctionnalités de la version de l'API visée. Pour cette raison, un développeur peut volontairement choisir de créer une application visant une version de l'API antérieure à la dernière disponible. De cette façon, l'application peut être utilisable sur des appareils plus

Android

Couche applicative	Alarme, Navigateur, Calculatrice, Calendrier, Caméra, Horloge, Contacts, Téléphone, Courriel, Accueil, Messagerie instantanée, Visionneuse de contenu multimédia, Album photo, SMS/MMS, Appel vocal
Cadriciel Android	Fournisseurs de contenu (ou *Content Providers*), Gestionnaires (ou *Managers*), Système de vues
Bibliothèques logicielles natives	**Engin d'exécution Android**
Gestionnaire audio, Freetype, Libc, Cadriciel média, OpenGL/ES, SQLite, SSL, Gestionnaire de surface, WebKit	Bibliothèque logicielle élémentaire, MVD/ART
CAM	Audio, Bluetooth, Caméra, Gestion Numérique des Droits (Digital Rights Management, GND), Stockage externe, Graphiques, Entrées, Média, Capteurs, TV
Noyau Linux	Pilotes de périphériques, gestionnaire d'alimentation.

Tableau 1.1 – Architecture d'Android.

anciens. Une fois installées sur un appareil, les applications utilisent le cadriciel (ou *framework*) d'Android pour communiquer avec les différentes composantes du SE.

Cadriciel Android Cette couche est responsable d'offrir aux applications les services applicatifs qu'elles nécessitent pour accomplir leurs tâches. Le tableau 1.2 reprend la description de John [67] et présente les principales composantes de cette couche. Les différentes abstractions offertes par cette couche sont ensuite redirigées vers les couches subséquentes pour le traitement.

Composante	Description
Activity Manager	Gestion du cycle de vie d'une application (démarrage, transitions, reprise, etc.)
Content Providers	Gestion des communications inter applications
Telephony Manager	Gestion des appels vocaux, habituellement téléphonique.
Location Manager	Gestion de la géolocalisation.
Resource Manager	Gestion des différents types de ressources sur l'appareil.

Tableau 1.2 – Composantes principales du cadre d'applications Android.

Engin d'exécution Android Cette couche est responsable de l'exécution des applications en tant que telles. En plus de contenir le nécessaire à l'exécution des applications Android, cette couche est responsable de la conversion du *bytecode* Dalvik en instructions-machines exécutables par le processeur de la plateforme.

Jusqu'à la version 4.4.4, cette conversion est réalisée par la compilation à la volée (Just-In-Time Compilation, JIT) par un interpréteur contenu dans la machine virtuelle Dalvik (Dalvik Virtual

Machine, MVD). L'interpréteur est conçu pour conserver en mémoire vive le code compilé des sections de *bytecode* les plus fréquemment parcourues pendant l'exécution de l'application. Cela permet de minimiser l'impact sur la performance de la compilation à la volée, tel que décrit par Cheng et Buzbee [24]. Depuis la version 5.0 d'Android, Android Runtime (ART) remplace la MVD officiellement. Ce nouvel environnement d'exécution délaisse la compilation à la volée et utilise plutôt la compilation anticipée (Ahead-Of-Time Compilation, AOT). Tel qu'il est expliqué par Ghuloum *et al.* [42], ce changement de paradigme améliore les performances et réduit la consommation énergétique. Ces optimisations se font au coût d'un temps d'installation plus long et de plus d'espace de stockage utilisé par installation.

Bibliothèques logicielles natives Il arrive que certaines parties d'une application nécessitent d'être optimisées afin de répondre aux besoins de performances d'une utilisation donnée. Le rendu 3D, la cryptographie ou encore la lecture de contenu multimédia en sont de bons exemples. Dans ces cas, des bibliothèques logicielles natives (C/C++) offrent aux applications Android la capacité d'exécuter du code natif afin d'optimiser la vitesse d'exécution de segments de code coûteux en temps de traitement ou requérant un contrôle plus fin de ses structures de données internes.

Couche d'abstraction matérielle Étant un SE ouvert, Android permet aux fabricants de choisir les composantes qu'ils désirent intégrer à leurs appareils. La couche d'abstraction matérielle (Hardware Abstraction Layer, CAM) offre une interface universelle permettant d'interagir avec les périphériques sans se soucier des particularités d'implémentations de chaque fabricant. Ces particularités sont déléguées aux pilotes de périphérique. L'implémentation de ces derniers est typiquement la responsabilité des manufacturiers d'appareils Android puisqu'ils adaptent le SE lors de la conception de ces dispositifs [45]. Le tableau 1.3, inspiré de Google Inc [45], présente les familles de périphériques offertes par cette couche.

Couche d'abstraction matérielle (CAM)				
Audio	Caméra	Bluetooth	DRM	Stockage Externe
Graphiques	Entrées	Média	Capteurs	Télévision

Tableau 1.3 – CAM sur Android.

Noyau Linux Le SE Android est construit sur la base d'un noyau modifié de Linux [46]. Ce noyau orchestre les fonctionnalités élémentaires d'Android telles que la gestion des fils d'exécutions, des périphériques, de l'alimentation, de la mémoire vive et du contrôle des accès et privilèges. Ce noyau est compilé à partir d'une mouture appelée Bionic [49] découlant de *glibc* et de *BSD*. L'une des caractéristiques principales de Bionic est qu'il a été optimisé pour les processeurs à faible fréquence d'horloge, ce qui était, à l'origine, le cas des processeurs des premiers appareils Android, tel que décrit par Devos [27]. Cette composante d'Android est la plus critique puisqu'une erreur ou une corruption à ce niveau est susceptible de compromettre

l'entièreté de l'appareil. De ce fait, il s'agit de l'une des cibles d'attaques les plus prisées pour l'exploitation d'Android (voir [47]).

1.2 Applications Android

L'interaction d'une application avec le SE s'effectue via un API permettant d'utiliser les ressources disponible sur Android comme la géolocalisation, état de l'appareil ou l'envoi de messages textes. Le langage de programmation utilisé pour la création des applications est majoritairement le Java. Il est également possible d'intégrer du code en C/C++ à l'aide de l'interface Java de programmation native ou Java Native Interface (JNI).

Lors de la compilation, le code Java est traduit dans un langage intermédiaire, appelé *bytecode Dalvik*. Cette forme intermédiaire permet de décrire les comportements de l'application dans avec une fine granularité (c.-à-d. de façon détaillée). Les instructions sous cette forme demeurent agnostiques à une architecture du microprocesseur et ne peuvent être directement exécutées par celui-ci. La conversion vers les instructions-machine appropriées pour un microprocesseur donné s'effectue par la plateforme elle-même lors de l'installation ou de l'exécution de l'application, selon l'engin d'exécution utilisé (ART ou MVD). Les sections de code en C/C++ sont, quant à elles, directement compilées en instructions-machines. Cela requiert que les bibliothèques logicielles C/C++ soient recompilées à l'aide des outils de compilation adaptés à chaque architecture de microprocesseur viser par le développeur.

Les applications Android sont un assemblage de différentes composantes. Ces composantes sont résumées au tableau 1.4. La description complète de ces composantes est présentée par Google Inc [53]. Lorsqu'une composante échange des informations avec une autre composante, il est question de communication intercomposante (CIC). Les CIC peuvent être réservées à des composantes détenant des permissions explicitement attribuées ou encore en étant directement sollicitées par l'émetteur.

Afin d'être valide, le code de l'application est accompagné de son manifeste (*AndroidManifest.xml*). Il s'agit d'un fichier en format eXtensible Markup Language (XML) décrivant les paramètres d'exécutions nécessaires pour faire fonctionner correctement l'application. Il contient :

- le nom de l'application et sa version ;
- la description de certaines composantes (*Activities*, *Intent Filters*, *Broadcast Receivers*, *Services* et *Content Providers*) ;
- les permissions demandées par l'application pour communiquer avec des composantes protégées de l'API ;
- le point d'entrée pour l'exécution de l'application ;
- la version minimum de l'API que requiert l'application pour fonctionner correctement et

— l'ensemble des bibliothèques logicielles externes auxquelles l'application est liée.

Lorsque l'application est construite en vue d'être déployée, le code compilé (sous la forme d'un ou plusieurs fichiers *classes[x].dex* où x est la numérotation du fichier si le code est réparti en plusieurs fichiers), le manifeste et les différentes ressources (bibliothèques logicielles, images, base de données, etc.) sont consolidés dans une archive compressée en format Zip nommée Android Package (APK) [48]. Les fichiers inclus dans l'archive sont signés à l'aide d'une clef privée incluse dans un certificat généré par le développeur ou automatiquement si l'application est en mode déverminage. La clef publique du certificat et ses informations sont incluses dans l'APK de même que les valeurs des signatures des fichiers validant leur intégrité. Cette clef publique est utilisée pour valider la signature des fichiers et ainsi garantir que les fichiers présents dans l'APK sont ceux d'origine.

1.3 Sécurité du SE

Au cours de son existence, Android a subi plusieurs modifications visant à améliorer la sécurité de la plateforme. L'équipe de l'Android Open Source Project (AOSP) publie des correctifs de sécurité pour limiter le risque d'actions malicieuses visant l'exploitation d'Android (vol de donnée, logiciel espion, élévation de privilèges, etc.). Ces correctifs visent également à corriger les vulnérabilités identifiées dans le SE. Bien que cette approche permette de corriger les menaces rapportées, elle n'est pas efficace contre celles qui ne sont pas rapportées ou dont le correctif n'est pas encore disponible. De plus, avant d'être déployés sur les appareils, ces correctifs sont révisés par les manufacturiers respectifs puis acceptés par les utilisateurs individuellement. Les détails de ce processus sont décrits par Wyatt [129]. Cela introduit d'importants délais dans le déploiement de correctifs et peut même l'empêcher si ceux-ci s'adressent à des appareils que le manufacturier ne souhaite plus supporter.

Pour ces raisons, des efforts ont été investis dans l'ajout de mécanismes proactifs de sécurité pour contenir les dommages potentiels que l'exploitation d'une faille de sécurité peut avoir sur Android. Le bac à sable applicatif, SELinux et la distribution aléatoire de l'espace d'adressage (Address Space Layout Randomization, ASLR) sont des éléments importants de cette sécurité et sont détaillés ci-après [1].

1.3.1 Bac à sable applicatif

L'une des principales défenses d'Android est son *bac à sable applicatif*, c'est-à-dire un environnement où les interactions avec les autres ressources ou applications de l'appareil sont impossibles ou fortement contrôlées. Ce contrôle s'exerce à l'aide de mesures de contrôle d'accès et des restrictions de privilèges. Le bac à sable s'applique à toutes les applications Android

1. La liste complète de ces mécanismes de protection sur Android est présentée par Google Inc [55] et le lecteur est invité à s'y référer pour un exposé détaillé.

exécutées sur un appareil et vise tant la portion Java que la partie native du code puisqu'il fait appel à plusieurs mécanismes de sécurité présents dans le noyau Linux [51]. Les identifiants d'utilisateur (User Id, UID), les identifiants de groupe (Group Id, GID) et le mécanisme de permissions sont la pierre angulaire du bac à sable applicatif d'Android.

Identifiant unique d'utilisateur (UID)

Le contrôle des accès sur la base des rôles est implanté dans Linux sous la forme d'identifiant d'utilisateur (User Id, UID). En attribuant un UID par application Android au moment de l'installation et en les forçant à s'exécuter uniquement via ce UID, chaque application est confinée dans un espace de fichiers isolé des autres applications du système. La seule exception est s'il s'agit d'un utilisateur privilégié (c.-à-d. *root*) qui peut outrepasser ces restrictions. Cependant, l'utilisateur *root* sur Android n'est pas disponible en dehors des processus du système, à moins de modification du SE ou par l'exploitation d'une faille de celui-ci. Si une application nécessite plus d'accès pour s'exécuter (par exemple, communiquer avec un serveur Web ou si elle souhaite écrire sur l'espace de stockage), elle doit obtenir des *permissions* [52].

Permissions

Les permissions sont une mesure de contrôle d'accès à l'API pour limiter l'accès à certaines fonctionnalités qui peuvent porter préjudice à l'utilisateur si elles sont octroyées à une application illégitime ou malicieuse. Pour cette raison, les permissions doivent être entérinées par l'utilisateur avant d'être données au demandeur. En plus des permissions faisant partie de l'API d'Android, il est possible de créer de nouvelles permissions afin de protéger certains accès sensibles aux données ou services qu'une application peut souhaiter exposer de façon restreinte.

Chaque permission correspond à un GID du noyau Linux. Chaque GID possède les accès aux ressources du SE requises pour l'exécution des comportements rattachés à cette permission. Pour chaque permission octroyée, Android ajoute le UID de l'application au groupe qui y correspond. C'est de cette façon que l'application obtient les privilèges lui permettant d'agir en accord avec la permission demandée.

Avant la version 6.0 d'Android, toutes les permissions demandées par une application doivent être déclarées dans son manifeste. Lors de l'installation, l'utilisateur doit accepter la totalité des permissions demandées ou refuser l'installation. La version 6.0 introduit la possibilité de demander les permissions à l'exécution et de révoquer les permissions à la pièce si l'utilisateur le désire. Cela permet un contrôle plus granulaire sur le type de comportement qu'un utilisateur autorise à une application et permet de limiter les demandes de permissions jugées exagérées. Par exemple, un utilisateur pourrait souhaiter refuser l'accès à ses contacts et à la messagerie texte de son téléphone pour une application de réveil-matin.

1.3.2 SELinux

Tel qu'expliqué par Shabtai [105], Security Enhanced Linux (SELinux) est un module de sécurité Linux introduisant le concept de contrôle d'accès obligatoire par l'utilisation d'étiquettes de sécurité pour établir des règles d'accès visant à renforcer la sécurité d'Android et protéger les accès aux composantes du noyau. À l'origine présenté par Loscocco et Smalley [79] pour PC, SELinux fait officiellement partie d'Android depuis la version 4.3 parue en 2010. Il permet de surveiller et bloquer l'accès à différents *objets* accédés par un processus. Un *objet* correspond à une ressource du SE tel un fichier, un *socket* ou un périphérique. Ils sont classés par type de ressources requises selon le niveau d'accès requis (lecture, écriture, etc.) et ont une étiquette de sécurité qui leur est assignée. Le tableau 1.5 décrit les principales composantes d'une étiquette de sécurité. Ces différentes caractéristiques servent à décrire quel utilisateur peut utiliser quel processus pour poser quelle action lors de l'accès à une ressource.

L'étiquette de sécurité peut être exprimée à l'aide d'un contexte de sécurité, ce qui correspond à une chaîne de caractère prenant la forme présentée à la figure 1.1 décrivant l'étiquette de sécurité sous une forme lisible.

Utilisateur:*Rôle*:*Type d'accès*:*Niveau de sécurité*

FIGURE 1.1 – Contexte de sécurité.

L'approche de SELinux est le refus par défaut. Cela implique que tous les accès aux ressources désirés sur le système doivent avoir été préalablement autorisés par le biais de règles d'accès se trouvant dans un fichier de politiques de sécurité faute de quoi l'accès est automatiquement refusé et journalisé par le gestionnaire d'accès de SELinux. Puisque SELinux est une composante indépendante du noyau Linux, il est en mesure de contrôler les accès de tous les utilisateurs du système, incluant ceux de l'utilisateur *root* [79]. Cette technique permet de limiter les actions qui pourraient provenir d'une élévation de privilège, c'est-à-dire d'une attaque qui obtient un accès *root* à partir d'un contexte sans privilège en exploitant une faille de sécurité. En effet, si le changement d'utilisateur se produit à partir d'un processus qui n'a pas explicitement la permission de le faire en vertu des politiques de sécurité, les actions subséquentes seront bloquées par le gestionnaire d'accès de SELinux.

1.3.3 ASLR

La distribution aléatoire de l'espace d'adressage (Address Space Layout Randomization, ASLR) vise à rendre la topologie du contenu de la mémoire vive imprévisible pour un attaquant. Ce mécanisme de sécurité complique l'exploitation d'une faille de sécurité reposant sur un débordement de tampon (*buffer overflow*). En effet, il devient difficile pour un attaquant d'injecter des appels à des ressources (p.ex. fonction exploitable, bibliothèque logicielle, exécutable, position de la pile ou du tas, etc.) en mémoire vive si leur position est inconnue. La première

apparition de l'ASLR sur Android a été appliquée à partir de la version 4.0. Il est possible de désactiver l'ASLR à la compilation du SE.

Résumé

Android est un système qui évolue rapidement afin d'offrir nouvelles fonctionnalités et sécurité accrue à ses utilisateurs. Toutefois, les concepteurs de maliciels s'adaptent continuellement aux nouvelles techniques déployées et il est essentiel de recourir à des techniques d'analyse de maliciels afin de découvrir les nouvelles menaces visant Android. Une revue du fonctionnement des maliciels sur Android ainsi que les différentes techniques existantes pour en faire l'analyse sur Android sont l'objet du chapitre 2.

Composantes	Description
Intent et Intent Filter	Un *Intent* est un message avec une demande d'action et, optionnellement, des données sérialisées à l'intention d'une autre composante utilisant un *Intent Filter* indiquant qu'elle accepte ce type de message. Les trois principales utilisations sont de lancer un *Activity*, un *Service* ou d'effectuer un *Broadcast*. Un *Intent* est explicite s'il déclare la composante destinatrice explicitement et dit implicite s'il laisse les paramètres du système et/ou l'utilisateur déterminer la composante (ou l'application) devant traiter ce message.
Fragments	Un *Fragments* est une composante modulaire interchangeable utilisée pour remplir l'interface des *Activities*.
Activity	Un *Activity* reçoit l'ensemble des interactions avec l'utilisateur. Il s'agit des différentes interfaces graphiques pouvant être utilisées pour interagir avec l'application.
Loader	Un *Loader* facilite le chargement des données depuis une source de données vers un composant d'affichage graphique comme un *Fragment* ou un *Activity*.
Service	Un *Service* est une composante d'une application pouvant s'exécuter en arrière-plan, sans interface graphique et qui est généralement responsable d'opérations longues. Le *Service* peut être utilisé dans le cadre de CIC, c'est-à-dire, permettre de passer des données entre composantes (p. ex. téléchargement d'un fichier, écoute de musique, etc.).
Broadcast Receiver	Un *Broadcast* est un message avec ou sans contenu sérialisé envoyé à l'intention de toutes composantes s'étant abonné par un *Broadcast receiver* à la catégorie visée par le *Broadcast* et qui possèdent les permissions associées, si requises. Il est possible de déclarer un *Broadcast receiver* soit dans le manifeste ou par programmation dans le code Java à l'exécution.
Content Provider	Un *Content Provider* définit une interface par laquelle une application peut permettre la création, modification et suppression de ses données à d'autres composantes tout en permettant d'encapsuler cette donnée et d'appliquer des mesures de sécurité pour la protéger. Il permet également de contrôler l'accès à ces données en obligeant toute application à avoir des permissions sur mesures avant d'être autorisé à accéder les données.
App Widget	Un *App Widget* est une interface graphique offrant des contrôles simplifiés et des informations résumées sur une application en vue d'en faire l'affichage sur l'écran d'accueil d'Android.

Tableau 1.4 – Composantes principales des applications Android.

Nom	Description	Exemples
Utilisateur	L'identifiant de l'utilisateur demandant l'accès à la ressource	*user*, *system*
Rôle	Le rôle du processus impliqué, par exemple, s'il s'agit d'un processus natif du système ou un processus utilisateur	*system_r*, *user_r*
Type	Type de l'opération demandée	*process transition*, *file write*
Niveau de sécurité	Spécifie le niveau d'accès requis dans le cadre d'un système de sécurité multi niveau (non utilisé sur Android)	*s0*

Tableau 1.5 – Éléments d'une étiquette de sécurité.

Chapitre 2

Analyse et détection des maliciels

L'étude de l'analyse des maliciels sur Android nécessite de comprendre le fonctionnement de la plateforme (comme décrit au chapitre précédent) et des maliciels qui y résident. La section 2.1 vise à décrire les tendances recensées chez les maliciels Android. C'est avec cette connaissance qu'il est possible d'établir les cibles que doivent atteindre les techniques d'analyse de maliciels sur la plateforme afin d'être efficaces. La section 2.2 recense l'état de l'art des méthodes d'analyse statique ou dynamique des comportements malicieux sur Android. Elle décrit également les principales limitations des approches actuelles et, ce faisant, légitime l'exploration d'autres techniques d'analyse (p. ex. l'analyse de la mémoire vive) complémentaires à celles existantes.

2.1 Maliciels Android

Chaque SE présente des particularités qui régissent le *modus operandi* des maliciels qui y sont destinés. Android n'échappe pas à cette logique. Il est important pour comprendre les défis que tentent de résoudre les approches d'analyse de maliciels d'expliquer les principaux vecteurs d'attaques utilisés par ceux-ci. Cette section détaille les principales techniques utilisées par les maliciels Android pour compromettre en partie ou en totalité l'appareil d'un utilisateur. Dans un premier temps, les principales approches utilisées par les maliciels pour infecter la plateforme sont présentées. Dans un second temps, les techniques relatives à l'évasion de la détection sont détaillées.

2.1.1 Vecteurs d'attaques

Les vecteurs d'attaques sont les techniques utilisées par les maliciels afin d'infecter un appareil Android. La compréhension des points d'entrées potentiels des maliciels permet d'identifier les points d'inflexion que doivent être capables de documenter les analyses de maliciels pour accroître les chances de les détecter. Ces vecteurs d'attaques sont détaillés ci-après.

Exploitation du noyau Linux Ce type d'attaque vise à exploiter directement le noyau Linux (p. ex. Lineberry [78]) ou les bibliothèques logicielles chargées dans Android (p. ex. ZLabs [139]). Elle peut permettre de poser des actions demandant plus de privilèges qu'autorisés pour un processus. Dans certains cas, elle permet de compromettre l'ensemble du système. Ces techniques d'exploitation peuvent aussi être volontairement utilisées afin d'obtenir les privilèges de l'utilisateur *root* pour étendre les fonctionnalités du SE. Un exemple de ce type d'utilisation est l'exploit TowelRoot de Hotz [63] qui peut servir à élever les privilèges de l'utilisateur pour lui permettre de modifier les configurations du noyau Linux ou encore de remplacer la version d'Android sur son appareil même si celui-ci est verrouillé par le manufacturier.

La découverte d'une faille pouvant mener à l'exploitation du noyau Linux et la création de l'exploit qui l'utilise demandent une expertise poussée sur le fonctionnement du SE ainsi que des connaissances techniques avancées (lecture d'instructions-machine, désassemblage, décompilations, examen de la pile ou du tas, lecture de rapports d'erreur, etc.). En revanche, une fois découverte et exploitée, l'exploit conçu peut être rendu public et utilisable avec un minimum d'effort sous la forme d'un outil libre ou de code réutilisable. Dans ce cas, la complexité pour abuser d'un système par cette faille de sécurité est grandement réduite. Des plateformes, comme *Metasploit*[97] et *Drozer*[87], sont d'ailleurs spécialisées dans la distribution de ce type d'exploits prêts à l'emploi.

Pour opérer, un maliciel de ce type utilise une ou plusieurs vulnérabilités du noyau, connues ou non, mettant en place les conditions favorables pour déclencher le comportement malicieux désiré. Certains exploits procèdent en altérant des structures internes du noyau contenues en mémoire vive. Celles qui sont particulièrement d'intérêt sont habituellement impliquées dans des fonctions critiques de l'exécution du SE comme la gestion du contrôle d'accès des utilisateurs. L'un des objectifs de ce type d'attaque est d'augmenter leurs accès ou privilèges sur la plateforme. Par exemple, l'exploit TowelRoot[63] utilise une vulnérabilité dans la gestion des *futex* (ou *fast userspace mutex*) permettant de suspendre un fil d'exécution si une condition donnée est remplie afin de protéger l'accès à une ressource déjà utilisée[134–136]. L'exploitation de cette vulnérabilité permet la modification de l'espace mémoire du noyau contenant la valeur de l'UID auquel appartient un fil d'exécution du noyau. Cela permet notamment d'attribuer l'UID "0" de l'utilisateur *root* à n'importe quel fil d'exécution, comme celui d'une invite de commande contrôlée par l'utilisateur. La présence de techniques de mitigation, comme l'ASLR, rend plus difficile l'exploitation du noyau par ce genre d'attaques. En effet, elles rendent l'exploitation du noyau Linux complexe et dépendante de la représentation de la mémoire vive au moment de leur exécution. De ce fait, un maliciel utilisant une même vulnérabilité sur des environnements différents (p. ex. architecture du processeur, version des bibliothèques logicielles ou du SE, etc.) peut échouer même si celui-ci est vulnérable.

Exploitation d'une application légitime Ce type d'attaque a pour but d'exploiter la plateforme par le biais de vulnérabilités présentes dans une application légitime. Il nécessite une bonne connaissance du fonctionnement interne de l'application ciblée et des interfaces qu'elle offre. Cette connaissance est habituellement acquise par la rétro-ingénierie de cette dernière afin d'y trouver des erreurs logiques ou des communications intercomposantes (CIC) non protégées. L'impact dépend surtout des permissions détenues par l'application exploitée et du niveau de sensibilité des opérations dont elle est responsable. Les conséquences peuvent être une attaque par déni de service, l'exfiltration de données sensibles et même la prise de contrôle de l'application. Les impacts de ces maliciels peuvent être mineurs sur les capacités de l'appareil, mais être majeurs pour l'utilisateur. Par exemple, une attaque prenant le contrôle d'une application possédant uniquement la permission d'accès à Internet serait dangereuse s'il s'agit d'une application de gestion bancaire dont l'exploitation pourrait engendrer des pertes financières pour l'utilisateur. En revanche, dans cet exemple, la performance de l'appareil serait peu ou pas affectée. La principale limitation de ce type d'attaque est que le comportement malicieux est confiné au même environnement d'exécution et de permissions que l'application compromise, à moins d'être combinée à une élévation de privilèges. L'ajout du module de sécurité SELinux à Android contribue à limiter les répercussions de l'élévation de privilèges. En effet, tous processus Linux découlant d'une application Android sont confinés à un utilisateur qui n'est pas autorisé par SELinux à élever ses privilèges selon les politiques de sécurité.

Applications malicieuses Les applications malicieuses sont des applications Android à part entière qui peuvent être installées par l'utilisateur (p. ex. par hameçonnage) ou à son insu sur l'appareil (p. ex. prédéployées à l'achat [32]). Leur principale raison d'être est de porter atteinte à l'intégrité du SE et de son contenu comme les applications et les données privées de l'utilisateur. Selon l'étude de Zhou et Jiang [138] portant sur 1260 maliciels récoltés entre 2010 et 2011, près de 86% de cet échantillon découlent d'applications légitimes ayant été modifiées pour y inclure du contenu malicieux. Ces maliciels sont habituellement redistribués sur différents marchés d'applications Android et sont des calques de leur version légitime. De plus, près de 36,7% de cet échantillon utilise des mécanismes d'élévation de privilège. Leurs comportements varient selon l'intention du concepteur et peuvent inclure l'envoi de textos à des services payants, le chiffrement de données personnelles dans un but d'extorsion et le vol de données personnelles[68].

2.1.2 Techniques d'évasion de la détection

Les vecteurs d'attaques présentés ci-dessus permettent à un agent malicieux d'infecter la plateforme et d'y avoir un impact indésirable, voire même dommageable pour l'usager. Or, il n'est pas suffisant de pouvoir infecter un appareil pour les concepteurs de maliciel. En effet, pour être rentable ou utilisable pour des activités illicites, un maliciel doit avoir la possibilité d'y rester suffisamment longtemps pour y exercer le comportement malicieux désiré

et, potentiellement, rentabiliser l'infection. De plus, le maliciel doit également être capable de se faire passer pour une application légitime au moment d'être soumis sur les marchés d'applications Android, faute de quoi il ne sera pas publié. Pour cette raison, les concepteurs de maliciels ont développé des techniques permettant d'éviter la détection et de masquer l'activité illicite des techniques d'analyse. Le rapport de Symantec [115] rapportent que les maliciels Android sont de plus en plus sophistiqués sur ce point. Plusieurs recherches ont étudié les capacités d'évasion de la détection des maliciels existants ou en ont proposé de nouvelles. Il est essentiel de les relever afin de mieux comprendre les limitations des techniques d'analyse aux différentes stratégies d'évasion. Les principales techniques d'évasion retrouvée sur Android sont présentées ci-dessous.

Obfuscation Wroblewski [127] définit l'obfuscation comme étant les transformations appliquées au contenu d'une application visant à rendre la compréhension ou l'analyse de celle-ci plus complexe tout en préservant le comportement de l'application transformée équivalente à celle d'origine (c.-à-d. produisant le même résultat). L'obfuscation vise généralement à limiter les capacités de rétro-ingénierie d'une application en augmentant le temps, l'effort et le niveau de connaissances requis pour y parvenir. Elle est employée pour protéger la propriété intellectuelle d'une application ou encore pour limiter l'efficacité d'analyse des engins de détection de maliciels. Sur Android, des études se sont intéressés à l'injection d'instructions Dalvik permettant de masquer le flux d'exécution réelle d'une application aux décompilateurs tout en maintenant le comportement final de l'application. Les travaux de Kovacheva [71], Rastogi *et al.* [99], Schulz [104] démontrent qu'il est possible de limiter les capacités d'analyse d'une application tout en conservant le comportement désiré à l'exécution. Pour ce faire, ils utilisent l'injection d'instructions Dalvik nuisibles à la décompilation ou l'interprétation du code par des outils d'analyse automatisés .

Une autre approche utilisée sur Android consiste à appliquer l'obfuscation directement sur le code source de l'application, avant la compilation. Maiorca *et al.* [81] ont appliqué des techniques d'obfuscation sur des échantillons de maliciels connus. Leur étude vise à démontrer la robustesse des outils de détection de maliciels disponibles sur le marché. Ces résultats sont présentés au tableau 2.1. L'application de ces techniques a été démontrée efficace pour éviter la détection sur la plupart des outils d'analyses commerciaux retenus dans l'étude. Ces résultats soutiennent la pertinence de l'obfuscation en tant que technique d'évasion de la détection.

Peu d'études se sont intéressées à appliquer des techniques d'obfuscation pour les fonctions incluses dans les bibliothèques logicielles natives en C/C++ pouvant être utilisées par une application Android. En contrepartie, peu d'outils d'analyse s'y sont intéressés également.

Détection et évitement d'engin d'analyses Cette technique permet à un maliciel d'exprimer ou d'inhiber son comportement malicieux en fonction de caractéristiques de son environnement d'exécution. Pour ce faire, il est possible de sonder différentes variables pré-

Type d'obfuscation	Description
Triviale	Technique reposant sur l'usage de l'outil DexGuard pour renommer les classes, fonctions, nom des variables, *packages* et fichiers de code sources avec des noms générés aléatoirement en maintenant la cohérence entre ces noms et ceux inclus dans le *AndroidManifest.xml*.
Réflexion	Utilisation de la propriété du Java permettant d'invoquer les fonctions contenues dans une classe en utilisant la chaîne de caractère qui la représente plutôt qu'en utilisant l'appel direct à celle-ci(voir l'annexe C).
Chiffrement des chaînes de caractères	Chiffrement de toutes les chaînes de caractères contenus dans le code de l'application avec déchiffrement à l'accès.
Chiffrement de classe	Chiffrement et compression de chaque classe d'une application et stockage sous la forme d'un tableau d'octets dans le code source avec déchiffrement dynamique à l'exécution et chargement par réflexion.
Masquage des points d'entrées	Le masquage des classes contenant des points d'entrées de l'application (c.-à-d. *Activity*, *Intent* ou *Broadcast*) par le chiffrement de celle-ci et l'adaptation du *AndroidManifest.xml* où elles sont déclarées.
Chiffrement des ressources d'une application	Les ressources (c.-à-d. les *assets* d'Android) sont chiffrées pour éviter la détection par comparaison avec des signatures connues.

Tableau 2.1 – Techniques d'obfuscation du code source.

sentes sur Android pour identifier si certaines portent des valeurs typiques d'un environnement d'analyse. Petsas *et al.* [95], Vidas et Christin [122] font état d'un ensemble de propriétés pouvant être utilisées par les applications pour détecter si le système est émulé ou correspond à un appareil réel. Afin d'effectuer cette détection, les auteurs interrogent les propriétés du système, comparent la performance du système à des mesures étalons récoltées *a priori* et font l'inventaire du matériel et des logiciels présents sur la plateforme. Le tableau 2.2 rapporte des techniques employées par les auteurs pour effectuer cette détection.

Une autre approche, documentée dans l'étude de Fratantonio *et al.* [39], permet d'éviter la détection en ajoutant un segment de code appelé *bombe logique*. Ce code sert de déclencheur à du code malicieux lorsqu'une condition donnée est atteinte. Pour être efficace, ce déclencheur doit être un événement permettant de discriminer si le maliciel est en interaction avec une victime potentielle ou avec un système d'analyse automatisé. Ainsi, il peut être une tâche trop complexe pour qu'un système automatisé puisse la compléter (p. ex. compléter le premier niveau d'un jeu). Il peut également s'agir de l'introduction d'un délai avant l'exécution du comportement malicieux.

La détection de l'utilisateur peut aussi se faire par la détection d'un patron d'interactions correspondant à ce qui est attendu d'un appareil qui est dans les mains d'un utilisateur typique. Par exemple, Petsas *et al.* [95] ont démontré qu'il est possible de conclure qu'un environnement est simulé si les valeurs retournées par les capteurs de l'appareil sont prévisibles et constantes.

Marqueurs	Description	Exemples
Comportementale	Analyse des valeurs retournées par l'API d'Android, de la configuration réseau du système.	Propriété *Build.SERIAL* Valeur de *getVoiceMailNumber()* Masque d'adresse IP
Performance	Mesure du temps de calcul requis pour exécuter une fonction donnée et performance du rendu graphique d'une scène et comparaison avec un niveau de base attendu.	Calcul de la $1048576^{\text{ième}}$ décimale de π Images par seconde lors du rendu d'une forme géométrique.
Inventaire des composantes	Énumération des composantes matérielles ou logicielles présentes sur l'appareil.	Mesures des capteurs, coordonnées GPS brutes, applications par défaut de Google, etc.
Particularité de l'environnement d'analyse	Exploiter des contraintes des outils d'analyse qui ne sont pas les mêmes dans un scénario réel, notamment dues à la grande quantité d'échantillons de maliciels qu'ils doivent traiter dans un temps limite et en l'absence d'un usager interagissant avec l'appareil.	Délai d'exéc. de l'application Attente d'une interaction utilisateur.

TABLEAU 2.2 – Techniques de détection d'engin d'analyse.

Détection des modifications sur l'application d'origine La détection des modifications consiste à appliquer des mécanismes qui s'assurent qu'au moment de l'exécution, le contenu d'une application n'a pas été modifié depuis sa compilation. Sur Android, cela peut se faire en validant la somme de contrôle (*checksum*) des fichiers de l'application avant son chargement ou en validant la signature cryptographique de l'APK à l'exécution tel qu'il a été proposé par Alexander-Bown [5]. Si les résultats ne concordent pas avec les valeurs d'origines, l'exécution s'interrompt. Cette détection n'est pas infaillible puisque la vérification est à même le code de l'application et celui-ci peut être sujet à modification par rétro-ingénierie.

De plus, il est aussi possible d'effectuer la détection d'un environnement de déverminage tel qu'expliqué par Dominic *et al.* [30]. Cette détection est utile afin de modifier le comportement d'une application où un analyste aurait injecté la propriété permettant le déverminage dans le manifeste de l'application avant de la déployer afin d'utiliser cet outil pour en faire l'analyse.

Communication par canaux cachés La communication par canaux cachés sur Android permet de dissimuler des interactions d'une composante avec une autre composante ou avec un agent extérieur à la plateforme par la manipulation d'éléments qui peuvent ne pas avoir *a priori* de rôle de communication, mais qui sont utilisés à cette fin. Il peut s'agir de l'utilisation de chiffrement pour protéger l'envoi de messages. Marforio *et al.* [82] ont démontré l'efficacité de cette capacité à déjouer les systèmes de détection de fuites de données par l'étude d'une variété de canaux cachés dans le cas de collusion entre applications (p. ex. écriture/lecture dans les journaux systèmes, le décompte du nombre de fils d'exécution). À titre d'exemple, il est possible d'obtenir un scénario de collusion à l'aide de deux applications (A et B) ayant les permissions identifiées au tableau 2.3.

Application A	**Application B**
android.permission.READ_CONTACTS	android.permission.INTERNET

Tableau 2.3 – Permissions des applications A et B.

1. A est installée sur l'appareil de l'utilisateur (p. ex. par *hameçonnage*) et l'utilisateur est incité à installer B (p. ex. promotion de fonctionnalités additionnelles).
2. A déclenche son mécanisme de lecture des contacts sur le téléphone.
3. A crée des fragments d'information avec 5 caractères à titre d'en-tête (p. ex. '_____') auxquels sont ajoutés 10 caractères ou moins de l'information dont l'on désire faire l'exfiltration pour un total de 15 caractères (longueur maximale d'un nom de fil d'exécution sur la plateforme).
4. A crée un ensemble de fils d'exécution pour chaque contact et les nomme avec l'information fragmentée à l'étape précédente en allouant un temps d'exécution pour chaque fil de T ms.
5. A attend la fin de tous les fils d'exécution avant d'itérer au prochain contact.
6. B exécute *ps -t | grep* _____ à tous les F *ms* où $F >= T * 2$ ms pour capturer tous les fils créés par A. Le résultat obtenu par B est présenté à la figure 2.1.
7. B reconstruit l'information en triant les fils d'exécutions dont le nom débute par les 5 caractères d'en-tête en ordre croissant par numéro unique de fil d'exécution. Si B a déjà obtenu le contact reconstruit, le doublon est rejeté.
8. Lorsque B détecte qu'il n'y a plus de fil d'exécution de créé, il envoie les contacts accumulés à un serveur distant pour l'exfiltration.

```
1 USER      PID   PPID  VSIZE  RSS     WCHAN              PC  NAME
2 u0_a110   28200 28137 936760 51156 futex_wait 00000000 S _____Aaron Duma
3 u0_a110   28201 28137 936760 51156 futex_wait 00000000 S _____s_1 555-55
4 u0_a110   28202 28137 936760 51156 futex_wait 00000000 S _____5-5555
```

Listing 2.1 – Résultat de "ps -t |grep _____" pendant l'exécution de A.

Dans cet exemple, B n'a jamais eu la permission de lire les contacts, ni A d'accéder à l'Internet, mais l'utilisation d'une propriété des fils d'exécution de Linux a permis d'exfiltrer la donnée sans avoir recours à des canaux de communications dédiés à cette fin. La principale difficulté de ce type d'attaque est d'avoir les deux applications complices sur le même appareil (voir l'annexe B pour le code de l'application A). Il est possible d'utiliser d'autres canaux cachés pour arriver à ce même résultat (p. ex. récepteur FM de l'appareil [38]). De plus, même en utilisant une connexion réseau explicite, il est possible de dissimuler l'information transigée à l'aide du chiffrement de la communication (p. ex. via SSL). Des rapports de Kaspersky Lab et INTERPOL [68], Symantec [114] ont recensé des maliciels Android capables d'utiliser le protocole chiffré Tor afin d'établir des liaisons avec un poste distant en conservant l'anonymat de l'émetteur et du récepteur par l'utilisation de chiffrement et de relais de connexion.

2.2 Analyse des maliciels

L'analyse de maliciels est définie par l'ensemble des techniques permettant d'extraire des marqueurs qui permettent de déduire le dessein d'une application et d'en déterminer sa dangerosité. Les approches d'analyse se distinguent par la nature statique ou dynamique de la méthode d'acquisition des données utilisées et de l'analyse elle-même. Cette section décrit les principales techniques d'analyse statique et dynamique proposées pour le SE Android.

2.2.1 Analyse statique

L'analyse statique est l'examen du contenu de l'ensemble d'une application afin d'inférer ses comportements sans toutefois avoir à l'observer en exécution. Ce type d'approche évite d'exécuter le maliciel sur une plateforme, réelle ou simulée, et permet de limiter les risques d'une infection tout en ne nécessitant pas de ressource matérielle spécifique pour la tâche (p. ex. un téléphone Android). Sur Android, ce type d'approche repose habituellement sur l'analyse du contenu de l'APK d'une application. Cette section détaille les principales techniques qui permettent d'acquérir les artefacts requis à l'AS ainsi que les approches proposées pour les analyser et identifier les comportements malicieux.

Acquisition

Dans le contexte de l'AS sur Android, le code source de l'application (Java et natif C/C++), les instructions Dalvik/compilées ou une représentation intermédiaire du code de l'application sont tous des candidats potentiels d'analyse. L'annexe A présente un exemple de code source Java d'une application Android ainsi que trois représentations alternatives : les résultats de la décompilation en instructions Dalvik, en langage intermédiaire (c.-à-d. Baksmali [66]) et après reconstruction en Java.

L'utilisation du code source Java d'une application permet de recourir aux techniques génériques ciblant le Java en général et de les généraliser pour la plateforme Android. Un exemple de cette approche est le cadriciel *Soot*[73]. Ce dernier permet l'optimisation, l'instrumentation, l'analyse et la visualisation d'applications Java. Ses fonctionnalités opérant sur le code source Java ont été adaptées afin d'être compatibles avec les programmes Android. Le code source d'une application peut offrir des indications sur l'intention du développeur au travers des commentaires et de la structure du code elle-même. Cette approche est toutefois limitée dans son utilisation si le code source n'est pas disponible ou s'il a été obfusqué par un outil.

Le cas échéant, il est possible de reconstituer le comportement d'une application en analysant son code compilé en instructions Dalvik. Dans le cadre d'une application Android, le fichier *classes.dex* contient les instructions Dalvik d'une application et est récupérable depuis l'APK de l'application. Cette technique peut également être appliquées pour des bibliothèques logicielles externes à l'APK si celles-ci sont en Java. Bien que l'analyse du code peut se faire

directement sur les instructions Dalvik, celui-ci peut être aride à interpréter. Pour cette raison, ces instructions peuvent être converties dans un langage intermédiaire comme le Smali[66]. Ce langage permet d'obtenir une représentation intermédiaire du code plus convivial à l'analyse et de l'utiliser pour apporter des modifications au code d'origine de l'application. Pour cette raison, il s'agit d'un outil d'intérêt pour effectuer l'acquisition de données statiques pertinentes à l'analyse de maliciel sur Android.

Il est possible de décompiler une application, c'est-à-dire de reconstruire son contenu sous la forme de code Java à partir du code compilé en utilisant des outils spécialisés comme *Jadx* [107] ou *ded decompiler* [35]. Le résultat obtenu est une reconstruction du code source tel qu'interprété par l'outil, ce qui peut différer significativement du code source d'origine. Dû à cet écart, il est possible que la reconstruction ne soit pas équivalente au code source de l'application. De plus, certaines techniques de décompilation peuvent ne pas parvenir à inférer un code équivalent en présence d'obfuscation. Schulz [104] a démontré qu'il est possible d'injecter des instructions Dalvik dans le fichier *classes.dex* de l'APK (voir 1.2) afin de masquer à la décompilation des segments d'instructions d'un fichier compilé.

Les *assets* (ou ressources) contenues par l'application peuvent aussi être extraites à des fins d'analyse. Maiorca *et al.* [81] ont démontré que cette approche est communément appliquée dans les logiciels antivirus. L'étude démontre que ces outils utilisent les *assets* pour reconnaître les fichiers étant typiquement contenus par des maliciels connus (p. ex. des images, des fichiers textes, etc.).

Le manifeste de l'application est également une cible d'intérêt pour l'AS. Plusieurs travaux (p. ex. [26, 116]) ont utilisé les permissions énumérées dans le manifeste afin de déterminer si une application possède un patron de permissions similaire à celui d'une ou plusieurs applications malicieuses. D'autres travaux (p. ex. [57, 128]) ont utilisé la déclaration des composantes dans le manifeste afin de déterminer si une application expose publiquement des composantes susceptibles de fuites potentielles de données sensibles.

Techniques d'analyse

Plusieurs techniques d'AS sur Android ont été proposées. Notamment, l'analyse d'accessibilité (et plus particulièrement, l'analyse par marqueurs ou *Taint Analysis*), du flux de données, du flux de contrôle et l'analyse des permissions ont contribué à l'amélioration de la détection de maliciels statiquement sur Android. Ces techniques sont résumées ci-après.

Analyse d'accessibilité Les techniques de cette famille tentent d'identifier les liens permettant à une variable de prendre la valeur d'une autre dans l'ensemble de l'application. Le contexte de l'analyse peut être une section du programme (p .ex. une méthode ou une classe) ou son entièreté. Selon Nikolić et Spoto [89], une variable v est dite accessible par w s'il y

Techniques	Descriptions
Analyse des effets de bords (*Side-effect Analysis*)	Détermine si la valeur d'un énoncé dépend de la valeur d'un autre énoncé présent dans l'application.
Analyse de l'initialisation des variables (*Field Initialization Analysis*)	Recherche des variables n'ayant pas été initialisées avant d'être utilisées.
Analyse de cyclicité (*Cyclicity Analysis*)	Détermine si des champs peuvent s'accéder mutuellement
Analyse de la longueur du chemin (*Path-length Analysis*)	Établit le nombre maximal de déférences de pointeur pouvant être fait à partir d'une variable
Analyse par découpage d'application (*Program Slicing*)	Isole uniquement les instructions d'un programme permettant de déterminer la valeur d'une variable d'intérêt
Analyse par marqueurs (*Taint Analysis*)	Établit les points de fuite potentiels de données sensibles d'un système en marquant toutes les variables étant directement ou indirectement en contact avec des éléments contenant ces données.
Analyse de pointeurs (*Notamment, Pointer Analysis*)	Résout les valeurs possibles pointées par les pointeurs. Sur Android, l'intérêt est davantage porté sur la résolution de la réflexion. En effet, la mémoire étant gérée automatiquement par Java, il n'y a pas de notion de pointeurs.
Analyse de partage (*Sharing Analysis*)	Identifie les ressources référencées par deux objets différents.

Tableau 2.4 – Techniques de l'analyse d'accessibilité.

a une ou des opérations présentes dans l'application qui permettent à w d'en accéder son contenu. Le tableau 2.4 présente ces techniques telles que décrites par Schmeelk *et al.* [103].

L'analyse par marqueurs (Taint Analysis, AM) a été utilisée dans le contexte d'Android par différents auteurs (p. ex.[12, 37, 76, 128]) pour effectuer la détection de comportements malicieux. Ce modèle d'analyse a pour objectif de déterminer si une variable atteignant un point de sortie d'une application, appelé drain, possède une valeur ayant été directement ou indirectement déterminée par une *source* de données sensibles, comme un accès à la base de données des contacts ou des messages textes. L'AM permet de déterminer si une application tente d'exfiltrer une donnée sensible. Le code présenté au listage 2.2 illustre le concept à la base de l'AM par un exemple d'exfiltration de données sensibles.

```
1 String sensitiveData = aSourceFunction();
2 byte[] data = sensitiveData.getBytes();
3 /*...*/
4 aSinkFunction(data);
```

Listing 2.2 – Marquage de variable et détection d'exfiltration.

— *Ligne 1 :* Appel à une fonction retournant une donnée jugée sensible. La variable *sensitiveData* est marquée pour être suivie.

— *Ligne 2 :* Comme l'attribution de *data* dépend d'une fonction de *sensitiveData*, *data* est également marquée.

— *Ligne 4 :* La variable *data* est passée à une fonction drain qui permet l'exfiltration. Comme *data* est marquée, une fuite potentielle est identifiée.

Sur Android, l'analyse des pointeurs vise l'identification des variables, fonctions et classes qui sont ciblées par les appels par réflexion. Barros *et al.* [12] ont proposé d'utiliser les types des variables fournies en arguments à cette méthode de même que le type de la valeur de retour d'un appel de méthode par réflexion pour établir une signature de celle-ci. Cette signature est utilisée pour identifier les méthodes qui ont la même dans le code. Cette liste établit les fonctions candidates pouvant être visées par l'appel et peut permettre d'inférer la fonction ciblée même si elle n'est pas explicitement invoquée.

Analyse du flux de données L'analyse du flux de données (AFD) permet de tracer le parcours d'une donnée dans une application depuis son point d'entrée ou de création jusqu'à sa sortie ou sa destruction. Ce traçage peut se limiter au contexte d'une méthode (intra-procédurale), à l'ensemble des méthodes contenues dans une composante Android analysée (interprocédurale) ou à l'ensemble des méthodes comprises dans une composante (intracomposante) ou à l'ensemble des composantes constituant l'application (intercomposante).

FlowDroid[8] combine l'approche intracomposante et l'AM afin d'établir les points d'exfiltration de données sensibles à l'intérieur des composantes d'une application. La gestion des appels intercomposants est conservatrice, c'est-à-dire que tout appel intercomposant sortant de l'application est traité comme un drain et tout appel entrant comme une source pour l'AM ce qui favorise les faux positifs. Afin d'améliorer l'évaluation des flux de données intercomposants de FlowDroid, Li *et al.* [76] ont proposé IccTA, un complément à FlowDroid permettant de tenir compte du flux des données lorsque celles-ci sont partagées entre les composantes au sein d'une même application.

DroidBench [8] a été créé afin d'évaluer la performance des outils visant à détecter l'exfiltration de données. Ce banc d'essai présente différents scénarios d'exfiltration de données permettant d'évaluer l'efficacité des solutions proposées dans la littérature et d'établir un comparatif d'efficacité entre ceux-ci. DroidBench a été développé et proposé conjointement à FlowDroid. Conséquemment, certains tests reflètent la vision des auteurs sur l'AFD et les cas limites considérés lors de la création de FlowDroid. Cette considération peut expliquer sa bonne performance dans l'étude de Arzt *et al.* [8] lorsque comparée à d'autres outils visant à détecter l'exfiltration de données sensibles. Étant ouvert à la communauté, DroidBench a été diversifié depuis sa création par différents auteurs (p. ex. [76, 108]).

Analyse du flux de contrôle L'analyse de flux de contrôle (AFC) vise à représenter l'ensemble des contextes d'exécution (composantes, objets, méthodes, etc.) et les transitions entre ceux-ci. Cela a pour but d'établir si une application possède un enchaînement d'états pouvant mener à l'expression d'un comportement malicieux. Il s'agit d'une approche répandue dans les outils d'analyses statiques visant Android (p. ex. [7, 56, 62, 70, 132]). Les résultats de l'AFC sont généralement représentés sous la forme d'un graphe où les nœuds représentent les états (composantes, objets, méthodes, etc.) et où les arcs représentent les transitions possibles

entre ces états. Comme présenté dans l'étude de Yang *et al.* [132], l'AFC sur Android doit tenir compte du modèle évènementiel de la plateforme. En effet, certains appels de fonctions et points d'entrée dans l'application ne sont déclenchés que lors de la survenue d'évènements asynchrones et imprévisibles (p. ex. l'utilisateur appuie sur un bouton d'aide) plutôt que de suivre un flux d'exécution avec un point d'entrée unique (p. ex. une méthode *main*). L'approche utilisée dans l'outil FlowDroid [8] est de construire le flux de contrôle en créant un point d'entrée unique artificiel dans lequel tous les autres points d'entrée de l'application sont appelés séquentiellement pour effectuer l'analyse. Cette approche ne tient pas compte des accès concurrents qui peuvent survenir si des opérations sont effectuées dans des fils d'exécution parallèles.

Les flux de contrôles implicites, c'est-à-dire si une méthode du cadriciel d'Android est appelée par une autre méthode également contenue dans celui-ci ou encore si une méthode est appelée par un appel de retour (c.-à-d. *callbacks*), doivent être également explorés afin d'assurer la validité du flux de contrôle extrait de l'application. Si ce n'est pas le cas, le graphe de contrôle résultant ne tiendra pas compte de toutes les interactions possibles ce qui pourrait fausser les résultats de l'analyse. Afin de pallier ce problème, Cao *et al.* [18] proposent une technique qui analyse le cadriciel d'Android et ses fonctions sous-jacentes afin d'utiliser les relations qui s'y trouvent lors de la construction du flux de données d'une application. Près de 20000 flux de contrôle implicites sont identifiés dans l'API d'Android, ce qui démontre l'importance de considérer cet aspect dans l'analyse. Plusieurs études (p. ex. [8, 18, 76, 96]) utilisent l'AFC de façon complémentaire à l'AFD. L'AFC permet d'établir les états et leur transition facilitant le suivi des données pour l'AFD.

Analyse des permissions L'analyse des permissions (AP) est l'étude des permissions utilisées ou déclarées dans une application afin d'établir si un patron malicieux peut être dégager de celles-ci. Ce type d'analyse a été démontrée efficace dans le classement malicieux ou bénin des applications dans différents travaux (p. ex. [26, 77, 102]). L'un des principaux avantages de l'AP est qu'elle demande peu de ressources puisqu'elle repose sur l'analyse du manifeste d'une application où sont déclarées les permissions. Une autre variante de l'AP, utilisée par Moonsamy *et al.* [84], fait la comparaison entre les permissions demandées dans le manifeste et celles utilisées dans le code de l'application. Cela permet de quantifier la fréquence d'utilisation de ces permissions et d'établir des modèles d'analyse tenant compte de ce critère. Ce type d'AP permet également de repérer les applications qui demandent plus de permissions que ce qu'elles utilisent dans le code contenu dans l'APK. Cela peut être un indicateur de mauvaise pratique de développement ou être un signe précurseur que le programme Android peut étendre l'éventail de ses comportements au-delà de ce qui déterminé dans son code d'origine. Cette approche demande davantage de puissance computationnelle que la précédente puisqu'elle nécessite d'analyser l'ensemble du code de l'application plutôt que de s'attarder uniquement sur les permissions déclarées dans le manifeste.

Sans égard à l'approche retenue, l'AP repose sur le principe que les permissions ou leur utilisation diffèrent entre les applications bénignes et malicieuses. Or, il est possible pour une application malicieuse de calquer ses permissions et la manière de les utiliser sur une application bénigne afin d'éviter la détection. De plus, Ramírez [96] et Shabtai *et al.* [106] abordent la possibilité d'utiliser la propriété *SharedUserId* afin de permettre de combiner les permissions d'une application avec celles d'une autre. En effet, *SharedUserId* est une propriété optionnelle du manifeste d'une application qui permet à deux applications signées par un même certificat de développeur de partager le même UID et, par extension, les mêmes GID. Comme expliqué précédemment (voir 1.3.1), ces identifiants uniques servent à gérer les accès aux ressources autorisées en fonction des permissions de l'application. De ce fait, deux applications qui partagent un même UID peuvent poser des actions qui requièrent des permissions détenues. Cela peut leur permettre de combiner l'ensemble de leurs permissions pour entraîner des dommages. Dû à ces problématiques, il est suggéré que l'AP soit combinée à d'autres approches d'analyse pour valider les résultats obtenus.

Limitations

L'utilisation de l'analyse statique n'est pas en mesure de garantir que tous les comportements possibles d'une application durant son exécution soient atteignables statiquement. Sur Android, cette problématique est d'autant plus présente puisque le Java permet le chargement dynamique de code. L'utilisation de la réflexion permet de faire cela (voir annexe C pour un exemple). Elle nuit à la capacité de suivre les transitions possibles dans le code d'une application statiquement. Rasthofer *et al.* [98] ont estimé le taux d'applications utilisant la réflexion à 76% sur un échantillon de 235000 avec une provenance majoritaire du Google Store. Ce ratio justifie les efforts déployés pour améliorer la capacité de l'AS à gérer le chargement dynamique de code.

Certaines études comme Arzt *et al.* [8], Li *et al.* [76] ont proposé de résoudre les appels par réflexion en déterminant la fonction appelée lorsque l'appel se fait à l'aide de chaînes de caractères constantes. Ces études n'ont pas proposé de solutions lorsque la valeur de ces chaînes de caractères est déterminée à l'exécution. Également, l'AS ne peut pas identifier la méthode ciblée si les arguments de celle-ci sont construits à l'aide de ressources externes au contenu de l'APK. Par exemple, l'AS ne peut inclure dans l'analyse une bibliothèque logicielle chargée dynamiquement par réflexion si celle-ci est téléchargée à l'exécution. Au moment de la rédaction, aucune technique d'analyse statique capable de résoudre avec certitude la réflexion n'a été relevée.

La communication intercomposante (CIC) complique également l'AS puisqu'elle permet de communiquer entre les composantes de différentes applications sur un même appareil. Cela peut être utilisé pour déclencher des comportements malicieux ou divulguer des données sensibles si cette communication survient entre des applications malicieuses ou si une application est

exploitée pour divulguer des données sensibles à une autre. Il n'est pas possible pour l'AS de déterminer si et quand une composante exportée (c.-à-d. publiquement accessible sur la plateforme) sera sollicitée ni de garantir si elle est en train d'interagir avec une composante de confiance. Afin de répondre à ce problème, il est possible d'adopter une approche conservatrice et de stipuler que toutes communications avec une composante externe à celle analysée sont suspectes. Conséquemment, toutes données envoyées via un CIC seront vues comme une fuite d'information et toutes données reçues seront traitées comme étant sensibles[8]. Bien que cette approche augmente le nombre de faux positifs observé, elle diminue les risques qu'une exfiltration de données passe inaperçue. Une approche proposée dans la littérature (p. ex. [12, 76, 90, 132]) tient compte du contenu des composantes externes avec lesquelles la composante courante interagit pour établir s'il y a présence d'une exfiltration de données. Pour ce faire, il est nécessaire que la composante externe à l'application soit elle-même disponible à l'AS.

Android est un système dont l'expression de certains comportements repose sur la survenue d'évènements déclencheurs externes aux applications (p. ex. l'utilisateur qui appuie sur un bouton d'aide). Certains de ces comportements ne surviennent qu'en présence d'actions qui peuvent être asynchrones et impossibles à prévoir. Ce modèle évènementiel peut affecter dynamiquement le flux d'exécution de l'application. Par exemple, la réception d'un SMS par un serveur de commande et contrôle pourrait contenir une clef de chiffrement utilisée par l'application lors de son exécution pour décrypter du contenu malicieux dynamiquement et l'exécuter. Dans ce scénario, l'AS ne serait pas en mesure d'analyser le contenu chiffré si la clef de chiffrement est suffisamment forte puisqu'elle est externe au code du maliciel. Ce type de scénario est une contrainte réelle de l'AS qui nécessite une autre approche pour y remédier.

2.2.2 Analyse dynamique

À l'instar de l'AS, l'AD dans un contexte d'analyse de maliciels vise à identifier les comportements malicieux d'une application. Contrairement à l'AS qui vise à inférer la présence d'un comportement malicieux par l'analyse des éléments contenus dans les fichiers de la cible, l'AD vise à détailler les impacts d'une application par l'observation de son comportement et de ses effets sur le système. Les artefacts pertinents à l'AD de même que les techniques utilisées pour les analyser sur Android sont détaillés dans cette section.

Acquisition

La tableau 2.5 présente différents éléments associés à l'exécution d'une application sur la plateforme Android et qui peuvent être significatifs dans un contexte d'analyse de maliciels.

Différentes techniques permettent de faire l'acquisition de ces données. Pour Android, les principales sont résumées au tableau 2.6 et détaillées ci-après.

Marqueurs	Description
Traces d'exécution	L'ensemble des méthodes appelées par une application et la séquence d'exécution de celles-ci. La granularité observée des appels de méthodes varie grandement en fonction de la technique d'acquisition utilisée. Ces appels peuvent être l'ensemble des appels système fait au noyau Linux pendant la durée de l'exécution de l'application d'intérêt, l'ensemble de ceux faits à l'API d'Android (pour une application ou pour toutes celles présentes au moment de l'analyse) ou l'ensemble de ceux faits aux méthodes internes au programme utilisées pendant son exécution.
Variables et données	Suivi des variables, leur contenu et les échanges de données pendant l'exécution.
Liste des processus	L'ensemble des processus créés, en exécution et/ou terminés pendant la durée de vie de l'application analysée.
Entrées et sorties	L'ensemble des interactions permettant l'échange des données avec un agent externe à la plateforme. Les données échangées par des périphériques (p. ex. Webcam, récepteur radio, Near-Field Communication (NFC)), etc.), par communications réseaux (SMS, Internet, etc.) et les fichiers accédés en lecture ou en écriture en sont des exemples.
Permissions demandées à l'exécution	Depuis Android 6.0, il est possible pour une application de demander les permissions nécessaires à son exécution à l'accès. De plus, celle-ci peut avoir des fonctionnalités utilisant des permissions qu'elle ne détient pas. Par exemple, elle peut utiliser une bibliothèque logicielle d'un tiers possédant des capacités plus étendues que celles requises et qui dépendent des permissions disponibles sur un appareil. Dans ce cas, une exception est lancée et doit être gérée dans l'application, sans quoi, le SE force sa fermeture et lance une erreur.
Mémoire vive	Le contenu de la mémoire vive de la plateforme ou d'une application. Le chapitre 3 traite explicitement de ce type de données.

TABLEAU 2.5 – Éléments d'intérêt pour l'analyse dynamique.

Techniques	TE	VD	LP	IO	P	MV
Fonctionnalités diagnostiques du SE	X		X	X	X	X*
Dévermineur d'applications	X	X		X	X	X*
Surveillance du trafic réseau				X*	X*	
Virtualisation/Émulation	X	X	X	X	X	X
Loadable Kernel Module (LKM)[1]	X	X	X	X	X	X
Interception des appels à l'API Android	X	X		X	X	
Interception des appels système[1]	X	X	X	X	X	X
Modification et recompilation du SE[1]	X	X	X	X	X	X
Dévermineur du SE[1]	X	X	X	X	X	X

TE : Trace d'exécution
VD : Variable et données
LP : Liste des processus
IO : Entrées et sorties
P : Permissions
MV : Mémoire vive.
* : Acquisition partielle.
[1] : L'accès *root* est habituellement requis.

TABLEAU 2.6 – Technique d'acquisitions.

Fonctionnalités diagnostiques du SE Cette technique consiste à utiliser l'ensemble des outils disponibles à même le SE pour extraire le contenu dynamique nécessaire à l'analyse. Il peut s'agir d'utilitaires et de méthodes offerts par le noyau Linux. La liste complète d'outils est accessible par la commande *ls /system/bin* sur un appareil Android. Les extraits de consoles présentés en 2.3, 2.4, 2.5 et 2.6 sont les résultats produits par les commandes *ps*, *lsof*, *netstat* et *strace* respectivement.

L'utilitaire *ps* permet d'obtenir la liste hiérarchique de l'ensemble des processus et fils d'exécution si l'option *-t* est incluse. *lsof* permet d'obtenir l'ensemble des fichiers ouverts (virtuels ou physiques) par les processus en exécution sur le système. L'identifiant du processus qui en détient le contrôle est également listé. *netstat* liste les *sockets* de type UNIX, TCP ou UDP utilisés ou ayant été utilisés pour communiquer, leur état et le processus propriétaire de ceux-ci. *strace* permet d'obtenir l'ensemble des appels système faits par une application. Pour obtenir l'information d'une application, *strace* doit être lancé sous un *UID* détenant le processus visé. Cela peut se faire par l'utilisation du même *UID* que celui de la cible ou en utilisant le *UID* de l'utilisateur *root*. Le listage 2.6 représente un extrait de la sortie de *strace* lorsque celui-ci est appliqué à une application Android qui manipule des fichiers au moment de la capture.

L'ensemble des outils sont décrits dans leur documentation respective. Le lecteur est invité à s'y référer pour plus amples détails.

```
USER      PID   PPID  VSIZE  RSS     WCHAN    PC         NAME
root      1     0     2616   628     ffffffff 00000000 S /init
root      2     0     0      0       ffffffff 00000000 S kthreadd
[...]
install   66    1     3196   656     ffffffff 00000000 S /system/bin/installd
keystore  67    1     6276   1860    ffffffff 00000000 S /system/bin/keystore
root      68    1     441076 46632   ffffffff 00000000 S zygote
[...]
media_rw  454   1     5164   708     ffffffff 00000000 S /system/bin/sdcard
media_rw  456   454   5164   708     ffffffff 00000000 S sdcard
media_rw  457   454   5164   708     ffffffff 00000000 S sdcard
[...]
u0_a25    1033  68    463312 34208   ffffffff 00000000 S com.android.email
u0_a25    1037  1033  463312 34208   ffffffff 00000000 S Signal Catcher
u0_a25    1040  1033  463312 34208   ffffffff 00000000 S JDWP
u0_a25    1041  1033  463312 34208   ffffffff 00000000 S ReferenceQueueD
u0_a25    1042  1033  463312 34208   ffffffff 00000000 S FinalizerDaemon
u0_a25    1043  1033  463312 34208   ffffffff 00000000 S FinalizerWatchd
u0_a25    1044  1033  463312 34208   ffffffff 00000000 S HeapTrimmerDaem
[...]
shell     1111  293   3200   752     c002a14c b6ec30d4 S sh
shell     1112  1111  4464   968     00000000 b6f7dd18 R ps
```

Listing 2.3 – Extrait de "ps -t" sur un appareil Android.

```
COMMAND     PID     USER     FD   TYPE     DEVICE  SIZE/OFF  NODE    NAME
[...]
com.andro   1033    u0_a25   exe  ???      ???     ???       ???     /system/bin/app_process32
com.andro   1033    u0_a25   20   ???      ???     ???       ???     /system/app/Email/Email.apk
com.andro   1033    u0_a25   21   ???      ???     ???       ???     /data/data/com.android.email/
    ↪ databases/EmailProvider.db
com.andro   1033    u0_a25   25   ???      ???     ???       ???     /data/data/com.android.email/
    ↪ databases/EmailProviderBody.db
com.andro   1033    u0_a25   26   ???      ???     ???       ???     /data/data/com.android.email/
    ↪ databases/EmailProviderBody.db
com.andro   1033    u0_a25   mem  ???      00:04   0         1992    /dev/ashmem/dalvik-main
[...]
```

Listing 2.4 – Extrait de "lsof" sur un appareil Android avec un accès *root*.

```
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.1:5037          0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:5555            0.0.0.0:*               LISTEN
tcp        0      0 10.0.2.15:5555          10.0.2.2:58080          ESTABLISHED
tcp6       0      0 ::ffff:10.0.2.15:37198  ::ffff:172.217.1.110:80 ESTABLISHED
```

Listing 2.5 – Extrait de "netstat -l -t" sur un appareil Android.

```
[...]
clock_gettime(CLOCK_MONOTONIC, {202, 282920727}) = 0
clock_gettime(CLOCK_MONOTONIC, {202, 283968091}) = 0
openat(AT_FDCWD, "/data/data/com.example.dex/app_dex/payload.jar", O_WRONLY|O_CREAT|
    ↪ O_TRUNC, 0600) = 33
fstat64(33, {st_mode=S_IFREG|0600, st_size=0, ...}) = 0
mprotect(0x73be6000, 4096, PROT_READ|PROT_WRITE) = 0
clock_gettime(CLOCK_MONOTONIC, {202, 291383567}) = 0
clock_gettime(CLOCK_MONOTONIC, {202, 294546763}) = 0
clock_gettime(CLOCK_MONOTONIC, {202, 295104601}) = 0
clock_gettime(CLOCK_MONOTONIC, {202, 295857676}) = 0
read(31, "PK\3\4\24\0\10\10\10\0M\202\214H\0\0\0\0\0\0\0\0\0\0\0\0\24\0\4\0ME"...,
    ↪ 8192) = 932
write(33, "\372\341\251\256\276\252\242\242\242\252\347
    ↪ (&\342\252\252\252\252\252\252\252\252
    ↪ \252\252\252\252\276\252\256\252\347\357"..., 932) = 932
read(31, "", 8192)                      = 0
getsockopt(33, SOL_SOCKET, SO_ERROR, 0xbec5e0c0, 0xbec5e0c4) = -1 ENOTSOCK (Socket
    ↪ operation on non-socket)
getsockopt(33, SOL_SOCKET, SO_LINGER, 0xbec5e038, 0xbec5e034) = -1 ENOTSOCK (Socket
    ↪ operation on non-socket)
close(33)                               = 0
[...]
```

Listing 2.6 – Extrait de "strace" sur un appareil Android.

Ces outils ne sont capables d'analyser que les applications exécutées sous le même UID que celui utilisé pour les lancer à moins d'utiliser les privilèges de l'utilisateur *root*. Si tel est le cas, ils peuvent être utilisés sur l'ensemble des processus en exécution.

Dévermineur d'applications Le dévermineur d'applications Android est habituellement employé comme un outil permettant de contrôler l'exécution et les variables internes d'une application pendant son développement. Il permet notamment d'accéder aux fils d'exécution actifs, aux valeurs des variables internes et aux connexions établies dans le contexte de l'application. Cette capacité d'extraction peut également être utilisée à des fins d'analyse. Sur Android, un dévermineur est inclus avec les outils de développement d'application de la plateforme. Ce dernier permet de contextualiser les informations et le contrôle de l'exécution avec le code source de l'application si ce dernier est disponible. Pour l'utiliser, une application doit posséder la propriété *android :debuggable=true* dans la section *<application>* de son manifeste afin que le dévermineur soit autorisé à s'y connecter. Une technique abordée dans l'ouvrage de Dominic *et al.* [31] permet de contourner cette limitation en modifiant le manifeste contenu dans un APK. Un nouvel APK est assemblé avec ce nouveau manifeste en conservant les autres fichiers intacts. Cette approche peut être détectée telle que mentionnée à la section 2.1.2

portant sur les techniques d'évasion des maliciels.

Également, l'utilisation d'un dévermineur comme *gdb* [44] ou *radare2* [93] avec les privilèges de l'utilisateur *root* constitue une solution de remplacement au dévermineur d'Android. Ces dévermineurs diffèrent de celui d'Android puisqu'ils opèrent sur les instructions-machine, la mémoire et le flot d'exécution d'un processus au niveau du noyau directement. De ce fait, les données extraites par ces outils ne sont pas contextualisées avec le code source de l'application ou en fonction des appels de l'API d'Android comme dans le cas précédent. Les données recueillies (séquences d'appels, valeurs de variables, pointeur en mémoire, etc.) comportent des éléments non pertinents à l'analyse puisqu'ils ne sont pas spécifiques à l'application et englobent les routines sous-jacentes du SE. Il peut s'avérer utile d'effectuer un prétraitement afin de débruiter ces données avant de les analyser. Par exemple, il pourrait être utile d'identifier les séquences d'instructions-machine correspondant à certains appels au noyau. De cette façon, les séquences pourraient être remplacées par le nom de la méthode correspondante dans la trace d'exécution et, ainsi, faciliter la lecture pour un analyste.

Surveillance du trafic réseau Plusieurs travaux ont utilisé l'observation des communications réseau sortantes et entrantes de l'appareil dans un contexte d'analyse de maliciels sur Android (p. ex. [36, 69, 133]). L'acquisition des données transigées sur le réseau peut se faire par l'écoute de celui-ci à partir d'un agent externe à l'appareil Android. Ce dernier capture alors l'ensemble du trafic avec un outil de capture et d'analyse de traces réseau comme *Wireshark*. Android offre des fonctionnalités permettant d'intercepter les paquets réseau à des fins de prétraitement avant la transmission vers le destinataire légitime. Ce mécanisme offre notamment la possibilité d'encapsuler les données transmises afin de les transmettre par un protocole sécurisé vers un destinataire faisant office de relais comme c'est le cas pour un Virtual Private Network (VPN), c'est-à-dire, un réseau privé virtuel avec un point d'accès distant. Or, Zaman *et al.* [133] ont utilisé ce mécanisme visant l'implémentation d'un VPN pour copier localement chaque paquet sortant avant de les relayer à leur destination légitime sans traitement additionnel. Cette technique a l'avantage de s'exécuter à même l'appareil et ne nécessite aucune modification du SE.

Virtualisation/Émulation La virtualisation d'un environnement consiste à créer un système invité constitué de composantes (p. ex. un microprocesseur ou la mémoire vive) virtualisées à partir des ressources physiques disponibles sur un système hôte et qui peuvent être partagées entre plusieurs machines virtuelles (Virtual Machine, MV). Ce partage de ressources est effectué de sorte à ce que le contenu qui s'y trouve, lui, est isolé des autres MV et de l'hôte. L'émulation est similaire à la virtualisation, mais, là où la virtualisation utilise des composantes matérielles dédiées pour optimiser l'exécution, l'émulation utilise principalement des composantes logicielles pour effectuer l'isolement des MV. Elle effectue également une traduction à la volée des instructions-machine du système émulé par ces composantes logicielles avant d'être acheminée

vers le processeur hôte. Dans le cas de la virtualisation, cette traduction se fait à l'aide de composantes matérielles dédiées. Dans le cas d'Android, l'émulation permet notamment d'effectuer la traduction d'instructions pour processeurs ARM, retrouvés couramment dans les tablettes et téléphones Android, vers celles pour processeurs Intel x86/x64, communément utilisés dans les ordinateurs personnels. L'émulateur Android fait partie de la suite d'outils de développement Android utilisés par les développeurs. Il permet la virtualisation d'appareils Android basés sur processeur Intel et l'émulation de ceux reposant sur processeurs ARM. Cela offre la possibilité de tester les applications en développement en fonction d'architectures de processeur différentes.

Dans le cadre de l'analyse de maliciel spécifiquement, l'utilisation de MV pour l'acquisition peut tirer avantage des *instantanés* (*snapshots*). Pour ce faire, l'état instantané d'une MV est stocké dans un fichier afin de permettre la reprise intégrale de l'état de la machine à ce moment précis. L'état de la mémoire vive, du processeur virtuel, du contenu de l'espace de stockage, et des périphériques virtualisés par le système invité sont préservés et sont restaurés [33]. Ce procédé élimine tous changements ayant pu survenir après la prise de l'instantané. En utilisant cette façon de faire, il est possible de créer un système d'analyse expérimental permettant de revenir à un état initial entre chaque analyse. Les étapes décrites à la figure 2.1 représentent cette approche et sont :

1. la prise d'un instantané de l'état initial désiré ;
2. le déploiement du maliciel dans l'environnement d'analyse ;
3. la capture des données dynamiques par l'utilisation d'outils d'analyses présents dans la MV ou par l'instrumentation de l'engin de virtualisation ou d'émulation pour effectuer la capture des comportements suspects (p. ex. CopperDroid [117]) ;
4. la prise d'un instantané avec des données sur l'état interne de la MV si désirée pour l'analyse ;
5. la restauration de l'état initial décrit en *1* et
6. recommencer le cycle à partir de *2*.

Comme mentionné à la section 2.1.2, certains maliciels sont capables de détecter ces environnements et d'inhiber leurs comportements en conséquence. Dans de tels cas, un écart peut être observé entre ce que le maliciel fait lorsqu'il est dans un environnement virtuel ou émulé et ce qu'il fait sur un système physique comme un téléphone intelligent.

Loadable Kernel Module (LKM) Un Linux loadable kernel module (LKM) est un module du noyau Linux dont le chargement se fait dynamiquement et qui permet à un composant d'interagir avec le SE et les périphériques d'un système en ayant les mêmes libertés que le noyau lui-même. Il s'agit d'une méthode pour étendre les capacités du noyau sans nécessité de recompiler celui-ci. De ce fait, les LKM ont un accès complet à l'ensemble des éléments présents dans le SE. Leur utilité dans un contexte d'analyse de maliciels a été démontrée sur

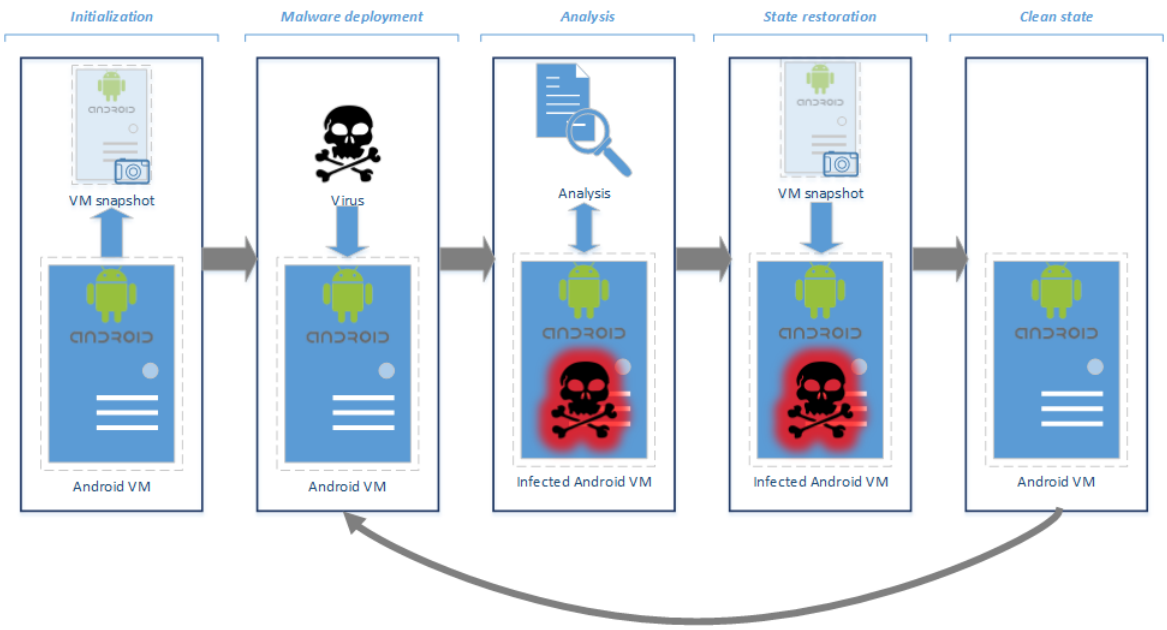

FIGURE 2.1 – Analyse de maliciels à l'aide d'une MV.

Android par des études de Min et Cao [83], Olipane [91] qui les ont utilisés pour effectuer l'instrumentation du SE. Ces derniers ont utilisé les LKM pour injecter des méthodes de journalisation des comportements du SE lors de l'exécution d'une application malicieuse. De plus, puisque les LKM possèdent un accès intégral et sans restriction à l'ensemble de la mémoire du SE où ils sont déployés, il est possible de les utiliser afin de faire la capture intégrale de la mémoire vive d'un système. Cette approche est détaillée au chapitre 3 qui traite de l'analyse de la mémoire vive.

Tel qu'il est expliqué par Brodbeck [16], le chargement dynamique des LKM n'est plus supporté par défaut depuis la version Android 4.3 pour des motifs de sécurité. Toujours selon ce même auteur, il a été démontré possible d'utiliser ces modules afin de manifester des comportements malicieux et d'éviter la détection. Ils sont réputés pour être un vecteur d'attaque sur les systèmes Linux. Toutefois, ils peuvent être réactivés lors de la compilation du noyau si cela est nécessaire.

Détournement des appels à l'API Android Par un mécanisme appelé *API Hooking*[1], il est possible de rediriger l'ensemble des appels aux méthodes de l'API Android vers un intermédiaire. Le but de ce relais est d'ajouter une procédure de journalisation de ces appels avant de les réacheminer vers les méthodes de l'API auxquelles elles sont destinées. La valeur retournée par la méthode est également interceptée avant d'être acheminée à l'appelant. Cette technique permet d'obtenir la trace des appels aux méthodes de l'API Android.

1. Traduction libre : détournement des appels à l'API.

Cette technique peut s'implémenter en substituant les appels à l'API d'Android dans le code décompilé d'une application par des appels à un autre API de journalisation appartenant à un tiers capable de les intercepter [10]. Une autre approche consiste à utiliser le concept de processus isolé (*Isolated Process*) introduit dès Android 4.1 et qui permet de lancer des processus enfants dépourvus de permission depuis une application mère. *Boxify* présenté par Backes *et al.* [9] utilise cette capacité afin de lancer l'ensemble des applications suspectes dans un processus isolé et enfant à une application maîtresse. Cette dernière intercepte les appels à l'API de la plateforme faits par l'application. D'un point de vue pratique, le traitement additionnel requis pour effectuer la journalisation de l'appel s'ajoute à celui déjà normalement requis par la fonction visée de l'API ce qui peut avoir un effet négatif sur la fluidité d'exécution de l'application.

Détournement des appels système Le détournement des appels système est similaire à celui des appels de l'API d'Android. Il vise plutôt les appels aux méthodes du noyau Linux. Contrairement à l'approche précédente, le détournement des appels système offre une visibilité sur tous les processus et toutes les opérations du SE plutôt que d'être limité aux opérations se produisant entre une application et l'API d'Android. Cette technique est soumise aux mêmes contraintes de performances que la précédente, c'est-à-dire que la charge de traitement additionnelle associée à la journalisation peut provoquer un ralentissement du système. De plus, pour effectuer ce type de redirection sans avoir recours à l'utilisation d'un exploit ou à la recompilation du noyau Linux, il est nécessaire d'avoir les privilèges de l'utilisateur *root* sur l'appareil. L'utilitaire Cydia Substrate de Freeman [40] offre la capacité de détourner les appels système sur Android. Il peut être utilisé afin d'injecter des appels de journalisation aux méthodes du noyau. Cet outil utilise le placement prévisible des plages d'adresses en mémoire vive pour modifier la valeur des adresses pointant vers les méthodes système afin que ces pointeurs réfèrent vers des méthodes équivalentes, mais ayant des capacités de journalisation additionnelles. Toutefois, l'utilisation de Cydia Substrate est limitée aux versions d'Android exemptes de distribution aléatoire de l'espace d'adressage (Address Space Layout Randomization, ASLR). En effet, l'ASLR empêche la mémoire vive d'avoir une topologie constante d'une exécution à l'autre. Cette prévisibilité de la disposition de la mémoire vive est requise par l'outil afin de localiser les tables contenant les pointeurs vers les méthodes système.

Une autre approche utilisée par Jeong *et al.* [65] détourne également les appels aux méthodes système en utilisant le chargement d'un LKM. Cette approche demande une recompilation du noyau pour rétablir le support des LKM, tel qu'expliqué précédemment. Également, tant pour l'approche de Jeong *et al.* [65] que Freeman [40], les privilèges de l'utilisateur *root* sont requis pour les utiliser.

Modifications et recompilation du SE Une autre méthode proposée consiste à modifier Android et ses composantes directement dans son code source afin d'obtenir une version

capable d'extraire les données dynamiques désirées à l'exécution. Le code source d'Android est publiquement disponible (voir [54]) ce qui permet à un analyste d'implémenter des capacités de journalisation de données dynamiques à même le SE. Une fois la modification apportée, le code est compilé puis déployé sur un appareil ou une MV pour lequel il a été prévu. Ce procédé nécessite les privilèges de l'utilisateur *root* pour le déploiement dans le cas d'un appareil physique. Toute version modifiée d'Android est fortement couplée à un modèle d'appareil et à la version du SE pour lesquels elle est compilée. Ces modifications peuvent demander des adaptations additionnelles pour chaque appareil ou version du SE devant être supporté. Cette situation complique la généralisation des modifications entre les appareils de différents modèles. De plus, chaque mise à jour d'Android est susceptible de modifier les composantes du système et peut entraîner des défectuosités dans le fonctionnement des composantes modifiées. Pour ces raisons, l'entretien d'une solution reposant sur la modification du code d'Android demande un effort important pour demeurer utilisable. Cette approche est notamment utilisée par TaintDroid [34].

Dévermineur du SE Il est possible de compiler un noyau Linux pour Android en activant les options permettant le déverminage dynamique de celui-ci, approche utilisée notamment par Zatuchna [136]. Tout comme le dévermineur pour application, il est possible de contrôler l'ensemble de l'application ciblée par le dévermineur du noyau. La cible de ce dévermineur est le noyau du SE ce qui lui permet d'accéder à l'ensemble du SE et de ce qui s'y trouve. Les privilèges de l'utilisateur *root* sont nécessaires pour déployer un noyau Linux déverminable ainsi que pour y connecter le dévermineur. De plus, le contrôle d'exécution imposé par un dévermineur du SE est susceptible de provoquer d'importants ralentissements sur la plateforme. Cela peut avoir un impact négatif sur l'exécution d'une application soumise à une contrainte temporelle (p. ex. la perte d'une liaison avec un serveur distant due à un trop long délai d'attente induit par le dévermineur).

Techniques d'analyse

Plusieurs approches permettent d'analyser les données dynamiques acquises afin de permettre à un analyste de discriminer les comportements malicieux des autres. Les principales techniques recensées dans la littérature d'Android sont décrites dans la présente section.

Analyse des appels aux méthodes L'analyse des appels aux méthodes infère le comportement d'une application par l'observation de la séquence des appels de méthodes, par le contenu passé en argument et par leur valeur de retour. Deux approches sont prédominantes sur Android pour ce type d'analyse : l'analyse des appels aux méthodes de l'API Android et celle des appels système du SE. Dans le premier cas, l'analyse est plus intuitive puisque les méthodes de l'API Android représentent des actions concrètes sur la plateforme. Chaque méthode est une abstraction de séquences d'actions devant être effectuées par le SE afin de

produire le comportement désiré. Par exemple, il est explicite en raison de son nom qu'un appel à la méthode *SmsManager.sendTextMessage()* vise à faire l'envoi d'un message texte. De plus, la documentation officielle de l'API d'Android [53] permet de connaître l'ensemble des méthodes offertes via l'API de la plateforme, leur utilisation et leurs effets ce qui facilitent l'interprétation des séquences observées. Bien que l'analyse des appels aux méthodes de l'API Android permette d'inférer le comportement global d'une application qui les utilise, il a été démontré par différents travaux qu'il est possible de contourner les appels aux méthodes de l'API tout en produisant des comportements équivalents. Par exemple, Hao *et al.* [60] ont démontré qu'il est possible d'utiliser les appels à des méthodes natives du SE directement afin de contourner l'API d'Android et ainsi, produire des comportements équivalents à des méthodes dangereuses sans laisser de trace. Également, une application Android peut demander l'exécution d'un fichier binaire externe à l'application par l'appel *Runtime.exec(<Application et paramètres>)* [74]. Un fichier binaire exécuté de cette façon peut interagir directement avec le SE pour accomplir des actions qui sont dans la limite des permissions détenues [2].

Afin d'avoir une vision complète de l'activité du système et non pas uniquement des appels à l'API Android, il est possible d'effectuer l'analyse de l'ensemble des appels système passés au noyau Linux pendant la durée de vie d'une application. Les appels système sont plus génériques et granulaires que ceux de l'API, ce qui signifie qu'une même fonction peut être impliquée dans l'expression de différents comportements. Par exemple, les méthodes système *read* et *write* visibles dans la sortie de *strace* au listage 2.6 sont utilisées afin de lire ou écrire des octets dans un fichier. Or, tous les comportements qui nécessitent de manipuler la lecture ou l'écriture de fichier utiliseront ces mêmes appels système, que une base de données SQLite ou des fichiers temporaires. Pour cette raison, le but d'un appel peut ne pas être apparent s'il est considéré en dehors de la séquence dans laquelle il survient. Les travaux de Burguera *et al.* [17], Tam *et al.* [117] ont utilisé l'analyse des appels système afin d'identifier des séquences représentant des comportements jugés d'intérêt comme ceux utilisés dans l'envoi d'un message texte. Également, Ham *et al.* [59] ont exploité ce type d'analyse afin d'extraire la fréquence des différents types d'appels système pour des applications réputées bénignes en fonction de la catégorie de celles-ci (p. ex. jeux, applications systèmes). À partir de ces résultats, ils ont pu établir qu'un maliciel ne présente pas le même patron (type d'appels et leur fréquence) qu'une application bénigne appartenant à la même catégorie. Ces travaux reposent sur la capacité à identifier un patron d'appels système. De ce fait, il est possible pour un concepteur de maliciels ingénieux de dissimuler l'activité malicieuse de son programme en effectuant des appels système additionnels avec pour seul but de calquer des traces d'exécution jugées bénignes.

Analyse du flux de données L'analyse du flux de données relève du même principe que son homologue statique présenté à la section 2.2.1. Contrairement à l'AS, l'AD est en mesure

2. L'exécution d'un fichier binaire est employée dans l'application utilisée à la section 4.

de suivre le flux de données même lorsque celui-ci résulte d'appels par réflexion, avec ou sans obfuscation et de CIC. Pour ce faire, l'analyse par marqueurs (Taint Analysis, AM) peut être utilisée afin de marquer les variables qui comportent des données sensibles durant l'exécution. Si une variable marquée est passée à une méthode vue comme un drain potentiel, il s'agit d'une fuite potentielle d'information sensible.

Pour implémenter cette capacité, deux approches ont été explorées. La première modifie l'émulateur Android afin de le doter d'une capacité externe à surveiller le fonctionnement du SE à partir d'un observateur externe à celui-ci. DroidScope [130] effectue un marquage des données sensibles ce qui lui permet de suivre leur passage dans l'application, et ce, au niveau du code natif de la MV Dalvik et de l'API Android. L'architecture de DroidScope est présentée à la figure 2.2 et est tirée de Yan et Yin [130]. De cette façon, il est possible de suivre les données sensibles dans la plateforme en incluant le noyau Linux. En revanche, une implémentation reposant sur la modification d'un émulateur est incompatible avec le déploiement sur un appareil physique. Cela est d'autant plus problématique que l'exécution dans un environnement émulé expose l'engin d'analyse à la détection d'environnement d'analyse. Une approche similaire a aussi été utilisée pour CopperDroid [117] où l'instrumentation est faite à même l'émulateur dans lequel s'exécute une version d'Android n'ayant pas été modifiée.

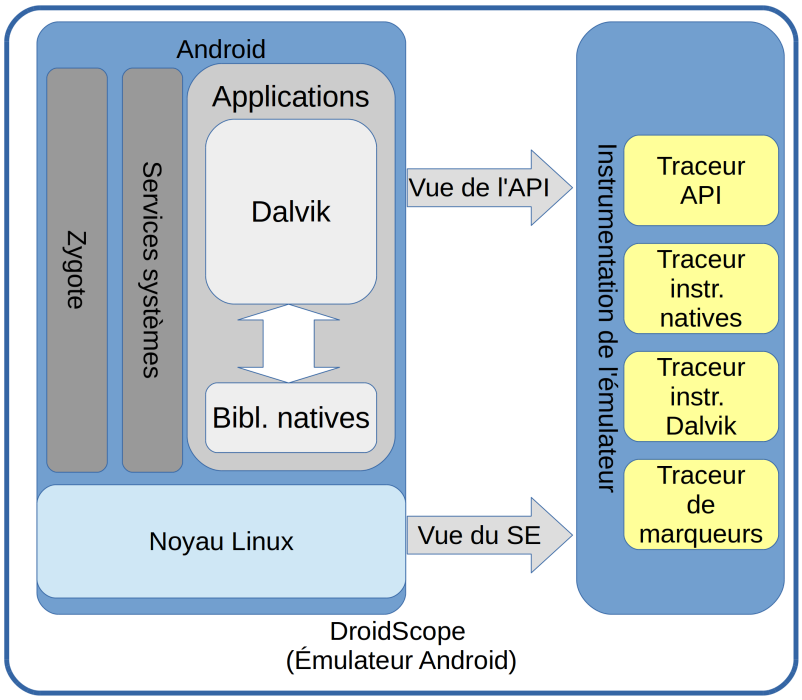

FIGURE 2.2 – Architecture de DroidScope.

La seconde approche, explorée par l'outil TaintDroid [34], repose sur la modification du code source du SE Android et, plus spécifiquement, la MVD. Cette modification vise à insérer le

marquage des données sensibles directement dans le traitement des variables d'une application Android, et ce, de façon transparente à son exécution. Il est possible de déployer cet outil sur un appareil physique puisqu'il repose uniquement sur une modification du SE. Lorsque c'est le cas, un maliciel obtient des données valides s'il sonde son environnement par l'utilisation des capteurs. Le traçage ne s'applique que pour la portion Java de l'application traitée par la MVD et exclut les appels aux méthodes natives.

Également, avec le remplacement de la MVD par ART, cette solution n'est plus adaptée pour les versions 5.0 et plus d'Android. Cette lacune a mené à des initiatives comme le projet ARTDroid de Costamagna et Zheng [25] afin d'offrir un cadriciel capable d'injecter des routines de journalisation des méthodes appelées dans une application analysée, tant dans la portion Java que native de son code. Bien que le projet n'offre pas de technique d'analyse en soi au moment de la rédaction, il propose un environnement permettant d'injecter de telles méthodes et constitue l'un des premiers pas dans le développement de techniques d'analyse adaptées à ART. TaintART de Sun *et al.* [112] est une version adaptée à ART de TaintDroid et utilise le compilateur AOT d'ART pour injecter des méthodes de marquage au moment de l'installation de l'application, moment où celle-ci est compilée en instructions-machine. Cette méthode repose néanmoins sur une version modifiée du SE et n'est pas triviale à généraliser à de nouvelles plateformes.

Analyse des entrants/extrants de la plateforme Ce type d'analyse traite une application suspecte comme une boîte noire dont la seule visibilité porte sur ses entrants et extrants. La nature de ces derniers diffère selon l'approche. Un type est l'ensemble des communications réseau établies par une application pendant son exécution. Cette approche par l'analyse de la trace réseau vise à inférer la nature malicieuse ou bénigne d'une application à partir des caractéristiques explicites (le message contenu dans la communication) ou implicites (taille du paquet, port de connexion, délai dans les envois, etc.) du trafic réseau qu'elle génère, dans la mesure où ces données sont accessibles et lisibles (chiffrement, obfuscation, fragmentation des paquets, etc.). Zaman *et al.* [133] ont proposé une méthode qui capture l'ensemble des paquets réseau entrants et sortants imputables à l'application analysée afin d'y retrouver les adresses Web contactées. Ces adresses sont comparées à une liste de domaines réputés malveillants. Les auteurs postulent que la présence d'un tel domaine est suffisante pour considérer l'application comme malicieuse. Feizollah *et al.* [36] ont pour leur part utilisé comme variables la taille des paquets réseau, la durée des communications, la taille de l'entête (*frame*) et le port d'origine des paquets capturés. Les données étaient accumulées pendant une durée de 30 minutes d'exécution d'applications bénignes et malicieuses. Une fois extraites, ces données ont été utilisées pour établir un modèle de classification permettant d'identifier correctement les échantillons malicieux.

Il a été proposé d'analyser le contenu des messages textes entrants et sortants. En effet,

certaines applications malicieuses utilisent l'envoi de messages textes à des numéros payants afin de faire des profits à l'insu de l'utilisateur. Certains de ces maliciels utilisent des techniques pour dissimuler ces envois tels que décrits par Olipane [91]. Dans cette optique, Abah *et al.* [4] ont utilisé les messages textes de même que l'ensemble des appels téléphoniques, l'état de l'appareil et la liste des applications en exécution comme variables pour générer un modèle statistique capable de reconnaître une application malicieuse.

Analyse par détection d'anomalies L'analyse par détection d'anomalies établit un niveau de base à partir de mesures prises sur l'appareil de capteurs différents puis compare celui-ci avec des mesures obtenues pendant l'exécution d'une application afin de déterminer s'il y a présence d'un comportement malicieux. Les mesures retenues ne sont pas, en général, directement associées à un comportement malicieux, mais peuvent indiquer une activité atypique qui y serait associée. Cette relation peut être accentuée si plusieurs mesures atypiques surviennent simultanément. Par exemple, Dixon *et al.* [28] ont corrélé le niveau de consommation de la pile à la géolocalisation de l'appareil pour bâtir un modèle de détection d'anomalies. La prémisse de leur étude est que l'utilisation d'un appareil intelligent diffère en fonction de l'emplacement de son utilisateur. Si une augmentation de la consommation de l'appareil est observée à un emplacement inhabituel (p. ex. pendant un déplacement), ce comportement est jugé suspect. Également, Yang et Tang [131] ont utilisé le profilage de la consommation énergétique des applications par leur famille (p. ex. jeux, réseaux sociaux, etc.). Cette technique permet de relever les applications qui possèdent un profil de consommation qui diffère de la famille à laquelle elles appartiennent et ainsi identifier des candidats effectuant des actions pouvant indiquer une activité malicieuse. Finalement, Kurniawan *et al.* [72] ont combiné la température de la pile et le niveau d'activité sur les communications réseau à la consommation de puissance pour générer modèle de classification permettant de déterminer si l'ensemble des informations lues sur la plateforme est le signe d'une menace.

Il est difficile d'appliquer cette approche dans un environnement virtuel comme la plupart des données recueillies proviennent de la lecture de la consommation énergétique et des valeurs retournées par les capteurs présents sur la plateforme au moment de l'analyse. En effet, ce type d'environnement simule ou fige la valeur des mesures retournées par les capteurs (p. ex. l'accéléromètre ou la position GPS) ou de la consommation électrique ce qui ne permet pas d'observer des variations sur ces variables pouvant indiquer des anomalies.

Limitations

À la lumière des travaux présentés dans cette section, il est possible de conclure que l'analyse dynamique est une approche complémentaire à l'analyse statique. Toutefois, les techniques d'AD sont elles-mêmes soumises à des contraintes qui en limitent leur efficacité. Notamment, elles requièrent plus de temps pour collecter les données nécessaires à l'analyse puisqu'elles sont accessibles uniquement par l'exécution de l'application. Le succès de l'analyse repose également

sur l'expression du comportement malicieux qui doit survenir pendant la durée de l'analyse ou il ne sera pas capturé. En plus de la durée de l'analyse, l'application doit être stimulée avec des évènements internes (p. ex. changement d'heure, alarmes, etc.) ou externes (p. ex. interaction avec l'interface, envoi/réception de messages textes, changement de coordonnées GPS, etc.) pour détoner les bombes logiques qui sont parfois présentes dans les maliciels dans un but d'évasion de la détection.

Étant donné que l'AD requiert l'exécution d'un échantillon pour obtenir les comportements à analyser, l'utilisation de MV pour effectuer cette analyse est courante. Comme discuté précédemment, l'utilisation de la virtualisation permet de restaurer l'état initial entre chaque analyse. Cependant, l'utilisation de MV provoque des disparités au niveau de l'environnement d'exécution. La recherche n'a pas été en mesure de démontrer qu'il était possible d'imiter en tout point un environnement réel par cette technique. Or, comme il a été présenté à la section 2.1.2, ces imperfections peuvent servir à la détection de l'environnement d'analyse ce qui peut limiter l'étendue des comportements observés dynamiquement.

Également, certaines techniques d'acquisition et d'analyse discutées précédemment sont implémentées en modifiant le SE Android ou l'environnement d'exécution (émulateur ou machine virtuelle). Comme les mises à jour peuvent apporter des modifications importantes aux composantes internes d'un SE, il est possible qu'elles apportent des modifications invalidant les outils qui altèrent ces mêmes composantes pour fonctionner. Par exemple, le remplacement de la MVD par ART a rendu inutilisables les techniques qui reposaient sur l'instrumentation de cette composante lors de l'exécution d'une application (p. ex. TaintDroid [34]).

Résumé

Ce chapitre a présenté les principaux enjeux reliés à l'analyse de maliciels sur Android. Les techniques d'analyse présentées apportent des pistes intéressantes afin d'effectuer la détection de maliciels sur Android. Ce chapitre a permis de mettre en lumière qu'il serait pertinent d'explorer une méthode d'analyse capable d'extraire du contenu dynamique tout en étant résiliente aux méthodes d'évasion de la détection. L'approche par l'AMV, un type d'AD, est discutée au chapitre suivant et semble être une bonne candidate pour l'analyse des maliciels sur Android. Elle a été relativement peu étudiée sur ce SE mobile malgré son importance dans l'analyse de maliciels portant sur Windows, Linux et OS X où les outils sont bien documentés [58]. Le prochain chapitre présente les principaux travaux ayant été réalisés pour Android en termes d'AMV et cadre l'état de la connaissance de ce domaine.

Chapitre 3

Analyse de la mémoire vive

Importante dans le domaine de l'informatique d'enquête (*cyberforensics*), l'analyse de la mémoire vive (AMV) fait partie de la famille de l'analyse dynamique (AD). Il s'agit d'une approche éprouvée pour effectuer l'analyse des maliciels sur les ordinateurs personnels. Hale Ligh *et al.* [58] ont rédigé un livre complet couvrant l'acquisition et l'analyse de la mémoire vive pour les SE Windows, Linux et Mac OS X principalement. Sur Android, plusieurs travaux (p.ex. [80, 113, 118]) ont démontré qu'il était possible d'extraire les SMS, des informations sur l'état interne du noyau Linux et les classes Java chargées dans l'application par l'AMV. Ce chapitre détaille l'état de la recherche sur l'AMV portant sur Android.

La section 3.1 présente les principales techniques d'acquisition de la mémoire vive présentes sur Android. La section 3.2 fait état des techniques d'analyse applicables à ces captures. L'outil d'AMV appelé *Volatility* et les approches l'utilisant y sont également présentés.

3.1 Acquisition

L'acquisition de la mémoire vive est l'action d'extraire la mémoire volatile d'un appareil en exécution et de la copier dans un fichier appelé capture. Une considération importante de l'acquisition décrite dans les travaux de Chan *et al.* [23] ainsi que Solomon *et al.* [109], est que les données contenues en mémoire vive peuvent être perdues si elles ne sont pas acquises rapidement après la survenue de l'évènement d'intérêt. Le choix de l'algorithme de gestion de la mémoire utilisée dans le SE, l'espace en mémoire disponible, le type d'opérations effectuées sur le système ainsi que la charge de travail du processeur peuvent affecter l'espérance de vie des données qui y sont contenues. Pour ces raisons, il est important qu'un processus d'acquisition soit rapide et affecte lui-même le moins possible ce contenu lors de l'opération de capture de la mémoire vive afin d'éviter la corruption des données d'intérêt.

L'un des différenciateurs des techniques disponibles est l'impact sur le contenu de la mémoire vive. L'étude de Aljaedi *et al.* [6] a démontré que l'acquisition de la mémoire vive par une

application exécutée à même la plateforme analysée peut entraîner une dégradation des données d'intérêt en mémoire vive. Cela se produit lorsqu'une application réserve une plage de mémoire pour y stocker des données sensibles puis libère ces emplacements pour des allocations futures. Dans cette situation, le contenu n'est effacé que lorsque du nouveau contenu y est réécrit. Comme l'utilisation d'une application d'acquisition de la mémoire vive nécessite qu'elle y soit elle-même chargée, il se peut que les espaces en mémoire qu'elle vient occuper aient appartenu à de telles données d'intérêts. Ils sont alors réécrits et irrécupérables à l'AMV. Pour cette raison, il est nécessaire de minimiser la quantité d'espace mémoire qui est affectée par le programme d'acquisition.

Carrier et Grand [19] ont proposé l'utilisation de matériels dédiés de type Peripheral Component Interconnect (PCI). Il a été également démontré possible d'utiliser des périphériques déjà présents sur une plateforme afin d'acquérir la mémoire vive. Notamment, les études de Gladyshev et Almansoori [43] et Zhang *et al.* [137] ont démontré cette capacité par l'utilisation du FireWire alors que Balogh et Mydlo [11] se sont intéressés à utiliser la carte réseau d'un système pour cette tâche. L'utilisation de matériel dédié permet de lire l'entièreté de la mémoire vive, et ce, sans nécessiter l'intervention du SE et du microprocesseur. En effet, l'interaction avec la mémoire vive s'effectue à l'aide d'une composante appelée Direct Memory Access (DMA) permettant à un périphérique de lire ou écrire directement avec la mémoire vive de façon indépendante de l'exécution du microprocesseur. Cette approche permet d'éviter l'envoi d'instructions au microprocesseur nécessitant des lectures et écritures en mémoire vive pour l'acquisition ce qui limite les risques de corruption des données qui s'y trouvent. Cette technique nécessite néanmoins la présence ou l'ajout de composantes électroniques sur le système ciblé et n'a pas été répliquée sur des appareils mobiles Android.

Une autre technique d'acquisition utilisée par Breeuwsma [15] fait appel au bus Joint Test Action Group (JTAG). Le JTAG est, à l'origine, un outil de développement permettant de charger un microprogramme sur un microcontrôleur et d'en faire le déverminage. Il permet d'interrompre l'exécution du microprocesseur pour contrôler son état interne, comme la valeur de ses registres ou des plages mémoires qu'il contient. Le contrôle de l'exécution du microprocesseur est l'une des fonctionnalités du JTAG qu'il est possible d'utiliser pour interrompre l'exécution de celui-ci et de lire l'entièreté de sa mémoire vive. Cette approche peu utilisée dans la littérature demande un accès physique au microcontrôleur de l'appareil visé. Également, elle peut nécessiter des modifications physiques comme des soudures de précision pour relier un connecteur JTAG au microcontrôleur. Ces altérations peuvent entraîner la destruction des composantes en cas d'erreur de manipulations. De plus, certains appareils n'exposent pas de connecteur JTAG ou celui-ci a été désactivé de façon permanente une fois l'installation d'usine terminée.

Sur Android, l'approche matérielle est généralement limitée puisque les plateformes mobiles, principal marché d'utilisateurs du SE, sont peu propices à l'ajout ou l'utilisation de matériel

d'acquisition de la mémoire vive. Pour cette raison, la plupart des approches relevées dans la littérature sont de nature logicielle. *dd* est l'un de ces logiciels utilisés pour faire l'acquisition de la mémoire vive. Il s'agit d'un utilitaire permettant de faire une copie binaire intégrale d'un espace de stockage (volatile ou non) vers un fichier ou un autre périphérique. Il est notamment utilisé pour faire la copie d'image de disque d'un média à l'autre. Il peut être également utilisé pour faire la copie du contenu de la mémoire vive exposée par le noyau Linux à un utilisateur avec les privilèges *root*. La copie s'opère généralement directement sur la mémoire vive, représentée sous la forme d'un fichier virtuel sur Linux (p. ex. */dev/fmem*). Pour ce faire, le noyau du SE doit avoir été compilé avec l'option correspondante pour en activer le support, sans quoi le fichier virtuel représentant la mémoire vive n'existera pas.

Il a été observé par Sylve *et al.* [113] que l'utilisation de *dd* modifie une quantité appréciable de champs en mémoire lorsqu'il est exécuté. Pour un émulateur Android possédant 512 Mo de mémoire vive, l'utilisation de *dd* provoque des écritures sur près de 20% de la totalité du contenu de la mémoire vive, détruisant toute information d'intérêt qui s'y trouve. La propension du SE à réécrire sur des plages mémoires pouvant contenir des données d'intérêt diminue avec l'augmentation de la quantité d'espace mémoire disponible. Plus le système dispose d'espace mémoire de libre, moins il est nécessaire de réutiliser des plages mémoires ayant déjà servi pour faire l'allocation de nouveaux espaces mémoire.

Il est également possible d'acquérir la mémoire vive d'un processus par l'utilisation d'un dévermineur comme *gdb* [44]. Sur Android, il est nécessaire d'avoir les privilèges de l'utilisateur *root* pour connecter *gdb* à un processus en exécution et y lire la mémoire vive si ce processus n'est pas un enfant du processus lançant *gdb*. De plus, cette méthode est limitée à un ou plusieurs processus, mais ne permet pas de cibler le noyau Linux lui-même à moins qu'il n'ait été recompilé pour supporter cette option. Si tel est le cas, il est possible de contrôler l'ensemble de son exécution par un dévermineur distant, mais aux coûts d'importants ralentissements pouvant interférer avec des opérations sensibles aux retards (p. ex. communications réseaux, périphériques, etc.).

Le dévermineur d'applications Android permet également d'analyser les variables en mémoire de celles-ci lors de leur exécution. Une application doit avoir été prévue pour être déverminée au moment de sa compilation, sans quoi le dévermineur ne sera pas autorisé à la contrôler. Cette approche ne permet pas d'observer l'ensemble du contenu de la mémoire vive du SE, des autres processus externes aux applications Android et des autres applications qui ne sont pas configurées pour être déverminables.

L'absence de matériel d'acquisition dédié pour plateforme Android, l'impact négatif de *dd* sur la mémoire vive et les contraintes des dévermineurs ont poussé l'exploration d'autres approches. Deux principales techniques d'acquisition de la mémoire vive se démarquent sur Android : l'acquisition par l'utilisation de l'outil Linux Memory Extraction (LiME) et par

l'utilisation d'une MV ou d'un émulateur. Elles sont présentées à la section suivante.

3.1.1 LiME

Sylve *et al.* [113] ont proposé *dmd*, devenu Linux Memory Extraction (LiME)[3], un LKM capable de faire l'acquisition de la mémoire vive sur les systèmes Linux. Il est possible d'utiliser ce module pour Android puisque le SE possède un noyau Linux. Le module LiME capture la mémoire vive de la façon suivante :

1. Le LKM récupère la première structure du noyau *iomem_resource* contenant la valeur de l'adresse d'un espace d'adressage atteignable. Ce-dernier peut désigner un emplacement en mémoire vive ou correspondre aux registres utilisables d'un périphérique. En effet, les registres internes de certains périphériques peuvent être également atteints par l'utilisation de l'adressage en mémoire.
2. La lecture de la structure du noyau *iomem_resource* permet de déterminer si l'espace mémoire en question correspond bien à un élément de la mémoire vive du système (attribut *iomem_resource->name == "System RAM"*). Dans la négative, celui-ci n'est pas d'intérêt et le module passe au prochain espace mémoire (attribut *iomem_resource->sibling*) et refait la même validation. Dans l'affirmative, il poursuit à l'étape suivante.
3. Les adresses physiques en mémoire de début et de fin du segment désigné par *iomem_resource* à la première étape sont traduites en adresses virtuelles. [1]
4. Le contenu qui se trouve entre l'adresse virtuelle de début et de fin est lu et écrit dans un fichier ou transmit par une communication TCP/IP vers l'acquéreur.

La boucle se termine lorsque la condition *iomem_resource->sibling == NULL* est vraie. Cela signifie que l'ensemble des espaces mémoire accessibles ont été parcourus. LiME permet l'écriture en 3 formats de capture, soit brute, remplissage et LiME. Le premier est une capture brute où toutes les données en mémoire vive sont disposées de façon contigüe dans un seul fichier. Le second format remplit de 0 les espaces en mémoire dont l'adresse ne correspond pas à la mémoire vive du système (par exemple, les plages désignant l'accès aux registres des périphériques). Beaucoup plus volumineux que le format brut, il est possible d'accéder à des données dans ce format par une transposition directe d'une adresse physique vers son emplacement correspondant dans le fichier. Finalement, le troisième format insère au début du fichier un entête présenté à la figure 3.1 et tiré de la documentation de LiME, également décrite par Sylve *et al.* [113].

1. Les adresses virtuelles constituent une représentation de l'espace d'adressage exposée par le SE à un processus. Les valeurs de ces adresses virtuelles doivent être traduites à la volée par le SE et des composantes physiques dédiées du processeur vers les adresses physiques afin d'obtenir l'emplacement réel des données dans les médias de stockage de la mémoire vive. Ce mécanisme permet notamment d'offrir une plage continue de valeurs d'adressage et est prévisible aux yeux d'un processus. Cette plage et les données qui s'y trouvent peuvent tout de même être réaffectées à une autre adresse physique en mémoire ou même être changées de médium de stockage sans impact pour le programme qui les utilise. Hale Ligh *et al.* [58] détaille la mémoire virtuelle au chapitre 1 de leur ouvrage et le lecteur est invité à s'y référer pour de plus amples détails.

```
typedef struct {
        unsigned int magic;     // Always 0x4C694D45 (LiME)
        unsigned int version;       // Header version number
        unsigned long long s_addr;  // Starting address of physical RAM range
        unsigned long long e_addr;  // Ending address of physical RAM range
        unsigned char reserved[8];  // Currently all zeros
} __attribute__ ((__packed__)) lime_mem_range_header;
```

Listing 3.1 – Entête d'une capture de mémoire vive LiME.

En détail, *magic* est une valeur fixe désignant le début de l'entête, *version* désigne la version utilisée de l'outil, *s_addr* indique l'adresse physique où débute le segment qui suit sur l'appareil, *e_addr* indique sa fin et *reserved* est réservé pour des usages futurs. La taille du segment est obtenue par l'équation suivante.

$$t = e\_addr - s\_addr$$

Une quantité de t octets se trouvent après l'entête et correspondent au contenu du segment en mémoire. Si un autre segment de mémoire vive doit être ajouté dans le fichier et que l'espace d'adressage n'est pas contigu au précédent, un autre entête est ajouté à la fin du segment précédent. Le nouveau segment est à son tour ajouté après ce deuxième entête. Ce processus se poursuit jusqu'à ce qu'il n'y ait plus de segment de mémoire vive à copier.

L'utilisation de LiME nécessite que le noyau Linux permette le chargement d'un LKM. Il s'agit d'une limitation importante puisque cette fonctionnalité est désactivée par défaut sur les versions d'origines d'Android. Toutefois, il est possible de recompiler un noyau Linux d'Android à partir du code source pour l'activer pour ensuite déployer ce noyau sur l'appareil visé. Les étapes pour la configuration de LiME sont présentées par Macht [80] et, plus récemment, sur la page Web du projet *Volatility*[2]. Elles se résument de la façon suivante :

1. Télécharger le code source de LiME[3].
2. Télécharger le code source du noyau Linux utilisé sur Android pour l'appareil ciblé. Le processus pour les appareils Nexus de Google est présenté dans la documentation officielle d'Android[4]
3. Compiler le noyau. L'annexe D présente le script de compilation utilisé dans le cadre de la maîtrise. Il est à noter que le script ajoute les instructions *CONFIG_MODULES=y* et *CONFIG_MODULE_UNLOAD=y* au fichier de configuration utilisé pour la compilation. Ces options sont responsables d'ajouter le support des LKM par le noyau.
4. Compiler LiME à l'aide du noyau précédemment compilé. L'annexe E détaille cette étape.
5. Déployer le noyau modifié sur la plateforme visée. La réalisation de cette étape varie grandement en fonction de l'appareil visé.

2. Voir `https://github.com/volatilityfoundation/volatility/wiki/Android`
3. Voir `https://github.com/504ensicsLabs/LiME` .
4. Voir `https://source.android.com/source/building-kernels.html` .

6. Débloquer l'utilisateur *root* de l'appareil. Ce processus varie également d'une plateforme à l'autre.
7. Téléverser la version compilée de LiME sur l'appareil.
8. Lancer l'acquisition au moment désiré à l'aide d'une commande passée à l'appareil via l'invite de commandes accessible par l'utilisation de l'Android Debug Bridge (ADB).

L'un des avantages de LiME rapportés par Sylve *et al.* [113] est qu'il a un impact significativement moindre que l'utilisation de *dd* sur Android sur le contenu de la mémoire vive. L'étude rapporte que LiME conserve près de 99% du contenu de la mémoire intacte pendant l'acquisition alors que l'utilisation de *dd* est de l'ordre de 80%. Les chances de corrompre une donnée sensible en mémoire vive par le processus d'acquisition lui-même sont donc plus faibles. Il permet également le transfert de la capture directement par une connexion TCP plutôt que d'en faire une sauvegarde locale sur l'appareil, évitant de modifier l'état de l'appareil examiné. LiME capture l'entièreté de la mémoire visible par le SE et peut être utilisé sur des appareils physiques comme des téléphones intelligents ou des tablettes, contrairement à l'approche par MV ou émulateur décrite à la section 3.1.2. Ce dernier point constitue un avantage important dans un cas de maliciel muni d'une capacité de détection d'environnement d'analyse comme celles décrites à la section 2.1.2.

Limitations LiME copie l'ensemble de la mémoire vive pendant l'exécution du système et requiert un délai pour compléter l'opération variant selon la capacité de stockage de la mémoire vive. Ce contexte de capture peut entraîner des incohérences dues à des accès qui peuvent survenir pendant l'acquisition. Par exemple, il est possible que les données dont l'adresse est référencée dans un segment de la mémoire n'existent plus, aient été déplacées ou misent en cache sur le disque au moment où cette adresse est lue par LiME. Également, une preuve de concept sur Windows intitulée *Shadow Walker*[110] a permis de démontrer qu'un maliciel peut parvenir à dissimuler sa présence à un outil d'acquisition de la mémoire vive. Pour ce faire, un maliciel doit parvenir à détourner les méthodes système effectuant la lecture de la mémoire vive afin que celles-ci retournent du faux contenu lorsque la plage mémoire contenant le maliciel est lue. Heureusement, ce scénario n'a pas été répliqué sur Android et est très complexe à mettre en exécution.

Un autre aspect négatif de cet outil est que le processus permettant de déployer LiME n'est pas trivial. Pour parvenir à une version fonctionnelle de l'outil, il est nécessaire de bien connaître les principes de compilation d'un noyau Linux. Ce couplage fort entre LiME et la version du noyau utilisée nécessite que les processus de compilation et de déploiement de ces deux composantes soient répétés pour chaque nouvelle version du SE. Stüttgen et Cohen [111] ont tenté de contourner cette contrainte. La technique proposée par les auteurs nécessite d'abord d'identifier un LKM qui utilise les mêmes fonctions et les structures internes du SE que celles requises pour faire l'acquisition de la mémoire vive. Une fois ce LKM identifié, les auteurs réordonnent son flot d'exécution et la valeur de ses structures internes afin que ceux-ci reflètent

le nouveau comportement visé. Conséquemment, le contenu exécutable de ce LKM hôte est remplacé dynamiquement par la routine d'extraction de la mémoire vive puis est exécuté par le noyau. Bien que cette technique ait été démontrée possible sur les plateformes Linux Ubuntu et Fedora, les auteurs soulignent que l'application de cette procédure n'a pas été testée sur Android et que plus de recherche est nécessaire afin de déterminer si elle peut y être adaptée. Également, comme il est expliqué à la section 2.2.2, les LKM ne sont plus supportés dans la version officielle de Google à moins de recompiler le noyau, ce qui complique les approches décrites précédemment.

En résumé, l'outil LiME est efficace pour l'acquisition de la mémoire vive et son utilisation a un faible impact sur celle-ci. Il est également possible de l'utiliser sur des systèmes physiques, pour autant que le code source du SE de ceux-ci soit accessible. Son déploiement demeure toutefois complexe par la nécessité de recompiler et déployer un noyau Linux de même que l'outil lui-même sur l'appareil visé. Ceci étant dit, son utilité demeure justifiée dans le cadre d'un environnement d'analyse de maliciels où il est possible de préconfigurer l'appareil d'analyse avant d'y déployer un maliciels.

3.1.2 Acquisition par machine virtuelle

Présentée à la section 2.2.2, l'utilisation de MV peut permettre d'obtenir la mémoire vive du système invité. La mémoire vive est contenue dans les instantanés générés par l'engin de virtualisation ou d'émulation. Il est possible d'extraire la mémoire vive d'une MV dans plusieurs des engins de virtualisation ou d'émulation disponibles sur le marché comme QEmu [113], VirtualBox [92] et VMware [123]. L'acquisition peut se faire directement par une commande passée à la plateforme de virtualisation ou utiliser l'instantané fait d'une MV pour y extraire la mémoire vive. Cette technique ne provoque pas de corruption de mémoire due à l'acquisition elle-même puisque le système invité est complètement figé à son insu pendant la capture.

Cette approche repose sur l'acquisition de la mémoire vive par des opérations qui sont externes au SE invité. De ce fait, il est improbable qu'un maliciel puisse interférer avec le processus de capture afin de dissimuler l'activité malicieuse. Finalement, l'utilisation d'instantané préserve l'état complet du système invité (p. ex. le stockage, l'état du processeur, etc.) donnant la possibilité de combiner les résultats de l'AMV à l'analyse d'autres composantes du système virtualisé au besoin.

Limitations Pour appliquer cette approche à un appareil physique, il serait nécessaire de pouvoir capturer l'ensemble de l'état de ce dernier afin d'être capable de le répliquer dans un environnement virtuel, et ce, pendant son exécution. À ce jour, le clonage d'un appareil Android avec la version d'origine du SE déployée par le fabricant n'a pas été démontré. Cette approche ne peut donc être utilisée dans le contexte d'une gestion d'incident réel où un analyste désire inspecter un appareil suspect.

Également, la capture de la mémoire vive par l'outil de virtualisation n'est plus supportée par l'émulateur Android officiel de Google. Le retrait de cette fonctionnalité ne permet plus d'utiliser la technique d'extraction décrite par Sylve *et al.* [113] pour Android. En effet, lorsque la commande d'acquisition est lancée sur l'émulateur, un message d'erreur indique que cette fonctionnalité est absente tel que présenté au listage 3.2. À la ligne 1, la connexion pour le contrôle de l'émulateur est établie via l'utilitaire *netcat* depuis le poste hôte. La commande *qemu monitor stdio* est envoyée à l'émulateur à la ligne 4. L'émulateur retourne un message d'erreur à la ligne 5 signifiant que le support de cette fonctionnalité a été retiré.

```
1 pc@current:~\$ nc 127.0.0.1 5554
2 Android Console: type 'help' for a list of commands
3 OK
4 qemu monitor stdio
5 KO: QEMU support no longer available
```

Listing 3.2 – Support de la capture de la mémoire vive retirée sur l'émulateur Android.

Pour cette raison, il est nécessaire d'utiliser des outils de virtualisation comme VirtualBox ou VMware pour obtenir une capture de cette façon. Or, ces systèmes permettent uniquement de virtualiser l'architecture de microprocesseur Intel x86/x64. Pour cette raison, ces outils ne sont compatibles qu'avec une mouture d'Android non officielle appropriée à ce type d'architecture appelée Android-x86 [64]. Or, cette version comporte des différences avec la version d'origine d'Android, ne virtualise pas l'activité de capteurs comme l'accéléromètre et n'est pas mise à jour au même rythme que la version officielle d'Android puisque ces mises à jour doivent être adaptées avant d'être applicables.

Finalement, tel qu'il est présenté à la section 2.1, il est possible pour un maliciel de détecter s'il est exécuté dans un environnement de virtualisation et d'inhiber un comportement malicieux si tel est le cas. Ainsi, les données recueillies par un environnement de virtualisation peuvent différer de ce qui serait collecté sur un appareil physique si un maliciel inhibe ses actions pour éviter la détection. Pour cette raison, la validité des résultats peut être en doute s'ils reposent sur des données recueillies par une MV.

3.2 Analyse

Les données acquises prennent la forme de valeurs binaires brutes nécessitant un traitement pour être utilisables à l'analyse. Comme décrit dans la méta-analyse de Garcia [41], il est possible de faire une recherche de chaînes de caractères identifiables contenues dans un fichier de capture afin d'extraire des valeurs indicatrices d'une activité malicieuse. Cette méthode a l'avantage d'être très rapide à exécuter puisque des outils optimisés pour ce type de recherche existent et sont communs. Par exemple, *grep* est un outil inclus dans le noyau Linux permettant d'identifier les chaînes de caractères contenues dans un fichier qui coïncident à une expression régulière soumise par l'utilisateur. Cette approche est limitée si la chaîne recherchée change

(p. ex. changement de l'adresse d'un serveur de commande et contrôle du maliciel), si elle est segmentée (p. ex. plusieurs segments non contigus plutôt qu'un tableau continu en mémoire) ou si elle est inconnue (p. ex. une clef de chiffrement binaire).

Une autre technique procède plutôt en parcourant l'ensemble du fichier de capture et en recensant les chaînes d'octets répondant à un ensemble de critères indiquant la présence d'une chaîne de caractères. Par exemple, l'outil *strings* identifie l'emplacement dans le fichier où se trouvent toutes séquences d'au moins 4 octets ou plus pouvant représenter un caractère et se terminant par un octet non imprimable (p. ex. l'octet nul '0x00'). Cette approche relève beaucoup de faux positifs devant être triés. Ce travail est généralement réalisé par un expert pouvant être assisté par un ou des algorithmes permettant de trier la pertinence des résultats (p. ex. Beebe et Clark [13]).

Les techniques reposant sur les chaînes de caractères ne sont pas en mesure de relever des variables d'un type autre que des chaînes de caractères comme la valeur d'un nombre à virgule flottante de type *float* ou une structure interne du noyau. Également, le contenu récupéré est décontextualisé, c'est-à-dire que les informations liées à son contexte d'exécution où est utilisé le texte ne sont pas récupérées par ces techniques. Cette contrainte peut entraîner des faux positifs si la chaîne de caractères visée apparaît dans un contexte légitime et sans présence malicieuse dans le système. Par exemple, un navigateur Web pourrait avoir dans son espace mémoire toutes les chaînes de caractères indiquant la présence d'un comportement malicieux si le contenu de la page Web affichée à l'écran décrit les chaînes de caractères du maliciel en question. Cet exemple représente le cas d'une application légitime qui serait faussement caractérisée comme un maliciel. À l'inverse, si une variante du maliciel utilise une adresse de serveur différente, celle-ci pourrait passer inaperçue.

Puisque le contexte d'exécution importe pour effectuer une AMV plus complète, une représentation de la topologie de la mémoire vive est préférable pour refléter l'état internes du SE au moment de la capture. Cela permet de récupérer des structures interne du SE comme l'inventaire des processus en exécution, la liste des fichiers utilisés de même que leur contenu tel que présenté par Case *et al.* [20]. Cette reconstruction de la topologie de la mémoire vive est une stratégie répandue dans les travaux portant sur l'AMV, tous SE confondus (p. ex. [29, 41, 58, 94, 121, 125]).

3.2.1 Volatility

Sur Android, l'analyse de la mémoire vive a été principalement étudiée à l'aide d'un logiciel libre appelé *Volatility*[5] programmé en *Python* [58, 80, 113]. Cette section vise à présenter le fonctionnement global de *Volatility* et son utilisation pour des systèmes Linux et, par extension, Android. Le lecteur est référé à l'ouvrage de Hale Ligh *et al.* [58] pour une compréhension

5. `https://github.com/volatilityfoundation/volatility`

approfondie de l'outil et de ses capacités, notamment en ce qui a trait à Mac OS X et Windows.

Description *Volatility* permet de reconstruire la représentation des structures spécifiques au SE présent sur le système d'où provient la capture depuis la mémoire vive. Pour se faire, *Volatility* charge un profil associé à une capture de mémoire vive lui permettant d'interpréter ce fichier en tenant compte du contexte de l'acquisition. Un profil comprend les éléments présentés au tableau 3.1 tiré de Hale Ligh *et al.* [58].

Composantes	Description
Métadonnées	Nom du SE, version, numéro du *build*.
Informations sur les appels systèmes	Indexes et noms.
Constantes du SE	Les variables à position fixe en mémoire du SE.
Type des primitives	Types des variables primitives (habituellement en *C*), incluant le nombre de bits des *integer*, *float*, etc.
System map	Structure contenant l'ensemble des adresses pour les variables et méthodes globales qui doivent être à un emplacement fixe en mémoire pour que le SE fonctionne correctement. Ce fichier est nécessaire pour les systèmes Macs et Linux.

TABLEAU 3.1 – Éléments d'un profil.

Un profil est constitué de 3 composantes requises pour reconstruire les structures en mémoire vive, soit les *VTypes*, les *Overlays* et les objets de *Volatility*.

Les *VTypes* sont des classes d'objets spécifiques à, et manipulables par *Volatility* qui permettent de transposer des valeurs binaires en mémoire vive vers leur structure C correspondante dans un format compatible avec *Python*. Une comparaison à la figure 3.1 présente une structure *C* et le *VType* équivalent en *Python*.

```
1 struct process {
2     int pid;
3     int parent_pid;
4     char name[10];
5     char * cmd;
6     void * ptv;
7 };
```

```
1 'process':[26, {
2     'pid':[0,['int']],
3     'parent_pid':[4,['int']],
4     'name':[8,['array',10,['char']]],
5     'cmd':[18,['pointer',['char']]],
6     'ptv':[22,['pointer',['void']]]
7 }]
```

FIGURE 3.1 – Structure *C* (gauche) et équivalent en *VType* (droite).

Le nom de la structure est indiqué à la ligne 1 du *VType* suivi du nombre d'octets qu'elle nécessite en mémoire. Chaque ligne subséquente présente le nom de la variable, l'*offset* en octets où elle se trouve à partir du début de la structure et finalement son type. Si le type est un tableau comme à la ligne 4, l'identifiant *array*, la taille du tableau et le type des variables qu'il contient sont également indiqués. Ces *VTypes* sont essentiels à la reconstruction des structures contenues en mémoire vive et chaque version de SE nécessite un ensemble de VTypes spécifique. Plus particulièrement sur Linux et Android, ils doivent être générés lors du processus de compilation du noyau à l'aide de *dwarfdump*, un outil de développement permettant d'extraire

automatiquement ces informations pour l'ensemble des structures présentes dans le noyau à partir du code source de celui-ci.

Il est possible d'ajouter ou de modifier les définitions des *VTypes* en appliquant des *Overlays* sur celles-ci. Les *Overlays* sont des définitions suivant le format d'un *VType* permettant de redéfinir certaines structures afin qu'elles correspondent au contexte d'analyse. Par exemple, un pointeur de type *void* pourrait correspondre à une structure connue de l'analyste et applicable pour une version modifiée de Linux. Ainsi, lorsque ce contexte d'analyse sera rencontré, l'*Overlay* correspondant peut être appliqué pour que *Volatility* reconstruise cette structure référencée et extraie son contenu de la capture de la mémoire vive.

En combinant l'utilisation des *VTypes* et des *Overlays* correspondants, il est possible de recréer des objets *Volatility*. Ces objets sont des structures reconstruites à partir de valeurs binaires contenues dans le fichier de capture de mémoire. Cette reconstruction s'effectue en récupérant le contenu situé à une adresse mémoire spécifique (p. ex. l'adresse d'une constante) ou par la reconnaissance d'un patron de valeurs appartenant à une structure du SE connue au moment de l'analyse (p. ex. l'entête d'une structure de données). Concrètement, les objets *Volatility* sont des classes *Python* contenant les variables définies par les *VTypes* et leur valeur. Ces classes peuvent être étendues pour offrir des fonctions permettant de manipuler leur contenu afin de raffiner l'extraction de marqueurs pertinents à l'analyse.

Dans le cas d'un système Linux, un profil est une archive Zip contenant l'information sur les structures de données du noyau Linux et ses *debug symbols* [124]. Contrairement à Windows pour lequel l'outil possède un répertoire de profils préconstruits, un profil Linux doit être construit par l'analyste spécifiquement pour chaque version du SE analysée.

La capture de mémoire vive peut être chargée dans l'outil une fois le bon profil *Volatility* choisi. Au moment du chargement, une succession de traductions d'espaces d'adressage est appliquée. Leur but est d'abstraire les accès au fichier de capture pour permettre à l'analyste de naviguer dans la capture de la mémoire vive comme s'il était sur le système acquis lui-même. Notamment, il permet d'explorer directement des emplacements en mémoire en utilisant des valeurs d'adresses telles qu'originalement vues par le SE lors de son exécution et non en fonction d'*offset* dans le fichier de capture. Par exemple, un premier filtre effectue la traduction d'adressage de la mémoire virtuelle vers l'espace d'adressage en mémoire physique en fonction de l'architecture du processeur, un second convertit cette valeur en un emplacement dans le fichier en fonction du type de fichier de la capture (p. ex. pour un format LiME, il faut tenir compte des entêtes du fichier) et finalement un dernier filtre effectue la récupération dans le fichier en fonction de l'*offset* calculé par les étapes précédentes. Il est possible de récupérer une information en n'utilisant qu'une partie de la séquence d'adressage, par exemple si l'analyste connaît directement l'adresse physique en mémoire qui l'intéresse.

Une fois le profil et la capture chargés, *Volatility* est en mesure d'appliquer des analyses sous la

Greffons	*Description*
linux_apihooks	Vérifie s'il y a détournements d'appels à l'API du noyau.
linux_arp	Affiche la table de routage Address Routing Protocol (ARP).
linux_banner	Affiche des informations sur le noyau du système.
linux_bash	Récupère l'historique des commandes *bash*.
linux_bash_env	Récupère les variables d'environnement dynamiques d'un processus *bash*.
linux_bash_hash	Récupère la table de hachage *bash* de la mémoire de ce processus.
linux_check_afinfo	Vérifie l'adresse des pointeurs des fonctions d'opération des protocoles réseau.
linux_check_creds	Valide si des processus partagent la même structure de privilèges.
linux_check_evt_arm	Vérifie la table d'exception afin de valider si un pointeur d'appel système a été modifié.
linux_check_fop	Valide si les structures d'opération ont été modifiées par un exploit de type *rootkit*.
linux_check_idt	Valide si l'IDT Interrupt Descriptor Table (IDT) a été modifiée
linux_check_inline_kernel	Recherche des altérations aux appels aux méthodes du noyau.
linux_check_modules	Compare la liste des modules avec les entrées dans *sysfs info*, si disponible.
linux_check_syscall	Valide si la table d'appels système a été modifiée.
linux_check_syscall_arm	Valide si la table d'appels système a été modifiée (pour processeur ARM).
linux_check_tty	Vérifie si des périphériques de type tty ont été détournés.
linux_dentry_cache	Extrait les fichiers présents dans la cache *dentry*.
linux_dmesg	Récupère le tampon de *dmesg*.
linux_dump_map	Écrit les champs de mémoire spécifiés sur stockage physique.
linux_dynamic_env	Récupère les variables d'environnement dynamiques d'un processus.
linux_elfs	Identifie et récupère les fichiers binaires ELF chargés dans l'espace mémoire des processus.
linux_enumerate_files	Répertorie l'ensemble des fichiers référencés par la cache du système de fichiers.
linux_find_file	Liste et récupère les fichiers en mémoire vive.

TABLEAU 3.2 – Partie 1 - Documentation des greffons pour Linux de *Volatility*.

forme de modules *Python* appelés greffons (ou *plug-ins*) permettant d'extraire les informations désirées sur le SE et ses différentes composantes au moment de la capture.

Android L'AMV sur Android à l'aide de *Volatility* a été principalement étudiée à l'aide de deux approches. La première consiste à utiliser les greffons visant Linux pour extraire l'information qui est commune à tous les SE ayant un noyau Linux. L'ensemble de ces greffons permet de sonder le noyau pour y extraire des traces d'activités anormales. Par exemple, il est possible de récupérer des fichiers entiers ou partiels manipulés en mémoire vive, d'obtenir la liste des processus en exécution au moment de la capture, d'obtenir l'état des connexions réseaux ainsi que les paquets en attentes de traitement par processus ou de vérifier s'il y a eu un détournement d'appels systèmes (ou *system call hooking*). L'ensemble de ces fonctionnalités est détaillé dans la documentation officielle de *Volatility* [124] de même que dans l'ouvrage de Hale Ligh *et al.* [58]. Les tableaux 3.2, 3.3 et 3.4 présentent la liste de greffons Linux ainsi que leur documentation officielle rattachée (traduction libre).

Greffons	*Description*
linux_getcwd	Liste le chemin d'accès courant pour chacun des processus.
linux_hidden_modules	Inspecte la mémoire vive pour y identifier des modules cachés du noyau.
linux_ifconfig	Répertorie les interfaces réseaux actives.
linux_info_regs	Méthode analogue à *info registers* dans *gdb*.
linux_iomem	Produit une sortie similaire à */proc/iomem* sur Linux.
linux_kernel_opened_files	Liste les fichiers chargés dans l'espace mémoire du noyau.
linux_keyboard_notifiers	Reconstruit la chaîne d'appels des notifications du clavier.
linux_ldrmodules	Compare la sortie de *proc maps* avec la liste des bibliothèques logicielles de l'entité *libdl*.
linux_library_list	Répertorie l'ensemble des bibliothèques logicielles chargées en mémoire d'un processus.
linux_librarydump	Extrait les bibliothèques logicielles partagées dans la mémoire d'un processus.
linux_list_raw	Répertorie les applications ayant des *sockets* en mode *promiscuité*.
linux_lsmod	Inventorie les modules du noyau chargés.
linux_lsof	Liste les descripteurs de fichier et leur chemin d'accès.
linux_malfind	Tente d'identifier la présence d'activités malicieuses par l'analyse de la topologie de la mémoire des processus.
linux_memmap	Extrait la topologie de la mémoire vive pour les tâches Linux.
linux_moddump	Extrait les modules du noyau chargés au moment de la capture.
linux_mount	Liste les *fs/devices* montés
linux_mount_cache	Liste les *fs/devices* montés depuis la *kmem_cache*.
linux_netfilter	Liste les détournements présents dans *Netfilter*.
linux_netscan	Identifie les structures de connexions réseaux présentes dans la mémoire vive.
linux_netstat	Identifie les ports ouverts et connexions établies.
linux_pidhashtable	Énumère les processus via la table de hachage des identifiant de processus (Process Id, PID).
linux_pkt_queues	Écrit l'ensemble des queues par processus de paquets présents en mémoire vive sur le disque.

TABLEAU 3.3 – Partie 2 - Documentation des greffons pour Linux de *Volatility*.

Afin d'améliorer l'analyse sur les plateformes Android, la seconde méthode proposée par Macht [80] et 504ensics Labs [1] procède en extrayant la représentation interne des variables et des structures des applications Android par la reconstitution de la MVD depuis une capture de la mémoire vive. En identifiant l'emplacement en mémoire de la MVD relativement au début du processus système *zygote*, il est possible de retrouver l'instance de la MVD dans l'ensemble des autres applications Android en appliquant ce même *offset* à ces applications. Cela est dû au fait qu'une application Android est d'abord initialisée par un appel système *fork*[6] sur *zygote* qui contient toutes les composantes de bases requises pour son lancement, dont la MVD. L'application charge ensuite les composantes qui lui sont uniques (p. ex. classes spécifiques, constantes, bibliothèques logicielles, etc.). En récupérant le contenu de la MVD, il est possible de récupérer l'ensemble des structures Java contenues dans l'application et d'avoir une pleine visibilité sur l'ensemble des structures internes et leur valeur pour une application Android.

6. *fork* permet de générer un nouveau processus identique à un autre et qui partage les mêmes références en mémoire vive que celui d'origine. Les références en mémoire sont modifiées vers un espace mémoire différent dès que l'un ou l'autre des processus effectue une écriture, évitant la corruption de la donnée pour l'autre processus.

Greffons	*Description*
linux_plthook	Recherche dans la table des liens des procédures (Procedure Linkage Table, PLT) pour des appels suspects.
linux_proc_maps	Récupère la topologie de la mémoire des processus.
linux_proc_maps_rb	Reconstitue la topologie de la mémoire des processus pour Linux via l'utilisation d'arborescence rouge-noir.
linux_procdump	Extrait l'exécutable d'un processus et le sauvegarde sur le disque.
linux_process_hollow	Recherche des indicateurs de processus évidés.
linux_psaux	Reproduit la sortie de la commande *ps aux* sur Linux qui permet d'obtenir l'ensemble des processus en exécution, leur chemin d'accès et l'heure de leur démarrage.
linux_psenv	Liste les processus avec les variables d'environnement statiques qui y sont rattachées.
linux_pslist	Reconstruit la liste des tâches Linux actives en parcourant la liste chaînée *task_struct->task list*.
linux_pslist_cache	Reconstruit les tâches Linux présentes dans la structure interne *kmem_cache*.
linux_pstree	Identifie les relations parents/enfants entre les processus en exécution.
linux_psxview	Tente de trouver des processus cachés en mémoire vive en croisant les résultats obtenus par les différentes façons de lister les processus présents en mémoire.
linux_recover_filesystem	Récupère l'ensemble du système de fichiers en cache depuis la mémoire vive, s'il y en a un.
linux_route_cache	Récupère la cache de la table de routage depuis la mémoire vive.
linux_sk_buff_cache	Récupère les paquets réseaux depuis la structure interne *sk_buff kmem_cache*.
linux_slabinfo	Duplique la structure */proc/slabinfo* sur un système en exécution.
linux_strings	Récupère les chaînes de caractères à partir d'une valeur d'adresse physique convertie en adresse virtuelle.
linux_threads	Liste les fils d'exécutions des processus.
linux_tmpfs	Récupère les systèmes de fichiers *tmpfs* en mémoire vive.
linux_truecrypt_passphrase	Tente d'extraire la phrase secrète en cache de l'utilitaire de chiffrement Truecrypt.
linux_vma_cache	Récupère les allocations de mémoire virtuelle à l'aide de la cache de *vm_area_struct*.

TABLEAU 3.4 – Partie 3 - Documentation des greffons pour Linux de *Volatility*.

Les variables, les objets et leur classe d'appartenance sont interprétables de cette façon. Nasim *et al.* [88] ont également utilisé cette représentation de la MVD en mémoire vive. Leur approche a permis d'identifier les classes Java qui s'y trouvent et de les comparer avec celles déclarées dans le code décompilé de l'application. Les classes présentent en mémoire et qui n'ont pas d'équivalence dans l'APK sont identifiées comme ayant été chargées dynamiquement et sont une source potentielle d'injection de comportements échappant à l'analyse statique. L'abolition de la MVD dans les versions actuelles d'Android rend ces techniques obsolètes. À ce jour, peu de travaux se sont intéressés à l'étude de l'AMV sur les versions d'Android reposant sur ART.

Résumé

Ce chapitre résume les principaux travaux réalisés à ce jour sur Android dans le domaine de l'AMV. Bien que la pertinence de cette approche ait été démontrée pour d'autres SE

appartenant à l'univers PC et Mac, les études menées sur Android demeurent limitées. Il est possible de dégager deux courants principaux de l'état de l'art qui se positionnent le long d'un continuum. À un pôle se trouvent les méthodes génériques qui sont résistantes aux mises à jour majeures comme c'est le cas pour l'approche par recherche de chaîne de caractères. À l'opposé se trouvent les approches spécifiques à la plateforme qui permettent d'extraire des résultats très précis et spécifiques à la plateforme, mais dont le bon fonctionnement est fortement couplé à la structure interne du SE et qui doivent refléter les changements à celle-ci. Les techniques reposant sur la MVD comme celles impliquées dans DalvikInspector de 504ensics Labs [1] et dans les modules *Volatility* de Macht [80] représentent bien cette réalité. Afin de combler cet écart, il importe de proposer une technique capable d'extraire des éléments spécifiques aux structures internes du SE à partir d'une capture de mémoire vive. Il incombe que la technique demeure suffisamment générique pour être valide avec peu ou pas d'adaptations entre les versions du SE dans un contexte d'analyse de maliciels. Le chapitre suivant propose une méthode répondant à ces critères et l'applique à un maliciel expérimental afin d'en démontrer l'efficacité.

Chapitre 4

Application de l'analyse de la mémoire vive

Ce chapitre propose une technique d'analyse de la mémoire vive visant un équilibre entre une approche suffisamment spécifique pour capturer des comportements malicieux particuliers à la plateforme Android, mais assez générique pour demeurer valide à travers les versions du SE. La méthode proposée utilise les greffons de *Volatility* ciblant l'extraction de structures internes du noyau Linux depuis une capture de la mémoire vive et les applique dans un contexte d'analyse de maliciels sur Android. Il est également proposé d'utiliser une technique proposée dans ce mémoire, l'analyse différentielle de la mémoire vive (ADMV), afin de concentrer l'effort d'analyse sur les structures affectées par l'exécution de l'application. Ce chapitre présente en premier lieu le rationnel et le fonctionnement de l'ADMV. En second lieu, il présente une étude dans laquelle l'approche est appliquée afin de capturer et d'extraire des artefacts de l'exécution d'un maliciel expérimental conçu à cette fin et vise à démontrer la pertinence de l'ADMV sur une plateforme Android utilisant ART.

4.1 Analyse différentielle de la mémoire vive

L'analyse différentielle peut être définie comme l'utilisation de la différence entre les données recueillies à deux temps de mesure t espacés par Δt. La résultante permet d'isoler uniquement les éléments ayant été créés, supprimés ou modifiés entre ces temps de mesure. Ce substrat réduit la quantité de données à analyser et vise à réduire le temps et la complexité de d'analyse. Dans le cas de l'analyse de maliciels, il est possible d'obtenir un état initial qui se définit par l'état de la plateforme au moment qui précède le déploiement de l'échantillon à analyser. Pour l'ADMV, cela signifie qu'il est possible d'obtenir une acquisition à un temps t_{pre} où l'application à analyser n'a pas encore été déployée et que le système fonctionne normalement. Après l'acquisition de cette capture initiale, l'application est déployée. Afin d'augmenter les chances qu'elle se révèle malicieuse si tel est le cas, elle peut être stimulée manuellement par

les actions d'un analyste, automatiquement par des outils simulant un utilisateur ou par des évènements divers survenant sur la plateforme (p. ex. réception de message texte, de courriel, délais, etc.). Une seconde acquisition est alors effectuée à un temps post-déploiement t_{post} où les artefacts de l'exécution de l'application sont possiblement encore présents dans la mémoire vive.

Afin d'effectuer la comparaison entre les deux temps de mesure, il est proposé de comparer les informations extraites à l'aide des modules visant Linux de *Volatility*. Tel qu'il est décrit précédemment, *Volatility* permet d'extraire des informations sur les structures internes du SE à partir de la mémoire vive. La comparaison des résultats obtenus par l'outil permet d'établir clairement quelles composantes du SE sont d'intérêt pour l'analyse. Il est postulé que les nouvelles entrées sont des éléments ayant été créés pendant l'exécution de l'application. Les éléments présents avant et après le déploiement de cette dernière sont considérés pour l'analyse s'ils ont été modifiés dans l'intervalle d'acquisition. Finalement, les éléments présents au temps t_{pre} et absent au temps t_{post} sont classés comme ayant été supprimés. Cette approche ne permet pas de capturer les artefacts ajoutés à la mémoire vive après le temps t_{pre} et retirés avant t_{post}.

L'ADMV a également été explorée par 504ensics Labs [2] en visant principalement les greffons Windows de *Volatility*. Leur approche est démontrée par l'outil Differential Analysis of Malware in Memory (DAMM) qui vise à automatiser la différenciation entre les captures de mémoire vive. Ce programme a été étudié dans les présents travaux et des efforts ont été menés afin de l'adapter aux greffons pour Linux de *Volatility*. Toutefois, le développement de l'outil a été abandonné par son créateur et les efforts pour le maintenir ont été interrompus. Conséquemment, DAMM n'a pas été retenu dans les présents travaux bien qu'il en démontre l'intérêt de l'approche.

Dans l'état actuel de l'approche proposée, un analyste est requis afin de filtrer les résultats de l'ADMV. En effet, le différentiel obtenu comprend tous changements apportés par l'application analysée, autant ceux apportés par la cible de l'analyse que ceux dus au fonctionnement normal du SE se produisant pendant la durée de vie de l'analyse. L'étude décrite à la section suivante a été réalisée afin d'évaluer si l'ADMV permet de cerner correctement les comportements malicieux d'une application en suivant les principes énoncés.

4.2 Étude de cas

L'approche de l'ADMV a reçu peu d'attention dans la littérature. Sur Android, il n'a pas été possible d'identifier des travaux s'étant penchés sur sa pertinence et son utilisation en tant qu'outil d'analyse de maliciels. C'est pour cette raison que l'étude suivante s'est intéressée à démontrer l'applicabilité de l'ADMV dans l'identification de comportements malicieux de même que sa résilience aux tactiques d'évasion de la détection comme l'obfuscation, le

chargement dynamique de code malicieux et l'élévation de privilège sur une version d'Android utilisant ART. La version 5.1 a été sélectionnée à cet effet.

Plus spécifiquement, l'expérimentation décrite vise à répondre aux questions suivantes : est-ce possible d'utiliser l'ADMV pour identifier, récupérer et analyser le contenu d'un fichier de code malicieux déchiffré, chargé dynamiquement et supprimé dès la fin de son chargement afin d'empêcher sa récupération sur le stockage physique de l'appareil ? Également, est-il possible par cette même technique d'identifier une élévation de privilèges et un cheval de Troie (ou porte dérobée) ?

Dans un premier temps, les aspects méthodologiques de l'étude sont présentés, suivis, dans un deuxième temps, des résultats obtenus. Dans un troisième temps, une discussion présente l'intérêt de ces résultats. Finalement, les limitations de l'étude ainsi que les pistes de recherche futures sont présentées.

4.3 Méthodologie

Un maliciel a été spécifiquement conçu pour manifester les comportements suivants visés par l'étude soit :

- l'obfuscation de ressources (code) malicieuses (du code malicieux dans le cas présent) ;
- le déchiffrement et chargement dynamique de code externe ;
- la suppression des fichiers intermédiaires ;
- l'exécution de ressources binaires externes ;
- l'élévation de privilèges et
- la connexion à distance à un serveur de commande et contrôle.

Ce maliciel expérimental couvre plusieurs problématiques de l'analyse de maliciels qui peuvent représenter un défi pour les méthodes d'analyse statique et d'analyse dynamique décrites au chapitre 2. Le maliciel en apparence bénin est déployé sur un téléphone intelligent Android muni d'une capacité d'acquisition de la mémoire vive afin de capturer l'activité suspecte.

L'appareil utilisé pour le déploiement de l'application est un téléphone Nexus 4 de LG avec la version 5.1 du système d'exploitation. Ce dernier a été recompilé pour supporter le chargement du module LiME selon la méthode présentée à la section 3.1.1. Le module est préchargé sur l'appareil avant le scénario. Également, pour le bien de la démonstration, le noyau a été modifié afin d'être vulnérable à l'élévation de privilèges utiliser dans l'exploit *towelroot* introduit par Hotz [63]. La révision du noyau utilisée correspond à une variante du noyau Linux, appelée Mako, et adaptée au Nexus 4. La version du code utilisée se situe sur la branche Git appelée *origin/android-msm-mako-3.4-lollipop-mr1*. La seule exception est le fichier *kernel/futex.c* qui est à une version antérieure, soit à la révision identifiée par l'identifiant unique Git *7e9543a0ebde3c1df0108523a3e9e152e254f962*. Cette dernière précède le correctif appliqué pour

protéger des exploits utilisant la même technique que *towelroot*. Cette modification a pour effet de retourner le noyau Linux à une version vulnérable tout en restant compatible avec la version 5.1 d'Android. Le fonctionnement de cet exploit est détaillé par Zatuchna sur *Nativeflow* [134–136]. La version utilisée dans le cadre de cette étude est celle de Timwr [120]. Finalement, l'utilisateur *root* est activé à l'aide du correctif SuperSU conçu par Chainfire [22] afin de permettre l'acquisition par LiME. Toutefois, l'utilisation de ces privilèges n'est limitée qu'à l'invite de commande accessible par l'interface de déverminage ADB et utilisée pour lancer l'acquisition.

L'application Android utilisée est une adaptation de celle proposée par Timrae [119]. Dans sa version d'origine, elle démontre le chargement dynamique d'un fichier de type Java Archive (JAR), une archive Java contenant un fichier Android « classes.dex » renfermant du code pouvant être chargé par une application. À l'origine, la version de Timrae n'est pas malicieuse. Pour l'étude, elle a été adaptée afin que le contenu du JAR soit chiffré et puisse être déchiffré à la volée avant d'être chargé dynamiquement. La figure 4.1 présente l'interface de l'application malicieuse. La figure 4.2 illustre la notification produit par le chargement dynamique.

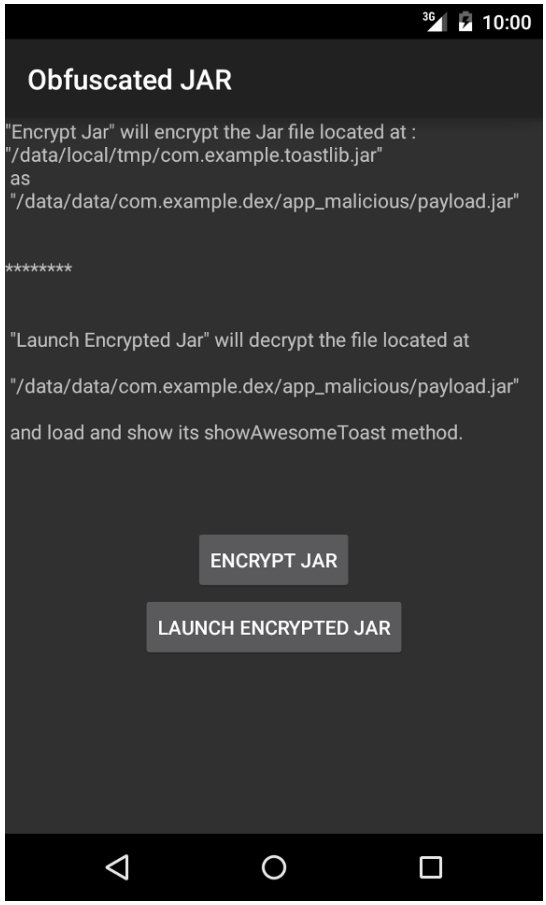

FIGURE 4.1 – Écran d'accueil de l'application.

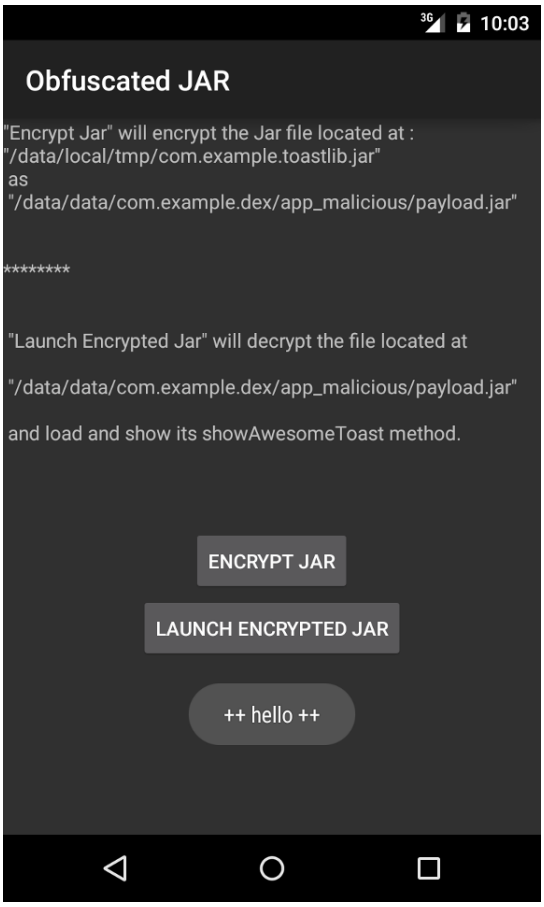

FIGURE 4.2 – Notification résultant du chargement dynamique de la ressource chiffrée.

D'autres modifications ont été apportées afin de représenter des comportements malicieux et d'évasion de la détection. Notamment, le programme récupère 3 fichiers depuis une source externe. Ces 3 fichiers sont le JAR contenant le code malicieux à exécuter, l'exploit d'élévation de privilège sous la forme de l'exécutable *towelroot* et l'utilitaire Linux *busybox*, comportant plusieurs programmes élémentaires n'ayant pas été inclus dans la version du noyau Linux d'Android. Pour simplifier le déploiement, l'emplacement d'où ces fichiers sont récupérés est, dans le cadre de cette étude, le dossier présent sur les systèmes Android à l'emplacement « */data/local/tmp* ». Ils sont chargés manuellement par l'expérimentateur avant l'exécution du scénario. Dans un contexte réel de maliciel, ces fichiers seraient probablement récupérés sur un serveur distant pour réduire le risque de détection au moment de l'installation. Ce choix méthodologique n'affecte pas la pertinence de l'étude puisque le chargement de ces ressources depuis un emplacement local correspondrait à l'étape suivant le téléchargement depuis un serveur distant.

Le flot d'exécution suit le modèle ci-dessous :

1. Le bouton « Encrypt JAR » permet à l'application de faire une copie chiffrée des 3

fichiers dans un espace de stockage réservé pour ses besoins uniquement, soit un dossier local situé à l'emplacement « */data/data/com.example.dex/* ». Le contenu est placé dans le sous-dossier « *app_malicious* ». Cette opération simule le téléchargement des fichiers chiffrés sur l'appareil et n'est effectuée que pour la configuration après l'installation du maliciel. Son principal but est de simplifier la manipulation par l'expérimentateur. Les fichiers sont chiffrés par le passage d'une opération binaire de type ou-exclusif (XOR) avec une clef correspondant à $0xAA$ en base hexadécimale. Cette opération n'est pas adaptée pour protéger le contenu des données, mais elle sert à illustrer avec simplicité qu'il est possible de chiffrer et déchiffrer du code à la volée.

2. Le bouton « Launch Encrypted JAR » est utilisé pour lancer le comportement malicieux. Lorsqu'il est appuyé, une méthode charge les fichiers chiffrés acheminés dans le dossier local à l'application, les déchiffre par le passage d'un XOR avec la même clef hexadécimale $0xAA$ et charge dynamiquement le contenu du JAR. Ce dernier effectue le comportement suivant.
   a) Il affiche la notification avec le message « ++ hello ++ » tel qu'illustré à la figure 4.2 .
   b) Cette étape correspond au moment où le comportement est jugé malicieux. Le maliciel lance l'exploit *towelroot* qui est utilisé pour obtenir les accès *root*. Avec ces privilèges, *busybox* est utilisé pour établir un serveur *netcat* à l'écoute sur le port 5550. Ce dernier relaye toutes les commandes reçues à une invite de commande *sh* lancée sur l'appareil Android avec les permissions *root* et retransmet au client distant tous les messages générés par les commandes envoyées. D'un point de vue pratique, le comportement décrit s'apparente à celui d'un cheval de Troie ayant les privilèges de l'utilisateur *root* sur l'appareil.
   c) Une fois le cheval de Troie déployé, le code chargé dynamiquement procède à la suppression de tous les fichiers déchiffrés ainsi que les fichiers intermédiaires engendrés par le chargement dynamique du JAR. Cette étape a pour but d'en empêcher la récupération par un analyste depuis le stockage de l'appareil.
   d) En arrière-plan, le cheval de Troie est libre de poursuivre son exécution puisque les fichiers nécessaires à son fonctionnement demeurent chargés en mémoire tant que l'application s'exécute.

L'acquisition de la mémoire vive est effectuée avec LiME. L'exécution d'une analyse débute après avoir chiffré les ressources à l'aide du bouton « Encrypt JAR » et se déroule de la façon suivante :

1. L'appareil est redémarré afin de vider le contenu de la mémoire vive et d'éviter le bruit dans l'acquisition. Rien d'autre n'est effectué avant que la plateforme et tous ses services ne soient démarrés.

2. L'application *Obfuscated JAR* est lancée.
3. La première acquisition de la mémoire vive est réalisée avec LiME.
4. Lorsque l'acquisition est complétée, l'utilisateur appuie sur « Launch Encrypted JAR ».
5. À l'aide d'un client *netcat*, une connexion est établie entre l'appareil et l'attaquant.
6. Une seconde acquisition de la mémoire vive est réalisée avec LiME.
7. À la fin de l'acquisition, l'expérimentation prend fin.

Volatility est utilisé pour analyser la capture. Il vise à récupérer les fichiers intermédiaires non chiffrés produits par le chargement dynamique et ayant été supprimé du stockage physique pendant l'analyse. Il est utilisé dans le but de récupérer toutes traces de l'exécution du cheval de Troie. Pour ce faire, une sélection de greffons *Volatility* visant le noyau Linux est retenue pour leur capacité à cibler les comportements malicieux en cause dans l'étude. Ces modules sont listés ci-dessous avec une justification de leur utilisation.

LINUX_PSTREE liste les processus en exécution au moment de l'acquisition ainsi que les relations parent-enfant entre ceux-ci. Une contribution au module proposé par Lebel et Ouellet [75] permet de récupérer également l'ensemble des informations contenues dans la structure *thread_info* et permet de lire les champs de l'identifiant du processus parent (Parent Process Id, PPID), l'identifiant d'utilisateur (User Id, UID), l'identifiant de groupe (Group Id, GID) et l'identifiant d'utilisateur effectif (Effective User Id, EUID). Ces informations permettent de savoir si de nouveaux processus sont créés pendant le temps d'analyse. Elles permettent également de valider si certains processus, nouveaux ou déjà présents, ont obtenu des privilèges *root*. Pour ce faire, les valeurs des champs UID, GID et EUID sont observées. Si la valeur de ceux-ci est 0, cela signifie que le processus s'exécute avec ces privilèges et peut constituer un marqueur de compromission.

LINUX_THREADS permet de répertorier l'ensemble des fils d'exécution présents au moment de la capture, leur processus d'appartenance, ainsi que les valeurs de UID, GID et EUID rattachées. Il est possible de connaître les fils d'exécution créés par une application et de voir toutes incongruités dans les privilèges de ceux-ci.

LINUX_PSAUX liste également les processus en exécution à l'instar de LINUX_PSTREE. Il fournit également l'ensemble des arguments utilisés pour lancer le processus. L'utilité de ce module est donc de pouvoir récupérer les commandes suspectes utilisées dans le lancement d'un processus. Cela permet de capturer les comportements de maliciels, comme celui utilisé dans l'étude, qui ont recours à l'invite de commande pour exécuter certaines actions comme démarrer un serveur pour recevoir les commandes distantes.

LINUX_ELFS liste l'ensemble des fichiers Linux exécutables (Executable Linux File, ELF) chargés en mémoire vive au moment de la capture. Ces fichiers contiennent des instructions binaires pouvant être exécutées directement par le SE ou utilisées à titre de ressources partagées par d'autres applications. Si des fichiers de type ELF sont chargés pendant l'analyse, il s'agit d'un indicateur d'utilisation de code natif dans le contexte de l'analyse. Plus précisément, cela peut indiquer que l'application utilise des bibliothèques logicielles natives du SE ou encore celles d'autres tiers. Ce type de comportement est particulièrement d'intérêt puisqu'il constitue un angle mort de plusieurs analyses recensées au chapitre 2.

LINUX_ARP extrait l'état de la table de routage au moment de la capture. Il est utilisé afin d'observer si les entrées ont changé dans cette table et inférer la présence de nouvelles connexions pendant l'exécution. De cette façon, il est possible d'établir si des connexions ont été effectuées pendant la durée de vie de l'analyse et de valider si celles-ci mènent à une source malicieuse. Ce module est utilisé en remplacement de LINUX_NETSTAT qui vise spécifiquement à récupérer les connexions réseau existantes dans le SE au moment de la capture. En effet, lors de tests préliminaires, il s'est avéré que ce dernier ne fonctionnait pas correctement sur Android et ne retournait aucune information lors de son exécution.

LINUX_FIND_FILE liste l'ensemble des fichiers ouverts par le système d'exploitation (SE) au moment de la capture. Cela permet de vérifier quels sont les fichiers utilisés par les processus, de valider si certains fichiers d'intérêt (p. ex. fichier ELF ou DEX) sont présents en mémoire vive et, le cas échéant, de les extraire dans leur intégralité pour analyse approfondie.

Il est postulé que l'ensemble de ces greffons permettent de reconstituer l'action malicieuse de l'application et de récupérer des artefacts de son exécution, même si ceux-ci sont supprimés de l'espace de stockage afin d'éviter la détection. Les résultats sont présentés à la section suivante.

4.4 Résultats

Deux fichiers contenant l'entièreté de la mémoire vive ont été recueillis, l'un avant l'opération de déchiffrement et du chargement dynamique et l'autre après. Les greffons *Volatility* sélectionnés ont été exécutés afin d'extraire les traces de l'exécution du maliciel. Les résultats obtenus pour l'analyse sur les captures au temps t_{pre} et t_{post} sont comparés afin d'extraire les éléments qui diffèrent. Les résultats de ces comparaisons sont présentés par modules exécutés ci-après.

LINUX_PSTREE : Les éléments d'intérêt résultant du différentiel entre les deux moments d'acquisition sont présentés au listage 4.1.

```
1 Offset          Name              PID   PPID  UID   GID   EUID
2 0xe5306000      init              1     0     0     0     0
3 0xec3ee800      com.example.dex   2849  203   10087 10087 10087
4 0xec8fdc00      towelroot-dec     3451  2849  0     0     0
5 0xec917c00      towelroot-dec     3465  0     0     0     0
```

Listing 4.1 – Résultats obtenus par l'analyse des différences entre les captures pré et post exécution du maliciel tel que rapporté par le module *linux_pstree*.

Sur la plateforme Linux, un PPID de valeur 0 est attribué au premier processus (*init*) et tous les autres sont des enfants directs (relation parent-enfant) ou indirects (plusieurs relations parent-enfant imbriquées) de celui-ci. Or, il apparaît que le processus *towelroot-dec* avec le PID 3465 ne respecte pas cette règle. Également, il s'exécute sous l'UID, le GID et l'EUID 0, correspondant à l'utilisateur *root* et les privilèges qui s'y rattachent. Le processus 3451 s'exécute lui aussi sous cet utilisateur. Ces éléments soulignent le caractère anormal de ce processus puisque sur un SE d'Android non modifié, l'utilisateur *root* n'est utilisé que pour le processus *init* ($PID == 1$), *kthreadd*($PID == 2$) et les enfants directs de ceux-ci ($PPID == [1,2]$). À noter qu'il se peut que des modifications du SE par l'utilisateur ou le fabricant créent des exceptions à cette règle. Dans ces cas, elle peut être adaptée à la plateforme en question pour y inclure ces exceptions légitimes. La sortie brute en annexe F de la capture *a priori* rapporte l'ensemble des processus s'exécutant sur la plateforme où il est possible d'observer cette relation. Dans le cas de la présente étude, comme il est expliqué par Zatuchna [134, 135, 136], l'exploit *towelroot* utilise une chaîne de vulnérabilités afin d'arriver à altérer les propriétés PID, PPID, UID, GID et EUID de son propre processus directement dans la mémoire vive lui permettant de s'élever en tant qu'utilisateur *root*. Cette modification viole la relation décrite et est vue comme une anomalie.

LINUX_THREADS : Les listages 4.2 4.3 et 4.4 rapportent les lignes qui diffèrent entre les deux temps d'analyse et qui sont pertinentes à l'analyse. Les listages 4.2 et 4.3 rapportent principalement des résultats normalement attendus de toutes applications Android lors de leur exécution. Il s'agit des fils d'exécution que lance la plateforme à l'exécution d'une application. Le listage 4.4 quant à lui dénote une anomalie. En effet, les valeurs 0 dans les champs *uid_cred*, *gid_cred* et *euid_cred* démontrent que le nouveau processus *towelroot-dec* détient des fils d'exécution sous l'utilisateur *root*.

```
1  Offset  NameProc  TGID  ThreadPid  thread_offset  Addr_limit  uid  gid  euid
       ↪ ThreadName
2  [...]
3  0xec3ee800  com.example.dex  2849  2849  0xec3ee800  0x00000058  10087  10087  10087
       ↪ com.example.dex
4  0xec3ee800  com.example.dex  2849  2853  0xec3edc00  0x00000008  10087  10087  10087
       ↪ Heap thread poo
5  0xec3ee800  com.example.dex  2849  2854  0xec3ef800  0x00000283  10087  10087  10087
       ↪ Heap thread poo
6  0xec3ee800  com.example.dex  2849  2855  0xec826800  0xe02cf8d0  10087  10087  10087
       ↪ Heap thread poo
7  0xec3ee800  com.example.dex  2849  2861  0xebc5cc00  0x12c0cea0  10087  10087  10087
       ↪ Signal Catcher
8  0xec3ee800  com.example.dex  2849  2862  0xed0bf000  0x0efe0ada  10087  10087  10087
       ↪ JDWP
9  0xec3ee800  com.example.dex  2849  2863  0xecd1dc00  0xf8d868ad  10087  10087  10087
       ↪ ReferenceQueueD
10 0xec3ee800  com.example.dex  2849  2864  0xecd1d800  0x465a4651  10087  10087  10087
       ↪ FinalizerDaemon
11 0xec3ee800  com.example.dex  2849  2865  0xecd1d400  0x1c3970f4  10087  10087  10087
       ↪ FinalizerWatchd
12 0xec3ee800  com.example.dex  2849  2866  0xecd1d000  0x3038f24a  10087  10087  10087
       ↪ HeapTrimmerDaem
13 0xec3ee800  com.example.dex  2849  2867  0xecd1cc00  0x1c10e140  10087  10087  10087
       ↪ GCDaemon
14 0xec3ee800  com.example.dex  2849  2868  0xecd1c400  0x47f0004e  10087  10087  10087
       ↪ Binder_1
15 0xec3ee800  com.example.dex  2849  2869  0xebc5c800  0x0000008a  10087  10087  10087
       ↪ Binder_2
16 [...]
```

Listing 4.2 – Résultats obtenus par l'analyse des différences entre les captures pré et post exécution du maliciel tel que rapporté par le module *linux_threads*.

```
Offset  NameProc  TGID  ThreadPid  thread_offset  Addr_limit  uid  gid  euid
    ↪ ThreadName
[...]
0xec3ee800  com.example.dex  2849  2877  0xecd1e800  0x1c48f643  10087  10087  10087
    ↪ Thread-268
0xec3ee800  com.example.dex  2849  2880  0xec896800  0x00310039  10087  10087  10087
    ↪ RenderThread
0xec3ee800  com.example.dex  2849  2883  0xec895400  0x12de0640  10087  10087  10087
    ↪ GL updater
0xec3ee800  com.example.dex  2849  2885  0xec895800  0x12c5d4c0  10087  10087  10087
    ↪ hwuiTask1
0xec3ee800  com.example.dex  2849  2886  0xec895c00  0x13e2cb20  10087  10087  10087
    ↪ hwuiTask2
0xec3ee800  com.example.dex  2849  2888  0xeb78c000  0x00001136  10087  10087  10087
    ↪ Binder_3
0xec3ee800  com.example.dex  2849  3441  0xebb4a400  0xf8d03a10  10087  10087  10087
    ↪ AsyncTask #1
0xec3ee800  com.example.dex  2849  3448  0xec914400  0x6841200c  10087  10087  10087
    ↪ .ProcessManager
0xec3ee800  com.example.dex  2849  3469  0xec914c00  0x000080e0  10087  10087  10087
    ↪ RenderThread
0xec3ee800  com.example.dex  2849  3472  0xecdbb400  0x00333ca8  10087  10087  10087
    ↪ RenderThread
0xec3ee800  com.example.dex  2849  3473  0xecdb8800  0x7c7af44f  10087  10087  10087
    ↪ RenderThread
0xec3ee800  com.example.dex  2849  3474  0xecdb9c00  0x1c3970e3  10087  10087  10087
    ↪ RenderThread
0xec3ee800  com.example.dex  2849  3475  0xeee49000  0x00000000  10087  10087  10087
    ↪ RenderThread
[...]
```

Listing 4.3 – (Suite...) Résultats obtenus par l'analyse des différences entre les captures pré et post exécution du maliciel tel que rapporté par le module *linux_threads*.

```
Offset  NameProc  TGID  ThreadPid  thread_offset  Addr_limit  uid  gid  euid  ThreadName
0xec8fdc00  towelroot-dec  3451  3451  0xec8fdc00  0xc000f8dc  0  0  0  towelroot-dec
0xec8fdc00  towelroot-dec  3451  3452  0xec8ffc00  0x45560a01  0  0  0  towelroot-dec
0xec8fdc00  towelroot-dec  3451  3453  0xec8fe000  0x4680b091  0  0  0  towelroot-dec
0xec8fdc00  towelroot-dec  3451  3454  0xec8fc000  0xf8d00124  0  0  0  towelroot-dec
0xec8fdc00  towelroot-dec  3451  3455  0xec917000  0xf8d9e7bd  0  0  0  towelroot-dec
0xec8fdc00  towelroot-dec  3451  3456  0xec8ff800  0x0066f2c7  0  0  0  towelroot-dec
0xec8fdc00  towelroot-dec  3451  3457  0xec8ff400  0x000cf850  0  0  0  towelroot-dec
0xec8fdc00  towelroot-dec  3451  3458  0xec8fc400  0x1cacf64c  0  0  0  towelroot-dec
0xec8fdc00  towelroot-dec  3451  3459  0xec4b1800  0x9e9fde6c  0  0  0  towelroot-dec
0xec8fdc00  towelroot-dec  3451  0001  0xec4b1c00  0xbc0c2b78  0  0  0  towelroot-dec
0xec917c00  towelroot-dec  3465  3465  0xec917c00  0xe1b4f8d9  0  0  0  towelroot-dec
```

Listing 4.4 – (Suite...) Résultats obtenus par l'analyse des différences entre les captures pré et post exécution du maliciel tel que rapporté par le module *linux_threads*.

LINUX_PSAUX : La comparaison réalisée a permis d'extraire le résultat présenté au listage 4.5.

```
PID     UID     GID     Arguments
3451    0       0       /data/data/com.example.dex/app_decrypted/towelroot-dec "/data/data/com.
    ↪ example.dex/app_decrypted/busybox-dec nc -l -p 5550 -e /system/bin/sh" &
```

Listing 4.5 – Résultats obtenus par l'analyse des différences entre les captures pré et post exécution du maliciel tel que rapporté par le module *linux_psaux*.

Ce résultat présente la commande lancée par l'exploit. Cette commande lance un serveur *netcat* qui écoute sur le port 5550 dans le but de rediriger les commandes reçues et les sorties qui y sont associées à une invite de commande */system/bin/sh* sur le téléphone.

LINUX_ELFS : Le listage 4.6 rapporte les entrées pertinentes ajoutées par le lancement du comportement malicieux.

```
Pid       Name                Start       End         Elf Path
--------- ------------------- ----------- ----------- -------------------------
3451 towelroot-dec     0xb6efb000 0xb6efe030 /system/lib/libnetd_client.so
3451 towelroot-dec     0xb6f20000 0xb6f827c8 /system/lib/libc.so
3451 towelroot-dec     0xb6f83000 0xb6f9b0b0 /system/lib/libm.so
3451 towelroot-dec     0xb6f9c000 0xb6f9f00c /system/lib/libstdc++.so
3451 towelroot-dec     0xb6fb6000 0xb6fbb864 /data/data/com.example.dex/app_decrypted/
    ↪ towelroot-dec
3465 towelroot-dec     0xb6fb6000 0xb6fbb864 /data/data/com.example.dex/app_decrypted/
    ↪ towelroot-dec
```

Listing 4.6 – Résultats obtenus par l'analyse des différences entre les captures pré et post exécution du cheval de Troie tel que rapporté par le module *linux_elfs*.

Le listage illustre, à nouveau, la présence du processus *towelroot-dec* de même que les fichiers ELF ayant été chargés par son lancement. L'observation de cette sortie permet de conclure que l'exécutable a été lancé à partir d'un dossier réservé à l'application *com.example.dex*, soit le maliciel à l'étude. Il est intéressant de noter que le chargement du fichier ELF */system/lib/-libnetd_client.so* constitue généralement un appel à une fonctionnalité réseau du SE. Dans le cas de *towelroot* [120], l'exploit utilise des *sockets* afin de mener son attaque ce qui explique le chargement de cette bibliothèque logicielle native.

LINUX_ARP : Le listage 4.7 présente les lignes qui diffèrent entre des deux moments d'analyse.

```
192.168.0.21 at 00:e6:1f:70:0f:f5 on wlan0
```

Listing 4.7 – Résultats obtenus par l'analyse des différences entre les captures pré et post exécution du maliciel tel que rapporté par le module *linux_arp*.

La route menant à l'adresse IP 192.168.0.21 avec l'adresse physique 00:e6:1f:70:0f:f5 via l'interface wlan0 correspond à la route établie pour connecter le client distant à l'appareil infecté par le cheval de Troie.

LINUX_FIND_FILE : Le listage 4.8 présente les éléments pertinents à l'analyse et chargés en mémoire pendant l'exécution du comportement malicieux.

```
#Inode    Inode      Chemin d'acces
------ ---------- ---------
[...]
603441 0xed318518 /data/data/com.example.dex/app_decrypted
---    0x00000000 /data/data/com.example.dex/app_decrypted/busybox-dec nc -l -p 5550 -e
603520 0xed3cbe20 /data/data/com.example.dex/app_decrypted/towelroot-dec
---    0x00000000 /data/data/com.example.dex/app_decrypted/decrypted.bad
---    0x00000000 /data/data/com.example.dex/app_decrypted/busybox-dec
603439 0xed40ce38 /data/data/com.example.dex/app_outdex
603521 0xed3cbbd8 /data/data/com.example.dex/app_outdex/decrypted.dex
603437 0xed410088 /data/data/com.example.dex/app_malicious
603444 0xed2d0088 /data/data/com.example.dex/app_malicious/towelroot
603443 0xed3c19a0 /data/data/com.example.dex/app_malicious/payload.jar
603442 0xed3199a0 /data/data/com.example.dex/app_malicious/busyboxenc
[...]
```

Listing 4.8 – Résultats obtenus par l'analyse des différences entre les captures pré et post exécution du maliciel tel que rapporté par le module *linux_find_file*.

Dans ce listage, le *Inode Number* correspond à l'identifiant unique de la structure de données comportant le fichier dans la mémoire vive du système, le champ *Inode* à l'adresse où celui-ci se trouve et le champ *File Path* au chemin d'accès du fichier sur le système au moment du chargement.

L'utilisation du module *linux_find_file* avec l'option « *-i <Addr>* » permet d'extraire un fichier situé à l'adresse *<Addr>*, lorsque celle-ci n'est pas nulle ($0x00$) [1]. Avec cette option et en utilisant les valeurs d'adresse $0xed3cbbd8$ et $0xed3cbe20$, il est possible d'extraire les fichiers *decrypted.dex* et *towelroot-dec* respectivement. Le premier correspond au fichier de code chargé dynamiquement par le maliciel et le second à l'exploit *towelroot*. En mémoire, ces fichiers sont tous les deux décryptés. Il a été possible de confirmer que le fichier récupéré est identique à l'originale en comparant le résultat de leur somme de contrôle MD5 qui s'est révélé identique. En revanche, *busybox-dec* n'a pas été maintenu en mémoire vive dans un format décrypté et est, ainsi, irrécupérable par cette méthode. Ce résultat démontre bien le caractère imprévisible et sensible de l'AMV. Son contenu peut changer sans préavis et altérer la capture obtenue.

4.5 Discussion

L'utilisation de *Volatility* a permis de décomposer le contenu brut des captures de mémoire vive en différentes structures internes du noyau Linux du SE Android. L'utilisation de l'analyse différentielle a permis d'établir le delta entre t_{pre} et t_{post} afin de relever uniquement les changements d'intérêt dans ces structures étant survenus pendant la durée de l'exécution de l'application. Finalement, le triage final de ce différentiel par un analyste a permis d'identifier

1. La valeur nulle indique que le contenu du fichier n'est plus présent en mémoire.

	Comportements	Module(s)	Détection
1.	Obfuscation par le chiffrement des ressources	LINUX_ELFS LINUX_FIND_FILE	Partielle
2.	Chargement dynamique de code externe	LINUX_FIND_FILE	Détecté
3.	Suppression des fichiers intermédiaires	LINUX_FIND_FILE	Partielle
4.	Exécution de ressources binaires externes	LINUX_PSTREE LINUX_THREADS LINUX_PSAUX LINUX_ELFS	Détecté
5.	Élévation de privilèges	LINUX_PSTREE LINUX_THREADS LINUX_PSAUX	Détecté
6.	Connexion à un serveur de commande et contrôle	LINUX_PSAUX LINUX_ARP LINUX_FIND_FILE	Partielle

TABLEAU 4.1 – Comportements malicieux détectés par l'ADMV.

les comportements qui sont liés à l'exécution de l'application par opposition à ceux qui proviennent du fonctionnement normal du SE.

L'approche par l'ADMV a permis d'extraire des marqueurs révélant la présence du maliciel sur la plateforme. Le tableau 4.1 présente les stratégies employées par le maliciel, les greffons ayant permis de détecter le comportement décrit et si les comportements ont pu être détectés.

L'obfuscation par le chiffrement des ressources a pu être en partie détectée. En effet, il a été possible de détecter que des ressources n'appartenant pas à l'application d'origine ont été chargées par celle-ci au moment de l'exécution. Il a été également possible de récupérer certains de ces fichiers depuis la mémoire vive. Les ressources chiffrées n'ont pu être détectées avant d'être chargées par l'application et l'action du déchiffrement lui-même n'est pas captée par l'ADMV.

Le chargement dynamique de code externe à l'application a été détecté. Le code chargé dynamiquement a pu être détecté et récupéré en mémoire vive. Il a également été possible de valider que le contenu du code récupéré correspond au code d'origine.

La suppression des fichiers utilisés pour lancer le comportement malicieux a pu être détectée en partie. Le contenu du fichier déchiffré *busybox-dec* utilisé pour déployer le cheval de Troie n'a pu être récupéré. Cependant, le descripteur du fichier comportant notamment son nom a tout de même pu être relevé. Cela peut être suffisant pour mener à une analyse plus en profondeur de l'application.

L'élévation de privilèges a été détectée dans le contexte de l'application qui lance des opérations avec l'utilisateur *root*.

La connexion à un serveur de commande et contrôle a également été partiellement détectée. Les connexions et les commandes utilisées pour lancer le maliciel ont été clairement identifiées par l'ADMV. Il n'a été possible d'identifier les ports réseau ouverts par l'application qu'en partie. En effet, la commande « busybox−dec nc −l −p 5550 −e /system/bin/sh & » identifiée par le greffon LINUX_PSAUX comporte le numéro de port utilisé, mais ne peut constituer un marqueur fiable pour détecter des communications réseau. Un maliciel pourrait procéder autrement pour ouvrir une connexion distante avec un serveur de commande et contrôle. Dans un contexte où une application malicieuse dissimule ses communications dangereuses au travers de communications légitimes, les ports ouverts sur un appareil et les connexions avec les postes distants qui y sont rattachés faciliteraient le filtrage entre contenu légitime et illégitime. Il serait intéressant d'avoir une version fonctionnelle de LINUX_NETSTAT sur Android afin d'obtenir cette information et d'enrichir l'analyse des communications réseau sur l'appareil.

Également, cette méthode a été démontrée applicable sur appareils physiques en ne demandant qu'une modification mineure du noyau Linux. De ce fait, l'ADMV offre une alternative résiliente aux techniques de détection d'une machine virtuelle ou un émulateur.

4.6 Limites

L'étude menée dans le cadre de cette maîtrise comporte des limites. Tout d'abord, l'approche proposée a été évaluée sur une application conçue pour n'exprimer que le comportement malicieux étudié. Or, il est commun qu'une application exécute différents comportements (p. ex. lecture de média, affichage de contenu publicitaire, etc.) pendant son exécution. Dans un contexte d'AMV, cela peut avoir un impact négatif sur le contenu récupérable et sur la quantité de données inutiles à filtrer pendant la phase d'analyse. Il serait souhaitable d'approfondir la recherche afin d'identifier une approche qui permettrait d'obtenir successivement plusieurs captures de la mémoire vive. De cette façon, il serait possible de garder la trace du contenu qui y transige et de limiter les risques de perdre des marqueurs importants. Considérant le stockage requis pour chacune des captures obtenues dans les présents travaux qui sont égales à la capacité de la mémoire vive de l'appareil analysé (2 Go pour le Nexus 4) et le long temps d'acquisition nécessaire (environ 2 min), la technique est encore trop exigeante pour être utilisée sur des appareils destinés à des utilisateurs.

Une autre limitation des présents travaux est que l'application analysée a été conçue expressément pour l'étude. De ce fait, le comportement malicieux recherché était entièrement connu *a priori*. Dans une situation réelle, le concepteur du maliciel et l'analyste sont typiquement des entités différentes. Ce dernier doit avoir une compréhension du fonctionnement du SE afin d'être en mesure d'inférer s'il y a des activités suspectes à partir du résultat des différences entre les captures de la mémoire vive.

Dans l'optique d'améliorer l'efficacité de la technique, le devis a été créé en cherchant à limiter les comportements à risque d'interférer avec l'analyse, comme l'action d'autres applications. Cette décision méthodologie a pour effet de minimiser la corruption des artefacts d'intérêt en mémoire, mais également de minimiser la validité écologique de l'étude. Toutefois, dans un contexte d'un laboratoire d'analyse de maliciels, cette décision est justifiée, souhaitable et applicable.

4.7 Travaux futurs

Au final, l'AMV par l'utilisation d'un différentiel entre des captures aux temps t_{pre} et t_{post} est une approche complémentaire aux autres approches d'analyses dynamiques et statiques. Néanmoins, plusieurs pistes de recherche sont souhaitables pour l'expansion de ce domaine prometteur. Notamment, il est important que les travaux futurs s'intéressent à réduire le temps requis afin d'effectuer une capture intégrale de la mémoire vive sur l'appareil. Pour ce faire, il est nécessaire de développer des techniques d'acquisitions qui seraient capables, tout comme LiME, de faire une capture complète de la mémoire vive, mais également de cibler spécifiquement des processus d'intérêt. Cela aurait pour effet de réduire la quantité de mémoire à parcourir pour chaque capture. Une telle technique favoriserait des temps d'acquisition plus courts et limiterait le risque de manquer un comportement d'intérêt. De plus, cela permettrait d'effectuer des captures fréquentes afin de profiler les interactions d'une application avec son espace mémoire alloué pour détecter d'éventuelles anomalies en cas d'exploit. Il serait également intéressant d'automatiser le triage des résultats de l'ADMV afin d'isoler automatiquement les changements qui sont uniquement imputables à l'échantillon analysé plutôt que de nécessiter une inspection manuelle par un analyste.

Un autre axe de recherche pour ce domaine serait d'identifier une technique permettant d'injecter un module de capture de la mémoire vive dans la plateforme Android sans nécessiter de remplacer son noyau Linux au préalable. L'étude de Stüttgen et Cohen [111], dans laquelle un processus d'acquisition est injecté à même un LKM déjà présent en mémoire, a tenté de répondre à cette problématique. Toutefois, l'approche proposée dans leur étude nécessiterait tout de même de recompiler un noyau pour activer le support des LKM en plus de devoir être adaptée à Android. Dans le même but, l'approche par l'ajout à Android de modules de sécurité proposée par Heuser *et al.* [61] pourrait offrir des fonctionnalités permettant d'acquérir la mémoire vive de façon légitime. L'approche n'ayant pas reçu d'appui par Google dans le code source d'origine du SE, cette solution a reçu peu d'attention par les principaux fabricants d'appareils intelligents.

Résumé

Les résultats obtenus dans ce chapitre démontrent qu'il est possible d'utiliser les greffons de *Volatility* plus génériques (c.-à-d. qui ciblent le noyau Linux en général plutôt que de s'attarder aux structures spécifiques à Android) pour détecter des activités malicieuses sur un appareil physique, et ce, même si le maliciel tente d'éviter la détection. Il est donc pertinent de poursuivre la recherche dans le domaine de l'AMV afin d'élaborer de nouveaux outils d'analyse plus flexibles que ceux présentés dans les travaux précédents, plus efficaces que ceux utilisés dans l'étude de cas et plus spécialisés sur le SE Android afin d'obtenir des résultats plus détaillés. La proposition de l'ADMV dans un contexte d'analyse de maliciels est un pas en ce sens.

Conclusion

Le présent mémoire s'est intéressé à l'analyse de la mémoire vive dans un contexte d'analyse de maliciels sur Android. Le chapitre 1 a détaillé le fonctionnement global d'Android afin de mettre en lumière les particularités du système d'exploitation. Ces connaissances ont permis de comprendre les spécificités des maliciels sur la plateforme et les techniques d'analyse au chapitre 2. Cette revue de la littérature a permis de comprendre le fonctionnement des différentes approches d'analyse statique et dynamique proposées dans la littérature pour la plateforme. Elle a également identifié les principaux défis à relever afin de résister aux stratégies d'attaques des concepteurs de maliciels. Plus spécifiquement, il a été démontré que les méthodes existantes sont pour la plupart vulnérables à l'une ou l'autre de techniques d'évasion de la détection comme l'obfuscation du code, la détection de l'environnement d'analyse, l'exécution de code natif malicieux ou encore le chargement dynamique du code. Afin de combler ces lacunes, le chapitre 3 a proposé de s'intéresser à l'analyse de la mémoire vive (AMV) dans un contexte d'analyse de maliciels et adapté à Android. L'exposé de la littérature présenté à ce chapitre a démontré l'existence de techniques permettant d'extraire la mémoire vive sur la plateforme mobile, que ce soit dans un contexte de système virtuel ou encore sur un appareil physique. Cette dernière configuration est particulièrement d'intérêt puisqu'elle permet d'analyser des maliciels qui inhibent leur comportement en présence de marqueurs d'un environnement virtuel. Les techniques permettant d'analyser les captures de la mémoire vive sur Android ont également été présentées à ce chapitre. Cette exploration a permis de présenter l'outil *Volatility* qui permet d'automatiser la récupération de données présentes dans la mémoire vive. Pour ce faire, l'outil utilise des modules d'analyse spécifiques permettant l'identification et la reconstruction des structures de données visées à partir d'un fichier de capture. Il a été constaté que peu d'approches actuelles sont fonctionnelles ou adaptées aux versions du SE supérieure à 5.0. En effet, les variations dans la structure interne d'Android imputables aux mises à jour fréquentes de la plateforme rendent complexe la conception d'un module d'analyse de la mémoire vive visant des structures spécifiques à Android. Afin d'extraire ces éléments de la mémoire vive, il est requis d'en connaître précisément leur contenu et leur structure afin de pouvoir les reconnaître dans la capture. Or, si ces éléments changent ou sont retirés du SE (p. ex. la MVD), le module d'analyse peut devenir obsolète.

Le chapitre 4 a démontré l'applicabilité de l'AMV dans un contexte d'analyse de maliciels

afin d'extraire des indices de compromission survenant après l'installation d'une application suspicieuse. L'étude de cas qui y est présenté propose une approche d'analyse de maliciels complémentaire aux méthodes dynamiques et statiques existantes, mais plus facile à maintenir que les approches d'AMV existantes dans la littérature. En effet, cette approche repose sur l'utilisation de greffons *Volatility* visant l'ensemble des plateformes ayant un noyau Linux plutôt que de viser spécifiquement Android. Dans le cadre de l'étude, une application dite malicieuse a été développée à titre de preuve de concept. Pour en faire la détection, l'approche utilisant l'analyse différentielle de la mémoire vive (ADMV) a été proposée et détaillée. Les résultats de l'expérience ont révélé que l'ADMV permet de capturer des comportements comme le déploiement dynamique de code malicieux obfusqué et même, dans certains cas, de détecter des attaques sérieuses comme l'élévation de privilèges. Cela révèle bien la pertinence de l'approche et atteint l'objectif initial qui visait à démontrer que l'ADMV est une technique d'analyse de maliciels capable de capturer des comportements malicieux sur cette plateforme. Elle se distingue particulièrement par sa capacité à pouvoir prendre ses données d'analyse directement à partir d'un appareil physique plutôt que de dépendre obligatoirement d'une infrastructure virtuelle ou d'un SE dont le fonctionnement interne a été altéré considérablement pour y inclure des capacités d'analyse. Elle permet aussi une visibilité sur l'ensemble du SE et non pas seulement sur ce qui s'exécute dans le contexte d'une application.

Bien que l'ADMV soit une approche d'intérêt, l'analyse manuelle des résultats demeure fastidieuse dans le cas où plusieurs maliciels doivent être analysés massivement. Plusieurs pistes de recherches sont possibles pour alléger ce traitement. L'une des approches proposées serait d'automatiser l'inférence des comportements à partir des données extraites par l'ADMV à l'aide de technique comme l'apprentissage automatique. Ce type d'approche permet d'entraîner un modèle à extraire les caractéristiques pertinentes dans un jeu de données et d'utiliser ces dernières pour classer les données selon des catégories connues. Le défi est d'avoir un ensemble de données suffisamment structuré pour permettre ce type de traitement et de connaître la valeur réelle de ces catégories (p. ex. malicieuses ou non). Or, dans le domaine de l'analyse de maliciels, ces catégories ne sont pas toujours clairement définies. Par exemple, il est possible qu'une application jugée « saine » comporte un morceau de code malicieux qui a échappé à la détection ou qu'elle puisse être exploitée par un attaquant. Il se peut également qu'une application « malicieuse » soit catégorisée ainsi parce qu'elle tente de valider la sécurité d'un appareil en exerçant des tests de vulnérabilités sur celle-ci. Une autre approche serait d'outiller les analystes avec des techniques de visualisation de données capables de rendre l'exploration d'une capture de mémoire vive plus intuitive (p. ex. [100]).

La complexité requise pour déployer un environnement permettant l'ADMV sur Android demeure un obstacle important de l'approche. En effet, la taxe imposée par la modification, la recompilation et le déploiement du noyau Linux impose une charge de travail additionnelle pour un analyste qui désire recourir à cette méthode. De plus, elle ne peut pas être utilisée

à la pièce sur un appareil intelligent d'origine. En effet, l'appareil doit être réinitialisé pour activer les fonctionnalités nécessaires à la capture de la mémoire vive. Elle demeure toutefois une source importante d'informations sur l'état interne de l'appareil et peut être un outil d'analyse efficace dans un contexte de laboratoire. Il reste que l'extraction précise et complète des marqueurs de compromission depuis la mémoire vive demeure un défi et n'est pas un problème résolu, même pour des SE où l'AMV est plus mature comme Windows, Linux ou MacOS. C'est pourquoi cette approche, et plus particulièrement, l'ADMV est proposée comme une méthode complémentaire aux autres outils d'analyse. En effet, il apparaît que les concepteurs de maliciels s'adaptent constamment aux mesures visant à identifier les menaces sur Android et que l'approche la plus résiliente à ces attaques est probablement de les combiner si les ressources et le temps disponibles au moment de l'analyse le permettent. La diversité des approches est susceptible de rendre difficile la création d'un maliciel capable de couvrir l'ensemble des méthodes de détection existantes. Il serait intéressant d'explorer dans des travaux futurs comment les avancées en termes d'analyse de maliciels sur Android pourraient être combinées pour améliorer le filtrage des menaces dans les marchés d'applications.

Au final, la plus importante mesure de protection sur Android demeure d'éduquer les utilisateurs sur de meilleures habitudes d'utilisation. Notamment, il importe d'enseigner aux usagers comment reconnaître les signes qu'une application puisse être malicieuse (p. ex. applications de sources inconnues, trop de permissions demandées, mauvaises évaluations des autres utilisateurs, etc.). Également, il est nécessaire d'inculquer l'importance d'appliquer les mises à jour disponibles aux appareils connectés à l'Internet, sans égard à la nature de leur SE respectif. Cette mesure est susceptible de réduire la quantité d'appareils comportant des vulnérabilités pouvant être la cible des concepteurs de maliciels. Malheureusement, bien que beaucoup d'utilisateurs appliquent les mises à jour avec diligence et souhaitent le faire, il reste que les manufacturiers doivent fournir les correctifs lorsqu'une faille est divulguée. À ce jour, beaucoup d'appareils ne sont plus supportés par les manufacturiers, et ce, même si les usagers continuent d'en dépendre. Cette façon de faire des fabricants doit être revue si l'on souhaite contribuer à la sécurisation du cyberespace.

Annexe A

Décompilation de l'application HelloWorld

Par souci de simplicité, seul le résultat de la décompilation de la classe HelloWorld est présenté ci-après. Le code A.1 présente le code source de l'application. Les extraits de code A.2, A.3 et A.4 présentent les résultats obtenus par la décompilation en *bytecode* Dalvik (dex) à l'aide de l'outil *dexdump*, en baksmali[66] tel qu'obtenu par l'utilisation de *apktool*[126] et en Java tel qu'obtenu par l'utilisation de l'outil *jadx*[107] respectivement.

Listing A.1 – Code source (Java)

```
1  package com.example.helloworld.helloworld;
2
3  import android.support.v7.app.AppCompatActivity;
4  import android.os.Bundle;
5
6  public class HelloWorld extends AppCompatActivity {
7
8      @Override
9      protected void onCreate(Bundle savedInstanceState) {
10         super.onCreate(savedInstanceState);
11         setContentView(R.layout.activity_hello_world);
12     }
13 }
```

Listing A.2 – Dalvik *bytecodes* via *dexdump*

```
Class #1436            -
  Class descriptor  : 'Lcom/example/helloworld/helloworld/HelloWorld;'
  Access flags      : 0x0001 (PUBLIC)
  Superclass        : 'Landroid/support/v7/app/AppCompatActivity;'
  Interfaces        -
  Static fields     -
  Instance fields   -
  Direct methods    -
    #0              : (in Lcom/example/helloworld/helloworld/HelloWorld;)
      name          : '<init>'
      type          : '()V'
      access        : 0x10001 (PUBLIC CONSTRUCTOR)
      code          -
      registers     : 1
      ins           : 1
      outs          : 1
      insns size    : 4 16-bit code units
1183c8:                                        |[1183c8] com.example.helloworld.helloworld.
    ↪ HelloWorld.<init>:()V
1183d8: 7010 f327 0000                         |0000: invoke-direct {v0}, Landroid/support/
    ↪ v7/app/AppCompatActivity;.<init>:()V // method@27f3
1183de: 0e00                                   |0003: return-void
      catches       : (none)
      positions     :
        0x0000 line=6
      locals        :
        0x0000 - 0x0004 reg=0 this Lcom/example/helloworld/helloworld/HelloWorld;

  Virtual methods   -
    #0              : (in Lcom/example/helloworld/helloworld/HelloWorld;)
      name          : 'onCreate'
      type          : '(Landroid/os/Bundle;)V'
      access        : 0x0004 (PROTECTED)
      code          -
      registers     : 3
      ins           : 2
      outs          : 2
      insns size    : 10 16-bit code units
1183e0:                                        |[1183e0] com.example.helloworld.helloworld.
    ↪ HelloWorld.onCreate:(Landroid/os/Bundle;)V
1183f0: 6f20 fe27 2100                         |0000: invoke-super {v1, v2}, Landroid/
    ↪ support/v7/app/AppCompatActivity;.onCreate:(Landroid/os/Bundle;)V // method@27fe
1183f6: 1400 1900 047f                         |0003: const v0, #float
    ↪ 1754586025038489865150631306382876344 32.000000 // #7f040019
1183fc: 6e20 a23b 0100                         |0006: invoke-virtual {v1, v0}, Lcom/example/
    ↪ helloworld/helloworld/HelloWorld;.setContentView:(I)V // method@3ba2
118402: 0e00                                   |0009: return-void
      catches       : (none)
      positions     :
        0x0000 line=10
        0x0003 line=11
        0x0009 line=12
      locals        :
        0x0000 - 0x000a reg=1 this Lcom/example/helloworld/helloworld/HelloWorld;
        0x0000 - 0x000a reg=2 savedInstanceState Landroid/os/Bundle;

  source_file_idx   : 1797 (HelloWorld.java)
```

Listing A.3 – Code intermédiaire Baksmali via *apktools*

```
.class public Lcom/example/helloworld/helloworld/HelloWorld;
.super Landroid/support/v7/app/AppCompatActivity;
.source "HelloWorld.java"


# direct methods
.method public constructor <init>()V
    .locals 0

    .prologue
    .line 6
    invoke-direct {p0}, Landroid/support/v7/app/AppCompatActivity;-><init>()V

    return-void
.end method


# virtual methods
.method protected onCreate(Landroid/os/Bundle;)V
    .locals 1
    .param p1, "savedInstanceState"    # Landroid/os/Bundle;

    .prologue
    .line 10
    invoke-super {p0, p1}, Landroid/support/v7/app/AppCompatActivity;->onCreate(Landroid/os/
        ↪ Bundle;)V

    .line 11
    const v0, 0x7f040019

    invoke-virtual {p0, v0}, Lcom/example/helloworld/helloworld/HelloWorld;->setContentView(
        ↪ I)V

    .line 12
    return-void
.end method
```

Listing A.4 – Code Java via *jadx*

```
1  package com.example.helloworld.helloworld;
2
3  import android.os.Bundle;
4  import android.support.v7.app.AppCompatActivity;
5
6  public class HelloWorld extends AppCompatActivity {
7      protected void onCreate(Bundle savedInstanceState) {
8          super.onCreate(savedInstanceState);
9          setContentView((int) R.layout.activity_hello_world);
10     }
11 }
```

Annexe B

Code de l'Application A

Listing B.1 – Activité de A : HelloWorld.java

```
1  package com.example.helloworld.helloworld;
2
3  import android.Manifest;
4  import android.content.pm.PackageManager;
5  import android.support.v4.app.ActivityCompat;
6  import android.support.v4.content.ContextCompat;
7  import android.support.v7.app.AppCompatActivity;
8  import android.os.Bundle;
9  import android.widget.Button;
10 import android.widget.Toast;
11
12
13 public class HelloWorld extends AppCompatActivity {
14
15     protected final int MY_PERMISSIONS_REQUEST_READ_CONTACTS = 0b00001000;
16
17     @Override
18     protected void onCreate(Bundle savedInstanceState) {
19         super.onCreate(savedInstanceState);
20         setContentView(R.layout.activity_hello_world);
21         if (ContextCompat.checkSelfPermission(this,
22                 Manifest.permission.READ_CONTACTS)
23                 != PackageManager.PERMISSION_GRANTED) {
24             if (ActivityCompat.shouldShowRequestPermissionRationale(this,
25                     Manifest.permission.READ_CONTACTS)) {
26                 Toast.makeText(this, "Contacts Permission needed!", Toast.LENGTH_LONG).show
                        ↪ ();
27             }
28             else{
29                 ActivityCompat.requestPermissions(this,
30                         new String[]{Manifest.permission.READ_CONTACTS},
31                         MY_PERMISSIONS_REQUEST_READ_CONTACTS);
32             }
33
34         }
35         else{
36             (findViewById(R.id.exporter)).setOnClickListener(new Ripper(this));
37         }
38
39
40     }
41     @Override
42     public void onRequestPermissionsResult(int requestCode,
43                                            String permissions[], int[] grantResults) {
```

```
44          Button extractor = (Button) findViewById(R.id.exporter);
45          switch (requestCode) {
46              case MY_PERMISSIONS_REQUEST_READ_CONTACTS: {
47                  if (grantResults.length > 0
48                          && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
49                      extractor.setOnClickListener(new Ripper(this));
50                  }
51
52              }
53          }
54      }
55  }
```

Listing B.2 – Activité de A : Ripper.java

```
1  package com.example.helloworld.helloworld;
2
3  import android.app.Activity;
4  import android.content.ContentResolver;
5  import android.database.Cursor;
6  import android.os.Handler;
7  import android.os.Looper;
8  import android.provider.ContactsContract;
9  import android.util.Log;
10 import android.view.View;
11 import android.view.View.OnClickListener;
12 import android.widget.Button;
13
14 import java.util.ArrayDeque;
15 import java.util.ArrayList;
16 import java.util.List;
17 import java.util.Queue;
18 import java.util.concurrent.atomic.AtomicBoolean;
19
20
21 public class Ripper implements OnClickListener {
22     private final Activity mActivity;
23
24     public Ripper(Activity activity) {
25         this.mActivity = activity;
26     }
27
28     @Override
29     public void onClick(View v) {
30         Log.e("OUTPUT", "Outputting!");
31         final ContentResolver cr = v.getContext().getContentResolver();
32         final Cursor cur = cr.query(ContactsContract.Contacts.CONTENT_URI,
33                 null, null, null, null);
34         if (cur.getCount() > 0) {
35             final Button exportButton = (Button) this.mActivity.findViewById(R.id.exporter);
36             final AtomicBoolean sigkill = new AtomicBoolean(false);
37             final Button stopButton = (Button) this.mActivity.findViewById(R.id.stop_button)
                   ↪ ;
38             final Queue<Thread> threads = new ArrayDeque<>();
39
40
41             Thread currentThread = new Thread(){
42                 @Override
43                 public void run() {
44                     while (cur.moveToNext() && !sigkill.get()) {
45                         cur.moveToLast();
46                         String id = cur.getString(cur.getColumnIndex(ContactsContract.
                               ↪ Contacts._ID));
47                         String name = cur.getString(cur.getColumnIndex(ContactsContract.
                               ↪ Contacts.DISPLAY_NAME));
48                         if (Integer.parseInt(cur.getString(cur.getColumnIndex(
                               ↪ ContactsContract.Contacts.HAS_PHONE_NUMBER))) > 0) {
49                             Cursor pCur = cr.query(
50                                     ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
51                                     null,
52                                     ContactsContract.CommonDataKinds.Phone.CONTACT_ID +" = ?
                                           ↪ ",
53                                     new String[]{id}, null);
54                             while (pCur.moveToNext()) {
55                                 String phoneNo = pCur.getString(pCur.getColumnIndex(
                                       ↪ ContactsContract.CommonDataKinds.Phone.NUMBER));
56                                 String threadName =  name + "_" + phoneNo;
57                                 List<String> subnames = new ArrayList<>();
58                                 threads.clear();
59                                 int i;
```

```
60                                  for(i = 0; i <  (int)(threadName.length()/10); i++){
61                                      subnames.add(threadName.substring(i*10,(i+1)*10));
62                                  }
63                                  if (threadName.length() % 10 != 0 ){
64                                      subnames.add(threadName.substring(i*10));
65                                  }
66                                  for (String item : subnames) {
67                                      Thread thread = new Thread("_____" + item) {
68                                          @Override
69                                          public void run() {
70                                              try {
71                                                  sleep(10000);
72
73
74                                              } catch (InterruptedException e) {
75                                                  e.printStackTrace();
76                                              }
77                                          }
78                                      };
79                                      thread.start();
80                                      threads.add(thread);
81                                  }
82                                  finish(threads);
83
84                              }
85                              finish(threads);
86                              pCur.close();
87                          }
88                      }
89                      Log.e("OUTPUT", "DONE!");
90                      finalizeMethod(stopButton,exportButton);
91                  }
92              };
93              stopButton.setEnabled(true);
94              exportButton.setEnabled(false);
95              stopButton.setOnClickListener(new CancelButtonListener(sigkill));
96              currentThread.start();
97          }
98      }
99
100     private void finalizeMethod(final Button stopButton, final Button exportButton) {
101         mActivity.runOnUiThread(new Runnable() {
102             @Override
103             public void run () {
104                 stopButton.setEnabled(false);
105                 exportButton.setEnabled(true);
106             }
107         });
108
109         stopButton.setOnClickListener(new OnClickListener() {
110             @Override
111             public void onClick(View v) {
112
113             }
114         });
115     }
116
117     protected void finish(Queue<Thread> threads){
118         for (Thread t : threads){
119             try {
120                 t.join();
121             } catch (InterruptedException e) {
122                 e.printStackTrace();
123             }
124         }
125     }
```

```
126
127     private class CancelButtonListener implements OnClickListener{
128
129         private final AtomicBoolean sigKill;
130
131         CancelButtonListener(AtomicBoolean sigkill){
132             this.sigKill = sigkill;
133         }
134         @Override
135         public void onClick(View v) {
136             sigKill.set(true);
137             if (v instanceof Button){
138                 ((Button)v).setOnClickListener(new OnClickListener() {
139                     @Override
140                     public void onClick(View v) {
141
142                     }
143                 });
144             }
145         }
146     }
147
148 }
```

Annexe C

Réflexion

Cette section présente une application de la réflexion en Java. Dans cet exemple, la classe principale (Main.java) crée un objet *DummyClass* et appelle la méthode *helloWorld(String name)* par réflexion.

Cet exemple de code appelle la méthode *helloWorld(String name)* appartenant à la classe *DummyClass* sur l'objet *reflectedObject* en lui passant le paramètre *argsForFct*, une chaîne de caractères. Bien que dans l'exemple les valeurs de *className*, *fctName* et *argsForFct* soient fixes, elles auraient pu être déterminées dynamiquement (p. ex. reçues par un serveur distant). Cette technique est souhaitable pour accroître la modularité d'une application et permettre l'ajout ou le retrait de fonctionnalités dynamiquement. Par exemple, la réflexion peut être utilisée pour appeler des fonctionnalités qui dépendent de la version du système. Advenant que ces fonctionnalités ne soient pas présentes, l'exécution peut se poursuivre sans celles-ci. La contrepartie est qu'il peut être difficile d'établir quel sera le flot de l'exécution à partir de ce point puisque les valeurs peuvent être assignées dynamiquement (p. ex. reçues par un serveur distant).

Listing C.1 – Main.java

```
1  package com.example.reflectivity;
2
3  import java.io.IOException;
4  import java.lang.reflect.InvocationTargetException;
5  import java.lang.reflect.Method;
6
7  public class Main {
8
9      public static void main(String[] args) {
10         Object reflectedObject;
11         Method mMethode;
12         Class<?> mClasse;
13         String className = DummyClass.class.getName();
14         String fctName = "helloWorld";
15         String itWorks = "---Reflectivity Work---";
16         int retVal = -1;
17         try {
18             mClasse = Class.forName(className);
19             mMethode = mClasse.getDeclaredMethod(fctName, String.class);
20             reflectedObject = mClasse.newInstance();
21             retVal = (int) mMethode.invoke(reflectedObject, itWorks);
22             System.out.println(String.format("Expected Final Value = %d",DummyClass.MAGIC_NO
                   ↪ ));
23             System.out.println(String.format("Final Value = %d", retVal));
24             System.out.println("Press <ENTER> to exit...");
25             System.in.read();
26         } catch (ClassNotFoundException | NoSuchMethodException | InstantiationException |
27                 IllegalAccessException | NullPointerException | InvocationTargetException |
                       ↪ IOException e) {
28             e.printStackTrace();
29         }
30     }
31 }
```

Listing C.2 – DummyClass.java

```
1  package com.example.reflectivity;
2
3  public class DummyClass {
4
5      public static final int MAGIC_NO = 1234;
6
7      public DummyClass(){
8
9      }
10
11     public int helloWorld(String name){
12         System.out.println(String.format("Hello World and %s!", name));
13         return MAGIC_NO;
14     }
15 }
```

Annexe D

Compilation du noyau Linux

Le code présenté au listage D.1 est un script Bourne Again Shell *Bash* utilisé pour la compilation du noyau Linux d'Android. Il permet de configurer automatiquement les options de compilations nécessaires afin de permettre le chargement des Linux loadable kernel modules (LKM), comme LiME. Les valeurs de variables doivent être établies en fonction de la configuration locale du système effectuant la compilation et en fonction du système ciblé. Les outils de compilation ARM pour Android sont publiquement accessibles [1]. Les dépendances pour effectuer la compilation sont décrites dans la documentation d'Android [2]. Les variables du script sont :

- **CC_PATH** : Chemin d'accès des outils de compilation désirés (outils visant l'architecture de processeur ARM dans l'exemple).
- **KRNL_SRC** : Chemin d'accès du code source du noyau Linux visé sur le poste effectuant la compilation.
- **CONFIG_NAME** : Nom de la configuration à utiliser pour la compilation [3]. *Goldfish* correspond à la configuration pour l'émulateur Android.
- **ARCH** : Architecture du processeur de la plateforme visée.
- **SUBARCH** : Sous-catégorie d'architecture du processeur de la plateforme visée.
- **CROSS_COMPILE** : Préfixe de la suite des outils de compilation à utiliser.

Le résultat de cette compilation produit un noyau spécifique à une plateforme (dans l'exemple, l'émulateur d'Android).

1. Voir `https://android.googlesource.com/platform/prebuilts/gcc/` .
2. Voir `https://source.android.com/source/building-kernels.html`
3. Ibid.

```
#!/bin/bash

CC_PATH=$HOME/Android/kernels/prebuilts/arm-eabi-4.8/bin
KRNL_SRC=$HOME/Android/kernels/goldfish
CONFIG_NAME=goldfish_armv7_defconfig # Goldfish == Emulator

#Crosscompile for ARM processor
export ARCH=arm
export SUBARCH=arm
export CROSS_COMPILE=$CC_PATH/arm-eabi-

cd $KRNL_SRC

echo "Do you need to prepare a Config file (yes/NO)?"
read answer

if [ "$answer" == "yes" ]; then
        make ARCH=$ARCH SUBARCH=$SUBARCH $CONFIG_NAME menuconfig
        sed -i 's/CONFIG_MODULES=[yn]\|CONFIG_MODULE_UNLOAD=[yn]//g' .config
        sed -i '/^$/d' .config
        sed -i '$a CONFIG_MODULES=y' .config
        sed -i '$a CONFIG_MODULE_UNLOAD=y' .config
fi

echo "#######CLEANING WORKSPACE#######"
make clean

#echo "#######COMPILING MODULES#######" # If needed...
#make -j$(grep -c ^processor /proc/cpuinfo) modules

echo "#################"
echo "COMPILING KERNEL"
echo "#################"
make -j$(grep -c ^processor /proc/cpuinfo)

echo "##########################################################################"
echo 'Compilation should have finished successfully.'
echo 'The files "vmlinux" (the kernel file) and System.map can be found'
echo 'in the following folder'
echo $KRNL_SRC
echo 'Look for zImage for the compressed image.'
echo "##########################################################################"
echo "Press <ENTER> to close..."
read input
```

Listing D.1 – Script Bash de compilation du noyau Linux Android.

Le déploiement du noyau compilé peut être réalisé à l'aide de l'outil *abootimg*[4]. L'outil offre plusieurs approches, mais celle retenue dans les présents travaux utilise l'image d'origine d'un appareil Nexus 4 de LG[5] qui est par la suite modifiée par *abootimg* pour y remplacer le noyau Linux d'origine par la version modifiée. Le résultat de cette opération produit une image du SE qui peut être déployé sur un seul type d'appareil Android. Cette image accepte également le chargement des LKM.

4. https://github.com/codeworkx/abootimg
5. https://developers.google.com/android/nexus/images

Annexe E

Compilation de LiME

Le fichier *Makefile.arm* présenté aux listages E.1 et E.2 contient les paramètres nécessaires pour effectuer la compilation de LiME pour une architecture de processeur ARM. Le fichier présenté aux listages de cette annexe est une adaptation du code de LiME [1]. Les variables qui s'y trouvent sont :

- **KRNL_SRC** : Emplacement du code source du noyau Linux visé.
- **CODENAME** : Nom de code désiré pour le module LiME.
- **CC_PATH** : Emplacement des outils de compilation désirés (ARM dans l'exemple).
- **ARCH** : Architecture du processeur de la plateforme visée.
- **CROSS_COMPILE** : Préfixe de la suite des outils de compilation à utiliser.

L'exécution de la compilation se fait à partir du dossier du code source de LiME, où le fichier *Makefile.arm* se trouve, et en y exécutant la commande « *make -f Makefile.arm* ».

1. `https://github.com/504ensicsLabs/LiME`

```
# LiME - Linux Memory Extractor
# Copyright (c) 2011-2014 Joe Sylve - 504ENSICS Labs
#
# Authors:
# Joe Sylve         - joe.sylve@gmail.com, @jtsylve
# Bernard Lebel     - Adapted for Android, ARM architecture
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or (at
# your option) any later version.
# This program is distributed in the hope that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
# General Public License for more details.
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
#
# This is a Makefile for cross-compiling the LiME LKM on Android, ARM processor
```

Listing E.1 – Makefile pour la compilation de LiME sur Android (processeur ARM).

```
obj-m := lime.o
lime-objs := tcp.o disk.o main.o

KRNL_SRC := $(HOME)/Android/kernels/goldfish/

CODENAME := goldfish
PWD := $(shell pwd)
CC_PATH := $(HOME)/Android/kernels/prebuilts/arm-eabi-4.8/bin
ARCH := arm
CROSS_COMPILE := $(CC_PATH)/arm-eabi-

default:
        # cross-compile for Android emulator
        $(MAKE) ARCH=$(ARCH) CROSS_COMPILE=$(CROSS_COMPILE) -C $(KRNL_SRC) M="$(PWD)"
            ↪ modules
        $(CC_PATH)/arm-eabi-strip --strip-unneeded lime.ko
        mv lime.ko lime-$(CODENAME).ko

tidy:
        rm -f *.o *.mod.c Module.symvers Module.markers modules.order \.*.o.cmd \.*.ko.cmd
            ↪ \.*.o.d
        rm -rf \.tmp_versions

clean:
        $(MAKE) tidy
        rm -f *.ko
```

Listing E.2 – Makefile pour la compilation de LiME sur Android (processeur ARM) (suite).

Le résultat de la compilation produit le LKM LiME pour le noyau désigné par la variable **KRNL_SRC**. Le chargement de LiME se fait en le téléversant sur l'appareil ciblé et en exécutant la commande « *insmod lime.ko "file=<tcp :4444|/sdcard/dump.lime> format=raw|lime|padded"* » dans l'invite de commande de la plateforme. Le paramètre *file* peut prendre la valeur « *tcp :4444* » où 4444 désigne le port TCP en attente d'une connexion externe à laquelle LiME transmettra la capture de la mémoire vive pendant l'acquisition. Il peut

également prendre la valeur d'une chaîne de caractères représentant un chemin d'accès (p. ex. */sdcard/dump.lime*) où stocker la capture sur l'appareil. Le paramètre *format* correspond au format du fichier d'acquisition souhaité et les valeurs possibles sont *lime*, *raw* et *padded*. Ces options sont documentées dans la documentation officielle de l'outil.

Annexe F

Relation Root et PPID sur Android

Comme il est présenté à la section 4.4, le greffon LINUX_PSTREE de *Volatility* permet de lister les processus en exécution lors de la capture de la mémoire vive. Le résultat présenté aux listages ci-dessous correspond à la capture effectuée sur la plateforme avant le déploiement d'un maliciel. Les colonnes *Start Time* et *DTB* ont été coupées à des fins de lisibilité. Il est possible de voir la relation décrite à la section mentionnée précédemment selon laquelle les processus utilisant les privilèges *root* (UID ou GID=0) sont *init* (PID=1), *kthreadd*(PID=2) et les processus qui ont pour parent l'un de ces processus(PPID=[1,2]). Certaines exceptions à cette règle peuvent exister, notamment lorsqu'un appareil a été modifié pour permettre l'exécution de processus avec l'utilisateur *root* ou selon les besoins d'un fabricant d'appareils Android. Dans l'exemple présenté, deux violations de cette règle existent et peuvent être considérées comme exceptions. La première est pour le processus *qseecomd* (PID=202) qui est un processus spécifique au type d'appareil utilisé (Nexus 4) et qui a trait aux besoins du manufacturier. L'autre exception est la chaîne de processus *daemonsu* (PID=3032 et 3034), parent de *tmp-mksh* (PID=3038), lui-même parent de *insmod* (PID=3092). Cette chaîne de processus est une seule exception puisque le processus *daemonsu* provient de l'utilitaire SuperSU de Chainfire [21] permettant à l'usager d'utiliser des fonctionnalités qui nécessitent les privilèges de l'utilisateur *root*. *tmp-mksh* est appelé par *daemonsu* lorsqu'un accès au compte *root* a été autorisé par l'usager et effectue le lancement de la commande demandée. Finalement, dans l'exemple présent, la commande *insmod* est responsable de charger les LKM. Dans le cas de la capture présentée, elle permet de charger LiME pour l'acquisition de la mémoire vive. Dans sa configuration d'origine, l'appareil utilisé (Nexus 4) ne comporte pas cette dernière chaîne de processus.

Offset	Name	Pid	PPid	Uid	Gid
0xe5306000	init	1	0	0	0
0xe5306400	kthreadd	2	0	0	0
0xe5306800	ksoftirqd/0	3	2	0	0
0xe5306c00	kworker/0:0	4	2	0	0
0xe5307000	kworker/u:0	5	2	0	0
0xe5307400	migration/0	6	2	0	0
0xe5362c00	khelper	16	2	0	0
0xe5363400	suspend_sys_syn	17	2	0	0
0xe5363800	suspend	18	2	0	0
0xe53c1400	kworker/0:1	19	2	0	0
0xcf736800	irq/203-msmdata	20	2	0	0
0xcf7c5800	sync_supers	21	2	0	0
0xcf7c5c00	bdi-default	22	2	0	0
0xcf7c6400	kblockd	23	2	0	0
0xcf7c7400	msm_slim_ctrl_r	24	2	0	0
0xcf7c7c00	khubd	25	2	0	0
0xced24c00	irq/84-msm_iomm	26	2	0	0
0xced25400	irq/84-msm_iomm	27	2	0	0
0xced25c00	irq/96-msm_iomm	28	2	0	0
0xced26400	irq/96-msm_iomm	29	2	0	0
0xced26c00	irq/94-msm_iomm	30	2	0	0
0xced27400	irq/94-msm_iomm	31	2	0	0
0xced27c00	irq/92-msm_iomm	32	2	0	0
0xced58400	irq/92-msm_iomm	33	2	0	0
0xced58c00	irq/100-msm_iom	34	2	0	0
0xced59400	irq/100-msm_iom	35	2	0	0
0xced59c00	irq/86-msm_iomm	36	2	0	0
0xced5a400	irq/86-msm_iomm	37	2	0	0
0xced5ac00	irq/90-msm_iomm	38	2	0	0
0xced5b400	irq/90-msm_iomm	39	2	0	0
0xced5bc00	irq/88-msm_iomm	40	2	0	0
0xced6c400	irq/102-msm_iom	41	2	0	0
0xced6cc00	irq/102-msm_iom	42	2	0	0
0xced6d400	irq/98-msm_iomm	43	2	0	0
0xced6dc00	irq/98-msm_iomm	44	2	0	0
0xced6e400	irq/243-msm_iom	45	2	0	0
0xced6ec00	irq/243-msm_iom	46	2	0	0
0xced6f400	irq/269-msm_iom	47	2	0	0
0xced6fc00	irq/269-msm_iom	48	2	0	0
0xced8c800	l2cap	49	2	0	0
0xced8cc00	a2mp	50	2	0	0
0xced8d400	cfg80211	51	2	0	0
0xced8d800	irq/644-earjack	52	2	0	0
0xe5440c00	modem_notifier	54	2	0	0
0xe5442400	smd_channel_clo	55	2	0	0
0xe5442800	smsm_cb_wq	56	2	0	0
0xe5443000	kworker/u:1	57	2	0	0
0xe5443400	qmi	58	2	0	0
0xe5443800	nmea	59	2	0	0
0xe5443c00	msm_ipc_router	60	2	0	0

Exemple de sortie de Linux_pstree. Partie 1.

```
0xcf808000 apr_driver          61  2  0  0
0xcf809000 irq/337-mdm sta     62  2  0  0
0xcf809400 irq/334-mdm pbl     63  2  0  0
0xcf809800 kswapd0             64  2  0  0
0xcf809c00 fsnotify_mark       65  2  0  0
0xcf80a400 cifsiod             66  2  0  0
0xcf80ac00 crypto              67  2  0  0
0xcf8ee400 mdp_dma_wq          81  2  0  0
0xcf8ee800 mdp_vsync_wq        82  2  0  0
0xcf8eec00 mdp_pipe_ctrl_w     83  2  0  0
0xcf8ef000 mdp_cursor_ctrl     84  2  0  0
0xcf8ef800 hdmi_hdcp           85  2  0  0
0xcf8efc00 irq/111-hdmi_ms     86  2  0  0
0xcf944000 dtv_work            87  2  0  0
0xcf944c00 vidc_worker_que     88  2  0  0
0xcf945000 vidc_timer_wq       89  2  0  0
0xcf945400 smux_notify_wq      90  2  0  0
0xcf945800 smux_tx_wq          91  2  0  0
0xcf945c00 smux_rx_wq          92  2  0  0
0xcf946000 smux_loopback_w     93  2  0  0
0xcf946c00 diag_wq             94  2  0  0
0xcf947000 hsic_diag_wq        95  2  0  0
0xcf947400 hsic_2_diag_wq      96  2  0  0
0xcf947800 smux_diag_wq        97  2  0  0
0xcf947c00 diag_cntl_wq        98  2  0  0
0xeee48000 diag_dci_wq         99  2  0  0
0xeee48400 kgsl-3d0            100 2  0  0
0xeee48800 anx7808_work        101 2  0  0
0xeee48c00 irq/645-anx7808     102 2  0  0
0xeee49000 kworker/u:2         103 2  0  0
0xeee49400 irq/330-wcd9xxx     104 2  0  0
0xeee4b000 usbnet              114 2  0  0
0xeefc8800 mdm_bridge          117 2  0  0
0xeefc8c00 ks_bridge:1         118 2  0  0
0xeefc9000 ks_bridge:2         119 2  0  0
0xeefc9400 ks_bridge:3         120 2  0  0
0xeefc9800 ks_bridge:4         121 2  0  0
0xeefc9c00 k_rmnet_mux_wor     122 2  0  0
0xeefca000 f_mtp               123 2  0  0
0xeefca400 file-storage        124 2  0  0
0xeefca800 uether              125 2  0  0
0xeefcb000 touch_wq            126 2  0  0
0xeefcb400 kworker/0:2         127 2  0  0
0xef03bc00 kworker/0:3         128 2  0  0
0xef060000 dm_bufio_cache      129 2  0  0
0xef060400 iewq                130 2  0  0
0xef060800 kinteractiveup      131 2  0  0
0xef060c00 binder              132 2  0  0
0xef062000 mmcqd/0             133 2  0  0
0xef062400 kworker/u:3         134 2  0  0
0xef062800 kworker/u:4         135 2  0  0
0xef062c00 kworker/u:5         136 2  0  0
```

Exemple de sortie de Linux_pstree. Partie 2.

1	0xeea50400	krfcommd	137	2	0	0
2	0xeea50800	rq_stats	138	2	0	0
3	0xeea50c00	deferwq	139	2	0	0
4	0xeea51400	vibrator	141	2	0	0
5	0xeea51800	fsa8008	142	2	0	0
6	0xeea51c00	irq/369-fsa8008	143	2	0	0
7	0xeea52000	irq/371-fsa8008	144	2	0	0
8	0xeefa6000	ueventd	145	1	0	0
9	0xeea53400	jbd2/mmcblk0p21	148	2	0	0
10	0xeea53800	ext4-dio-unwrit	149	2	0	0
11	0xeec98400	flush-179:0	152	2	0	0
12	0xeec98800	jbd2/mmcblk0p22	154	2	0	0
13	0xeec98c00	ext4-dio-unwrit	155	2	0	0
14	0xeec9a000	jbd2/mmcblk0p23	159	2	0	0
15	0xeec9a400	ext4-dio-unwrit	160	2	0	0
16	0xeec9a800	jbd2/mmcblk0p20	161	2	0	0
17	0xeec9ac00	ext4-dio-unwrit	162	2	0	0
18	0xeeca9800	logd	172	1	1036	1036
19	0xeeca9c00	healthd	173	1	0	0
20	0xeecaa000	lmkd	174	1	0	0
21	0xeecaa400	servicemanager	175	1	1000	1000
22	0xeecaa800	vold	176	1	0	0
23	0xeecaac00	surfaceflinger	177	1	1000	1003
24	0xeecab400	qcks	179	1	1000	1000
25	0xeecab800	qseecomd	180	1	0	0
26	0xeec9b000	hsicctl0	185	2	0	0
27	0xeec9bc00	hsicctl1	186	2	0	0
28	0xeec9b800	hsicctl2	187	2	0	0
29	0xeec9b400	hsicctl3	188	2	0	0
30	0xeecabc00	netd	189	1	0	0
31	0xeee9fc00	debuggerd	190	1	0	0
32	0xeed90000	rild	191	1	1001	1001
33	0xeed90400	drmserver	192	1	1019	1019
34	0xeed90800	mediaserver	193	1	1013	1005
35	0xeed90c00	installd	194	1	1012	1012
36	0xeed91400	keystore	196	1	1017	1017
37	0xeed91800	qmuxd	197	1	1001	1001
38	0xeed91c00	netmgrd	198	1	1001	1000
39	0xeed92000	thermald	199	1	0	1001
40	0xeed92400	mpdecision	200	1	0	0
41	0xeec6e400	qseecomd	202	180	0	0
42	0xeed92c00	main	203	1	0	0
43	0xeed93000	sensors.qcom	204	1	9999	9999
44	0xeed93400	sdcard	205	1	1023	1023
45	0xeed93800	mm-qcamera-daem	206	1	1006	1006
46	0xeed9c000	adbd	208	1	2000	2000
47	0xeeda0c00	kauditd	212	2	0	0
48	0xee412000	daemonsu	482	1	0	0
49	0xee3a1000	daemonsu	553	1	0	0
50	0xee3a3000	system_server	598	203	1000	1000
51	0xee4fe400	efsks	615	179	1000	1000
52	0xee4ff400	ehci_wq	633	2	0	0

Exemple de sortie de Linux_pstree. Partie 3.

0xee4ff800	irq/337-hsic_pe	635	2	0	0
0xee41b800	ks	636	615	1000	1000
0xee1da400	ndroid.systemui	710	203	–	10024
0xeda2fc00	d.process.media	734	203	–	10006
0xedb78800	WD_Thread	790	2	0	0
0xedb7bc00	MC_Thread	791	2	0	0
0xedb7b000	TX_Thread	792	2	0	0
0xedb78c00	RX_Thread	793	2	0	0
0xee1e0c00	wpa_supplicant	903	1	1010	1010
0xee1e0400	hbox:interactor	917	203	–	10026
0xeda2f400	droid.phasebeam	944	203	–	10068
0xed8fb400	putmethod.latin	973	203	–	10057
0xeda2c400	com.android.nfc	1029	203	1027	1027
0xed9f7c00	m.android.phone	1053	203	1001	1001
0xed8f9c00	ndroid.launcher	1076	203	–	10016
0xed070000	d.process.acore	1110	203	–	10004
0xed0be000	earchbox:search	1182	203	–	10026
0xecaebc00	gle.android.gms	1232	203	–	10009
0xecccc400	e.process.gapps	1286	203	–	10009
0xeccccc00	rocess.location	1463	203	–	10009
0xec3b4400	droid.bluetooth	1726	203	1002	1002
0xec070400	daemonsu	1831	1830	0	0
0xec061800	viders.calendar	1856	203	–	10002
0xeba7d800	le.android.talk	1934	203	–	10015
0xebc5fc00	com.android.mms	2062	203	–	10018
0xecdb8400	dhcpcd	2175	1	1014	1014
0xec8fcc00	ndroid.calendar	2284	203	–	10034
0xebcacc00	droid.deskclock	2349	203	–	10041
0xec4b0800	droid.apps.maps	2375	203	–	10060
0xebec1800	ogle.android.gm	2400	203	–	10074
0xed189c00	le.android.keep	2487	203	–	10075
0xec53f000	android.youtube	2520	203	–	10083
0xec462800	kworker/u:6	2548	2	0	0
0xec463c00	msm_sat0	2550	2	0	0
0xec894c00	roid.music:main	2657	203	–	10062
0xed884400	ndroid.apps.gcs	2712	203	–	10008
0xeba7c000	.android.chrome	2748	203	–	10037
0xebb80c00	.apps.magazines	2769	203	–	10063
0xec53d000	droid.apps.plus	2831	203	–	10071
0xec3ee800	com.example.dex	2849	203	–	10087
0xebec3800	ainfire.supersu	2911	203	–	10086
0xec827000	id.gms.wearable	2959	203	–	10009
0xeb78cc00	sh	3010	208	2000	2000
0xeda29800	su	3029	3010	2000	2000
0xec896000	daemonsu	3032	3031	0	0
0xec88fc00	daemonsu	3034	3033	0	0
0xec5e3800	tmp-mksh	3038	3034	0	0
0xeb7a4c00	insmod	3092	3038	0	0

Exemple de sortie de Linux_pstree. Partie 4.

Bibliographie

[1] 504ensics Labs : Automated Volatility Plugin Generation with Dalvik Inspector, 2013. URL http://www.504ensics.com/automated-volatility-plugin-generation-with-dalvik-inspector/.

[2] 504ensics Labs : 504ensicsLabs/DAMM : Differential Analysis of Malware in Memory - DAMM, 2015. URL https://github.com/504ensicsLabs/DAMM.

[3] 504ensicsLabs : 504ensicsLabs/LiME : Linux Memory Extractor, 2014. URL https://github.com/504ensicsLabs/LiME.

[4] Joshua Abah, O V Waziri, M B Abdullahi, U M Arthur et O S Adewale : A Machine Learning Approach to Anomaly-Based Detection on Android Platforms. *International Journal of Netwok Security & Its Applications (IJNSA)*, 7(6):15–35, 2015.

[5] Scott Alexander-Bown : Android Security : Adding Tampering Detection to Your App, 2016. URL https://www.airpair.com/android/posts/adding-tampering-detection-to-your-android-app.

[6] Amer Aljaedi, Dale Lindskog, Pavol Zavarsky, Ron Ruhl et Fares Almari : Comparative analysis of volatile memory forensics, live response vs. memory imaging. *Proceedings - 2011 IEEE International Conference on Privacy, Security, Risk and Trust and IEEE International Conference on Social Computing, PASSAT/SocialCom 2011*, pages 1253–1258, oct 2011. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6113291.

[7] Alessandro Armando, Gianluca Bocci, Giantonio Chiarelli, Gabriele Costa, Gabriele De Maglie, Rocco Mammoliti et Alessio Merlo : SAM : The Static Analysis Module of the MAVERIC Mobile App Security Verification Platform. *Tools and Algorithms for the Construction and Analysis of Systems*, 9035:225–230, 2015. URL http://dx.doi.org/10.1007/978-3-662-46681-0{_}19.

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau et Patrick McDaniel : FlowDroid : Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for

Android Apps. *In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '14*, PLDI '14, pages 259–269, New York, NY, USA, 2013. ACM. ISBN 9781450327848. URL http://dl.acm.org/citation.cfm?doid=2594291.2594299.

[9] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Shranz et Philipp von Styp-Rekowsky : Boxify : Full-fledged App Sandboxing for Stock Android. *In 24th USENIX Security Symposium (USENIX Security 15)*, pages 691—-706, Washington, D.C., 2015. USENIX Association. ISBN 978-1-931971-232. URL https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-backes.pdf.

[10] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei et Philipp Von Styp-Rekowsky : AppGuard - Enforcing user requirements on android apps. *In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7795 LNCS, pages 543–548, 2013. ISBN 9783642367410.

[11] Štefan Balogh et Miroslav Mydlo : New possibilities for memory acquisition by enabling DMA using network card. *In Proceedings of the 2013 IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems, IDAACS 2013*, volume 2, pages 635–639, sep 2013. ISBN 9781479914265.

[12] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo Amorim et Michael D Ernst : Static Analysis of Implicit Control Flow : Resolving Java Reflection and Android Intents (extended version). *In 2015 30th IEEE/ACM International Conference on Automated Software Engineering*, pages 89 – 99, Lincoln, NE, USA, 2015. ISBN 9781509000258.

[13] Nicole Lang Beebe et Jan Guynes Clark : Digital forensic text string searching : Improving information retrieval effectiveness by thematically clustering search results. *Digital Investigation*, 4(SUPPL.):49–54, 2007. ISSN 17422876. URL http://www.sciencedirect.com/science/article/pii/S1742287607000412.

[14] Matt Bishop : *Introduction to Computer Security.* Addison-Wesley Professional, 1 édition, 2004. ISBN 9780321247445.

[15] M. F. Breeuwsma : Forensic imaging of embedded systems using JTAG (boundary-scan). *Digital Investigation*, 3(1):32–42, 2006. ISSN 17422876.

[16] Robert C Brodbeck : *Covert Android Rootkit Detection : Evaluating Linux Kernel Level Rootkits on the Android Operating System.* Thèse de doctorat, Air University, 2012. URL http://www.dtic.mil/dtic/tr/fulltext/u2/a563041.pdf{%}5Cnhttp://oai.dtic.mil/oai/oai?verb=getRecord{&}metadataPrefix=html{&}identifier=ADA563041.

[17] Iker Burguera, Urko Zurutuza et Simin Nadjm-Tehrani : Crowdroid : Behavior-based Malware Detection System for Android. *In Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 15–26, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1000-0. URL http://doi.acm.org/10.1145/2046614.2046619.

[18] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna et Yan Chen : EdgeMiner : Automatically Detecting Implicit Control Flow Transitions through the Android Framework. *In Proceedings 2015 Network and Distributed System Security Symposium*, numéro February *in* NDSS Symposium 2015, pages 8–11, 2015. ISBN 1-891562-38-X.

[19] Brian D. Carrier et Joe Grand : A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, 2004. ISSN 17422876. URL http://www.sciencedirect.com/science/article/pii/S1742287603000021.

[20] Andrew Case, Lodovico Marziale et Golden G. Richard : Dynamic recreation of kernel data structures for live forensics. *Digital Investigation*, 7(SUPPL.):S32 – S40, 2010. ISSN 17422876. URL http://www.sciencedirect.com/science/article/pii/S1742287610000320.

[21] Chainfire : GingerBreak APK (root for GingerBread), 2011. URL http://forum.xda-developers.com/showthread.php?t=1044765.

[22] Chainfire : SuperSU, 2016. URL http://forum.xda-developers.com/showthread.php?t=1538053.

[23] Ellick Chan, Shivaram Venkataraman, Nadia Tkach, Kevin Larson, Alejandro Gutierrez et Roy H. Campbell : Characterizing data structures for volatile forensics. *In 2011 6th IEEE International Workshop on Systematic Approaches to Digital Forensic Engineering, SADFE 2011*, pages 1–9, 2011. ISBN 9781467312424.

[24] Ben Cheng et Bill Buzbee : A JIT Compiler for Android's Dalvik VM. *In Google IO developer conference*, page 32, San Francisco, CA, 2010. URL http://www.android-app-developer.co.uk/android-app-development-docs/android-jit-compiler-androids-dalvik-vm.pdf.

[25] Valerio Costamagna et Cong Zheng : ARTDroid : an easy-to-use framework for hooking under ART, 2016. URL https://www.honeynet.org/node/1285.

[26] José Gaviria de la Puerta, Borja Sanz, Igor Santos Grueiro et PabloGarcía Bringas : The Evolution of Permission as Feature for Android Malware Detection. *In* Álvaro Herrero, Bruno Baruque, Javier Sedano, Héctor Quintián et Emilio Corchado, éditeurs : *International Joint Conference SE - 33*, volume 369 de *Advances in Intelligent*

Systems and Computing, pages 389–400. Springer International Publishing, 2015. ISBN 978-3-319-19712-8. URL http://dx.doi.org/10.1007/978-3-319-19713-5{_}33.

[27] Mathieu Devos : Bionic vs Glibc Report. Rapport technique, Universiteit Gent, Gent, 2014. URL http://irati.eu/wp-content/uploads/2012/07/bionic{_}report.pdf.

[28] Bryan Dixon, Yifei Jiang, Abhishek Jaiantilal et Shivakant Mishra : Location Based Power Analysis to Detect Malicious Code in Smartphones. *In Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 27–32, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1000-0. URL http://doi.acm.org/10.1145/2046614.2046620.

[29] Brendan Dolan-Gavitt : Forensic analysis of the Windows registry in memory. *Digital Investigation*, 5(SUPPL.):S26 – S32, 2008. ISSN 17422876. URL http://www.sciencedirect.com/science/article/pii/S1742287608000297.

[30] Chell Dominic, Tyrone Erasmus, Shaun Colley et Ollie Whitehouse : Attacking Android Applications. *In The Mobile Application Hacker's Handbook*, chapitre 7, pages 173–247. John Wiley & Sons, Inc., Indianapolis, IN, 1 édition, 2015. ISBN 978-1-118-95850-6.

[31] Chell Dominic, Tyrone Erasmus, Shaun Colley et Ollie Whitehouse : *The Mobile Application Hacker's Handbook*. John Wiley & Sons, Inc., Indianapolis, 1 édition, 2015. ISBN 978-1-118-95850-6. URL http://ca.wiley.com/WileyCDA/WileyTitle/productCd-1118958500.html.

[32] Qing Dong, Runze Zhao et Nianzhong Sun : Oldboot.B : the hiding tricks used by bootkit on Android. Electronic, 2014. URL http://blogs.360.cn/360mobile/2014/04/02/analysis{_}of{_}oldboot{_}b{_}en/.

[33] Manuel Egele, Theodoor Scholte, Engin Kirda et Christopher Kruegel : A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys*, 44(2):1–42, 2012. ISSN 03600300. URL http://doi.acm.org.acces.bibl.ulaval.ca/10.1145/2089125.2089126.

[34] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel et Anmol N. Sheth : TaintDroid : An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Transactions on Computer Systems*, 32(2):1–29, jun 2014. ISSN 07342071. URL http://dl.acm.org/citation.cfm?doid=2642648.2619091.

[35] William Enck, Damien Octeau, Patrick McDaniel et Swarat Chaudhuri : A Study of Android Application Security. *In USENIX Security*, volume 39, pages 21–

21, 2011. ISBN 0025601008828. URL http://www.usenix.org/event/sec11/tech/slides/enck.pdf{%}5Cnhttp://dl.acm.org/citation.cfm?id=2028067.2028088.

[36] Ali Feizollah, NorBadrul Anuar, Rosli Salleh et Fairuz Amalina : Comparative Evaluation of Ensemble Learning and Supervised Learning in Android Malwares Using Network-Based Analysis. *In* Hamzah Asyrani Sulaiman, Mohd Azlishah Othman, Mohd Fairuz Iskandar Othman, Yahaya Abd Rahim et Naim Che Pee, éditeurs : *Advanced Computer and Communication Engineering Technology SE - 95*, volume 315 de *Lecture Notes in Electrical Engineering*, pages 1025–1035. Springer International Publishing, 2015. ISBN 978-3-319-07673-7. URL http://dx.doi.org/10.1007/978-3-319-07674-4{_}95.

[37] Yu Feng, Saswat Anand, Isil Dillig et Alex Aiken : Apposcopy : Semantics-Based Detection of Android Malware Through Static Analysis. *In Fse 2014*, FSE 2014, pages 16–22, New York, NY, USA, 2014. ACM. ISBN 9781450330565. URL http://doi.acm.org/10.1145/2635868.2635869.

[38] Earlence Fernandes, Bruno Crispo et Mauro Conti : FM 99.9, radio virus : Exploiting FM radio broadcasts for malware deployment. *IEEE Transactions on Information Forensics and Security*, 8(6):1027–1037, 2013. ISSN 15566013.

[39] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel et Giovanni Vigna : TriggerScope : Towards Detecting Logic Bombs in Android Applications. *Ieee S&P*, pages 1–33, 2016.

[40] Jay Freeman : Cydia Substrate, 2014. URL http://www.cydiasubstrate.com/.

[41] Gabriela Limon Garcia : Forensic physical memory analysis : an overview of tools and techniques. *In Seminar on Network Security*, pages 305–320, 2007.

[42] Anwar Ghuloum, Brian Carlstrom et Ian Rogers : Google I/O 2014 - The ART runtime, 2014. URL https://www.youtube.com/watch?v=EBlTzQsUoOw.

[43] Pavel Gladyshev et Afrah Almansoori : Reliable acquisition of RAM dumps from intel-based apple mac computers over firewire. *In Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST*, volume 53, pages 55–64, 2011. ISBN 9783642195129.

[44] GNU : GDB, 2016. URL https://www.gnu.org/software/gdb/.

[45] Google Inc : Android Interfaces and Architecture, 2015. URL https://source.android.com/devices/.

[46] Google Inc : Application Fundamentals, 2015. URL http://developer.android.com/guide/components/fundamentals.html.

[47] Google Inc : Application Security, 2015. URL https://www.google.com/about/appsecurity/android-rewards/.

[48] Google Inc : Managing Projects Overview, 2015. URL https://developer.android.com/tools/projects/index.html.

[49] Google Inc : platform/bionic.git - Git at Google, 2015. URL https://android.googlesource.com/platform/bionic.git.

[50] Google Inc : Security, 2015. URL https://source.android.com/devices/tech/security/index.html.

[51] Google Inc : System and kernel security, 2015. URL http://source.android.com/devices/tech/security/overview/kernel-security.html.

[52] Google Inc : System Permissions, 2015. URL http://developer.android.com/guide/topics/security/permissions.html.

[53] Google Inc : Android Developers, 2016. URL https://developer.android.com.

[54] Google Inc : Building Kernels, 2016. URL https://source.android.com/source/building-kernels.html.

[55] Google Inc : Security Enhancements, 2016. URL https://source.android.com/security/enhancements/index.html.

[56] Michael Grace, Yajin Zhou, Zhi Wang et Xuxian Jiang : Systematic Detection of Capability Leaks in Stock Android Smartphones. *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS 2012)*, 2012.

[57] John B Guptill : *Examining Application Components to Reveal Android Malware*. Air Force Institute of Technology, 2013. URL https://books.google.ca/books?id=Z2e9oAEACAAJ.

[58] Michael Hale Ligh, Andrew Case, Jamie Levy et AAron Walters : *The Art of Memory Forensics : Detecting Malware and Threats in Windows, Linux, and Mac Memory*. Wiley, Indianapolis, 1 édition, 2014. ISBN 978-1-118-82509-9.

[59] You Joung Ham, Daeyeol Moon, Hyung Woo Lee, Jae Deok Lim et Jeong Nyeo Kim : Android mobile application system call event pattern analysis for determination of malicious attack. *International Journal of Security and its Applications*, 8(1):231–246, 2014. ISSN 17389976.

[60] Hao Hao, Vicky Singh et Wenliang Du : On the Effectiveness of API-level Access Control Using Bytecode Rewriting in Android. *In Proceedings of the 8th ACM SIGSAC*

Symposium on Information, Computer and Communications Security, ASIA CCS '13, pages 25–36, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1767-2. URL http://doi.acm.org.acces.bibl.ulaval.ca/10.1145/2484313.2484317.

[61] Stephan Heuser, Adwait Nadkarni, William Enck et Ahmad-Reza Sadeghi : ASM : A Programmable Interface for Extending Android Security. *In 23rd USENIX Security Symposium (USENIX Security 14)*, pages 1005–1019, San Diego, CA, 2014. USENIX Association. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/heuser.

[62] Benjamin Holland, Tom Deering, Suresh Kothari, Jon Mathews et Nikhil Ranade : Security Toolbox for Detecting Novel and Sophisticated Android Malware. *CoRR*, abs/1504.0, 2015. URL http://arxiv.org/abs/1504.01693.

[63] George Hotz : towelroot by geohot, 2014. URL https://towelroot.com/.

[64] Chih-Wei Huang : Android-x86, 2016. URL http://www.android-x86.org/.

[65] Youn-Sik Jeong, Hwan-Taek Lee, Seong-Je Cho, Sangchul Han et Minkyu Park : A kernel-based monitoring approach for analyzing malicious behavior on Android. *Proceedings of the 29th Annual ACM Symposium on Applied Computing - SAC '14*, pages 1737–1738, 2014. URL http://dl.acm.org/citation.cfm?doid=2554850.2559915.

[66] JesusFreke : JesusFreke/smali : smali/baksmali, 2015. URL https://github.com/JesusFreke/smali.

[67] Shiju P. John : Android Architecture, 2015. URL http://www.eazytutz.com/android/android-architecture/.

[68] Kaspersky Lab et INTERPOL : Mobile Cyber Threats. Rapport technique, Kaspersky Lab, 2014. URL http://media.kaspersky.com/pdf/Kaspersky-Lab-KSN-Report-mobile-cyberthreats-web.pdf.

[69] Pallavi Kaushik et Amit Jain : DroidCheck : Android Malware Detection by Behavioral Techniques and Honeypot. *International Journal of Computer Science and Mobile Computing*, 4(4):829–834, 2015. URL http://ijcsmc.com/docs/papers/April2015/V4I4201599a74.pdf.

[70] Hong Kong, Patrick P F Chan et Lucas C K Hui : DroidChecker : Analyzing Android Applications for Capability Leak Categories and Subject Descriptors. *WISEC '12 Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 125–136, 2012.

[71] A Kovacheva : *Efficient Code Obfuscation for Android*. Thèse de doctorat, Université du Luxembourg, 2013. URL http://link.springer.com/chapter/10.1007/978-3-319-03783-7{_}10.

[72] Harry Kurniawan, Yusep Rosmansyah et Budiman Dabarsyah : Android anomaly detection system using machine learning classification. *In 2015 International Conference on Electrical Engineering and Informatics*, pages 288–293, Denpasar, Bali, Indonesia, aug 2015.

[73] Patrick Lam, Eric Bodden, Ondrej Lhotak et Laurie Hendren : The Soot framework for Java program analysis : a retrospective. *In Cetus '11*, 2011. URL https://sable.github.io/soot/resources/lblh11soot.pdf.

[74] Patrik Lantz et Bjorn Johansson : Towards bridging the gap between Dalvik bytecode and native code during static analysis of Android applications. *In IWCMC 2015 - 11th International Wireless Communications and Mobile Computing Conference*, pages 587–593, 2015. ISBN 9781479953448.

[75] Bernard Lebel et Jonathan Ouellet : Added unified outputs and additionnal output info useful in finding some rootkits on Android #224, 2015. URL https://github.com/volatilityfoundation/volatility/pull/224.

[76] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau et Patrick McDaniel : I know what leaked in your pocket : uncovering privacy leaks on Android Apps with Static Taint Analysis. *CoRR*, abs/1404.7(April 2014), 2014. URL http://arxiv.org/abs/1404.7431.

[77] Shuang Liang et Xiaojiang Du : Permission-combination-based scheme for Android mobile malware detection. *In 2014 IEEE International Conference on Communications, ICC 2014*, pages 2301–2306, 2014. ISBN 9781479920037.

[78] Anthony Lineberry : RageAgainstTheCage, 2010. URL http://dtors.org/2010/08/25/reversing-latest-exploid-release/.

[79] Peter Loscocco et Stephen Smalley : Integrating flexible support for security policies into the Linux operating system, 2001. URL http://dl.acm.org/citation.cfm?id=647054.

[80] Holger Macht : *Live Memory Forensics on Android with Volatility*. Ph.d. thesis, Friedrich-Alexander University Erlangen-Nuremberg, 2013. URL http://www1.informatik.uni-erlangen.de.

[81] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu et Giorgio Giacinto : Stealth attacks : An extended insight into the obfuscation effects on Android malware. *Computers & Security*, 51:16–31, jun 2015. ISSN 01674048. URL http://www.sciencedirect.com/science/article/pii/S016740481500022X.

[82] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon et Srdjan Capkun : Analysis of the communication between colluding applications on modern smartphones. *In Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 51–60, New York, NY, USA, 2012. ACM. ISBN 9781450313124. URL http://dl.acm.org/citation.cfm?doid=2420950.2420958.

[83] Luo Xu Min et Qing Hua Cao : Runtime-Based Behavior Dynamic Analysis System for Android Malware Detection. *In Information Technology Applications in Industry, Computer Engineering and Materials Science*, volume 756 de *Advanced Materials Research*, pages 2220–2225. Trans Tech Publications, 2013.

[84] Veelasha Moonsamy, Jia Rong et Shaowu Liu : Mining permission patterns for contrasting clean and malicious android applications. *Future Generation Computer Systems*, 36:122–132, 2014. ISSN 0167739X.

[85] A Moser, C Kruegel et E Kirda : Exploring Multiple Execution Paths for Malware Analysis. *In Security and Privacy, 2007. SP '07. IEEE Symposium on*, pages 231–245, 2007.

[86] Motive Security Labs : Motive Malware Report 2014 H2. Rapport technique, Alcatel-Lucent, 2014. URL https://resources.alcatel-lucent.com/asset/184652.

[87] MWR Labs : Drozer, 2015. URL https://labs.mwrinfosecurity.com/tools/drozer/.

[88] Faisal Nasim, Baber Aslam, Waseem Ahmed et Talha Naeem : Uncovering Self Code Modification in Android. *Codes, Cryptology, and Information Security : First International Conference, C2SI 2015*, 9084:297–313, 2015. URL http://dx.doi.org/10.1007/978-3-319-18681-8{_}24.

[89] Đurica Nikolić et Fausto Spoto : Reachability analysis of program variables. *ACM Transactions on Programming Languages and Systems*, 35(4):1–68, 2013. ISSN 01640925. URL http://dl.acm.org/citation.cfm?id=2560142.2529990.

[90] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha et Patrick Mcdaniel : Composite Constant Propagation : Application to Android Inter-Component Communication Analysis, 2015. URL http://siis.cse.psu.edu/pubs/octeau-icse15.pdf.

[91] Robert J Olipane : *Short Message Service (SMS) Command and Control (C2) Awareness in Android-based Smartphones Using Kernel-level Auditing*. Thèse de doctorat, Air University, Wright-Patterson Air Force Base, 2012. URL https://books.google.ca/books?id=pq5OMwEACAAJ.

[92] Oracle : VirtualBox, 2016. URL https://www.virtualbox.org/.

[93] Pancake : Radare2, 2016. URL http://radare.org/r/.

[94] Nick L. Petroni, AAron Walters, Timothy Fraser et William a. Arbaugh : FATKit : A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation*, 3(4):197–210, 2006. ISSN 17422876. URL http://www.sciencedirect.com/science/article/pii/S1742287606001228.

[95] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis et Sotiris Ioannidis : Rage Against the Virtual Machine : Hindering Dynamic Analysis of Android Malware. *In Proceedings of the Seventh European Workshop on System Security*, EuroSec '14, pages 5 :1—-5 :6, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2715-2. URL http://doi.acm.org.acces.bibl.ulaval.ca/10.1145/2592791.2592796.

[96] Ernesto Isaac Ramírez : *Static Data Flow Analysis for Vulnerable Permission Configurations of Android Applications*. Master's, Hamburg University of Technology (TUHH), 2015. URL http://www.sts.tu-harburg.de/pw-and-m-theses/2015/silva15.pdf.

[97] Rapid7 : Penetration Testing Software, 2016. URL https://www.metasploit.com/.

[98] Siegfried Rasthofer, Steven Arzt, Max Kolhagen, Brian Pfretzschner, Stephan Huber, Eric Bodden et Philipp Richter : DroidSearch : A Tool for Scaling Android App Triage to Real-World App Stores. *In 2015 Science and Information Conference (SAI)*, 2015. URL http://www.bodden.de/pubs/rak+15droidsearch.pdf.

[99] Vaibhav Rastogi, Yan Chen et Xuxian Jiang : Catch Me If You Can : Evaluating Android Anti-Malware Against Transformation Attacks. *Information Forensics and Security, IEEE Transactions on*, 9(1):99–108, jan 2014. ISSN 1556-6013.

[100] Wannes Rombouts : wapiflapi/veles : Visual reverse engineering tool., 2016. URL https://github.com/wapiflapi/veles.

[101] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang et Dongyan Xu : GUITAR : Piecing Together Android App GUIs from Memory Images. *CCS '15 Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security*, pages 39–41, 2015. ISSN 15437221.

[102] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Pablo Garcia Bringas et Gonzalo ? ?lvarez : PUMA : Permission usage to detect malware in android. *In Advances in Intelligent Systems and Computing*, volume 189 AISC, pages 289–298, 2013. ISBN 9783642330179.

[103] Suzanna Schmeelk, Junfeng Yang et Alfred Aho : Android Malware Static Analysis Techniques. *In CISR '15 Proceedings of the 10th Annual Cyber and Information Security Research Conference*, numéro 5 *in* CISR '15, pages 1–8, New York, NY, USA, 2015. ACM. ISBN 9781450333450. URL http://doi.acm.org/10.1145/2746266.2746271.

[104] Patrick Schulz : Dalvik Bytecode Obfuscation on Android, 2012. URL http://dexlabs.org/blog/bytecode-obfuscation.

[105] Asaf Shabtai : Securing Android-Powered Mobile Devices Using SELinux. *IEEE Security & Privacy*, 8(3):36–44, nov 2010. ISSN 1540-7993. URL http://doi.ieeecomputersociety.org/10.1109/MSP.2009.144.

[106] Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi Dolev et Chanan Glezer : Google Android : A Comprehensive Security Assessment. *IEEE Security and Privacy*, 8(2):35–44, mar 2010. ISSN 1540-7993. URL http://dx.doi.org/10.1109/MSP.2010.2.

[107] Skylot : skylot/jadx : Dex to Java decompiler, 2015. URL https://github.com/skylot/jadx.

[108] Secure software engineering : secure-software-engineering/DroidBench : A micro-benchmark suite to assess the stability of taint-analysis tools for Android, 2015. URL https://github.com/secure-software-engineering/DroidBench.

[109] Jason Solomon, Ewa Huebner, Derek Bem et Magdalena Szezynska : User data persistence in physical memory. *Digital Investigation*, 4(2):68–72, 2007. ISSN 17422876. URL http://www.sciencedirect.com/science/article/pii/S174228760700028X.

[110] S. Sparks et J. Butler : Shadow Walker - Raising The Bar For Rootkit Detection, 2005. URL http://blackhat.com/presentations/bh-usa-05/bh-us-05-sparks.pdf.

[111] Johannes Stüttgen et Michael Cohen : Robust Linux memory acquisition with minimal target impact. *Digital Investigation*, 11(0):S112–S119, 2014. ISSN 17422876. URL http://www.sciencedirect.com/science/article/pii/S174228761400019X.

[112] Mingshen Sun, Taofile :///home/shade/Desktop/Sun2016.pdf Wei et John C.S.Lui : TaintART : A Practical Multi-level Information-Flow Tracking System for Android RunTime. *In CCS*, pages 331–342, 2016. ISBN 9781450341394.

[113] Joe Sylve, Andrew Case, Lodovico Marziale et Golden G. Richard : Acquisition and analysis of volatile memory from android devices. *Digital Investigation*, 8(3-4):175–184, 2012. ISSN 17422876. URL http://www.sciencedirect.com/science/article/pii/S1742287611000879.

[114] Symantec : 2015 Internet Security Threat Report. Rapport technique, Symantec Corporation, Mountain View, CA, USA, 2015. URL https://www.symantec.com/security{_}response/publications/whitepapers.jsp.

[115] Symantec : 2016 Internet Security Threat Report. Rapport technique, Symantec Corporation, Mountain View, CA, USA, 2016.

[116] Kabakus Abdullah Talha, Dogru Ibrahim Alper et Cetin Aydin : APK Auditor : Permission-based Android malware detection system. *Digital Investigation*, 13:1–14, jun 2015. ISSN 17422876. URL http://www.sciencedirect.com/science/article/pii/S174228761500002X.

[117] Kimberly Tam, Salahuddin J Khan, Aristide Fattori et Lorenzo Cavallaro : CopperDroid : Automatic Reconstruction of Android Malware Behaviors. *In 22nd Annual Network and Distributed System Security Symposium (NDSS)*, numéro To Appear *in* NDSS, San Diego, CA, 2015. ISBN 189156238X.

[118] Vrizlynn L L Thing, Kian Yong Ng et Ee Chien Chang : Live memory forensics of mobile phones. *Digital Investigation*, 7(SUPPL.):S74 – S82, 2010. ISSN 17422876. URL http://www.sciencedirect.com/science/article/pii/S174228761000037X.

[119] Timrae : timrae/custom-class-loader : Custom class loading in Dalvik (Android Studio Version), 2014. URL https://github.com/timrae/custom-class-loader.

[120] Timwr : timwr/CVE-2014-3153 aka towelroot, 2016. URL https://github.com/timwr/CVE-2014-3153.

[121] Timothy Vidas : The Acquisition and Analysis of Random Access Memory. *Journal of Digital Forensic Practice*, 1(4):315–323, 2007. ISSN 1556-7281. URL http://dx.doi.org/10.1080/15567280701418171.

[122] Timothy Vidas et Nicolas Christin : Evading Android Runtime Analysis via Sandbox Detection. *In Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 447–458, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2800-5. URL http://doi.acm.org/10.1145/2590296.2590325.

[123] VMware Inc : VMware Workstation Player, 2016. URL https://www.vmware.com.

[124] VolatilityFoundation : Linux volatilityfoundation/volatility Wiki, 2016. URL https://github.com/volatilityfoundation/volatility/wiki/Linux.

[125] Stefan Vömel et Felix C. Freiling : A survey of main memory acquisition and analysis techniques for the windows operating system. *Digital Investigation*, 8(1):3–22, 2011. ISSN 17422876. URL http://www.sciencedirect.com/science/article/pii/S1742287611000508.

[126] Ryszard Wiśniewski et Connor Tumbleson : APKTool - A tool for reverse engineering Android apk files, 2015. URL https://ibotpeaches.github.io/Apktool/.

[127] Gregory Wroblewski : General Method of Program Code Obfuscation. *In International Conference on Software Engineering Research and Practice*, page 8, Wroclaw, Poland, 2002.

[128] Daoyuan Wu, Xiapu Luo et Rocky K. C. Chang : A Sink-driven Approach to Detecting Exposed Component Vulnerabilities in Android Apps. *arXiv preprint arXiv :1405.6282*, 2014. URL http://arxiv.org/abs/1405.6282.

[129] Tim Wyatt : Inside the Android Security Patch Lifecycle | Lookout Blog, 2011. URL https://blog.lookout.com/blog/2011/08/04/inside-the-android-security-patch-lifecycle/.

[130] Lk Yan et H Yin : Droidscope : seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. *Proceedings of the 21st USENIX Security Symposium*, page 29, 2012. URL https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final107.pdf{%}5Cnhttp://dl.acm.org/citation.cfm?id=2362793.2362822.

[131] Hongyu Yang et Ruiwen Tang : Power Consumption Based Android Malware Detection. *J. Electrical and Computer Engineering*, 2016:6, 2016. ISSN 2090-0147. URL http://dx.doi.org/10.1155/2016/6860217.

[132] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang et Atanas Rountev : Static Control-Flow Analysis of User-Driven Callbacks in Android Applications, 2015. URL http://web.cse.ohio-state.edu/{~}rountev/presto/pubs/icse15.pdf.

[133] M Zaman, T Siddiqui, M R Amin et M S Hossain : Malware detection in Android by network traffic analysis. *In Networking Systems and Security (NSysS), 2015 International Conference on*, pages 1–5, Dhaka, Bangladesh, jan 2015. IEEE. URL http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=7043530.

[134] Dany Zatuchna : Escalating Futex, 2014. URL https://www.appdome.com/blog/escalating-futex.

[135] Dany Zatuchna : Pwning the kernel && root, 2014. URL https://www.appdome.com/blog/pwning-the-kernel-root.

[136] Dany Zatuchna : The Futex Vulnerability, 2014. URL https://www.appdome.com/blog/the-futex-vulnerability.

[137] Lei Zhang, Lianhai Wang, Ruichao Zhang, Shuhui Zhang et Yang Zhou : Live memory acquisition through firewire. *In Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering*, volume 56, pages 159–167, 2011. ISBN 9783642236013.

[138] Yajin Zhou et Xuxian Jiang : Dissecting Android Malware : Characterization and Evolution. *In Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109, may 2012.

[139] ZLabs : The Latest on Stagefright : CVE-2015-1538 Exploit is Now Available for Testing Purposes, 2015. URL https://blog.zimperium.com/.