Vers une détection automatique des applications malveillantes dans les environnements Android

Eric Finickel
Université de Lorraine, Loria
Vandoeuvre-lès-Nancy
F-54506, France
eric.finickel@loria.fr

Abdelkader Lahmadi
Université de Lorraine, Loria
Vandoeuvre-lès-Nancy
F-54506, France
abdelkader.lahmadi@loria.fr

Olivier Festor
Inria
Villers-lès-Nancy
F-54600, France
olivier.festor@inria.fr

ABSTRACT

Dans ce papier, nous présentons l'état de l'art sur les attaques et les menaces dans les environnements Android ainsi que les approches de détection associées. La plupart de ces approches utilisent des informations obtenues par instrumentation de la machine virtuelle ou par rétro-ingénierie du *bytecode* des applications. Nous proposons ainsi une nouvelle méthode moins coûteuse qui repose sur l'analyse des journaux des événements applicatifs et systèmes générés par la plate-forme Android. Cette analyse nous permettra d'établir des signatures des applications Android associant leurs structures et leurs comportements dynamiques.

In this paper, we present the state of the art regarding attacks and threads in Android environment and their associated detection methods. Most of the existing approaches rely on information obtained using the instrumentation or the reverse-engineering of the bytecode of the analyzed applications. We present therefore a novel approach relying on the analysis of system and applications logs generated by the Android platform. Our developed analysis tool allows us to establish fingerprints of Android applications associating their execution structure and dynamics.

Keywords

Android, Applications, Logs , Analyse, Sécurité

1. INTRODUCTION

L'environnement Android est devenu le système d'exploitation de première importance pour un nombre considérable d'équipements, notamment les smartphones et les tablettes ainsi que des équipements futurs. Cet environnent offre aux usagers une multitude d'applications à télécharger depuis le *Market* officiel fourni par Google ou des *Markets* alternatifs. Ces applications peuvent être de toutes sortes afin de se divertir ou d'aider les usagers. Le comportement de ces applications et leur impact sur les données privées des usagers, ainsi que les fonctions sensibles associées aux capteurs et la téléphonie, n'est pas vérifié par les fournisseurs de ces plate-formes de téléchargement. Seul l'usager a le choix d'accepter ou non au moment de l'installation de l'application, les permissions associées pour lui garantir l'accès à ces fonctions ou aux données sensibles. L'usager est souvent laissé à son bon sens pour faire ce choix. En revanche, les usagers sont de plus en plus exposés à des attaques ciblant les environnements Android. Récemment, deux projets ont publié des listes d'applications malveillantes ciblant l'environnement Android. Il s'agit du projet *malware* Genome Projet qui a publié 1260 applications et du projet VirusTotal qui en a publié récemment 20 000. Des attaques de sécurité peuvent être lancées par le biais de ces applications pour plusieurs raisons. Premièrement, lors de la soumission d'une application au Market officiel ou aux Markets alternatifs, aucun contrôle stricte n'est effectué. Deuxièmement, une application doit être signée avant d'être soumise. Cependant, les applications peuvent être signées à l'aide d'un certificat auto-signé et cela ne respecte donc pas la chaîne de certification. Donc n'importe quelle application peut être diffusée sur les différentes plate-formes de téléchargement.

Il existe plusieurs outils permettant d'analyser et de détecter les différents *malwares* ciblant les équipements mobiles. On trouve essentiellement les outils classiques d'anti-virus basés sur un mécanisme de signature de la même manière que sur les PCs avec par exemple *Antivirus Free*, *Lookout Security & Antivirus* et *Norton Mobile Security Lite*. Cependant il a été montré [15] que ces outils sont peu efficaces pour détecter des *malwares* inconnus ou avec des signatures différentes de celles présentes dans les bases de détection. Plusieurs travaux de recherche [4, 13, 7, 16] ont proposé d'autres approches pour détecter des *malwares* Android en s'appuyant essentiellement sur l'analyse des permissions sollicitées par les applications, l'instrumentation du système pour analyser les appels aux fonctions sensibles ou la rétro-ingénierie du *bytecode* des applications. En revanche, ces approches malgré, leur efficacité en terme de détection, reposent sur des mécanismes coûteux et intrusifs pour collecter des informations caractérisant les applications Android.

Dans ce travail, notre objectif est de présenter une nouvelle méthode basée sur les traces d'exécution fournies par l'environnement Android pour inférer la structure et d'établir un modèle comportemental de ses applications. Ces modèles nous permettront de construire des signatures des applications malveillantes.

L'article est organisé de la façon suivante : en section 2 nous présentons l'environnement Android, son architecture et son fonctionnement. Nous détaillons aussi les menaces de sécurité ainsi que les vecteurs d'attaques associés aux *malwares* ciblant cet environnement. Dans la section 3, nous étudions les différentes approches existantes pour détecter ces *malwares*. Dans la section 4, nous détaillons notre approche basée sur l'analyse des logs pour générer des signatures de ces *malwares*. Nous concluons et nous présentons les perspectives dans la section 5.

2. LES ENVIRONNEMENTS ANDROID

L'environnement Android comme l'indique la figure 1 est construit au dessus d'un noyau Linux modifié pour gérer les ressources systèmes et le matériel de l'équipement mobile. Les services systèmes, les applications natives et les applications Java exécutées dans cet environnement reposent sur des processus Linux. À chaque application installée est associée un unique identifiant utilisateur (userId) et un ensemble d'identifiants de groupes (gids) correspondant aux permissions demandées par l'application. Ces deux paramètres sont utilisés pour contrôler l'accès aux ressources du système. Chaque application Android est exécutée dans sa propre machine virtuelle Java Dalvik optimisée pour les équipements mobiles afin de réduire sa consommation mémoire et obtenir une meilleure isolation entre les applications exécutées sur l'équipement.

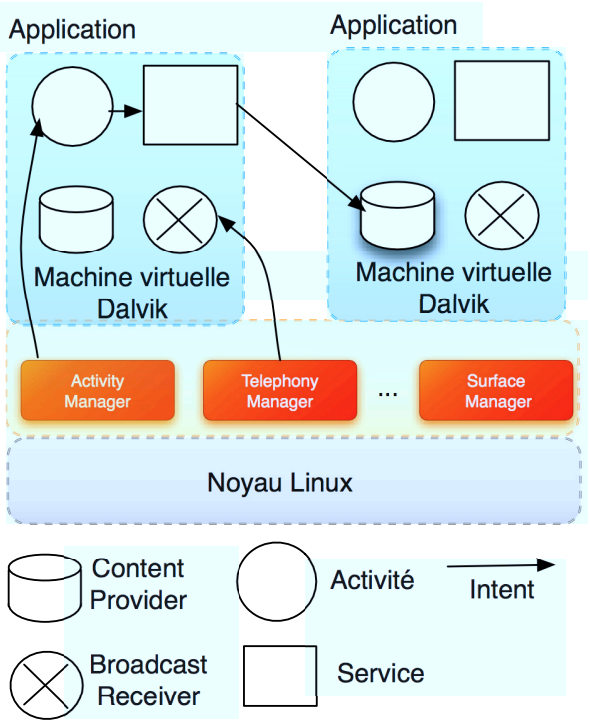

Figure 1: Architecture générale du framework Android d'après [11].

Une application Android repose sur différents composants natifs ou Java. Les composants Java sont essentiellement les activités, les services, les *broadcastReceivers* et les *content providers*. Ces composant communiquent entre eux et avec le système en utilisant des messages spécifiques nommés les Intents. Il s'agit des communications inter-processus spécifiques à la plate-forme Android. Les composants d'activités sont définis pour construire des interfaces utilisateurs. Les composants de services sont des tâches en arrière plan s'exécutant sans interface utilisateur. Ces derniers sont instanciés soit par un composant d'activité soit par un Broadcast Receiver via le mécanisme d'Intent. Les composants de gestionnaire de contenu sont des bases de données utilisées pour fournir un contenu spécifique. Les Broadcast Receiver sont utilisés pour recevoir les Intents de broadcast envoyés par le système ou d'autres applications. Les Intents peuvent être utilisés par tous les composants, sauf les gestionnaires de contenu, afin d'activer, de désactiver ou d'échanger des données entre ces composants. Les composants natifs sont des bibliothèques partagées et chargées dynamiquement à l'exécution. Ainsi, un composant écrit en java est d'abord compilé en bytecode java (.class) pour ensuite être converti en bytecode Dalvik (.dex). Le code natif peut aussi être compilé pour obtenir les bibliothèques partagées (.so) grâce au support JNI qui permet de faciliter les communications entre les composants java et natif. Un fichier Manifest contenant la liste des permissions et les composants de l'application est défini par le développeur. Cette liste de permissions doit être acceptée par l'utilisateur pour pouvoir installer l'application. Les fichiers dex, les librairies partagées, le fichier manifest et les différentes autres ressources sont utilisés pour créer une archive d'extension *apk* qui servira pour la distribution et l'installation de l'application sur les équipements.

2.1 Les menaces

Les *malwares* Android ont introduit notamment quatre grands types de menaces qui pèsent sur les usagers de cet environnement : le Financial Charging, la collecte et l'utilisation d'informations privées, l'utilisation abusive des ressources de l'équipement et les actions nuisibles.

Pertes financières. C'est une action destinée à faire perdre de l'argent au profit de l'attaquant en appelant ou en envoyant des SMS vers des numéros surtaxés. Les messages surtaxés sont moins facilement détectables par l'usager pour deux raisons. Premièrement, la permission *sendTextMessage* d'Android permet d'envoyer les messages en arrière plan sans notifier l'utilisateur. Ce comportement a été observé dans le *malware* GGTracker [15]. Deuxièmement, les appels surtaxés gardent la communication ouverte [6] et ainsi l'usager pourrait détecter le comportement malveillant de l'application émettrice de l'appel. Le numéro surtaxé utilisé peut être codé soit en dur dans l'application, soit chargé au lancement de l'application en communiquant avec un serveur distant [15]. Certains *malwares* peuvent aussi envoyer des SMS à d'autres téléphones impliquant des conséquences financières dans le cas où l'abonnement téléphonique de l'utilisateur ne supporte pas les SMS illimités. Par exemple, le *malware* DogWars envoie des SMS à tous les numéros du répertoire d'un téléphone cible [15].

La collecte et l'utilisation des informations privées. Certaines attaques ciblent les contenus des SMS, des MMS, des mails, le journal d'appel, les identifiants des comptes utilisateurs, les détails des contacts et les informations stockées par les applications [5]. Par exemple, l'application FakeNetflix

tente de subtiliser les comptes et les mots de passe des utilisateurs [15]. Un attaquant est capable d'écouter les conversations téléphoniques des usagers en utilisant l'enregistrement vocal, prendre des photos et connaître en temps réel sa localisation géographique grâce au GPS [5]. Bickford et O'Hare [3] ont implanté un *rootkit* permettant d'écouter ou d'enregistrer une conversation confidentielle.
Les graywares [6] sont des applications légitimes qui collectent les données privées pour établir des profils commerciaux des usagers.
La liste des contacts d'un utilisateur s'avère très utile pour envoyer des spams et des liens de *phising*. Les spammers utilisent des *malwares* pour diffuser leurs publicités car dans la plupart des pays, l'envoi de SMS pour ce type d'opération est illégal. Les *malwares* permettent ainsi de réduire le risque pour le spammer car la provenance des spams est plus difficile à détecter.

L'Utilisation des ressources de l'équipement. Certains attaquants visent à exploiter les ressources de calcul de l'équipement ainsi que sa connectivité réseau pour déployer des *botnets* [5]. Des applications malveillantes sont utilisées pour envoyer des requêtes vers des moteurs de recherches afin d'augmenter les rangs de certains sites web. Ils effectuent ensuite une consultation du site dont l'on souhaite augmenter le rang. ADRD est un exemple de *malware* qui exploite l'accès réseau d'un smartphone Android pour générer un nombre important de requêtes afin d'optimiser les rangs de certains sites web au niveau des moteurs de recherche [6].

Le déni de services. Ces sont des attaques qui visent à générer une gêne temporaire plutôt que d'effectuer des actions malveillantes. Bien que de telles attaques soient plus facilement détectables, elles essayent avant tout de provoquer le plus de dégât possible, comme consommer la batterie, générer un trafic réseau considérable ou même dans le cas extrême, désactiver l'appareil de façon permanente [5]. Certains *malwares* provoquent une gêne sans conséquences majeures, par exemple en modifiant le fond d'écran de l'appareil, la sonnerie, etc [6]. Bickford et O'Hare [3] ont aussi implanté un *rootkit* permettant de vider la batterie d'un smartphone en activant des fonctionnalités gourmandes en ressources, comme le GPS, le bluetooth ou le WIFI.

2.2 Les vecteurs d'attaques

Depuis que les plate-formes mobiles ressemblent de plus en plus aux systèmes PCs, les vecteurs d'attaques classiques y ont migré [5].
Les chevaux de Troie sont aussi utilisés sur le smartphone pour prendre le contrôle total d'un équipement. C'est un moyen utilisé pour obtenir des informations privées ou installer d'autres applications malicieuses comme des *botnets* ou des vers. Ils sont utilisés pour transmettre des informations obtenues par le biais du *phising*.
Les *botnets* [9] sont un ensemble d'équipements compromis qui peuvent être contrôlés et coordonnés à distances grâce aux serveurs C&C. Les messages C&C sont transmis via SMS [8]. Ce type de *malware* est utilisé pour effectuer des attaques DDoS. Une fois le *malware* installé, il est possible de prendre le contrôle du téléphone à distance en utilisant le protocole HTTP. Les URLs des serveurs C&C peuvent être chiffrés ainsi que les communications. Par exemple, l'application DroidKungFu utilise le chiffrement AES et l'application Geimini utilise un chiffrement DES. Les serveurs C&C peuvent être enregistrés dans des domaines contrôlés par l'attaquant ou ils peuvent être hébergés des services de cloud.
Les vers sont des applications malicieuses auto-répliquantes qui se répandent automatiquement sur les systèmes non-infectés. Par exemple le vers Ikee qui change le fond d'écran sur iphone.
Les *rootkits* sont des *malwares* qui modifient furtivement le code de l'OS et les données pour atteindre des objectifs malveillants. Un *rootkit* permettant de vider la batterie d'un smartphone a été développé par Bickford et O'Hare [3].

Plusieurs moyens sont utilisés afin d'atteindre et d'infecter une plate-forme mobile. Premièrement, les services de réseau mobile [5] avec les SMS, les MMS et les appels vocaux sont utilisés pour délivrer du contenu malveillant. Deuxièmement, les équipements mobiles sont connectés en permanence à Internet [5]. Ensuite, la connectivité bluetooth [5] permettant la diffusion de proche en proche. Une fois qu'une connexion est établie entre deux équipements, celui qui a été compromis peut envoyer son contenu malveillant à destination de l'autre. Cependant, ce vecteur d'attaques est limité car d'une part en règle générale la période pendant laquelle un appareil cherche à détecter les appareils susceptibles de se connecter à lui est faible. D'autre part, l'utilisateur doit confirmer un transfert de fichier que ce soit avec ou sans mot de passe.

Plus spécifique au système Android, différentes techniques sont utilisées afin d'installer les applications malicieuses sur l'équipement d'un usager à savoir : le repackaging, l'update attack, le drive-by download [15].

Le repackaging. L'auteur télécharge une application, la désassemble, y ajoute du code malicieux, la ré-assemble afin de déposer la nouvelle application sur un Market. De plus, pour essayer de cacher le contenu malicieux, l'auteur peut tenter d'utiliser des noms de classes semblant légitimes. Par exemple, AnserverBot utilise *com.sec.android.provider.drm* comme nom de package de son code malicieux.
D'autres *malwares* n'effectuent pas de transformation mais imitent des applications légitimes pour réaliser des actions malveillantes. Par exemple, le *malware* FakeNetFlix dérobe le compte et le mot de passe NetFlix de l'utilisateur.

L'update attack. C'est une technique plus difficile à détecter. Au lieu de contenir un payload directement malicieux, L'application possède un composant de mise à jour qui charge ou télécharge le code malicieux. Ce code peut être ajouté grâce au repackaging. Par exemple, les *malwares* DroidKungFuUpdate proposent une mise à jour à l'utilisateur permettant d'installer via Internet une nouvelle application contenant du code malicieux.

Le Drive-by download. C'est une technique traditionnelle déjà utilisée dans les attaques sur les PCs. Elle propose aux utilisateurs de télécharger des applications censées être utiles

mais qui sont en effet des *malwares*. Par exemple, GGTracker propose à l'utilisateur via une publicité, une application d'analyse de la consommation de la batterie. En réalité, cette dernière application souscrit à un service surtaxé.

La complexité de l'environnement Android dûe au couplage du noyau Linux avec l'ensemble de ses 90 bibliothèques embarquées, a introduit des vulnérabilités au niveau du système qui peuvent être exploitées pour augmenter les privilèges d'une application et dans le meilleur des cas devenir root [15]. Nous pouvons citer par exemple les exploits RATC ou exploid. D'autres *malwares* comme DroidKungFu n'embarquent pas directement d'exploit. Ils sont plutôt chiffrés et stockés comme des ressources ou d'autres fichiers pour être utilisés au lancement de l'application. Ce qui pose plus de problèmes de détection.

3. APPROCHES EXISTANTES

Il y a deux grands types d'approches permettant la détection des *malwares*. La première est l'analyse statique qui porte sur la rétro-ingénierie du *bytecode* de l'application afin d'analyser son code source. La deuxième est l'analyse dynamique en s'appuyant sur les données collectées par instrumentation de la machine virtuelle Dalvik ou le noyau Linux.

3.1 Analyse statique

Batyuk et Herpich [2] ont proposé un service capable d'accéder aux applications du *Market* Android et de fournir un rapport à l'utilisateur via une analyse statique révélant les potentiels menaces de sécurité et de violation de la vie privée. Cette analyse est effectuée en quatre étapes. Premièrement, un utilisateur effectue une requête de rapport pour une application. Ensuite, l'application subit un reverse-engineering en utilisant l'outil apktool pour obtenir le contenu du fichier Manifest.xml initial de l'application ainsi que son code Java obtenu après décompilation des fichiers smali. Des opérations d'analyse sont effectuées par une ensemble d'algorithme de détection en réalisant du pattern matching des fonctions sensibles appelées dans le code. Enfin un rapport de sécurité est créé et fourni à l'utilisateur pour l'informer des fuites de données possibles. Ils proposent aussi un moyen de diminuer les menaces en faisant une correspondance avec des patterns malicieux et non désiré au niveau du code source de l'application, par un reverse-engineering automatique, avant une recréation de l'application selon les préférences de sécurité proposées par l'utilisateur. Ces préférences représentent un ensemble de détecteurs qui eux mêmes sont associés à une modification du code et sera effectué si l'application correspond au détecteur.

3.2 Analyse dynamique

Burguera et Iker [4] ont conçu une approche de détection basée sur une technique de crowdsourcing afin d'obtenir les traces des appels systèmes d'une application depuis le noyau Linux. Leur approche s'appuie sur trois composants. Le composant d'acquisition de données est responsable d'obtenir les données via l'application Crowdroid installée sur les équipements des usagers. Une donnée peut être une information basique du téléphone, les liste des applications installées ou des traces d'exécution. Le deuxième composant analyse les données collectées, stocke les informations basiques et crée des vecteurs caractéristiques. Le troisième composant de détection de *malware* utilise les vecteurs caractéristiques pour créer des modèles de référence. Ces modèles sont utilisés pour détecter des comportements anormaux en utilisant des algorithmes de classification de type *k-means clustering*.

Isohara et Takamasa [10] ont proposé une approche d'analyse de comportement des applications Android. Ils utilisent principalement deux composants. Un collecteur de log réalisé sur le smartphone pour enregistrer tous les appels systèmes et filtres les événements liés à l'application ciblée. Les logs générés par une application sont collectés sur deux niveaux différents. Les logs au niveau application qui sont générés par les applications elles mêmes et les logs niveaux noyau qui sont générés par le système d'exploitation. Au niveau application, ils permettent de reconstituer son comportement avec précision. En revanche, ces logs peuvent être évités assez facilement par une application malicieuse. Au niveau noyau, ces logs permettent de reconstituer le comportement avec moins de précision mais ne peuvent pas être évités. L'analyse des applications est réalisée sur un serveur après avoir sélectionné les lignes de logs utiles pour une application en particulier et qui correspondent à des signatures décrites par des mots clés et des expressions régulières servant à détecter celles malicieuses. Les signatures permettant d'identifier les fuites d'informations sont générées automatiquement en utilisant l'identifiant du smatphone, le numéro de téléphone, etc. L'inconvénient de leur approche est d'avoir un ensemble de règles trop ou pas assez restrictif générant des signatures pouvant révéler des applications comme des faux positifs ou respectivement des faux négatifs.

DroidScope [13] est une plate-forme d'analyse de *malware* dans un environnement virtualisé. Cette technique possède deux avantages. Elle est totalement virtuelle, cela permet d'observer la plupart des attaques sur le noyau système et c'est difficile pour un attaquant de d'arrêter l'analyse.
Le système va construire à la fois la sémantique au niveau OS afin de comprendre les activités d'un *malware* et ses composants natifs et la sémantique au niveau Java pour comprendre le comportement des composants java.
Pour cela, quatre outils sont utilisés. L'analyseur natif et l'analyseur des instructions de la machine virtuelle Dalvik permettent d'observer le comportement du *malware* en enregistrant des traces d'instructions détaillées. L'analyseur des APIs fournit une vue haut niveau des interactions d'une application ou ses composants avec le système permet la surveillance des activités d'un *malware* au niveau API pour savoir comment il communique avec l'environnement : composant java avec le framework ; composants natifs avec le système Linux ; composants natifs interagissent avec les composants java ; et comment ces deux derniers communiquent avec la JNI. Le Taint Tracker permet l'analyse de la propagation des données dans la machine virtuelle Dalvik en utilisant des marqueurs. Cette analyse permet de savoir comme le *malware* obtient et fait fuir des informations.

3.3 Analyse hybride

Zhou et Wu [14] ont conçu un système appelé DroidRanger capable de détecter des applications appartenant à des familles de *malwares* connues et à des familles de *malwares* inconnues. Le système repose sur deux schémas d'analyse.
Un premier schéma d'empreintes comportementales basées sur les permissions est utilisé pour détecter les applications appartenant à des familles de *malwares* connus. Il filtre les

applications selon les permissions requises par les familles de *malwares* connus (matching avec permissions choisi par leur soin et contenu dans le fichier manifest des apps). Le modèle des signatures comportementales comprend des informations sémantiques contenues dans le fichier manifest, les méthodes appelées et la structure de l'application. Les signatures créées seront ensuite évaluées avec les signatures de *malwares* connus.
Un deuxième schéma est utilisé pour identifier des *malwares* appartenant à des familles inconnues. Ce dernier schéma applique des heuristiques de filtrage en chargeant dynamiquement le code et le code natif utilisé. Puis en effectuant une analyse basée sur la surveillance dynamique de l'exécution. Cela permet d'identifier la nature malveillante ou non de l'application. Ensuite l'empreinte est créée et ajoutée à la base de donnée qui sera utilisée par le premier schéma.

Grace et Michael [7] proposent un schéma proactif pour repérer les *malwares* inconnus (zero-day) et implémenté dans un système appelé RiskRanger. Ceci sans utiliser des signatures de *malwares* connues pour évaluer les risques potentiels de sécurité (divisés en 3 catégories) des applications et dresser une liste d'applications pouvant être dangereuses méritant plus d'analyses.
Il y a deux niveaux d'analyse de risques. Le premier niveau permet d'identifier les applications non offusquées présentant un risque haut ou moyen. Un risque haut exploite des vulnérabilités logicielles au niveau de la plate-forme. Un risque moyen n'exploite pas ces vulnérabilités mais peut causer des pertes financières ou des fuites d'informations sensibles. Par exemple, pour détecter les applications à hauts risques, ils utilisent des signatures créées pour chaque famille de vulnérabilités connues pour identifier leurs caractéristiques essentielles. Les applications à analyser sont celles qui présentent uniquement du code natif. Le deuxième niveau permet d'identifier les applications avec des comportements suspects. Cela concerne principalement des applications de cryptage permettant à d'autres applications de ne pas être détectées par le premier niveau d'analyse de risque ou des applications de chargement dynamique de code. Il arrive que le système génère des faux négatifs. Soit les *malwares* comprennent des comportements malicieux que le système n'arrive pas à détecter car ils ne comportent pas de risques ni haut, ni moyen. Soit ils ne partagent pas le même payload malicieux que les autres dans les mêmes familles de *malwares*. Soit ils sont "coupables par association".

Ces approches utilisent des mécanismes de collecte de données pour détecter les applications malveillantes sont difficiles à mettre en œuvre à cause de l'offuscation du code pour les approches statiques et l'évolution continue des environnements Android pour les approches dynamiques. Nous nous sommes donc focalisés sur une approche plus légère basée sur l'analyse des logs générés par l'environnement Android au cours de l'exécution des applications installées.
Notre approche permet d'analyser une application en fonction d'une structure recomposée partiellement grâce aux journaux de logs et d'un comportement élaboré aussi bien au niveau des permissions attribuées qu'au niveau des interactions entre les différents composants.
Afin d'obtenir les informations comportementales concernant une application, nous extrayons des informations directement d'une image de l'état du système. Ainsi, contrairement aux approches classiques d'analyses statiques, nous obtenons les permissions attribuées aux applications sans rétro-ingénierie pour obtenir le fichier manifest.xml contenant les permissions requises.
Nous pouvons créer des schémas d'exécution d'applications en corrélant les informations comportementales au sens interaction entre composants et les informations structurelles. De plus, ces informations sont obtenues d'une manière moins lourde que celle proposée par les approches dynamique avec l'instrumentation de la machine virtuelle.

4. ANALYSE DES LOGS ET EXTRACTION DES SIGNATURES

La détection des *malware* nécessite l'obtention des traces d'exécution détaillées de tous les événements ayant lieu sur les équipements Android. Ces traces d'exécution sont obtenues par la collecte des logs générés par l'environnement Android. Comme l'indique la figure 2 nous avons mis en œuvre une sonde embarquée pour collecter et d'envoyer les logs vers un serveur distant pour être stockés et analysés. Cette sonde s'appuie sur l'outil *logcat* pour collecter périodiquement les logs. Un outil est ensuite exécuté sur le serveur pour découper les données et de les insérer dans une base de données. Nous pouvons ainsi procéder à l'extraction des informations relatives aux structures et aux comportements des applications exécutées sur l'équipement d'un usager.

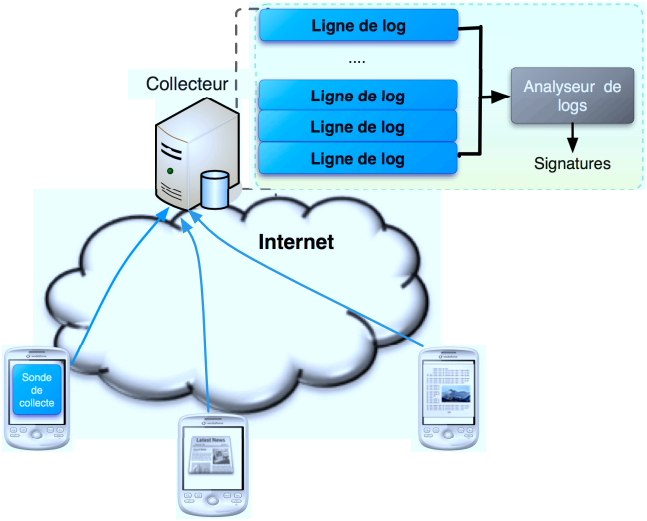

Figure 2: Architecture globale de notre approche.

4.1 Comportement structurel

Nous sommes en mesure de spécifier la structure d'une application Android en extrayant les lignes de logs issus de sonde de collecte, relatifs à l'exécution de ses services et ses activités. Parmi les actions possibles sur les activités ou les services, on retrouve la création, la suppression, la reprise, la suspension, etc. Ces actions présentent chacune des paramètres qui vont nous servir à identifier le nom de l'activité ou du service concerné. Par exemple, pour l'action de création de service am_create_service, on retrouve les paramètres Intent, Name, PID et Service Record. Le paramètre Name identifie donc le nom du service qui a été créé.

À titre d'exemple, nous avons extrait une partie des logs

```
ROW                    COLUMN+CELL
com.evernote1907750    column=component:, timestamp=1365580639682,value=am_create_activity
com.evernote1907750    column=params:Action, timestamp=1365580639687,value=android.intent.action.MAIN
com.evernote1907750    column=params:Component Name, timestamp=1365580639686,value=com.evernote/.ui.HomeActivity
com.evernote1907750    column=params:Flags, timestamp=1365580639690,value=270532608
com.evernote1907750    column=params:MIME Type, timestamp=1365580639688,value=NULL
com.evernote1907750    column=params:Task ID, timestamp=1365580639685,value=28
com.evernote1907750    column=params:Token, timestamp=1365580639684,value=1087427576
com.evernote1907750    column=params:URI, timestamp=1365580639689,value=NULL
com.evernote1907755    column=component:, timestamp=1365580639699,value=am_restart_activity
com.evernote1907755    column=params:Component Name, timestamp=1365580639702,value=com.evernote/.ui.HomeActivity
com.evernote1907755    column=params:Task ID, timestamp=1365580639701,value=28
com.evernote1907755    column=params:Token, timestamp=1365580639700,value=1087427576
com.evernote1907765    column=component:, timestamp=1365580639800,value=am_create_service
com.evernote1907765    column=params:Intent, timestamp=1365580639803,value=act=com.evernote.action.LOG_IN_PREP
com.evernote1907765    column=params:Name, timestamp=1365580639802,value=com.evernote/.client.EvernoteService
com.evernote1907765    column=params:PID, timestamp=1365580639804,value=4908
com.evernote1907765    column=params:Service Record, timestamp=1365580639801,value=1084600144
com.evernote1907770    column=component:, timestamp=1365580639824,value=am_destroy_activity
com.evernote1907770    column=params:Component Name, timestamp=1365580639827,value=com.evernote/.ui.HomeActivity
com.evernote1907770    column=params:Task ID, timestamp=1365580639826,value=28
com.evernote1907770    column=params:Token, timestamp=1365580639825,value=1087427576
com.evernote1907772    column=component:, timestamp=1365580639836,value=am_destroy_service
com.evernote1907772    column=params:Name, timestamp=1365580639838,value=com.evernote/.client.EvernoteService
com.evernote1907772    column=params:PID, timestamp=1365580639839,value=4908
```

Figure 3: Exemple de traces d'exécutions de l'application Android Evernote.

(figure 3) générés au lancement de l'application Evernote. La première colonne sert à identifier les différentes lignes de logs et dans la deuxième colonne, on retrouve les différents champs contenus dans une ligne de log. Ainsi, la première ligne de log représentée par les 8 premiers enregistrements (identifiés par com.evernote1907750), nous indique le type d'opération effectuée à savoir la création d'une activité nommée com.evernote/.ui.HomeActivity. En analysant ces différentes lignes, nous pouvons observer que l'application possède l'activité com.evernote/.ui.HomeActivity et le service com.evernote/.client.EvernoteService.

Cette première analyse des logs nous permet de représenter la structure d'une application en terme de création, de suppression d'activité ou de service ainsi qu'avec les autres actions associées. Maintenant, il faut pouvoir construire avec le plus d'exactitude possible, le comportement dynamique de l'application et son interaction avec les services du système.

4.2 Comportement dynamique

Pour étudier le modèle comportemental des applications, nous avons développé une application malveillante que nous avons nommé *mymalware*. Celle-ci intercepte les appels sortants issus du mobile et les enregistre dans un fichier. Nous avons ensuite utilisé l'outil *Dumpsys* disponible dans la plateforme Android afin d'extraire les logs caractérisant l'état des différents services et les applications installées sur le système. Nous avons donc pu obtenir des informations statiques concernant l'application. Parmi les éléments identifiés, nous avons la liste des permissions qui lui ont été attribuées, son uid et son gid. En particulier, la permission android.permission.PROCESS_OUTGOING_CALLS qui permet de surveiller les appels sortants. Cette permission permet de recevoir l'Intent android.intent.action.NEW_OUTGOING_CALL.

En effet comme on l'indique la figure 4, la liste des BroadcastReceivers qui doivent recevoir cette Intent contient notre application *mymalware*. Ce dernier est inscrit dans deux groupes. Le premier 3003 se présente comme étant un groupe permettant d'accéder au réseau. Le deuxième 1015 est celui des applications pouvant écrire sur des supports externes.

```
Package [fr.collimator.mymalware] (40cbf990):
  userId=10190 gids=[3003, 1015]
  sharedUser=null
  pkg=Package{40b61cd8 fr.collimator.mymalware}
  .....
  dataDir=/data/data/fr.collimator.mymalware
  targetSdk=10
  .....
  grantedPermissions:
    android.permission.READ_PHONE_STATE
    android.permission.READ_SMS
    android.permission.WRITE_EXTERNAL_STORAGE
    android.permission.INTERNET
    android.permission.PROCESS_OUTGOING_CALLS
    android.permission.WRITE_SMS
    android.permission.RECEIVE_SMS
    android.permission.CALL_PHONE
    android.permission.READ_CONTACTS

APP* UID 10190 ProcessRecord{4054da80 11394:fr.collimator.mymalware/10190}
  .....
  pid=11394 starting=false lastPss=0
  .....
  services=[ServiceRecord{40bd11b0 fr.collimator.mymalware/.background}]
  receivers=[ReceiverList{40a98510 11394 fr.collimator.mymalware/10190 remote:40569cb8}]


* ServiceRecord{40bd11b0 fr.collimator.mymalware/.background}
  intent={cmp=fr.collimator.mymalware/.background}
  .....
  app=ProcessRecord{4054da80 11394:fr.collimator.mymalware/10190}
  .....
  startRequested=true stopIfKilled=false callStart=true lastStartId=1

ReceiverList{40a98510 11394 fr.collimator.mymalware/10190
             remote:40569cb8}
  app=ProcessRecord{4054da80 11394:fr.collimator.mymalware/10190}
  pid=11394 uid=10190
  Filter #0: BroadcastFilter{40a985b0}
    ...
    Action: "android.intent.action.NEW_OUTGOING_CALL"
    ...

Historical Broadcast #36:
  BroadcastRecord{40a54748 android.intent.action.NEW_OUTGOING_CALL}
  Intent { act=android.intent.action.NEW_OUTGOING_CALL (has extras) }
  ...
  Receiver #0: BroadcastFilter{40a985b0 ReceiverList{40a98510 11394 fr.collimator.mymalware/
10190 remote:40569cb8}}
```

Figure 4: Extrait des logs obtenus avec l'outil Dumpsys.

Nous avons aussi pu obtenir des informations dynamiques qui correspondent en fait à l'historique des Intents envoyés ou des services créés. Par exemple, toujours sur la figure 4, la partie Historical Broadcast #36. On peut y voir l'Intent signalant qu'un nouvel appel est en cours et qu'il a été envoyé par le composant com.android.phone. On voit aussi la liste des BroadcastReceivers qui reçoivent cet Intent. Nous pouvons donc établir un lien entre le composant com.android.phone, émetteur de l'Intent android.intent.action.NEW_OUTGOING_CALL et le récepteur *mymalware*. Nous avons pu reconstituer le comportement de cette application, en observant notamment sa réception d'un Intent lorsque un appel

était émis depuis le téléphone. D'une manière plus générale, ces éléments, mis en corrélation avec les logs de la section précédente vont nous permettre d'obtenir un modèle comportemental et structurel de l'application.

5. CONCLUSIONS ET TRAVAUX FUTURS

Dans ce papier, nous avons dans un premier temps établi un état de l'art sur les attaques et les menaces dans les environnements Android. Au niveau des menaces, on retrouve les pertes financières, la collecte et l'utilisation des informations privées, l'utilisation des ressources de l'appareil et les actions purement nuisibles. Ensuite, nous avons détaillé les différents vecteurs d'attaques utilisés pour mettre en œuvre ces menaces, notamment par l'installation des applications malveillantes sur l'équipement de l'usager. Nous avons ensuite montrer notre approche basée sur l'analyse des journaux de logs, pour pouvoir construire un modèle caractérisant la structure d'une application Android. Nous avons ensuite enrichie cette structure avec d'autres sources d'information liées aux interactions de l'application avec les services disponibles sur le système.

Nos travaux futurs portent notamment sur l'élaboration d'une chaîne automatisée d'extraction et d'analyse des logs Android. Nous utiliserons les cartes auto-organisatrices de Kohonen [12, 1] avec des vecteurs de *malwares* connus, afin d'identifier les régions caractérisant des comportements malicieux. Nous envisageons ensuite l'évaluation de notre approche à large échelle afin de déterminer son efficacité en terme de faux positifs, de faux négatifs et sa consommation énergétique sur les équipements Android.

6. REFERENCES

[1] David Barrera, H. Güneş Kayacik, Paul C. van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 73–84, New York, NY, USA, 2010. ACM.

[2] Leonid Batyuk, Markus Herpich, Seyit A. Camtepe, Karsten Raddatz, Aubrey-Derrick Schmidt, and Sahin Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. In *6th International Conference on Malicious and Unwanted Software (MALWARE 2011)*, pages 66–72, Fajardo, Puerto Rico, USA, October 2011. IEEE Conference Publications.

[3] Jeffrey Bickford, Ryan O'Hare, Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Rootkits on smart phones : Attacks, implications and opportunities. *HotMobile'10. Annapolis, Maryland, USA*, February 2010.

[4] Burguera, Iker, Zurutuza, Urko, Nadjm-Tehrani, and Simin. 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.

[5] G. Delac, M. Silic, and J. Krolo. Emerging security threats for mobile platforms. *MIPRO 2011. Opatija, Croatia*, May 2011.

[6] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steven Hanna, and David Wagner. A survey of mobile malware in the wild. *SPSM'11. Chicago, Illinois, USA*, October 2011.

[7] Grace, Michael, Zhou, Yajin, Zhang, Qiang, Zou, Shihong, Jiang, and Xuxian. Riskranker : scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, pages 281–294, New York, NY, USA, 2012. ACM.

[8] Hamandi, Khodor, Elhajj, Imad H., Chehab, Ali, Kayssi, and Ayman I. Android sms botnet : a new perspective. In *MOBIWAC*, pages 125–130. ACM, 2012.

[9] Martin S. Olivier Heloise Pieterse. Android botnets on the rise : Trends and characteristics. In *ISSA'12*, pages 1–5, 2012.

[10] Isohara, Takamasa, Takemori, Keisuke, Kubota, and Ayumu. Kernel-based behavior analysis for android malware detection. In *Proceedings of the 2011 Seventh International Conference on Computational Intelligence and Security*, CIS '11, pages 1011–1015, Washington, DC, USA, 2011. IEEE Computer Society.

[11] Amiya K. Maji, Fahad A. Arshad, Saurabh Bagchi, and Jan S. Rellermeyer. An empirical study of the robustness of inter-component communication in android. In *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[12] Saeed Moghaddam and Ahmed Helmy. Modeling of internet usage in large mobile societies using self-organizing maps. *SIGMOBILE Mob. Comput. Commun. Rev.*, 14(4) :13–15, November 2010.

[13] Yan, Lok Kwong, Yin, and Heng. Droidscope : seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[14] Zhou, Wu, Zhou, Yajin, Jiang, Xuxian, Ning, and Peng. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, CODASPY '12, pages 317–326, New York, NY, USA, 2012. ACM.

[15] Yajin Zhou and Xuxian Jiang. Dissecting android malware : Characterization and evolution. *Proceedings of the 33rd IEEE Symposium on Security and Privacy. San Francisco*, May 2012.

[16] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market : Detecting malicious apps in official and alternative android markets. *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS 2012), San Diego, CA*, February 2012.