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

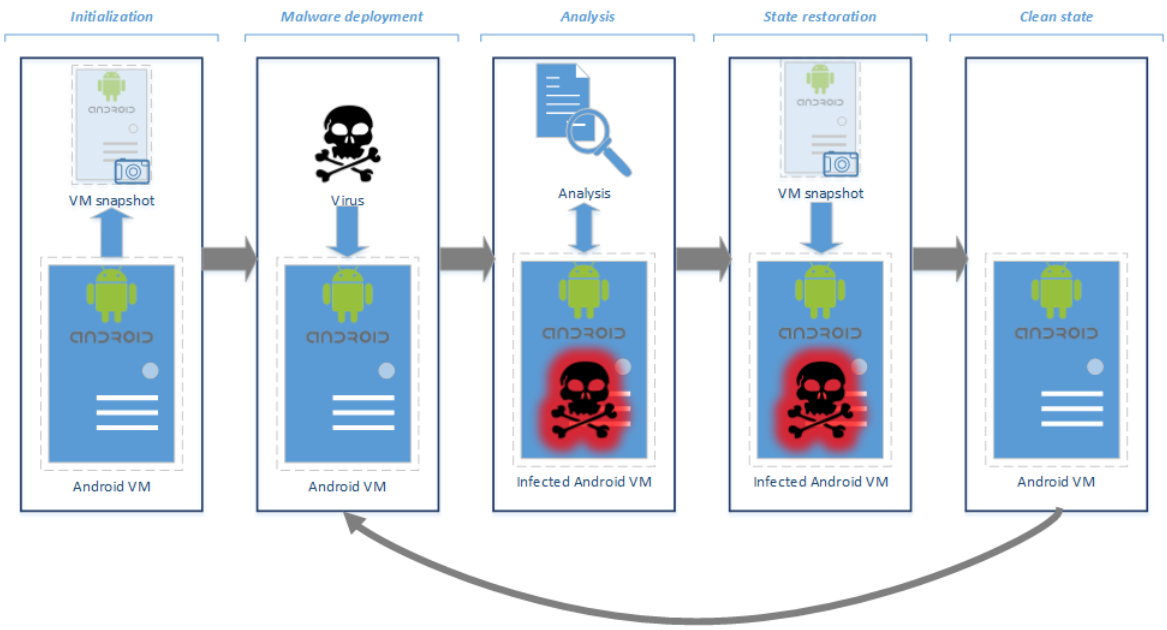

FIGURE 2.1 – Analyse de maliciels à l'aide d'une MV.

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.

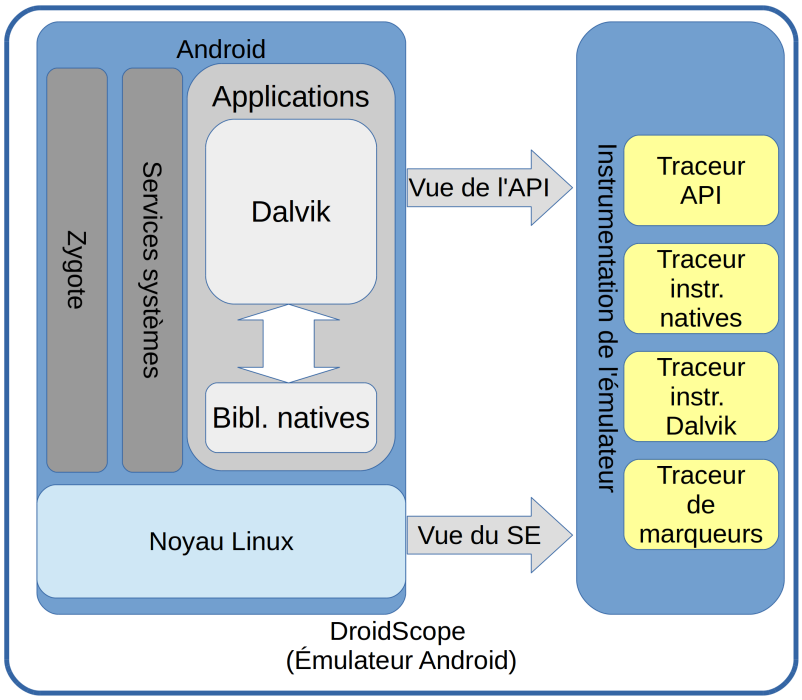

FIGURE 2.2 – Architecture de DroidScope.

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.