Master research Internship

Master Thesis

Analyse Automatisée de Malware Android

Cryptography and Security - Mobile Computing

Author:
Adrien Abraham

Supervisors:
Valérie Viet Triem Tong
Jean-François Lalande
Équipe CIDre (CentraleSupélec / INRIA)

UNIVERSITÉ DE RENNES 1

Abstract. Les smartphones Android, contenant de nombreuses informations privées, sont la cible de pirates par le biais de malware, des applications au comportement malveillant. Ces malware sont dissimulés dans des applications bénignes et installés par les utilisateurs sans qu'ils sachent les intentions des pirates : par exemple voler des informations ou envoyer des SMS.

Différentes méthodes existent pour tenter de détecter les malware Android, comme les analyses dynamiques qui étudient le comportement d'une application pendant son exécution. Les pirates savent cependant que leurs malware seront probablement analysés, et mettent en place des techniques pour faire échouer les analyses dynamiques, en ne s'exécutant que sous certaines conditions.

Pour améliorer les analyses dynamiques existantes, nous proposons une méthode originale de ciblage de code suspect et d'instrumentation de l'application pour forcer l'exécution du code ciblé. Pour mettre en place ce système, un nouveau type de graphe de flot de contrôle représentant toute l'application est développé.

Contents

1 Introduction **1**

2 Android **2**
2.1 Système d'exploitation Android 2
2.2 Applications Android 3
2.2.1 Langages et environnement 3
2.2.2 Composants 3
2.2.3 Format de distribution 3
2.3 Mécanismes de sécurité 5
2.3.1 Permissions 5
2.3.2 Modèle de sécurité Unix 5

3 Malware Android **6**
3.1 Nature des malware Android 6
3.2 Attaques contre le système et l'utilisateur 7
3.3 Déclencheurs et précautions 8

4 Analyse du bytecode d'une application **10**
4.1 Analyse de bytecode et représentations intermédiaires 11
4.2 Graphe de flot de contrôle 13
4.2.1 Représentation graphique du programme 13
4.2.2 Limites des graphes de flot de contrôle 13
4.3 Spécificités d'Android 14
4.3.1 Points d'entrée 15
4.3.2 Flots de contrôle implicites 15
4.3.3 Cycles de vie 16

5 Exécution ciblée **16**
5.1 État de l'art 17
5.2 Ciblage de code suspect 18
5.3 Détermination d'un chemin d'exécution 20
5.3.1 Graphe de flot de contrôle d'application 20
5.3.2 Algorithme de détermination du chemin 22
5.4 Exécution automatique par instrumentation 24
5.4.1 Concept d'instrumentation 24
5.4.2 Instrumentation de programmes Java et Android avec Soot 24
5.4.3 Exécution forcée d'un chemin prédéfini 24

6 Travaux liés **24**

7 Conclusion **24**

A Instructions Jimple **27**

B Heuristiques de ciblage de code **28**

1 Introduction

TODO relire et reformatter ça

Les smartphones sont devenus des compagnons de la vie de tous les jours pour une grande partie de la population, et les données stockées sur ces machines sont donc souvent privées : ils contiennent des conversations privées, notre réseau de connaissance, des mots de passe ou des photos. Par conséquent, les smartphones sont également devenus des cibles de choix pour les pirates informatiques qui cherchent à récupérer ces informations pour les utiliser ou les revendre.

Le système d'exploitation pour smartphone le plus utilisé aujourd'hui est Android [7] ; son code est *open source* contrairement à ses concurrents principaux, ce qui permet à la communauté scientifique de s'impliquer plus activement dans la recherche en sécurité sur ce système. Un utilisateur d'Android peut télécharger des applications pour son smartphone sur des boutiques en ligne, appelées *markets*, la plus importante étant le Google Play, comptant plus d'un million d'applications [13].

Une application ayant un comportement malveillant est appelée *malware* : à l'insu de l'utilisateur, un malware peut subtiliser des informations sensibles, envoyer des SMS à des services payants, accéder aux appareils photos ou au GPS du smartphone, intégrer un botnet et d'autres actions indésirables. Les malware sont généralement diffusés sur les markets sous l'apparence d'applications inoffensives : cela peut être des applications simplistes développées spécifiquement pour le malware, mais dans la majorité des cas, il s'agit d'applications saines (non malveillantes) déjà existantes qui ont été modifiées pour y introduire le malware.

Pour contrer ces intrusions des malware dans les markets et les smartphones, la littérature sur les malware Android contient de multiples techniques souvent basées sur les méthodes classiques de détection d'intrusion, telles qu'utilisées dans des systèmes plus communs comme les serveurs ou les ordinateurs de bureau.

Dû au volume trop important d'applications sur le Google Play, il semble impossible de vérifier manuellement si les applications qui y sont ajoutées sont des malware ou non. TODO compléter

Un des problèmes récurrents rencontrés par ce genre d'analyse (demandant d'exécuter l'application à analyser) est qu'une exécution "à l'aveugle", c'est à dire sans intervention humaine, peut facilement passer à côté des actions malveillantes de l'application. En effet, un malware dissimulé dans une application ne va pas forcément se déclencher dès le démarrage de celle-ci : il peut attendre une certaine durée ou un certain événement pour se déclencher. Ces déclencheurs peuvent être des précautions mises en place par les pirates pour éviter d'être facilement détectés par ces analyses.

Nous nous plaçons dans un contexte de documentation des comportements des malware. Si pour réaliser une analyse, des malware sont exécutés mais leur code malveillant n'est pas déclenché, les résultats de l'analyse seront faussés. Par conséquent, il faut s'assurer qu'un malware exécute son code malveillant lorsqu'il est analysé. Cela implique d'avoir une idée préalable sur ce que l'on souhaite voir exécuté. Nos contributions pour répondre à cette problématique sont les suivantes :

- ciblage
- graphes
- instru

TODO plan de la suite :
commencer avec l'OS / l'écosystème puis voir les malware

2 Android

Dans cette section, nous présentons les spécificités d'Android et de ses applications, ainsi que les mécanismes de sécurité déjà mis en œuvre.

2.1 Système d'exploitation Android

Le nom d'Android désigne un système d'exploitation complet, du noyau jusqu'aux programmes utilisateurs. Plus précisément, on distingue plusieurs couches logicielles séparées dans ce système d'exploitation, présentées dans la liste suivante suivant leur proximité croissante avec le matériel.

1. Les *applications* sont des programmes que l'utilisateur peut installer depuis des markets, utiliser et supprimer comme il le souhaite. Hormis certaines applications spécifiques pré-installées, elles ne sont pas indispensables au fonctionnement du système. Elles permettent à l'utilisateur d'effectuer la plupart des actions qu'il attend de son smartphone, comme naviguer sur le Web, lire et répondre à ses mails ou passer des appels téléphoniques.

2. Les *services système* sont les programmes gérant le fonctionnement du système et les interactions des applications avec le matériel.

3. Le *HAL* (*Hardware Abstraction Layer*) est un mécanisme d'abstraction du matériel, permettant de fournir une interface uniforme aux services systèmes lorsqu'ils souhaitent utiliser les fonctions du matériel, sans avoir besoin de connaître l'API de son pilote.

4. Le *noyau* employé par Android est une version modifiée du noyau Linux, améliorée de façon à mieux convenir à un matériel de smartphone, notamment au niveau de l'économie d'énergie. Les pilotes du matériel sont des modules directement intégrés au noyau.

Ce découpage permet d'isoler les différentes parties du système entre elles, notamment les communications de chaque application avec les autres composants du système, qu'on désigne par ICC (*Inter-Component Communication*), vu que chaque requête d'une application vers un autre composant (que ce soit un composant système ou une application) passe par des services systèmes.

Dans cette étude, nous ne nous intéressons qu'aux applications Android car ce sont parmi elles que se trouvent la grande majorité des malware, qui est le vecteur d'attaque principal des pirates. Ce n'est néanmoins pas le seul point du système qui est attaqué : les services systèmes ont déjà été montré vulnérables à certaines attaques (*e.g.* RageAgainstTheCage [3]), et les exploitations de vulnérabilités du noyau Linux sont le plus souvent applicables à Android également (*e.g.* Asroot [2]). Ces attaques nécessitent cependant souvent l'installation d'une application pour opérer, renforçant le caractère critique de la sécurité au niveau des applications.

Un des composants caractéristique d'Android est sa machine virtuelle Java. Ces machines sont un atout de Java : pour faciliter la portabilité du code, le langage Java n'est pas compilé vers du code machine spécifique à une architecture de processeur, mais vers du *bytecode* Java. Ce bytecode, semblable à un langage assembleur d'assez haut niveau, est interprété par une machine virtuelle Java pour être exécuté. Le bytecode est donc utilisable sur tous les systèmes possédant une machine virtuelle Java ; cette portabilité permet aux développeurs d'implémenter des applications en Java qui seront déployables sur tous les systèmes Android en minimisant les problèmes de compatibilité entre les différents constructeurs de smartphones.

2.2 Applications Android

2.2.1 Langages et environnement

Les applications Android sont développées principalement en Java, en utilisant les composants fournis par le SDK Android (*Software Development Kit*). Il est tout de même possible d'utiliser d'autres langages comme le C++ avec l'aide des compilateurs du NDK (*Native Development Kit*) et de la JNI (*Java Native Interface*). Ce mécanisme de Java permet d'appeler des méthodes contenues dans des bibliothèques binaires depuis le code Java, permettant ainsi d'utiliser des bibliothèques complexes développées dans d'autres langages sans avoir à les redévelopper en Java.

La machine virtuelle Dalvik est la machine virtuelle Java d'Android, modifiée pour utiliser du bytecode Dalvik au lieu du bytecode Java. Lorsqu'un développeur veut exécuter son application, le code source Java est d'abord compilé vers du bytecode Java à l'aide d'un compilateur standard (par exemple *javac*), puis un outil du SDK Android, *dx*, convertit ce bytecode Java vers du bytecode Dalvik. La principale motivation derrière ce format de bytecode spécifique à Android est une gestion de la mémoire plus adaptée au smartphone. En effet, le bytecode Java gère les variables à l'aide d'une pile, alors que le bytecode Dalvik utilise principalement des registres, permettant de réduire le nombre d'instructions nécessaires et donc d'économiser du temps d'exécution [25].

L'utilisation d'une machine virtuelle a été remise en question depuis quelques années, et Google a développé un mécanisme nommé ART (pour *Android Runtime*) se chargeant de compiler *ahead-of-time* (à l'installation d'une application) son bytecode Dalvik vers un exécutable natif, du même genre que celui compilé par le NDK pour l'architecture du smartphone. Lors de l'exécution de l'application, il n'y a plus que du code natif et il n'y a donc plus besoin de machine virtuelle, permettant de diminuer d'environ 50% le temps d'exécution du processeur [6].

Ce mécanisme a été testé depuis plusieurs versions d'Android ; la version 5.0 (Lollipop), sortie en novembre 2014, remplace désormais par défaut la machine Dalvik par ART. Dans le cadre des travaux présentés ici, ce changement de paradigme ne nous impacte pas, bien qu'il remette en question de nombreux travaux de la littérature basés sur des modifications de la machine Dalvik.

2.2.2 Composants

TODO activité, services, etc

intents

2.2.3 Format de distribution

Les applications sont distribuées sous forme de fichier APK (*Application Package*), contenant toutes les ressources nécessaires à son fonctionnement. Ce fichier APK est en fait une archive JAR, qui est un format de distribution de programmes Java. Les fichiers JAR sont eux-mêmes de simples archives au format ZIP, respectant certaines règles comme la présence obligatoire de certains fichiers et dossiers. La Figure 1 représente le contenu possible d'une application telle qu'on peut la télécharger sur un market.

Parmi les fichiers nécessaires au fonctionnement de l'application, deux nous seront plus particulièrement intéressants dans cette étude :

classes.dex : ce fichier binaire contient toutes les classes Java du programme sous forme de bytecode Dalvik.

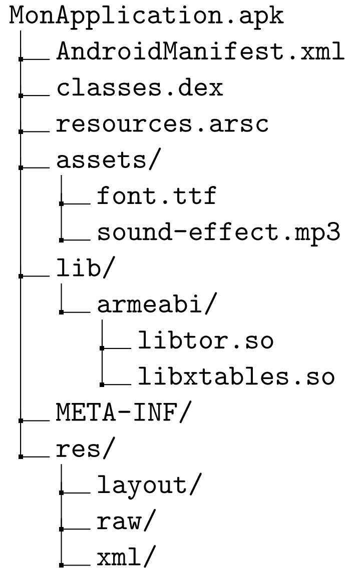

Figure 1: Exemple de contenu d'une application Android empaquetée

AndroidManifest.xml : ce manifeste contient des informations sur le contenu de l'application, comme son nom, les différents composants qu'elle implémente et les permissions qu'elle demande au système.

En ce qui concerne les autres fichiers, le dossier `META-INF` est un des composant du format JAR et contient entre autres le certificat de l'application ainsi que des empreintes cryptographiques des fichiers de l'archive. Les dossiers `assets` et `res` contiennent des ressources multimédia de l'application ; certains fichiers comme ceux au format XML peuvent être compilés sous forme binaire et empaquetés dans le blob `resources.arsc` pour pouvoir être chargés plus rapidement. Enfin, le dossier `lib` contient les différentes bibliothèques natives dont l'application est dépendante. Pour pouvoir fonctionner sur tous les smartphones Android, une application doit fournir des versions de ces bibliothèques compilées pour chaque architecture existante (ARM, x86, MIPS, ...).

Il est intéressant de noter ici que le code de toutes les classes Java de l'application est distribué avec l'application sous forme de bytecode, dans le fichier *classes.dex*. Bien que ça soit un problème non trivial, il est possible de décompiler ce bytecode vers du code source Java avec un succès relatif. Plus précisément, cette décompilation se déroule en deux étapes. Premièrement, le bytecode Dalvik peut être reconverti vers du bytecode Java avec des outils comme Dare [23], avec plus de 99% de classes converties avec succès. Ensuite, plusieurs outils de l'état de l'art en décompilation Java permettent d'obtenir du code source Java plus ou moins satisfaisant [8], c'est à dire plus ou moins lisible et proche de l'original.

Un analyste a donc la possibilité de retrouver du code source de l'application pour essayer de comprendre ses actions. Google encourage cependant l'utilisation d'outils d'obfuscation de code pour empêcher de récupérer du code source depuis une application, ce qui rend le travail de décompilation toujours plus compliqué.

2.3 Mécanismes de sécurité

2.3.1 Permissions

Le principal mécanisme de sécurité d'Android est la mise en œuvre de la politique de sécurité des applications, formée par les permissions qu'elles doivent demander pour effectuer certaines actions. Pour empêcher une application de faire ce qu'elle souhaite sur le smartphone, la plupart des interactions avec le reste du système sont soumises à un mécanisme de contrôle d'accès, dont la politique est représentée par les permissions déclarées et acquises par l'application pour effectuer une certaine classe d'action. Par exemple, une application souhaitant utiliser l'appareil photo du smartphone doit demander la permission `android.permission.CAMERA`. S'il ne demande pas cette permission, la méthode de l'API Android `android.hardware.Camera.open()` permettant d'obtenir une interface avec l'appareil photo ne sera pas disponible et l'application devra s'arrêter.

Ces permissions [1] doivent être déclarées dans le manifeste de l'application pour être accordées par le système. Ainsi, une application est forcée de lister explicitement les permissions qu'elle demande dans son manifeste. Lors de l'installation d'une application, le système montre à l'utilisateur la liste des permissions qu'elle demande et le laisse juger du bon sens de ses demandes. Un utilisateur éclairé n'installera sans doute pas un jeu sur son smartphone qui lui demande la permission d'envoyer des SMS.

Ce système de permission est intéressant mais malheureusement peu efficace en pratique. La documentation d'Android n'étant pas toujours claire sur les permissions à demander pour appeler telle ou telle méthode de l'API, les développeurs d'applications ont tendance à demander plus de permissions que nécessaire. Les utilisateurs ne sont pas toujours aussi éclairés que ce qui était supposé et ne comprennent pas toujours les permissions demandées, ou les acceptent sans les lire.

Pour aider les développeurs à mieux comprendre les permissions Android, une table de correspondance entre les méthodes de l'API et les permissions nécessaires associées a été établie par Felt et al. dans le projet StowAway [16] [2], avec pour principal constat qu'environ un tiers des applications du Google Play ont trop de permissions pour ce qu'elles veulent faire. Un premier effort devrait être fait par les développeurs pour minimiser les permissions utilisées par leurs applications afin d'améliorer la lisibilité des permissions restantes et aider les utilisateurs dans leur décision. Cependant, même en minimisant les permissions requises, les utilisateurs peuvent être démunis lorsqu'ils doivent les étudier. Un second effort serait de mieux expliquer le rôle de chaque permission demandée à l'utilisateur.

2.3.2 Modèle de sécurité Unix

Étant basé sur un noyau Linux, Android bénéficie du modèle de sécurité des systèmes de fichier Unix basé sur des triplets de permissions (lecture, écriture et exécution) attribués à chaque fichier pour le possesseur du fichier, le groupe auquel appartient le fichier et les autres processus du système. Plus précisement, Android tire parti de ce système de sécurité en créant un nouvel utilisateur pour chaque application installée. Ainsi, une application n'a accès qu'aux fichiers qu'elle crée et ne peut lire ou écrire les fichiers des autres applications.

Il est intéressant de noter que depuis la version 4.4 d'Android (sortie en octobre 2013), un nouveau système a été ajouté pour renforcer le mécanisme de contrôle d'accès aux fichiers. SE

[1] Permissions disponibles : `https://developer.android.com/reference/android/Manifest.permission.html`

[2] La page contenant la table des correspondances n'est plus en ligne, mais est disponible sur les Archives du Web : `https://web.archive.org/web/20130516095746/http://android-permissions.org/permissionmap.html`

Android [26] (*Security Enhanced Android*) est une adaptation de SE Linux [28] et implémente un système de contrôle d'accès obligatoire (ou MAC, pour *Mandatory Access Control*) au niveau du noyau. Le contrôle d'accès est appliqué à tous les composants du système, même les processus *root*. Ce projet étant issu d'un laboratoire de recherche indépendant de Google montre que l'entreprise reste à l'écoute des propositions faites par la communauté scientifique pour améliorer la sécurité sur son système d'exploitation.

Pour finir, la machine virtuelle Dalvik possède les mécanismes typiques des machines virtuelles Java, comme un vérificateur de bytecode [4]. Il vérifie que le bytecode est cohérent, par exemple en vérifiant que les sauts conditionnels ne sortent pas de la méthode, il facilite le travail du ramasse-miette de la machine virtuelle et peut même optimiser le code. Contrairement à une JVM classique qui vérifie le bytecode avant exécution, celui-ci est effectué *ahead-of-time*, à l'installation, pour ne pas ralentir le lancement d'une application.

3 Malware Android

Lorsqu'une application agit de façon néfaste pour l'utilisateur, on considère que c'est un malware. Il n'y pas de statistiques précises concernant le nombre de malware car cela dépend des markets analysés ainsi que des méthodes de détection. Entre 1% [17] et 9% [21] des applications trouvées sur Internet sont des malware.

Cette section présente en détail ce qui fait qu'une application est considéré comme étant un malware : son implémentation, sa diffusion vers les utilisateurs, ses actions malveillantes et ses capacités à éviter d'être détectée.

3.1 Nature des malware Android

En général, un malware est un programme qui est ajouté à une application saine déjà existante, puis redistribué en se faisant passer pour l'application originale sur d'autres markets. Vu que le code d'une application peut être récupéré, du moins en partie, depuis une application déjà empaquetée, il est facile pour les pirates d'ajouter leur code à celui d'une autre application, en modifiant juste ce qu'il faut dans le code original pour que le malware se comporte comme désiré une fois installé.

On les trouve surtout sur les markets alternatifs [17], notamment les markets asiatiques, non gérés par Google, car le contrôle sur les soumissions d'applications y est moins sérieux. Google a également communiqué sur un système de détection de malware, Bouncer [5], sur lequel peu d'informations sont disponibles [22] ; ce système est sans doute le rempart principal contre les pirates tentant de soumettre des malware au Google Play.

Dans leur étude sur les conditions dans lesquels les malware Android sont implémentés et distribués, Allix et al. [11] notent plusieurs points intéressants : (i) les malware sont souvent développés en copiant aveuglément le code source d'autres pirates ou depuis des bases de données publiques de vulnérabilité ou d'exemples de code, et (ii) les horaires pendant lesquels sont développés les malware concordent avec ceux d'un travail régulier (*e.g.* 5 jours par semaine). Les principaux résultats à tirer de ces observations sont que si les malware développés sont nombreux, on retrouve souvent des implémentations similaires et parfois peu fonctionnelles (serveurs distants rapidement mis hors ligne, problème de certificats, ...).

Ce phénomène de duplication d'un code malveillant avec des modifications mineures pour chaque pirate forme ce qu'on appelle des *familles* de malware, composées d'*échantillons* (*samples*). Un

échantillon correspond au fichier APK d'une application contenant le malware de sa famille ; comme il ne correspond qu'à ce fichier tel qu'il a été trouvé par l'analyste, il est souvent identifié par son empreinte MD5. Dans une famille de malware, on a donc différents échantillons dans lesquels on retrouve des morceaux de code similaires, c'est à dire plus ou moins modifiés mais venant certainement d'une même source.

En ce qui concerne le code du malware lui-même, il est souvent modifié de façon plus ou moins avancée, pour être moins facilement détectable ou compréhensible lors d'une analyse. L'obfuscation la plus simple, proposée par l'outil Proguard du SDK, consiste à renommer les classes et les variables avec des noms aléatoires. La sémantique du code reste la même, mais sa lecture est rendue plus difficile. Une méthode plus avancée d'obfuscation est d'utiliser la réflexion, un mécanisme de Java permettant d'exécuter des instructions dépendant de données dynamiques, c'est à dire déterminées pendant l'exécution. La Figure 2 montre la différence entre deux blocs d'instructions avec une sémantique similaire (appeler `foo.hello()`), mais le deuxième utilisant la réflexion. Le code avec réflexion peut récupérer les chaînes de caractères `"package.name.Foo"` et `"hello"` depuis des données chiffrées ou depuis le réseau, et ne fournit aucune autre indication sur cette classe et cette méthode.

```
// Sans utiliser la réflexion
Foo foo = new Foo();
foo.hello();

// En utilisant la réflexion
Class<?> cl = Class.forName("package.name.Foo");
Object instance = cl.newInstance();
Method method = cl.getClass().getDeclaredMethod("hello", new Class<?>[0]);
method.invoke(instance);
```

Figure 2: Obfuscation par réflexion

3.2 Attaques contre le système et l'utilisateur

Certains travaux (*e.g.* Delosieres et García [15]) ont tenté d'établir une taxonomie des malware Android, à l'instar des familles de malware sur des systèmes d'exploitations comme Windows ou Linux. Quelques exemples de familles : les *bots* attendent des commandes d'un serveur de C&C (*Command and Control*) pour effectuer des actions prédéterminées en conséquence, les *trojans* s'activent discrètement une fois installés pour transmettre de l'information à un pirate. Les classifications obtenues ne sont pas toujours cohérentes car un malware ne fait rarement partie que d'une seule catégorie, et il nous semble plus pertinent d'essayer plutôt de lister les différents types d'attaques perpétrés contre l'utilisateur et son smartphone.

- Vol d'informations (*spyware*) : la majorité des malware volent certaines informations qui peuvent leur être profitable, à l'insu de l'utilisateur. Les informations du smartphone comme les identifiants et les contacts peuvent être revendues à des sociétés peu scrupuleuses, les photos peuvent servir à faire du chantage, des mots de passe peuvent être récupérés pour pirater des comptes de l'utilisateur.

Le malware Gone in 60 Seconds récupère les messages, appels récents, marque-pages et historique de navigation du smartphone et les envoie sur un serveur distant, avant de se désinstaller lui-même.

- Envoi de SMS : certains malware envoient des SMS et/ou des MMS à des services payants leur appartenant (ou appartenant à des complices) ; c'est un de moyens les plus simples de gagner de l'argent aux dépens de l'utilisateur.

  Le malware Foncy, un clone de l'application SuiConFo de suivi de consommation, détermine le pays dans lequel l'utilisateur a acheté son abonnement téléphonique à l'aide du code pays associé à sa carte SIM. Il envoie ensuite des SMS surtaxés dans une boucle infinie à un numéro dépendant du pays déterminé.

  D'autres malware plus malveillants envoient des messages aux contacts du smartphone pour nuire à la réputation de l'utilisateur : la famille de malware Dogowar, cachée dans un jeu Android mettant en scène des combats de chiens et implémentée par des activistes pour les droits des animaux, envoie à tous les contacts du smartphone : *"I take pleasure in hurting small animals, just thought you should know that."*.

- Publicité agressive (*adware*, ou *scareware*) : pour pouvoir gagner de l'argent, certains malware font apparaître de nombreuses publicités sur le smartphone. Certaines vont jusqu'à afficher de faux messages inquiétants, informant l'utilisateur que son appareil est infecté et que tel ou tel service payant, administré par le pirate, lui permettra de le désinfecter.

  Parmi des exemples récents [3], une trentaine d'applications ont été supprimées du Google Play car elles poussaient l'utilisateur à souscrire à des services payants (4.80€ / mois) pour garder le smartphone libre de malware. Ces applications ont a priori été installées sur plus d'un million de smartphone, ce qui donne une idée des gains que les pirates peuvent espérer.

- Destruction / détournement de données : dans certains cas, le but du malware est d'endommager le smartphone en nuisant à son bon fonctionnement ou à l'intégrité de ses données.

  Dans ce deuxième cas, certains malware mettent en place des demandes de rançon contre les fichiers de l'utilisateur : en tirant parti de la cryptographie asymétrique, les malware comme *SimpLocker* [4] chiffrent les fichiers personnels de l'utilisateur, bloquent le smartphone et décrivent une procédure pour envoyer de l'argent contre quoi les fichiers seront déchiffrés et le smartphone débloqué. On parle dans ce cas-ci de *ransomware*.

Les différentes attaques des malware peuvent donc provoquer des dégâts matériels et financiers importants, en plus d'engendrer des situations sociales potentiellement tragiques.

3.3 Déclencheurs et précautions

Les circonstances du déclenchement du code malveillant des malware est une de leur caractéristique trop peu souvent prise en compte. En général, les malware semblent exécuter leur code malveillant

[3] 22 mai 2015, un groupe d'AdWare détectés sur le Google Play : `http://goo.gl/mkO2Kv`

[4] En fait, les premiers échantillons de SimpLocker utilisaient un chiffrement symétrique avec une clé codée en dur, ce qui rendait le déchiffrement trivial et sans avoir à payer. Des versions plus récentes utilisent des chiffrements asymétriques.

dès le lancement de l'application, notamment ceux dont les agissements sont explicites : un *scareware* n'a pas de raisons de rester cacher. En revanche, un *spyware* souhaite la plupart du temps agir à l'insu de l'utilisateur, et certaines familles mettent en place des précautions pour éviter d'être détecté.

Dans les échantillons que nous avons analysé, plusieurs types de déclencheurs reviennent régulièrement. Le tableau de la Figure 3 présente certains des déclencheurs utilisés par des familles de malware connues.

Type de déclencheur	Implémentation	Utilisé par
Boot du smartphone	Attente de la réception de l'Intent système `BOOT_COMPLETED`	DroidKungFu1 SimpleLocker Badnews
Délai (*timebomb*)	Compare le timestamp courant avec un timestamp préenregistré	DroidKungFu2

Figure 3: Déclencheurs fréquents, implémentation typique et familles de malware les utilisant

La Figure 4 donne un aperçu du code du déclencheur de DroidKungFu2. Celui-ci se déroule en plusieurs étapes : au lancement de l'application, le malware vérifie si un timestamp (nommé *start*) a été écrit dans les préférences de l'application, ce qui sert à déterminer si l'application a déjà été lancée. Le cas échéant, le malware vérifie si ce timestamp est vieux de plus de 30 minutes, et lance ensuite ses procédures malveillantes. Ce délai permet d'éviter d'être repéré lorsque l'exécution du malware est surveillée.

On peut également parler de précautions plutôt que de déclencheurs lorsque le malware effectue des vérifications avant de s'exécuter. Un test fréquent est de détecter si l'application est en train de s'exécuter sur un émulateur ou un smartphone réel. En effet, un utilisateur cible du malware ne va pas utiliser un émulateur : il est sans doute en train d'être testé ou analysé. Par conséquent, il vaut mieux ne rien faire de suspect, et donc ne pas se déclencher tout de suite. Des méthodes simples permettent de vérifier si l'exécution a lieu sur un émulateur (*e.g.* la valeur rendue par `getDeviceId()` est caractéristique, comme une suite de zéros). Il est possible de s'efforcer à rendre un émulateur le plus proche possible d'un smartphone réel, mais un pirate a à sa disposition suffisamment de moyens pour savoir sur quoi il est exécuté.

Dans la littérature, beaucoup de travaux se concentrent particulièrement sur l'algorithme ou l'implémentation de leur contribution aux méthodes de détection de malware, et au final assez peu sur les caractéristiques fondamentales des malware. De la même manière, les tests d'implémentation sont régulièrement effectués sur des ensembles de malware (des *datasets*) sur lesquels peu de détails sont donnés. Que font ces malware utilisés dans le test ? Quelle est la complexité de leurs actions ou le degré d'élaboration de l'obfuscation mise en œuvre ?

Ce travail s'inscrit dans un projet de compréhension approfondie des actions et méthodes des malware, avec pour but à terme de dresser une description qualitative et complète d'un ensemble d'échantillons, pour servir de base de connaissances à d'autres chercheurs. Construire une telle base de données implique plusieurs choses :

- Il faut utiliser le maximum de données disponibles sur un échantillon : ses méta-données, son manifeste, son bytecode, son code source si on est capable de le récupérer.

```java
public void onCreate() {
    SharedPreferences sharedPrefs = this.getSharedPreferences("sstimestamp", 0);

    // Si c'est la première exécution, écriture d'un timestamp et sortie.
    long startTime = sharedPrefs.getLong("start", 0);
    long currentTime = System.currentTimeMillis();
    if (startTime == 0) {
        SharedPreferences.Editor prefsEditor = sharedPrefs.edit();
        prefsEditor.putLong("start", currentTime);
        prefsEditor.commit();
        this.stopSelf();
        return;
    }

    // Sinon, vérification qu'au moins 30 minutes se sont écoulées
    // depuis la dernière exécution.
    long diff = currentTime - startTime;
    long delay = 1800000;   // 30 minutes, en millisecondes
    if (diff < delay) {
        this.stopSelf();
        return;
    }

    // Code malveillant
    this.updateInfo();      // Récupère des informations
    this.getPermission();   // Exploite une vulnérabilité
    this.provideService();  // Fournit le service
}
```

Figure 4: Extrait d'un échantillon de DroidKungFu2 (`F438ED38B59F772E03EB2CAB97FC7685`). Des parties du code ont été simplifiées et des variables renommées par souci de clarté.

- Il faut exécuter ces échantillons pour analyser leur comportement réel. Cela implique de savoir ce que l'on souhaite observer : est-ce que le malware s'est bien déclenché ? Peut-on s'en assurer ?

4 Analyse du bytecode d'une application

Dans la littérature sur la détection de malware, on divise les différentes méthodes en deux grandes catégories : les analyses *statiques* qui inspectent les fichiers d'une application, sans l'exécuter, et les analyses *dynamiques* qui observent toutes les interactions d'un programme avec son environnement pendant une exécution surveillée. Si les analyses statiques peuvent souvent donner des indices cohérents sur le comportement qu'aurait une application à son exécution, ce ne sont que des suppositions et seules les analyses dynamiques peuvent vraiment capturer ce comportement. Comme vue précédemment, une analyse dynamique peut tout de même échouer à capturer ce comportement si le malware a mis en place des déclencheurs pour gagner en discrétion.

Pour tenter de comprendre les actions d'une application Android, il est possible d'effectuer une première analyse statique sur le fichier APK pour inspecter le code de l'application, au format

Dalvik, le contenu de son manifeste et ses autres ressources. Nous regroupons ici les différentes informations que l'on peut tirer d'une analyse statique de l'application pour supposer quel est son comportement à l'exécution, et assister une analyse dynamique subséquente.

4.1 Analyse de bytecode et représentations intermédiaires

Le bytecode étant un format prévu pour être interprété par une machine virtuelle et non pour être lu par un humain, c'est un langage bas-niveau et proche d'un langage assembleur, au format binaire (*i.e.* présence de nombreux caractères non ASCII, donc illisible depuis un éditeur de texte). Pour pouvoir comprendre leur contenu, il est possible de désassembler ces fichiers avec un outil adapté. Les JDK (*Java Development Kits*) contiennent l'outil *javap* permettant de générer une version désassemblée de fichiers de bytecode Java. Pour Dalvik, des outils nommés *smali* et *baksmali* permettent respectivement d'assembler et désassembler du code Smali vers du bytecode Dalvik ; le Smali est un langage spécialement conçu pour servir d'assembleur Dalvik.

La Figure 5 présente une courte méthode Java sous forme de code source Java, de bytecode Java désassemblé et de Smali. Cette méthode de calcul de la suite de Syracuse nous servira d'exemple dans cette section.

```
// Code original Java
public static int
syracuse(int n) {
  while (n != 1) {
    if (n % 2 == 0)
      n /= 2;
    else
      n = n*3 + 1;
  }
  return n;
}
```

```
// Bytecode Java
// désassemblé par javap
public static int
syracuse(int);
   0: iload_0
   1: iconst_1
   2: if_icmpeq     27
   5: iload_0
   6: iconst_2
   7: irem
   8: ifne          18
  11: iload_0
  12: iconst_2
  13: idiv
  14: istore_0
  15: goto          0
  18: iload_0
  19: iconst_3
  20: imul
  21: iconst_1
  22: iadd
  23: istore_0
  24: goto          0
  27: iload_0
  28: ireturn
```

```
# Bytecode Dalvik
# sous forme Smali
.method public static
syracuse(I)I
  .locals 1
  .param p0, "n"    # I
  .prologue
  :goto_0
  const/4 v0, 0x1
  if-ne p0, v0, :cond_0
  return p0
  :cond_0
  rem-int/lit8 v0, p0, 0x2
  if-nez v0, :cond_1
  div-int/lit8 p0, p0, 0x2
  goto :goto_0
  :cond_1
  mul-int/lit8 v0, p0, 0x3
  add-int/lit8 p0, v0, 0x1
  goto :goto_0
.end method
```

Figure 5: Méthode `syracuse` d'exemple (i) en Java à gauche, (ii) en bytecode désassemblé par *javap* au centre, et (iii) en bytecode Dalvik désassemblé en Smali sur la droite.

S'il n'est pas impossible de comprendre ce que fait cette méthode Java dans l'une des deux versions bytecode, on constate que c'est plus compliqué à lire et que la sémantique de l'algorithme est bien moins intuitive. Les identifiants de variables ont pratiquement disparu et la plupart des

opérations consistent à manipuler directement la mémoire, par une pile ou des registres, plutôt que de décrire l'algorithme derrière la méthode.

Dans le domaine de l'analyse de programmes Java, un outil en particulier se présente aujourd'hui comme incontournable depuis plusieurs années : le framework Soot permet d'analyser des programmes Java, sous forme de code source ou de bytecode, en représentant le contenu des programmes sous forme d'objets Java. Par exemple, lorsque Soot analyse un programme contenant plusieurs méthodes Java, il instancie un ensemble d'objets `SootMethod` contenant les différentes caractéristiques de chaque méthode comme sa signature (son nom, le type de ses arguments, la classe à laquelle elle appartient), et son code.

Pour pouvoir représenter un programme Java sous forme d'objets Java, Soot est capable d'utiliser plusieurs représentations intermédiaires du code source, la principale étant le Jimple. Ce langage, semblable à du Java, conserve la sémantique du programme d'origine mais comporte un nombre d'instruction très réduit (15 seulement), permettant d'effectuer des opérations d'optimisation plus facilement. Ces instructions fournissent le strict minimum de fonctionnalités pour être compatible avec Java : les assignations, les appels et retour de fonction, etc. Une liste complète des instructions Jimple est disponible en annexe A.

La Figure 6 montre l'équivalent Jimple de l'exemple précédent. Les types sont explicites, les noms des variables les reflètent (dans le nom `$i0`, `i` signifie *int*), les sauts et les opérations sont faciles à suivre. Nous ne nous intéresserons pas ici sur les détails de la syntaxe du Jimple ; il faut simplement comprendre la façon dont les instructions s'enchaînent. Les `goto` permettent de sauter au `label` correspondant. Lorsqu'un `goto` est précédé par un `if`, c'est un saut conditionnel et le test en argument du `if` doit être évalué vrai à l'exécution pour que le saut soit pris.

```java
// Code original Java
public static int
syracuse(int n) {
  while (n != 1) {
    if (n % 2 == 0)
      n /= 2;
    else
      n = n*3 + 1;
  }
  return n;
}
```

```
// Version Jimple par Soot
public static int
syracuse(int) {
  int $i0, $i1;
  $i0 := @parameter0: int;
 label1:
  if $i0 != 1 goto label2;
  return $i0;
 label2:
  $i1 = $i0 % 2;
  if $i1 != 0 goto label3;
  $i0 = $i0 / 2;
  goto label1;
 label3:
  $i1 = $i0 * 3;
  $i0 = $i1 + 1;
  goto label1;
}
```

Figure 6: Représentation en Jimple de la méthode `syracuse`

Le nombre réduit d'instructions Jimple permet de représenter chacune d'entre elle par une interface Java, nommée `Unit`. Dans une méthode comme celle de la Figure 6, l'instruction `$i1 = $i0 % 2;` sera accessible et manipulable depuis un objet de type `AssignStmt` (pour *assignment statement*), depuis lequel on peut récupérer des `Value` comme la cible de l'assignation (`$i1`) et l'expression `RemExpr` (pour *remainder expression*) du modulo qui est assigné. Ce modulo étant une

expression binaire, il permet d'accéder aux deux paramètres `$i0` et `2`. De la même manière, les instructions de saut comme les *if* et les *goto* sont accessibles respectivement via des objets `IfStmt` et `GotoStmt`. Chacun possède une méthode `getTarget()` qui rend l'Unit cible du saut ; dans le cas d'un *if*, c'est la prochaine `Unit` à exécuter si la condition est évaluée vraie.

Tout ces objets sont des `Unit` de Soot. Avoir accès à ces objets permet de mettre en œuvre des analyses particulièrement poussées car chaque élément du code d'un programme est accessible de façon logiquement organisée. Dans la suite de ce rapport, nous utilisons `Unit` et instruction Jimple de façon interchangeable par abus de langage.

Soot était à la base un framework étudié pour les programmes Java uniquement et dédié à l'optimisation de code, mais ces dernières années l'équipe SSE développant Soot a implémenté un support pour le bytecode Dalvik constamment amélioré. L'équipe s'oriente également de plus en plus vers la recherche en détection de malware sur Android [9], laissant prévoir d'importantes contributions dans le domaine dans les années à venir.

4.2 Graphe de flot de contrôle

Avec une représentation de chaque instruction du programme sous forme d'`Unit`, depuis lesquelles on peut extraire des informations sur le déroulement du programme, on peut développer des objets plus formels pour observer le comportement d'un programme.

Nous allons utiliser ici des graphes de flot de contrôle, ou CFG (*Control-Flow Graph*). La notion de flot de contrôle désigne les exécutions possibles d'un programme à l'échelle des instructions, soit de façon plus imagée, les *chemins* empruntables.

4.2.1 Représentation graphique du programme

Pour pouvoir représenter l'ordre dans lequel les instructions d'une méthode sont exécutées, en tenant compte des sauts conditionnels, on peut utiliser des graphes de flot de contrôle. La représentation en Jimple du code permet de générer facilement ces graphes car il y a peu d'instructions de saut à prendre en compte (`if` et `goto`), contrairement au Java (boucles, exceptions, sections critiques, *switch* et autres).

La Figure 7 montre le CFG de notre méthode `syracuse` en Jimple. La sémantique de chaque élément de ce graphe orienté est définie ainsi : chaque nœud du graphe représente une `Unit` de la méthode, et chaque arc représente le successeur possible de chaque instruction à l'exécution. Ainsi, la plupart des instructions n'ont qu'un arc vers l'instruction suivante, à part pour les sauts conditionnels où il y a deux successeurs possibles, correspondant aux deux `Unit` à exécuter selon si la condition du saut est évaluée à vraie ou non pendant l'exécution.

Notons qu'il n'y a pas de spécification formelle sur ce que doit être exactement un CFG : dans d'autres travaux, les sauts conditionnels sont représentés sous forme de nœuds en losange, avec des arcs annotés, alors qu'ici les sauts et la condition associée ne forment qu'un même nœud. L'essentiel est que le graphe comporte l'information qui nous intéresse sur le flot de contrôle.

Ces graphes peuvent être générés par Soot au format DOT, un format flexible de représentation de graphes, pour chaque méthode d'une application.

4.2.2 Limites des graphes de flot de contrôle

Les CFG offrent une représentation pratique des déroulements possibles de l'exécution d'une méthode. Générer ces graphes en tenant compte des mécanismes de Java comme les exceptions n'est pas

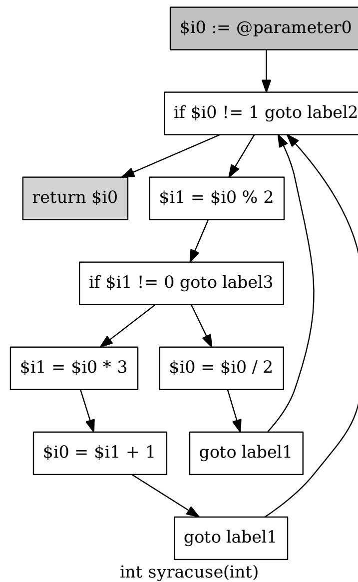

Figure 7: Graphe de flot de contrôle (CFG) de la méthode `syracuse`

très compliqué pour Soot car c'est un problème étudié depuis longtemps [14]. Cependant cette représentation s'arrête aux limites d'une méthode, et on ne connaît pas les relations de cette méthode par rapport au reste de l'application, comme les endroits d'où elle est appelée. Si on souhaite déterminer le comportement de l'intégralité d'une application, les graphes de flot de contrôle ne sont pas suffisants.

Un autre type de graphe, nommé graphe d'appel de fonction (*call-graph*), consiste à générer un graphe dont les nœuds sont des méthodes et les arcs sont les appels de ces méthodes allant vers les méthodes appelées. Par exemple, dans une application où une méthode `run` appelle une autre méthode `startJob`, le graphe d'appel de fonction de cette application contiendra un nœud par méthode et un arc orienté de `run` vers `startJob`.

Soot (et quelques autres outils) permet de générer cet autre type de graphe pour des programmes Java, et avec un succès relatif pour les applications Android (cf. Sous-section 4.3). Il est compliqué d'utiliser ces deux types de graphes mis en commun car il y a une différence d'échelle entre les deux : les nœuds des CFG représentent des instructions et les nœuds des graphes d'appels représentent des méthodes.

4.3 Spécificités d'Android

Si les applications Android sont toutes des programmes au moins en partie en Java, il y a plusieurs différences entre le comportement d'une application Android et celui d'un programme Java exécuté sur une JVM classique.

4.3.1 Points d'entrée

La première différence vient de la pluralité des *points d'entrée* d'une application Android. En effet, dans un programme Java, C, ou de la plupart des langages impératifs, un programme n'a qu'un point d'entrée, c'est à dire une seule fonction par laquelle l'exécution peut commencer. Cette fonction est généralement nommée `main` et est unique au programme.

Dans une application Android, ce point d'entrée est remplacé par une activité désignée comme étant le "main" parmi toutes celles de l'application, et est l'activité qui est doit être affichée lorsque l'utilisateur lance l'application en appuyant sur son icône. Dans le manifeste, cette activité "principale" se manifeste par la capture des Intents système `action.MAIN` et `category.LAUNCHER`. Bien que l'application ne sera généralement lancée que par le biais de cette activité, il est possible d'utiliser n'importe quelle activité de l'application comme point d'entrée, par exemple en passant par un invite de commande, ou par l'outil de débug *adb*.

Plus généralement, certains des composants Android de base peuvent être utilisés comme points d'entrée d'une application : les activités, mais aussi les services (s'ils sont paramétrés pour), et les récepteurs d'Intent. Ces multiples points d'entrée sont utiles pour permettre aux applications d'interagir entre elles par le biais d'Intents. Prenons le cas d'une application A de paiement en ligne sécurisé, on peut imaginer que l'activité principale est une interface de configuration des données bancaires. Si une autre application B souhaite utiliser ce système de paiement, elle voudra lancer l'application A par une activité permettant d'effectuer le paiement, et non l'activité principale.

```
# Lancer l'activité SecondActivity de l'application com.google.test
> adb shell am start com.google.test/.SecondActivity
# Lancer le service DummyService
> adb shell am startservice com.google.test/.DummyService
# Activer le BroadcastReceiver répondant à l'Intent SCREEN_ON
> adb shell am broadcast -a android.intent.action.SCREEN_ON
```

Figure 8: Commandes de lancement arbitraire de composants d'une application

La Figure 8 montre des commandes pour lancer ces composants indépendamment de l'activité principale d'une application.

4.3.2 Flots de contrôle implicites

La génération d'un graphe d'appel de fonction se fait généralement en inspectant chaque instruction d'appel dans toutes les méthodes d'une application pour déterminer les arcs du graphe. Ces flots de contrôle entre méthodes sont *explicites* : dans une méthode a, telle `Unit` est un appel à la méthode b. Certains mécanismes intrinsèques d'Android rendent cette génération compliquée ou incomplète.

Prenons pour exemple la procédure pour lancer une activité, décrite par la Figure 9. Depuis une certaine activité, on souhaite lancer l'activité `SecondActivity` : un Intent est crée avec l'objet `Class` de `SecondActivity`, et cet Intent est passé en paramètre à la méthode `startActivity` de l'API Android.

Il y a donc bien un lien en terme de flot de contrôle entre la méthode `startSecondActivity` et une méthode de `SecondActivity` car l'appel à `startActivity` va effectivement donner lieu à l'instanciation d'une `SecondActivity`. Lorsqu'une activité est instanciée, sa première méthode exécutée automatiquement par le système est `onCreate`. Dans la Figure 9, à aucun moment la

```
// Dans une classe d'exemple, FirstActivity
public void startSecondActivity() {
  // Flot explicite de cette méthode vers la méthode statique Log.i
  Log.i("MyApp", "About to start SecondActivity");
  // Flot implicite de cette méthode vers une méthode de SecondActivity
  Intent intent = new Intent(this, SecondActivity.class);
  startActivity(intent);
}
```

Figure 9: Flot implicite : procédure de lancement d'activité

méthode `startSecondActivity` n'appelle explicitement `onCreate` sur un objet SecondActivity ; il n'y a d'ailleurs même pas d'instanciation explicite de cet objet. Ce flot est alors dit *implicite* car il est décidé par le système et non par le programme.

Pour trouver ces flots implicites et générer un graphe d'appel complet, il faudrait avoir une connaissance totale des agissements des méthodes de l'API Android. Grace et al. [18] sont parvenus à contourner ce problème en observant que ces méthodes, bien qu'elles n'appellent pas explicitement une méthode cible, ont une sémantique bien définie dans la documentation d'Android. Par exemple un appel à `startActivity` donnera lieu à l'appel, par le système, de la méthode `onCreate` de l'activité cible, à un moment indéterminé dans le futur (en pratique, c'est quasiment instantané). Il est donc possible de créer un arc dans le graphe d'appel de fonction entre la méthode appelant `startActivity`, et la méthode `onCreate` de l'activité cible.

Un autre type de flot implicite est dû aux interactions de l'utilisateur avec le smartphone, par exemple via l'interface graphique. Pour chaque bouton d'une activité, l'application configure généralement une méthode à exécuter lorsqu'il est pressé. Ce lien peut être défini statiquement dans un fichier de ressources, mais il peut également être déclaré dynamiquement, pendant l'exécution. Lorsque l'utilisateur appuie sur un bouton de l'interface, la méthode associée est exécutée par le système mais là encore, à aucun point dans le programme cette méthode n'est appelée explicitement ; aucun arc du graphe d'appel ne va vers le nœud de cette méthode.

4.3.3 Cycles de vie

Une autre spécificité d'Android à prendre en compte est le système de cycle de vie des applications. Si l'utilisateur est en train d'utiliser une application et reçoit un appel téléphonique, l'application va se mettre en pause. Pour cela, le système appelle automatiquement la méthode `onPause` de l'application. Lorsque l'appel téléphonique est terminé, le système laisse l'application reprendre son exécution en appelant sa méthode `onResume`. La Figure 10 présente les différentes interactions du système avec l'application et les méthodes associées.

À l'instar des flots implicites provoqués par les interactions de l'utilisateur avec l'application, ces méthodes appelées par le système uniquement et en réponse à des événements extérieurs ne sont jamais appelées depuis une autre méthode de l'application.

5 Exécution ciblée

Pour obtenir des résultats exacts sur le comportement des malware, nous avons déterminé que nous allions avoir besoin de les exécuter, en s'assurant que les parties correspondant au code malveil-

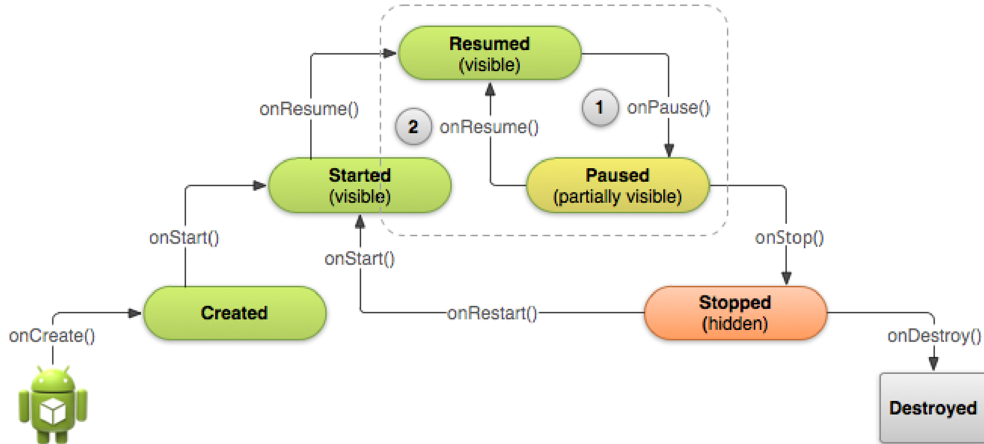

Figure 10: Cycles de vie d'une application Android

lant soit bien déclenchées. Les sections précédentes nous ont montré plusieurs outils pratiques pour analyser le code d'une application, mais aussi les difficultés liées à Android, comme les flots implicites.

Dans cette section, nous présentons nos contributions en trois points :

- reprendre itemize intro en simple

5.1 État de l'art

Depuis 2012, certains travaux se sont penchés sur le problème de l'analyse automatique d'applications Android, avec pour objectif principal la détection de malware Android. Le concept de *stimulation* d'application revient régulièrement dans ces travaux : on cherche à stimuler l'application pour exécuter, selon les besoins de l'analyse, soit le maximum de son code, soit les parties de son code qui nous intéresse.

Dans la bibliographie de ce stage, nous présentions les différentes méthodes utilisées par les travaux d'analyse dynamique pour interagir automatiquement avec l'application analysée. Le Monkey [1] est le choix le plus simple et le plus répandu : cet outil inclus dans le SDK Android envoie des événements utilisateurs pseudo-aléatoires à l'application pour simuler l'utilisation d'un être humain. Les événements générés sont par exemple des clics aléatoires sur l'écran ou les boutons du smartphone.

Parmi les travaux les plus intéressants, Andrubis de Lukas Weichselbaum [27, 19] récupère la liste des activités et services répertoriés dans le manifeste de l'application et les exécute tous, indépendamment les uns des autres. Zheng et al. [29] ont implémenté dans leur système SmartDroid une générateur de graphe d'appel de fonction mais aussi un graphe d'appel d'activité. Ce deuxième graphe détermine les chemins possibles entre les différentes activités de l'application, traversables à l'aide des éléments d'interface que l'activité propose. L'approche de SmartDroid est d'utiliser ces graphes pour trouver quels sont les chemins possibles menant à des appels d'API sensibles, et parcourir automatiquement les activités en effectuant les interactions utilisateurs nécessaires. Aucune implémentation publique ni d'Andrubis ni de SmartDroid n'est disponible.

Parmi les approches plus formelles, Anand et al. [12] et plus récemment Machiry et al. [20] ont essayé d'adapter les méthodes de *concolic testing* (mot-valise pour *concrete* et *symbolic*). Ces techniques consistent à utiliser de l'interprétation abstraite et des solveurs pour trouver les valeurs nécessaires à la prise de tel ou tel chemin dans le graphe de flot de contrôle. Des implémentations sont disponibles mais sont encore trop instables pour être utilisées sur un large ensemble d'échantillons.

Il est possible que le manque d'implémentation concrète et maintenue pour résoudre ce problème vienne de la complexité temporelle importante d'une exécution exhaustive du code de l'application, c'est à dire de s'assurer que chaque instruction du code soit parcourue au moins une fois. Le besoin d'avoir une exécution exhaustive est d'ailleurs plus liés au problème de *test* automatisé, plutôt que d'analyse automatisée. De cette observation, nous avons décidé de commencer une analyse en ciblant des parties précises du code que nous souhaitons voir exécutées. Dans le cadre de la détection de malware, cette approche nous paraît plus efficace que chercher à exécuter tout le code d'une application pour inclure celui du malware.

5.2 Ciblage de code suspect

Nous avons accès à une représentation du code en Jimple, avec des informations comme les classes utilisées dans chaque `Unit`. Prenons par exemple l'instruction d'envoi de SMS suivante.

```
$r2.sendTextMessage("+3336303630", null, "PREMIUM", null, null)
```

Cette instruction fait appel à la méthode `sendTextMessage` de l'objet référencé par `$r2`, avec en paramètre deux chaînes de caractère et trois `null`. L'objet `Unit` correspondant à cette instruction nous informe alors que cette instruction utilise les types `android.telephony.SmsManager` (type de `$r2`), `java.lang.String` (les chaînes de caractères), `null_type` (les `null`) et `void` (le type de retour de `sendTextMessage`).

Pour cibler du code suspect depuis cette représentation, nous avons cherché à déterminer des heuristiques basées sur ces classes utilisées lors de chaque instruction. Grace et al. [17], pour détecter simplement des comportements suspects parmi de larges ensembles d'échantillons, observent l'instanciation et l'utilisation de certaines classes de l'API Android, comme `DexClassLoader`. Cette classe sert à charger du code Java dynamiquement, ce qui est intéressant pour les malware car ça leur permet d'exécuter du code récupéré par le réseau ou depuis un fichier chiffré.

Plus récemment, Aafer et al. [10], pour la réalisation de leur prototype DroidAPIMiner, ont établi une liste des 20 méthodes Java de l'API Android utilisées plus fréquemment par les malware que par les applications classiques. Cette différence va de 20% à presque 50% (dans le cas de `getSubscriberId`) d'utilisation en plus ; en d'autres termes, cette méthode est utilisée 50% plus souvent par des malware que par des applications saines. Ci-dessous nous listons quelques-uns des composants de l'API les plus touchés par cette différence, avec les méthodes impliquées et une courte explication sur les raisons possibles de ces utilisations plus fréquentes par les malware. Les statistiques détaillées sont disponibles dans l'article de DroidAPIMiner.

android.telephony.TelephonyManager : ce composant permet d'obtenir des informations concernant le téléphone, dont des identifiants uniques permettant d'identifier un téléphone précis. Par exemple, `getSubscriberId` et `getDeviceId` permettent d'obtenir le numéro IMEI [5] sur les téléphones qui en sont dotés, et `getLine1Number` permet d'obtenir le numéro de téléphone.

android.app.Service : les Services fonctionnant en tâche de fond (dans un thread différent), c'est un composant pratique pour les malware qui attendent un certain temps ou des commandes d'un serveur C&C pour agir discrètement. Les malware implémentent plus fréquemment les méthodes `onDestroy`, `onCreate` et le constructeur de cette classe.

android.context.pm.PackageManager : ce composant permet de lister les applications installées sur le système, et d'en installer ou supprimer des nouvelles.

android.telephony.SmsManager : ce composant permet d'envoyer des SMS et des MMS. Sans surprise, on retrouve la méthode `sendTextMessage` permettant d'envoyer des SMS à un numéro donné dans tous les malware envoyant des messages à des services payants.

java.util.{Timer,TimerTask} : ces composants standards de Java sont souvent utilisés par les malware dans les implémentations de *time bombs*.

java.lang.{Runtime,Process} : ces composants, standards de Java également, servent à manipuler des processus natifs : lancer leur exécution (`exec`), capturer leurs sorties (`getOutputStream`), attendre la fin de leur exécution (`waitFor`). La majorité des malware contenant des processus natifs passent par ces méthodes pour gérer leur fonctionnement depuis le code Java.

Avec des différences de fréquence d'utilisation telles que pour la méthode `getSubscriberId`, on considère que du code appelant cette méthode a une probabilité supérieure à la normale d'être impliqué dans une action malveillante. Bien entendu, ce n'est pas un indicateur fiable pour déterminer du code malveillant car il est tout à fait possible qu'une application saine souhaite avoir accès à ces informations, mais c'est un indice suffisant pour déterminer quelles parties du code d'une application sont a priori plus suspectes que d'autres.

Nous avons alors mis en place un mécanisme simple de ciblage de code suspect basé sur ces différences de fréquence d'utilisation. Dans un premier temps, nous dressons une liste de différentes classes de l'API Android et des bibliothèques standards de Java qui sont utilisés plus fréquemment par les malware. Ces classes sont regroupées en différentes catégories de comportements suspects :

Binary : (Java) gestion de fichiers exécutables

Dynamic : (Android) chargement dynamique de code

Reflection : (Java) classes utilisées par la réflexion

Crypto : (Java) bibliothèque de protocoles cryptographiques

Network : (Java) gestion du réseau

SMS : (Android) gestion d'envoi et réception de SMS & MMS

Misc : (Java) classes généralement utilisées pour dissimuler ses actions

[5]Le numéro IMEI (*International Mobile Station Equipment Identity*) est unique à chaque terminal et permet par exemple aux opérateurs téléphoniques de bloquer les appels depuis et vers des terminaux volés.

Le détail complet des catégories et classes associées au moment de l'écriture de ce rapport est disponible dans l'annexe B. Ces catégories et classes ne sont pas définitives ; il faut encore les tester sur un ensemble de malware conséquent pour trouver les combinaisons donnant les meilleurs résultats, c'est à dire les classes se trouvant avec une fréquence bien supérieure parmi les malware.

Pour chaque instruction d'un programme, on inspecte via l'`Unit` associée par Soot les classes qu'elle utilise. Si parmi les classes utilisées certaines sont dans les catégories de comportement suspects, on attache un score de risque à l'`Unit`. Ces scores sont plus ou moins importants selon le risque encouru par l'exécution de l'instruction : une instruction utilisant le `SmsManager` aura un score de 10, une instruction utilisant la réflexion aura un score de 3.

Cette approche est simple et efficace en pratique : sur quelques malware testés, le code malveillant est correctement ciblé. Cependant, le nombre de faux positifs (code non malveillant ciblé) est assez important, notamment dû à certaines bibliothèques tierces. Beaucoup d'applications utilisent le réseau à des fins honnêtes, aussi est-ce cohérent de donner un score de risque, même faible, à toute instruction utilisant une connexion réseau ? Les catégories et scores associés devront sans doute être revus au fur et à mesure des tests.

De la même manière, la réflexion a des usages légitimes, dans la gestion des interfaces graphiques, le chargement de code dynamique également, dans les bibliothèques d'affichage de publicités. Comme on sait par avance que certaines bibliothèques tierces connues utilisent tel ou tel mécanisme que l'on peut considérer comme suspect, il est possible d'entretenir une liste blanche d'application autorisée pour réduire le nombre de faux-positifs.

Cette méthode de ciblage est limitée par l'utilisation de la réflexion dans les malware. En effet, une instruction appelant une méthode `m` par réflexion ne donne pas forcément statiquement le nom de cette méthode, et Soot ne peut pas toujours le récupérer. Notre solution pour contourner ce problème consiste à considérer l'utilisation de la réflexion comme suspect, avec un score faible. Cela signifie qu'une application entièrement obfusquée par du code utilisant la réflexion aura un score de risque positif à chaque instruction, ce qui n'est pas une information utile.

5.3 Détermination d'un chemin d'exécution

Une fois qu'une ou plusieurs `Unit` suspectes ont été notées, nous devons savoir comment exécuter ces instructions, *i.e.* trouver un chemin possible à parcourir pour atteindre ces instructions à l'exécution.

5.3.1 Graphe de flot de contrôle d'application

Un tel chemin part d'un point d'entrée de l'application pour arriver à l'instruction ciblée. Il traverse donc différentes instructions, potentiellement des sauts conditionnels, et sans doute différentes méthodes. Nous avons vu dans la Sous-section 4.2.2 que les graphes de flot de contrôle sont limités à une méthode, et que les graphes d'appels de fonction, qui permettent de représenter les relations interprocédurales, sont à un niveau différent. Nous avons donc créé un graphe de flot de contrôle de l'application entière, basé sur les CFG de chaque méthode, liés entre eux par les flot interprocéduraux explicites et implicites que nous savons déterminer.

Les flots explicites sont générés par les instructions d'appel à une méthode, représentées dans Soot par des `Unit` de type *InvokeStmt*. Pour chaque instruction de ce type, on peut récupérer la signature de la méthode appelée. Une signature de méthode est une chaîne de caractère représentant le nom complet (classe + nom) d'une méthode, son type de retour et ses arguments. Cette signature

permet de vérifier si la méthode appelée fait partie des méthodes de l'application, et non d'une API externe. Si elle fait bien partie de l'application, on peut tracer l'arc représentant le flot interprocédural de l'instruction de l'appel au point d'entrée de la méthode appelée. Le point d'entrée d'une méthode est la première instruction à être exécutée, la seule sans prédécesseur.

Les flots implicites nécessitent de déterminer quelles sont les différentes conséquences des appels à certaines méthodes de l'API Android. Nous avons déjà cité l'exemple de la méthode `startActivity` dans la Sous-section 4.3.2, mais nous en avons déterminé plusieurs autres et implémenté la création des arcs associés dans le graphe de flot de contrôle d'application.

Création d'activité : un appel à `startActivity` entraîne un appel à la méthode `onCreate` de l'activité créée.

Création de service : un appel à `startService` entraîne un appel à la méthode `onCreate` du service créé, puis à sa méthode `onStartCommand`.

Liaison à un service : si un service est déjà créé, un appel à `bindService` entraîne un appel à la méthode `onBind`.

Dans le cas où plusieurs flots implicites s'enchaînent comme dans le cas de la création de service, il faut placer les arcs de façon à bien représenter l'enchaînement des instructions à l'exécution. Si la méthode `onCreate` est implémentée par le service dans l'application (ce qui est facultatif), un premier flot implicite relie l'appel `startService` au point d'entrée de la méthode `onCreate`. Ensuite, un autre flot implicite relie chaque instruction de retour de cette méthode au point d'entrée de la méthode `onStartCommand`, sous la forme de plusieurs arcs, un par instruction de retour. Si `onCreate` n'est pas implémentée par le service, c'est la méthode `onCreate` de la classe de base du service, une classe de l'API Android, qui est exécutée suite à `startService`. Cette méthode n'est pas dans le graphe de flot de contrôle de l'application, qui ne contient que les classes incluses dans le bytecode de l'application. On trace alors le flot implicite comme reliant directement `startService` au point d'entrée de `onStartCommand`.

Obtenir suffisamment d'informations pour déterminer ces flots implicites est un problème compliqué d'analyse statique. Dans le cas de `startService`, pour déterminer le flot implicite vers le `onCreate` du service cible, il faut connaître le nom de la classe de ce service. L'argument passé à `startService` est un Intent, qui a été initialisé avec en paramètre l'objet `Class` associé à la classe du service à démarrer. Le lien entre `startService` et la classe du service cible subit donc plusieurs indirections. Dans le cas où l'Intent est initialisé dans la même méthode que l'appel à `startService`, c'est relativement simple de retrouver quel est la classe du service visé. Si l'Intent a été initialisé ailleurs, il faut commencer à trouver avec une analyse de flot de données dans quelles méthodes il a pu être initialisé, ce qui n'est pas trivial. Un Intent peut également être initialisé avec d'autres paramètres que cet objet `Class`, ce qui peut encore complexifié la recherche de la classe du service cible.

La Figure 11 montre en partie ce que donne un graphe de flot de contrôle d'application, dans le cas d'une application de test appelant un service comme décrit ci-dessus. Pour conserver des graphes cohérents tout en leur ajoutant de l'information, plusieurs codes graphiques s'appliquent. Les nœuds en ellipse ne sont pas des instructions mais de simple indications du point d'entrée d'une méthode (eux-mêmes en gris foncé). Ces indications, lorsqu'elles sont colorées en vert, signifient non seulement un point d'entrée d'une méthode, mais aussi qu'elles sont le point d'entrée de l'application, comme présenté en Sous-section 4.3.1. Les nœuds oranges sont ceux qui ont un

score de risque positif. Cette Figure comporte deux arcs tracés en tiret : ce sont les flots interprocéduraux implicites entre l'appel `startService` et les méthodes `onCreate` et `onStartCommand`, tracés différemment à cause de leur sémantique légèrement différente par rapport aux autres arcs du graphe.

Chaque nœud possède en attribut l'identifiant de sa méthode, de façon à ne pas mélanger les instructions de plusieurs méthodes pendant un parcours. Un graphe de flot de contrôle d'application peut sembler être une structure lourde, surtout lorsque l'on sait que certains échantillons possèdent plusieurs milliers de méthodes. En pratique, les bibliothèques de manipulation de graphes comme NetworkX n'ont pas de problème avec des graphes de cette envergure.

5.3.2 Algorithme de détermination du chemin

Avec ce graphe de flot de contrôle d'application, il est plus simple de relier un bout de code ciblé à un point d'entrée de l'application avec un chemin montrant toutes les instructions à parcourir. Par rapport aux graphes d'appel de fonction, on n'a pas seulement une liste de méthode à parcourir, mais la liste des instructions de chacune de ces méthodes qu'on doit parcourir pour atteindre l'instruction ciblée.

Pour trouver ce chemin, nous avons affaire à un problème de parcours de graphe qui peut être décrit formellement de la façon suivante.

Dans une application, il y a m méthodes, représentées sous la forme de control-flow graphs (CFG). Les CFG sont des graphes *(S,A)* orientés.

- Un sommet s de S représente une instruction Jimple.
- Un arc a de A représente une possible instruction suivante à exécuter. À part pour les instructions de saut, il n'y a qu'un arc sortant par sommet.

Chaque graphe a un sommet e sans prédécesseur, nommé *point d'entrée de la méthode*. L'ensemble de ces points d'entrées est nommé $\mathcal{E}$. Un sous-ensemble $\mathcal{E}^* \subset \mathcal{E}$ contient les *points d'entrées de l'application*, par lesquels l'exécution de l'application peut commencer.

Un sommet avec deux arcs sortants vers deux `Unit` de la même méthode est un branchement (*e.g.* `Unit` d'un *if*), et les deux `Unit` cibles sont des *branches*. Lors de l'appel à une autre méthode, un arc part du sommet de l'instruction d'appel vers le point d'entrée de la méthode cible ; ce sont les flots interprocéduraux.

Lors d'une exécution, on obtient une liste de certains sommets visités, à savoir des points d'entrée et des branches.

Avant une exécution subséquente, on détermine un (ou plusieurs) sommet t (pour *target*) que l'on souhaite atteindre. On cherche alors à trouver un chemin partant de l'un des sommets de $\mathcal{E}^*$ et allant vers t. On souhaite également obtenir la liste des branches situées sur le chemin trouvé pour pouvoir les forcer à l'exécution.

On note que ce chemin peut ne pas exister, dû aux spécificités d'Android vues précédemment. Typiquement, une méthode appelée en réaction à l'appui d'un bouton n'est explicitement appelée par personne, donc son graphe est disjoint du reste du graphe de l'application.

Il peut aussi y avoir plusieurs chemins, depuis plusieurs points d'entrée. Dans ce cas, il est préférable de trouver le chemin le plus court, pour que l'exécution soit la plus rapide possible.

On a alors besoin d'un algorithme avec les entrées et sorties suivantes :

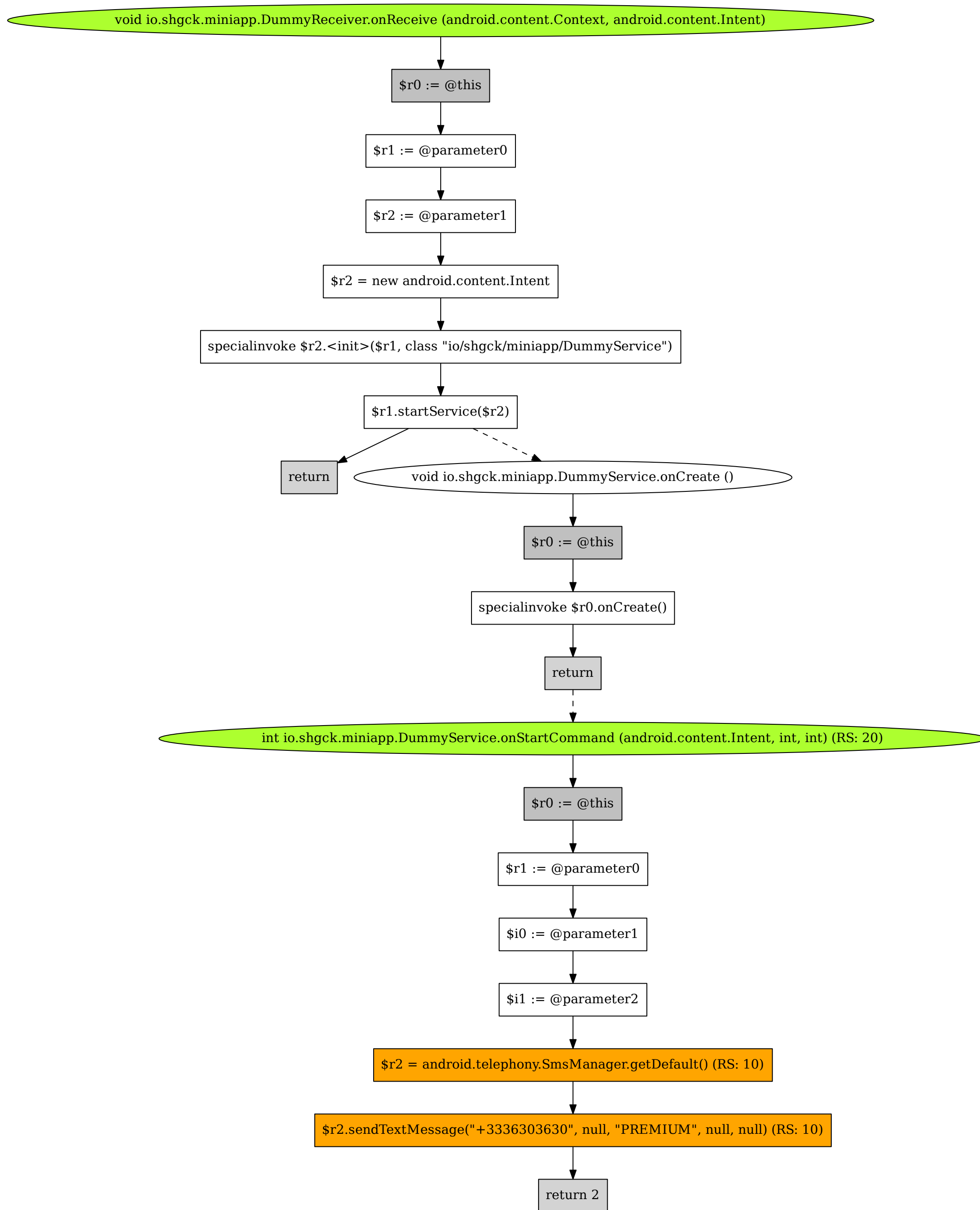

Figure 11: Extrait d'un graphe de flot de contrôle d'application, comportant trois méthodes, reliées par deux flots implicites

Inputs Les CFG, les ensembles $\mathcal{E}$ et $\mathcal{E}^*$, un sommet cible `t`

Output Une liste de sommets à parcourir dont un appartenant à $\mathcal{E}^*$.

gnéé presque fini mais dodo

5.4 Exécution automatique par instrumentation

5.4.1 Concept d'instrumentation

TODO (déjà fait need cp)

5.4.2 Instrumentation de programmes Java et Android avec Soot

TODO adapter ma prés

5.4.3 Exécution forcée d'un chemin prédéfini

utilisation de soot pour ajouter des tags
tags -¿ branches
exemple cfg3
boom on force des trucs
Un point sur la stabilité de la techno ?
Questions ouvertes : quid de la sémantique ?

6 Travaux liés

Lors de ce stage, un rapport technique a été publié par l'équipe SSE, qui développe Soot, sur l'outil Harvester [24]. Cet outil permet de modifier une application Android pour obtenir la valeur, pendant l'exécution, d'une variable désignée par analyste dans le code Jimple d'une application. En pratique, Harvester peut modifier le code d'une application pour forcer un certain chemin d'exécution, en supprimant le code non lié à l'instruction ciblée. Les sauts conditionnels sont adéquatement forcés, en prenant soin, à l'aide d'une analyse de flot de données, de ne pas forcer un saut qui ferait changer la valeur de la variable désignée à l'exécution. Cela permet par exemple de cibler une instruction utilisant la réflexion, donc indisponible statiquement, et de récupérer sa valeur pendant l'exécution.

Harvester n'est pas encore disponible mais les auteurs nous ont signalé leur intention de libérer le code d'ici quelques semaines. Si l'outil est facilement adaptable à nos besoins, il pourrait être une alternative intéressante à la partie instrumentation de notre implémentation, en lui donnant en entrée les parties du code que nous avons ciblé au préalable.

7 Conclusion

References

[1] Ui/application exerciser monkey. `https://developer.android.com/tools/help/monkey.html`.

[2] *Asroot*, CVE-2009-2692. http://www.cvedetails.com/cve/CVE-2009-2692, 2009.

[3] *RageAgainstTheCage*, exploitation d'une vulnérabilité d'adbd. http://c-skills.blogspot.fr/2010/08/droid2.html, 2010.

[4] Notes sur le vérificateur de bytecode (*bytecode verifier*) de Dalvik. https://android.googlesource.com/platform/dalvik/+/master/docs/verifier.html, 2010.

[5] Annonce du service Bouncer : Adding a new layer to Android security. http://googlemobile.blogspot.fr/2012/02/android-and-security.html, 2012.

[6] A closer look at Android RunTime (ART) in Android Lollipop. http://anandtech.com/show/8231/a-closer-look-at-android-runtime-art-in-android-l/, 2014.

[7] Parts de marché des OS pour smartphone. http://www.idc.com/getdoc.jsp?containerId=prUS24257413, 2014.

[8] Comparaison entre plusieurs décompilateurs Java et Android. http://shgck.io/docs/android/apk_decomp/, 2015.

[9] Secure Software Engineering blog. http://sseblog.ec-spride.de/, 2015.

[10] Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. *Security and Privacy in Communication Networks*, 127:86–103, 2013.

[11] Kevin Allix, Quentin Jerome, Tegawende F Bissyande, Jacques Klein, Radu State, and Yves Le Traon. A forensic analysis of Android malware – How is malware written and how it could be detected? In *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*, pages 384–393. IEEE, 2014.

[12] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 59. ACM, 2012.

[13] AppBrain. Applications Android sur Google Play. http://www.appbrain.com/stats/number-of-android-apps, 2015.

[14] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *ACM SIGSOFT Software Engineering Notes*, volume 24, pages 21–31. ACM, 1999.

[15] Laurent Delosières and David García. Infrastructure for detecting Android malware. In *Information Sciences and Systems 2013*, pages 389–398. Springer, 2013.

[16] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In ACM, editor, *18th ACM conference on Computer and communications security*, pages 627–638, October 2011.

[17] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. RiskRanker: scalable and accurate zero-day Android malware detection. In *10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 281–294. ACM, 2012.

[18] Michael C Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock Android smartphones. In *NDSS*, 2012.

[19] Martina Lindorfer and Matthias Neugschwandtner. ANDRUBIS-1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, San Jose, CA, USA, September 2014. IEEE Computer Society.

[20] Aravind MacHiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for Android apps. In *9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[21] Cheetah Mobile. 2014 half year security report. `https://developer.android.com/tools/help/monkey.html`.

[22] Jon Oberheide and Charlie Miller. Dissecting the Android Bouncer. *SummerCon2012, New York*, 2012.

[23] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting Android applications to Java bytecode. In *20th International Symposium on the Foundations of Software Engineering (SIGSOFT)*, page 6. ACM, 2012.

[24] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime data in Android applications for identifying malware and enhancing code analysis, 2015.

[25] Yunhe Shi, Kevin Casey, M Anton Ertl, and David Gregg. Virtual machine showdown: stack versus registers. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(4):2, 2008.

[26] Stephen Smalley and Robert Craig. Security Enhanced (SE) Android: bringing flexible MAC to Android. In *20th Annual Network & Distributed System Security Symposium*, San Diego, California, USA, February 2013. Internet Society.

[27] Lukas Weichselbaum. Andrubis: Android Malware Under The Magnifying Glass. Technical report, 2014.

[28] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: general security support for the Linux kernel. In *Foundations of Intrusion Tolerant Systems*, pages 213–213. IEEE Computer Society, 2003.

[29] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in Android applications. In *2nd ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104. ACM, 2012.

A Instructions Jimple

Ce tableau présente les instructions du langage Jimple. Celles-ci reflètent les instructions fondamentales du bytecode Java, dont les instructions sont détaillées ici :
`http://cs.au.dk/~mis/d0vs/jvmspec/ref-Java.html`

Instruction	Rôle
JAssignStmt	Assignation
JBreakpointStmt	Point d'arrêt (pour débug)
JEnterMonitorStmt	Prendre le lock d'une section critique
JExitMonitorStmt	Relâcher le lock d'une section critique
JGotoStmt	Saut inconditionnel
JIdentityStmt	?
JIfStmt	Saut conditionnel
JInvokeStmt	Appel d'une méthode
JLookupSwitchStmt	Comparaison et saut optimisé pour les *switch*
JNopStmt	*nop* (ne fait rien)
JRetStmt	Retour d'une *subroutine* (*e.g.* un bloc `finally`)
JReturnStmt	Retour d'une méthode
JReturnVoidStmt	Retour d'une méthode (sans valeur de retour)
JTableSwitchStmt	Saut pré-calculé
JThrowStmt	Lever une exception

B Heuristiques de ciblage de code

Le tableau ci-dessous présente, pour chaque catégorie de code suspect, les classes qui sont généralement associées, et le score de risque défini. Un nom de classe terminé par * signifie que toutes les classes de ce package sont incluses dans la catégorie.

Catégorie	Description	Classes	Score
Binary	Exécution de fichiers exécutables et gestion de processus natifs	java.lang.Runtime java.lang.Process java.lang.System	6
Dynamic	Chargement dynamique de code Java/Dalvik	dalvik.system.BaseDexClassLoader dalvik.system.PathClassLoader dalvik.system.DexClassLoader dalvik.system.DexFile	8
Reflection	Package du mécanisme de réflexion	java.lang.reflect.*	3
Crypto	Bibliothèques de protocoles et algorithmes cryptographiques	javax.crypto.* java.security.spec.*	3
Network	Gestion du réseau	java.net.Socket java.net.ServerSocket java.net.HttpURLConnection java.net.JarURLConnection	5
SMS	Gestion des contacts et données , envoi et réception de SMS & MMS	android.telephony.TelephonyManager android.telephony.SmsManager	10
Misc	Classes généralement utilisées pour dissimuler ses actions	java.util.Timer java.util.TimerTask	3