Détection d'anomalies sur Android

Vincent Turpaud

Rapport de stage

Détection d'anomalies sur Android

Author :
Vincent Turpaud

Supervisor :
Chamseddine Talhi

Résumé

Ce rapport a pour but d'expliquer les points importants du déroulement de mon stage, effectué à l'École de Technologie Supérieure (ÉTS) de Montréal au département Génie Logiciel. L'objectif de mon stage a été de créer un programme Android qui permet de monitorer les applications installées sur un téléphone et de détecter des comportements suspects. Le prototype mis au point lors de ce stage se démarque des programmes utilisés habituellement dans le domaine de la détection de logiciels malveillants. En effet, celui-ci n'a besoin d'aucune connaissance préalable de ce qu'on l'on considère comme anormal et ne se base pour faire son diagnostic que sur la connaissance des applications saines.

Mots-clé : détection de *malware*, Android, smartphone, détection d'anomalies, *monitoring*

Table des matières

1 Motivations

Le marché des téléphones intelligents, ou smartphones, a connu un essor considérable au cours de dernières années. Ces téléphones ont dépassé leur fonctionnalité première de communication vocale et sont désormais de véritables mini-ordinateurs, dotés d'un système d'exploitation propre qui permet à l'utilisateur d'installer toutes sortes d'application. Bien que le nombre de modèles de téléphone soit considérable, deux systèmes d'exploitation prédominent largement le marché : l'iOS d'Apple et Android de Google.

Ce dernier permet à n'importe quel utilisateur ayant quelques connaissances en programmation de créer et publier ses propres applications sur le site *Google Play*, où les autres utilisateurs peuvent les télécharger. Ce site se base sur un système de réputation et de signature pour assurer la sécurité des utilisateurs.

Ce système est loin d'être parfait. D'une part, des applications malicieuses (*malwares*) se retrouvent régulièrement sur ce marché. Dès qu'il prend connaissance d'une telle contamination, Google réagit en supprimant les applications suspectes du marché. Cependant, le temps nécessaire à cette réaction laisse le temps à de nombreux utilisateurs d'être infectés [1, 2]. D'autre part, de nombreux utilisateurs sont contraints d'utiliser des marchés alternatifs. Par exemple, à l'heure actuelle, *Google Play* ne supporte pas les applications payantes en Chine [3]. Les utilisateurs se tournent donc massivement vers des marchés alternatifs [4]. Ces marchés ne sont pas contrôlés et sont donc infestés de programmes malveillants : il est crucial pour ces utilisateurs d'être à même de les détecter afin de limiter les risques.

1.1 Enjeux de la détection de logiciels malveillants sous Android

Les domaines d'utilisation des smartphones dans la vie de tous les jours sont très variés. Ceux-ci suivent leurs utilisateurs partout et peuvent contenir énormément de données à leur sujet, que ce soit dans le cadre de leur vie privée ou professionnelle. En effet, on peut tout connaître des projets d'un cadre en ayant accès à ses mails, tout comme on peut facilement tracer les grandes lignes de ses habitudes de consommation en observant ses déplacements à l'aide d'un GPS. Qui plus est, le téléphone lui-même stocke qu'on le veuille ou non des informations telles que le numéro de téléphone de l'utilisateur, qui pourraient intéresser des démarcheurs peu scrupuleux.

Pour toutes ces raisons, les smartphones sont des cibles privilégiées pour les programmes malveillants. N'importe quel utilisateur peut être affecté par un *malware* qui tente de récupérer des données personnelles ou d'appeler un numéro surtaxé, ce qui rend la publication d'une telle application très lucrative. Qui plus est, l'impact sur l'utilisateur est direct. Outre l'intrusion dans la vie privée, qui est un problème rencontré de manière récurrente sur

internet, les envois de SMS surtaxés lui font perdre de l'argent directement et peuvent donc poser de gros problèmes si ils ne sont pas détectés rapidement.

1.2 Réalité de la menace

La menace de trouver des logiciels malveillants sur *Google Play* est bien réelle [2]. Le premier cas d'application suspicieuse retirée du marché par Google date de Janvier 2010. Il s'agissait d'un *spyware*, programme destiné à collecter à l'insu de l'utilisateur des données personnelles, en l'occurrence des informations bancaires.

En Août 2010 fut découvert le premier *Trojan* (cheval de Troie) destiné à envoyer des SMS à l'insu de l'utilisateur, répondant au nom de *Fake Player*. Le programme se fait passer pour un lecteur de vidéo et envoie à l'insu de l'utilisateur des SMS vers un numéro surtaxé, pour un coût de 5 dollars par message. Il s'agit du premier exemple de programme malveillant sur Android qui ne se contente pas d'espionner l'utilisateur et lui fait perdre de l'argent directement.

Depuis ces premiers exemples et en particulier à partir de 2011, le nombre de programmes malveillants détectés sous Android a fortement augmenté, de paire avec l'utilisation de cette plate-forme. Les rapports parlent d'une « augmentation de 400% des application malicieuses Android depuis l'été 2011 »[5]. Si les marchés d'applications alternatifs en sont la première source, on voit régulièrement Google annoncer qu'il vient de retirer des programmes suspicieux de son *Google Market*. Des antivirus sont disponibles pour essayer de limiter les dégâts, mais il faut reconnaître que leur efficacité est limitée. Ils sont en particulier totalement inefficaces pour détecter de nouveaux *malwares*, qui ne sont pas présents dans leurs bases de données.

1.3 Comportement des logiciels malveillants et méthodes de diffusion

Les logiciels malveillants utilisent différentes méthodes pour se dissimuler, que ce soit aux yeux de l'utilisateur ou des logiciels de détection. La méthode la plus courante de diffusion est le *repackaging* : l'auteur du logiciel malveillant désassemble une application connue, y ajoute du code, puis publie le programme modifié sur un marché alternatif en le faisant passer pour le programme d'origine.

Pour éviter d'être détecté par les antivirus, l'auteur du logiciel malveillant va souvent éviter de mettre directement toute sa *payload*, c'est à dire le code qui va mener l'attaque, dans son application. En effet, les *payload* contiennent en général des attaques permettant de briser la sécurité mise en place par Android. Ces attaques sont peu nombreuses et possèdent donc des signatures qu'un antivirus peut détecter. Pour éviter cela, l'auteur n'injecte pas directement la *payload* dans l'application, mais fait en sorte que l'application la

télécharge ultérieurement.

À ce jour, plusieurs familles de *malwares* implémentent une variante de cette attaque, parmi lesquelles on peut citer *BaseBridge*, *DroidKungFu*, *Plankton*, *BadNews* et *Anserverbot*. Les quatre premiers ont par ailleurs déjà été détectés sur *Google Play* d'après rapports de *Lookout*. Chaque variante a son propre scénario d'attaque et on peut imaginer que de nouvelles feront leur apparition dans le futur.

Une application infectée par *BaseBridge* attend qu'une nouvelle mise à jour soit disponible. Lorsque l'application doit afficher une fenêtre invitant à effectuer cette mise à jour, BaseBridge détourne celle-ci [6]. Au lieu de télécharger la vraie mise à jour, le programme installe une fausse mise à jour, déjà présente sur le téléphone sous la forme de ressource du programme infecté, contenant la *payload*. En évitant d'inclure celle-ci dans le code de l'application infectée dès le départ, les chances d'être détecté pour *BaseBridge* sont réduites.

DroidKungFu [1] agit d'une manière similaire, mais au lieu de stocker la « mise à jour »dans les ressources de l'application, celui-ci fait en sorte de la télécharger depuis le réseau. Pour cela, la notification de mise à jour est effectuée par une librairie tierce [6] qui permet d'effectuer des mises à jour de manière légitime. Cependant, cette librairie permet de télécharger les mises à jour depuis d'autres marchés d'application que *Google Play* : la mise à jour téléchargée est en fait une fausse mise à jour qui contient la *payload*.

Ces deux exemples nécessitent au moment de la mise à jour l'accord explicite de l'utilisateur. En effet, celui-ci reçoit une notification et il peut très bien refuser la mise à jour si il suspecte quelque chose en regardant par exemple les nouvelles permissions demandées. Les familles *Plankton* et *Anserverbot* se montrent en revanche plus discrètes [6]. Celles-ci téléchargent un fichier .jar contenant la *payload*, qui est exécuté dynamiquement par le système. Ceci permet d'échapper à l'analyse statique [7, 8].

Le dernier exemple en date, *BadNews*, a également eu un fort impact, puisque 32 applications infectées par ce dernier ont été téléchargées entre 2 et 9 millions de fois sur *Google Play*, selon les statistiques de ce dernier. *BadNews* se fait passer pour un réseau de publicité dans dans une librairie tierce [9]. Les librairies tierces sont des ressources utilisées par les programmeurs qui leur permettent d'implémenter des fonctionnalités classiques sans avoir à les réécrire, en l'occurrence l'intégration de publicités dans leur application. Par conséquent, une application peut être infectée à l'insu même de son développeur, puisqu'il suffit que celle-ci utilise une librairie infectée pour qu'elle soit également contaminée.

Une fois *BadNews* installé, celui-ci met le téléphone en écoute d'un serveur de contrôle, qui envoie des instructions toutes les 4 heures. Parmi ces instructions, le serveur peut essayer de faire télécharger à l'utilisateur d'autres applications malicieuses telles que *AlphaSMS* en les faisant passer pour des mises à jour d'applications courantes comme Skype.

Toute ces méthodes peuvent prendre l'utilisateur au dépourvu, puisqu'il a tendance à faire confiance à une application connue téléchargée sur le marché officiel. Les programmes infectés continuent de fonctionner normalement en apparence, mais un grand nombre d'actions nuisibles pour l'utilisateur peuvent avoir lieu à son insu. Très vite, celui-ci peut sans s'en rendre compte avoir envoyé un grand nombre de SMS surtaxés, s'être fait voler toutes ses données personnelles, ou même ses coordonnées bancaires.

1.4 Problème visé

Dans notre travail, nous nous intéressons aux mises à jour malicieuses, qui, comme nous venons de le voir, sont largement utilisées par de nombreuses applications malveillantes pour cacher leur activité. Nous partons du constat que, quelque soient les méthodes de dissimulation utilisées par les *malwares*, celles-ci impliquent à un moment ou l'autre un changement de comportement d'un programme. En effet, toutes ces méthodes suivent une même idée : éviter d'exécuter le code malveillant directement au premier lancement du programme afin d'éviter d'être détecté.

Si le programme télécharge sa *payload* ultérieurement, celle-ci va lui faire exécuter des actions nouvelles. De la même manière, un programme malicieux qui fait un sorte qu'un autre programme totalement bénin effectue une fausse mise à jour change le comportement de ce dernier.

Un exemple commun d'une telle attaque est le suivant : un utilisateur possède sur son téléphone diverses applications, dont Skype. Un jour, il télécharge une application sur *Google Play* qui s'avère être infectée par *BadNews*, chose qui est déjà arrivée plusieurs millions de fois [9, 10]. Cette application ne va rien faire de suspect pendant deux ou trois jours. Après ce laps de temps, l'utilisateur reçoit un message lui indiquant que Skype doit effectuer une mise à jour. Se disant que ce programme est connu et ne présente aucun risque, il accepte. Cette mise à jour est en fait une mise à jour malicieuse lancée par l'application malveillante : une fois celle-ci effectuée, Skype va commencer à collecter les données personnelles de l'utilisateur et à les envoyer sur un serveur privé.

Dans cet exemple, quelle que soit la faille exploitée par le logiciel malveillant pour lancer une mise à jour de Skype, et quelles que soient les actions malveillantes qu'il lui fait exécuter, nous pensons pouvoir observer un changement de comportement de Skype. En effet, on peut supposer que dans notre exemple, en temps normal, Skype n'envoie aucun SMS surtaxé. Même si Skype offre la possibilité de le faire, l'utilisateur peut très bien n'avoir jamais utilisé cette fonctionnalité. L'application malicieuse *AlphaSMS* envoie justement un grand nombre de SMS surtaxés [11]. On voit donc le comportement de Skype changer, ce qui peut d'après nous être un bon indicateur d'actions suspicieuses.

L'idée de notre approche est de surveiller ces changements de compor-

tement pour chaque application afin de détecter les actions malveillantes. De cette manière, si une application malicieuse temporise d'une manière ou d'une autre son attaque pour échapper à une détection classique, notre système devrait être à même de détecter le changement de comportement provoqué par l'attaque.

2 Approche proposée

Notre approche consiste à *monitorer*, c'est à dire surveiller, chaque application présente sur le téléphone. Une alerte est levée dès qu'un programme change de comportement. Cette idée est une reprise de l'approche comportementale utilisée dans le domaine de la détection d'intrusion, dont nous parlerons plus en détail dans la partie 2.2.

2.1 Objectifs

Le *monitorage* des applications s'effectue en deux phases. Une première phase, dite d'apprentissage, consiste à construire un modèle « normal »du programme, en se basant sur son comportement durant une période où l'on considère qu'il n'effectue aucune action malicieuse. Une fois ce modèle construit, la phase de détection à proprement parler peut commencer : on observe le comportement du programme au cours de son exécution et on le compare au modèle construit lors de la phase d'apprentissage. Si le comportement s'éloigne trop du modèle, une alerte est levée.

Le principal avantage de cette méthode et qu'elle ne nécessite aucune connaissance préalable des comportements suspicieux recherchés. En effet, on se base uniquement sur le modèle du comportement normal du programme pour choisir de lever ou non des alertes. Ceci est particulièrement intéressant pour détecter de nouveaux types de programmes malveillants, pas encore référencés, pour lesquels une approche de détection par signature ne pourra pas fonctionner, puisque la signature ne sera pas encore connue.

Cette approche a bien sûr ses limites. Comme nous l'avons précisé, nous supposons qu'aucune action malveillante n'a lieu durant la phase d'apprentissage. En effet, des actions malveillantes ayant lieu durant cette phase mèneraient à un modèle erroné, qui autoriserait des actions illicites. Un *malware* tel que ceux rencontrés sous Windows qui exécute son code immédiatement sans chercher à se cacher ne pourra donc pas être détecté. Cependant, de tels *malwares* ne sont pas répandus sur Android. En effet, le nombre de *failles* - faiblesses du système d'exploitation utilisée pour contourner sa sécurité - connues sur Android est assez limité [12], ce qui fait qu'un antivirus classique fonctionnant par signature aurait peu de mal à les détecter.

Notre travail se veut donc complémentaire. Un antivirus classique se charge de détecter les programmes malicieux connus en détectant leur signature, tandis que notre prototype cherche à détecter les symptômes d'une

contamination par une application malicieuse qui serait passée entre les mailles du filet.

2.2 Travaux précédents : détection d'intrusion par approche comportementale

Comme nous l'avons mentionné auparavant, nous nous sommes inspirés de nombreux travaux effectués dans le domaine de la détection d'intrusion. Le concept de modèle de comportement normal que nous utilisons a été introduit par Forrest et coll. en 1996 [13]. Cette publication décrit comment construire un modèle de comportement normal en se basant sur de courtes séquences de n appels système appelées *n-grams*. Le modèle est constitué de l'ensemble des *n-grams* observés lors de la phase d'apprentissage. Lors de la phase de détection, on regarde les séquences d'appels système effectuées par le programme et on compte celles qui ne sont pas présentes dans le modèle, que l'on considère comme anormales. Si le taux de séquences anormales dépasse un certain seuil, on considère que le programme se comporte de manière anormale.

Toute cette approche est destinée à surveiller l'activité de serveurs. Les services accessibles à distance qu'offre un serveur, tels qu'un site web, sont autant de portes d'entrées pour un attaquant. Celui-ci essaie de se servir de failles présentes dans ces services pour leur faire exécuter les commandes qu'il veut, et ainsi faire faire ce qu'il veut au serveur. L'idée de base de l'approche est de considérer que dans le cadre d'un fonctionnement normal, un service n'effectuera qu'un ensemble fini de suites d'appels système. La phase d'apprentissage décrite ci-dessus permet d'obtenir un ensemble le plus proche possible de l'ensemble des suites d'appels système que l'on peut considérer comme normales. Un attaquant prenant le contrôle du service lui fait effectuer des appels système qui sortent de cet ensemble : lors de la phase de détection, les séquences n'ayant pas été enregistrées dans le modèle sont considérées comme anormales.

Depuis ce travail, de nombreuses publications ont repris le concept de modèle de comportement normal en faisant évoluer les critères utilisés pour sa construction [14, 15]. Certaines de ces évolutions portent sur les données utilisées pour la construction du modèle, par exemple pour tenir compte des paramètres utilisés avec les appels système [16]. D'autres publications tiennent compte dans la construction du modèle de la fréquence d'apparition des séquences d'appels système [17, 15] : au lieu de simplement mémoriser les appels système qui apparaissent lors de la phase d'apprentissage, les auteurs comptent leurs apparitions. Un taux d'anomalie inversement proportionnel à la fréquence d'apparition est associé à chaque séquence, ce qui permet d'obtenir un modèle plus précis du comportement du programme, ainsi qu'une tolérance à quelques anomalies lors de la phase d'apprentissage : même si celles-ci sont présentes dans le modèle, elles le sont en faible nombre, et

conservent donc un taux d'anomalie important.

D'autres évolutions portent sur la manière dont construire le modèle. Au lieu d'utiliser des *n-grams* comme dans la publication initiale, des variantes utilisent des algorithmes d'apprentissage automatique comme des modèles de Markov [15, 18], des réseaux de neurones [19, 20], ou des modèles de Bayes [21]. Toutes ces approches essaient de créer un modèle plus précis des données, afin de réduire le taux de faux-positifs engendrés, mais présentent une complexité plus importante. [14]

La diversité des expériences et des conditions dans lesquelles elles sont effectuées empêche de pouvoir comparer ces méthodes en se basant uniquement sur les résultats fournis par les études : des implémentations dans un même cadre pour les tester sur un même jeu d'expérimentations est nécessaire. De rares études ont effectué une telle comparaison [14, 15] d'où sont sortis quelques tendances générales. Premièrement, aucun algorithme ne semble prédominer ou se situer en dessous des autres. Le choix des données utilisées pour l'entraînement a une bien plus grande influence sur les résultats que le choix du modèle. Deuxièmement, l'utilisation de méthodes sophistiquées pour la création du modèle engendrent une plus forte complexité, et donc une plus forte surconsommation en ressources.

Notre analogie est la suivante : nous considérons que chaque application installée sur un téléphone correspond à un service sur un serveur, potentiellement vulnérable. Nous supposons également que chaque application possède un comportement normal consistant en un ensemble fini de séquences d'appels système, que l'on essaie d'approcher lors de la phase d'apprentissage. Une mise à jour d'une application joue le même rôle qu'une communication du service avec un client : c'est une chose nécessaire, mais qui peut présenter un risque. Pour essayer de détecter une mise à jour malicieuse, qui correspond à une attaque, on *monitore* l'application. Tout comme l'attaque fait effecteur de nouvelles actions au service, la mise à jour malicieuse change le comportement de l'application. On peut donc espérer la détecter de la même manière.

L'idée de « comportement normal »pour une application Android n'a pas encore été démontrée et reste une supposition de notre part : nos expériences en partie 4 vont avoir entre autre pour but de vérifier cette hypothèse.

2.3 Travaux précédents : détection de malware sur Android

Deux approches sont utilisées pour la détection de logiciels malveillants sur Android et en général : l'approche statique [22] et l'approche dynamique [23]. L'analyse statique se base uniquement sur le code source d'un programme pour le scanner, tandis que l'analyse dynamique l'exécute dans un environnement dédié pour observer son comportement. Les analyses dynamiques font l'objet de plus de recherches dans le domaine des smartphones.

Historiquement, les premiers travaux portant sur la détection de logiciels

malveillants sur les smartphones se basaient principalement sur l'observation de la consommation de la batterie [24]. Ceux-ci consistaient à surveiller la consommation en batterie du téléphone et à la comparer à un patron de consommation normal pour détecter les anomalies.

L'architecture d'Android utilise un système de permission : chaque application indique lors de son installation à quelles ressources du téléphone elle a besoin d'accéder. L'analyse de ces permissions pour détecter des combinaisons suspectes a fait l'objet de plusieurs publications [25]. En effet, les logiciels malveillants ont tendance à demander de plus nombreuses permissions que les logiciels bénins. Si en théorie un utilisateur est à même de voir à l'installation d'une application si celle-ci demande trop de permissions pour ce qu'elle est sensée faire, en pratique à peu près toutes les applications affichent une liste de permission relativement longue que l'utilisateur ne prend pas forcément le temps de regarder. Une analyse automatique avertissant l'utilisateur si les permissions demandées deviennent trop dangereuses peut aider à résoudre ce problème.

Schmidt et coll. ont proposé de *monitorer* les événements au niveau du noyau de Linux [26]. Une modification de ce dernier leur a permis d'extraire des données telles que les appels système et les modifications de fichiers afin de construire un modèle de comportement normal. Cependant, l'article a été écrit avant que les *malwares* se répandent sous Android, et aucun test n'a donc pu être effectué. Dans une publication ultérieure [27], ces mêmes auteurs proposent d'effectuer le *monitorage* an lançant les applications dans un environnement sécurisé dédié, dit *sandbox*. Cette fois, le *monitorage* a pour but de détecter des signatures de *malwares*.

Une approche assez différente a été proposée sous le nom de TaintDroid [28] par Enck et coll. TaintDroid permet de tracer dynamiquement lors de l'exécution d'une application la circulation des données sensibles de l'utilisateur. Ce système permet de détecter les applications qui volent les informations sensibles telles que les coordonnées GPS ou le carnet d'adresse.

Si de nombreux paramètres sont utilisés pour modéliser le comportement d'un programme dans les différentes publication, l'absence d'étalon empêche de réellement pouvoir comparer les résultats obtenus. Cependant le choix même des paramètres a une influence sur le type d'attaque que l'on cherche à détecter. Surveiller la consommation de batterie peut être efficace pour détecter un logiciel malveillant peu discret qui génère énormément de calcul, tandis que surveiller les données sensibles comme TaintDroid peut permettre de détecter le vol d'informations personnelles. Surveiller les appels aux différentes API (*Application Programing Interface*) tels que les appels système donne des informations plus générales, mais également plus difficiles à interpréter pour l'homme.

Au cours de notre travail, nous nous servirons des appels système. En effet, ces paramètres présentent l'avantage d'être utilisés à la fois dans les travaux concernant la détection d'intrusion décrits dans le paragraphe pré-

cédent et dans le domaine de la détection de logiciels malveillants. Nous aurions également pu nous servir des appels à toutes les API, mais ceci aurait considérablement augmenté le nombre d'appels à surveiller. Un modèle construit de la sorte serait beaucoup plus volumineux qu'avec seulement les appels système et plus lourd à traiter : un filtrage préalable serait nécessaire. Étant donné les contraintes rencontrées sous Android (voire partie 2.5), nous avons donc décidé de nous limiter aux appels système.

2.4 Limites de notre approche

Comme nous l'avons vu, les logiciels malveillants en général et en particulier sous Android ont mis au point toute une panoplie de techniques pour échapper à la détection des antivirus. Bien souvent, les antivirus procèdent à un scan de l'application avant ou au cours de l'installation. Échapper à cette détection initiale est le but premier de ces techniques. Par conséquent, sous Android, un logiciel malveillant évite en général d'exprimer son comportement malveillant directement. En effet, si le *malware* ne télécharge sa *payload* que lors de la prochaine mise à jour du programme infecté, son comportement malveillant ne sera exprimé qu'après plusieurs jours.

Quelle que soit la méthode utilisée pour se cacher, on a donc deux phases de comportement. Une phase de comportement normale, puis l'exécution de la *payload*. L'idée même de notre approche est d'essayer de détecter ce changement de phase. À priori, la *payload* fait effectuer au programme des actions qu'il n'effectue pas lors de son comportement normal. Nous supposons donc que si l'on possède une représentation du fonctionnement normal, nous serons à même de détecter une exécution de *payload*. Cette méthode a l'avantage de ne faire aucune supposition sur la nature de la *payload*.

Bien entendu, un *malware* exprimant son comportement malicieux d'emblée ne pourra pas être détecté, puisqu'il introduirait des actions malicieuses dans le modèle normal. Nous supposons qu'un antivirus classique serait à même de détecter une telle attaque : notre travail se veut complémentaire.

Il faut aussi noter qu'une mise à jour d'application légitime peut également faire grandement changer celle-ci. Par exemple, l'introduction d'une nouvelle fonctionnalité légitime peut faire effectuer à une application de nouvelles actions : notre système les détecterait comme anomalies. Cependant, on peut espérer que de telles mises à jour soient plutôt rares. Notre but ici est de déterminer si on peut détecter de tels changements de comportement : la différenciation entre mise à jour légitime et malicieuse sera l'objet de travaux ultérieurs.

On peut également dores et déjà imaginer des attaques contre notre approche. En effet, un attaquant possédant une bonne connaissance de notre modélisation peut forger une attaque en faisant en sorte d'effectuer les actions malicieuses tout est restant dans les limites du modèle normal. Ceci dit, chaque smartphone construit son propre modèle. On peut donc espérer que

cette diversité rende l'implémentation systématique de ce genre d'attaque dans un *malware* plus compliquée.

2.5 Contraites rencontrées sur Android

Au cours de notre projet, nous devons être particulièrement vigilants à propos de la consommation de ressources. En effet, les ressources disponibles sur un smartphone sont très limitées et ne permettent pas d'effectuer une analyse aussi poussée que sur un ordinateur. Les limitations apparaissent à deux niveaux : la consommation du CPU et l'espace de stockage disponible.

La consommation du CPU est selon nous la contrainte principale. En effet, notre programme *monitore* les applications en temps réel durant leur fonctionnement et génère par conséquent une surconsommation. Si cette surconsommation devient trop importante, l'utilisateur risque de percevoir un ralentissement, ce qu'il faut éviter au maximum. De plus, cette surconsommation a un impact sur la batterie et nous ne voulons pas que notre prototype force l'utilisateur à recharger son téléphone plus fréquemment.

La consommation CPU est très inégale. Le processeur est très sollicité lorsque l'utilisateur utilise activement son téléphone pour jouer à un jeu ou regarder un film, mais la plupart du temps le téléphone est en veille. De plus, beaucoup d'utilisateurs branchent leur téléphone la nuit pendant qu'ils dorment : durant cette période, la batterie ne pose plus problème. Un système de *monitorage* viable doit donc être capable d'adapter son fonctionnement : fonctionner de manière ralentie lorsque l'utilisateur sollicite le processeur, et tourner à plein régime lorsque le téléphone est branché.

Il est dès lors assez facile d'imaginer la programmation d'un logiciel malveillant qui lance son attaque lorsque le processeur est très chargé pour éviter notre détection, mais une attaque de ce type nous semble peu probable. En effet, pour éviter d'être détectés, les *malwares* ont tendance à lancer leur activité malicieuse lorsque l'utilisateur n'utilise pas le téléphone, comme le tristement célèbre *DroidDream* [29], qui tire son nom de sa capacité à lancer sa charge utile entre 11h du soir et 8h du matin, c'est à dire lorsque l'utilisateur a le plus de chances de dormir, d'où son nom. On a donc finalement plus de chances de trouver une activité anormale lorsque le téléphone est en veille.

Une alternative serait de stocker les traces lorsque le téléphone est sollicité afin de reporter le traitement à un moment où celui-ci est au repos. En effet, ceci permettrait d'éviter le type d'attaque cité ci-dessus en traitant toutes les traces de manière égale. Ceci implique cependant de stocker celles-ci sur le téléphone en attendant de pouvoir effectuer le traitement. Un travail serait très probablement nécessaire pour trouver un compromis entre compression des traces et complexité de cette compression. En effet l'espace de stockage est limité sur un téléphone, ce qui risque de poser problème si un grand nombre de traces est stocké, mais une compression trop gourmande en

ressources ne résoudrait en rien le problème de consommation CPU. Si sur ordinateur un disque dur contient facilement 1To de données, un smartphone doit se limiter à quelques Go. Toutes les applications s'efforcent de ne pas occuper plus de quelques Mo maximum. Cette limite ne doit donc pas être dépassée pour la sauvegarde des modèles ou des traces des applications.

Si nous pouvons imaginer de déporter le stockage et le traitement sur un ordinateur distant pour contourner ces problèmes, un tel système provoquerait une forte utilisation du wi-fi pour transférer les données, ce qui solliciterait d'autant plus la batterie. De plus, ceci poserait des problèmes de disponibilité. Le réseau wi-fi peut ne pas être disponible, l'ordinateur distant ne pas répondre ou être en panne. Nous privilégierons donc les modélisations permettant d'effectuer l'ensemble des opérations sur le téléphone. Le but de nos expériences va être de trouver un bon compromis entre performances et consommation en ressource.

3 Notre contibution

Nous allons présenter dans cette partie notre prototype en détail : modules, choix de données traitées, algorithmes utilisés. Étant donné l'absence de travaux préalables de *monitorage* sur Android, nous n'avons dans certains cas pas été à même de choisir parmi plusieurs possibilités sans effectuer de tests. Par exemple, nous avons implémenté deux modélisations possibles, que nous allons comparer dans la partie dédiée aux résultats.

3.1 Schéma global

La figure 1 représente les différents modules de notre prototype.
Les modules ont les rôles suivants :

1. Strace exécute une commande Linux nommée *strace* qui récupère les appels système effectués par un processus. Les appels sont obtenus sous la forme d'un flux de données que nous récupérons et envoyons au module suivant sous forme de séquences d'appels système de taille fixe, appelées traces.
2. Le module de normalisation récupère ces séquences d'appels système et effectue tout le traitement nécessaire à la mise en forme pour notre analyse.
3. Le module de création des modèles est chargé de créer une modélisation d'un comportement normal à partir d'un jeu de traces.
4. Le module de scan de trace est chargé de comparer des traces à un modèle de comportement normal fabriqué au préalable pour déterminer si le comportement de l'application qui les a crée est suspicieux.

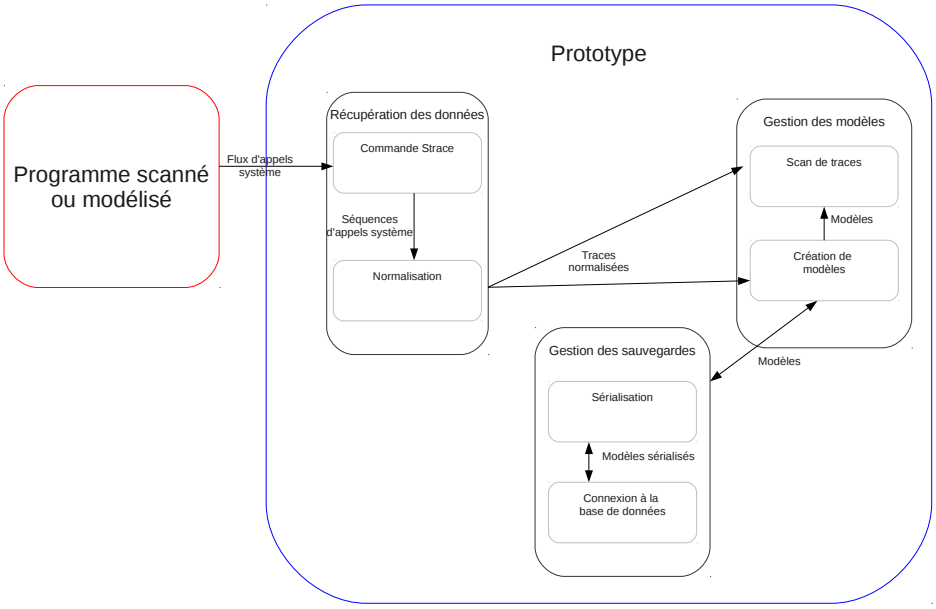


FIGURE 1 – Modules du prototype

5. Le module de sérialisation met les objets utilisés pour représenter les modèles sous forme de chaînes de caractères afin de pouvoir les sauvegarder.
6. Enfin, le module de connexion à la base de donnée se charge d'envoyer les requêtes qui permettent de sauvegarder ou charger des modèles.

Une description plus détaillée de chacun est apportée dans les parties 3.2 à 3.3.2. Ces modules sont interchangeables et permettent de facilement tester différents paramètres. En effet, nous pouvons changer la normalisation des traces ou le type de modèles utilisé sans toucher au reste du programme, ce qui permet à notre prototype d'être facilement réutilisé.

3.2 Extraction des données

3.2.1 Strace

Comme nous l'avons spécifié auparavant, *strace* est une commande Linux qui permet d'obtenir le flux des appels système effectués par une processus. Dans notre cas nous l'utilisons une fois par processus *monitoré*. Cette commande n'est pas disponible par défaut dans les versions récentes d'Android, aussi nous avons du l'installer manuellement. De plus, son utilisation nécessite les droits *root*.

Ce module lance *strace* avec les paramètres nécessaires pour récupérer le flux d'appels système de l'application voulue. Ce flux est coupé en séquences d'appels système qui sont transférées au module suivant. Le flux renvoyé par la commande *strace* renvoie également les paramètres passés aux appels système : nous avons choisi de ne pas tenir compte de ceux-ci pour diminuer le coût en ressources de la modélisation. En effet, s'ils pourraient permettre d'obtenir une modélisation plus précise, les prendre en compte reviendrait à multiplier le nombre de données contenues dans nos modèles et donc augmenter à la fois la complexité de l'analyse et l'espace de stockage nécessaire. Nous avons donc décidé de voir quelles performances sont possible pour une analyse qui ne tient pas compte des paramètres, quitte à revenir sur ce choix plus tard si les résultats obtenus ne sont pas satisfaisants.

3.2.2 Normalisation

Ce module se charge de faire le tri de ce qui nous intéresse ou pas dans les appels système.

Il nous permet de filtrer les appels système que nous souhaitons prendre en compte ou ignorer. En effet, nous pensons que filtrer des appels peu significatifs peut faire gagner du temps de calcul lors de la modélisation, ainsi que diminuer la taille des modèles. Cependant, la détermination de l'importance des différents appels est un travail à part entière que nous n'avons pas encore effectué à l'heure actuelle.

Nous utilisons également ce module pour donner aux séquences une taille standard. La taille des traces utilisées pour le scan a une certaine influence sur la précision de celui-ci, dont nous discutons dans la partie 4. Nous interagissons donc avec ce module pour faire des tests avec différentes tailles de trace.

3.3 Création du modèle et scan

Le module de création permet de créer des modélisations de comportements normaux des programmes à partir de séquences d'appels système. Les modélisations ainsi créées permettent de scanner, dans le module de scan de traces, d'autres jeux de séquences construits ultérieurement. Ces modélisations peuvent être sauvegardées pour une utilisation ultérieure via les modules de sauvegarde.

Nous avons implémenté deux types de modélisations, que nous décrivons dans les parties 3.3.1 et 3.3.2. Chaque modélisation a sa propre manière de parcourir et décrire les traces. Leurs performances sont comparées en partie 4.

La création du modèle est la phase la plus calculatoire du fonctionnement de notre prototype. Ceci dit, celle-ci est ponctuelle. Dans un premier temps on accumule les traces, puis lorsque l'on en a assez, on crée le modèle. Une

fois celui-ci crée on l'utilise pour scanner d'autres traces, mais on ne le modifie plus. Par conséquent, on observe simplement un pic de consommation pendant quelques secondes lors de la création du modèle, ce qui a une moins grande influence selon nous que la surconsommation permanente engendrée par le scan des traces, qui s'effectue normalement en temps réel (voire partie 4.4).

Par conséquent, dans nos choix de modèles, nous nous sommes focalisés sur la possibilité d'adapter la phase de scan aux ressources disponibles. En effet, nos deux modélisations permettent de « couper »les données tout en gardant une description cohérente et ainsi effectuer une analyse moins poussée, mais moins coûteuse.

3.3.1 Modèle de type Lookahead pairs

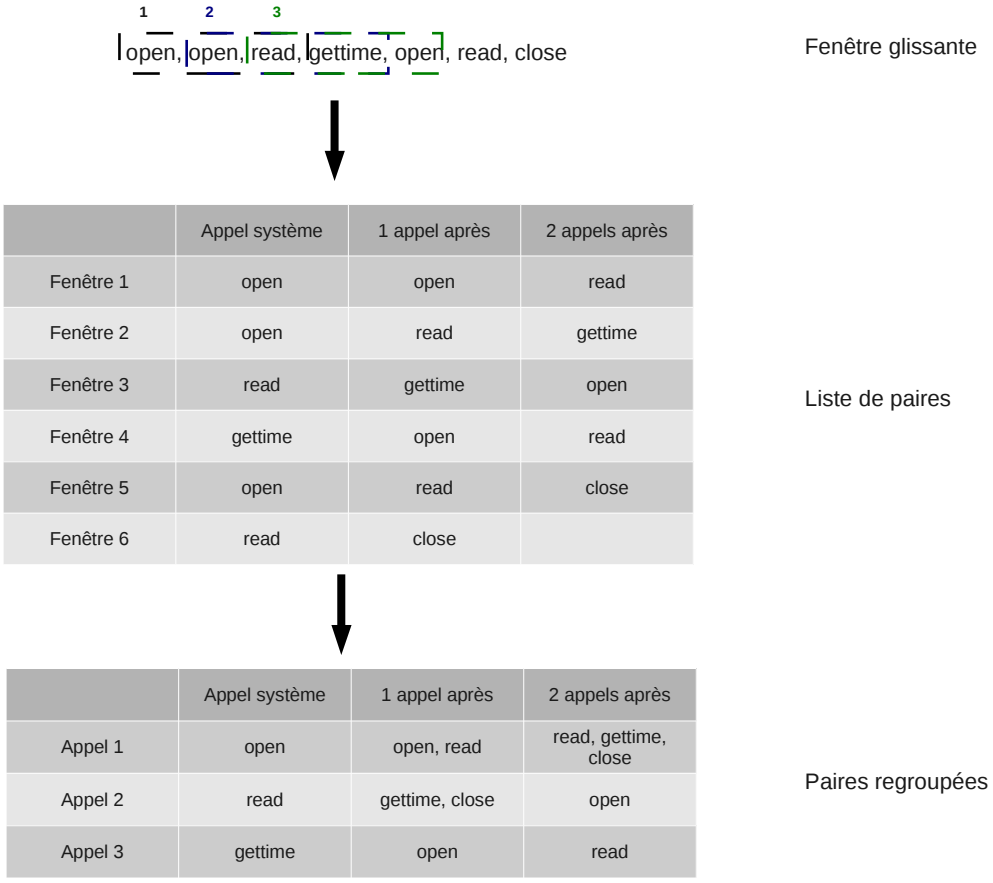
Ce modèle reprend le travail de Forrest [13]. Il consiste à associer à chaque appel système des listes d'appels système suivants possibles. La première liste associée à un appel système est la liste des appels pouvant suivre immédiatement celui-ci, la 2e liste est la liste des appels possibles en 2e position après, etc.

Pour construire le modèle, une fenêtre glissante de largeur n (nombre de listes + 1) parcourt une trace. Le premier appel de la fenêtre est le premier élément de la paire. Le deuxième appel est ajouté à la première liste, le troisième appel à la deuxième liste, et ainsi de suite. La figure 2 montre un exemple de construction des paires.

L'ensemble des paires ainsi formées constitue notre modèle : il s'agit de l'ensemble des paires que l'on considère comme normales. Le scan d'une trace consiste à faire glisser une fenêtre de la même manière que pour la construction et à comparer les paires trouvées avec celles présentes dans le modèle. Une paire qui n'est pas dans le modèle est considérée comme anormale. Une trace est considérée comme anormale lorsque son taux de paires anormales dépasse un certain seuil, qu'il nous faudra déterminer par l'expérience.

Nous avons choisi ce modèle pour son faible coût. En effet, il s'agit de la modélisation la plus simple que nous ayons trouvé, à la fois au niveau de l'analyse que de l'espace nécessaire à sa sauvegarde. Le nombre d'opérations effectué pour la construction du modèle est proportionnel à la longueur de la trace, puisque l'on parcourt une fois la trace et que l'on se contente de ranger dans un tableau les appels que l'on y trouve. La taille du modèle est dans le pire cas de l'ordre de $(nombredetrace)^2 * largeurdelafentre$. Comme nous le verrons dans la partie 4.5, en pratique nous avons obtenu beaucoup moins de paires que ce maximum.

De plus, la structure du modèle permet d'adapter l'analyse. Par exemple, si on construit un modèle contenant des paires jusqu'à 7 appels plus loin, on peut très bien lors de l'analyse se limiter à regarder les paires jusqu'à 4



	Appel système	1 appel après	2 appels après
Fenêtre 1	open	open	read
Fenêtre 2	open	read	gettime
Fenêtre 3	read	gettime	open
Fenêtre 4	gettime	open	read
Fenêtre 5	open	read	close
Fenêtre 6	read	close	

	Appel système	1 appel après	2 appels après
Appel 1	open	open, read	read, gettime, close
Appel 2	read	gettime, close	open
Appel 3	gettime	open	read

FIGURE 2 – Construction des paires

appels plus loin si les ressources disponibles ne nous permettent pas de faire plus.

Il faut néanmoins être conscient d'une certaine surévaluation du comportement avec ce modèle. En effet, en gardant en mémoire les paires et non les sous-séquences complètes (*n-grams*), notre modèle autorise certaines séquences qui n'apparaissent en réalité pas dans les traces utilisées pour le construire. Par exemple, dans la figure 3.3.1, la séquence *open, open, close* est considérée comme normale alors qu'elle n'apparaît pas dans la séquence initiale.

Nous ne sommes pas en mesure de dire à priori si ceci est un avantage ou un inconvénient. En effet, si la surévaluation peut mener à un taux de faux-négatifs plus élevé à cause de séquences anormales non détectées, elle peut aussi réduire le nombre de faux positifs en donnant un modèle plus général. Nous ne pouvons pour l'instant qu'effectuer des suppositions, que

nous allons mettre à l'épreuve lors de nos expériences.

3.3.2 Modèle de type arbre de n-grams

Ce type de modèle est une version plus complexe du modèle précédent, inspirée des travaux de Hubballi et coll. [17]. Au lieu d'être composé de paires, le modèle est composé des sous séquences qui apparaissent dans les traces, dites *n-grams*. La taille maximale des *n-grams* gardés en mémoire est la « profondeur »du modèle, de manière analogue à la distance jusqu'à laquelle les paires sont évaluées dans notre modèle précédent. Par ailleurs, les *n-grams* sont stockés sous la forme d'un arbre. La racine de l'arbre est un nœud fictif dont partent toutes les séquences. Les fils de la racine correspondent à tous les premiers appels système des séquences, et chaque nœud a pour fils les appels système suivants possibles. Si on met bout à bout les appels système obtenus en parcourant l'arbre de la racine à une feuille, on obtient un des *n-grams* enregistrés. La figure 3 montre un exemple de construction de cet arbre.

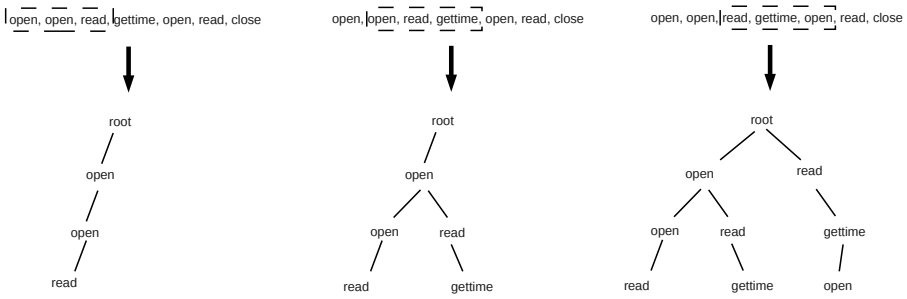


FIGURE 3 – Construction d'un arbre de *n-grams*

Avec cette construction, chaque nœud de l'arbre peut être vu comme le *n-gram* constitué par l'ensemble de ses parents et lui-même. De cette manière, si on décide de construire un arbre de profondeur n, on garde en mémoire non seulement toutes les séquences de taille n, mais également toutes les sous séquences de taille inférieure. Ceci nous permet d'adapter l'utilisation du modèle aux ressources disponibles en analysant des séquences de taille réduite si besoin, tout comme nous pouvions le faire avec notre modèle de paires.

De plus, au lieu de garder comme seule donnée la présence des séquences, ce modèle mémorise également leur nombre d'apparitions. De cette manière nous espérons former un modèle plus précis des applications. Puisque chaque nœud de l'arbre représente une séquence, il nous suffit de stocker les nombres d'apparitions dans chaque nœud de l'arbre. En pratique, l'arbre est construit de proche en proche en parcourant la trace avec une fenêtre glissante. À chaque fois qu'une séquence est trouvée, si celle-ci est déjà présente dans

l'arbre on se contente d'incrémenter son nombre d'apparitions. Sinon, on l'ajoute dans l'arbre avec un nombre d'apparition de 1. la figure 4 résume ces opérations.

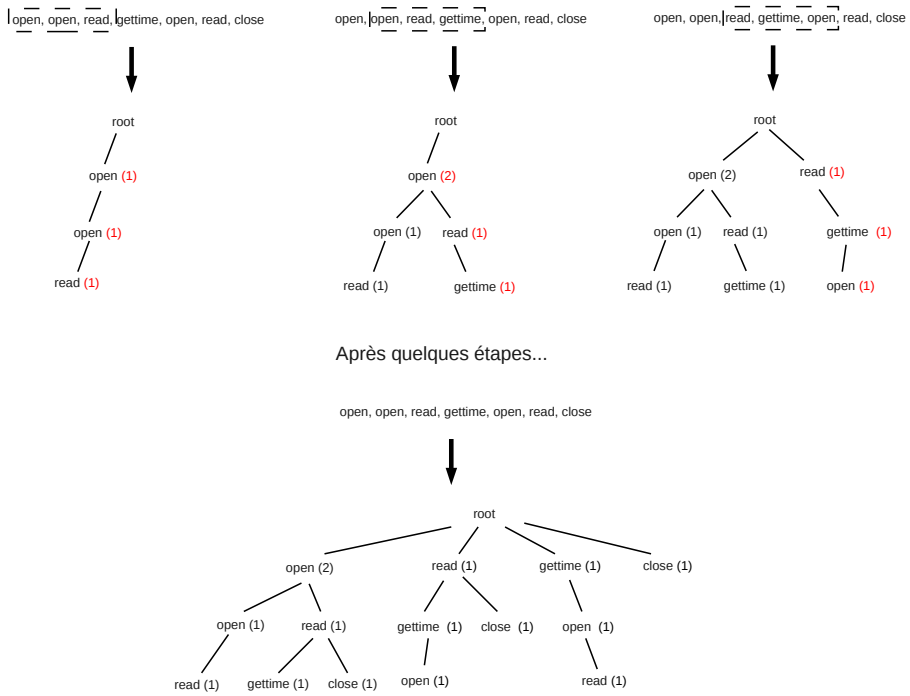


FIGURE 4 – Construction d'un arbre de *n-grams* avec nombres d'apparitions

Une fois la construction de l'arbre terminée, un taux d'anomalie est calculé pour chaque *n-gram* en fonction de son nombre d'apparition et du nombre total de *n-grams* ayant servi à construire le modèle grâce à la formule suivante :

$$anomaly = log(\frac{total}{apparitions})$$

Avec *total* le nombre total de *n-grams* ayant servi à construire le modèle et *apparitions* le nombre d'apparitions pour le *n-gram* dont le taux d'anomalie est calculé.

Lors de la phase d'analyse, on construit un arbre de manière similaire. Une fois cet arbre construit pour, celui-ci est parcouru en profondeur et comparé à l'arbre du modèle afin de déterminer les taux d'anomalies des différents *n-grams* qui le constituent. Un *n-gram* qui apparaît dans les deux arbres se voit attribuer le taux d'anomalie calculé précédemment dans le premier arbre, tandis qu'un *n-gram* ne s'y trouvant pas se voit attribuer un

taux d'anomalie maximal ayant pour valeur :

$$anomaly_{max} = log(total * 2)$$

Avec *total* le nombre total de *n-grams* ayant servi à construire le modèle. Cette valeur est arbitraire et correspond au taux d'anomalie qu'aurait une séquence apparaissant 2 fois moins qu'une séquence qui n'apparaît qu'une seule fois. Il nous semble normal de considérer les séquences ne figurant pas dans le modèle normal comme plus anormale que toute séquence qui y est présente. La valeur donnée ici est arbitraire et pourrait avoir besoin d'être changée ultérieurement. Cependant, une valeur trop grande rendrait les taux d'anomalies de traces apparaissant peu souvent négligeables devant le taux d'anomalie d'une trace qui n'apparaît pas. Seule l'information sur la présence ou l'absence des séquences dans le modèle initial serait visible lors de l'analyse, ce qui contredirait l'utilité de garder en mémoire le nombre d'apparition des séquences. Dans un tel cas, le comportement de ce modèle devrait s'approcher de celui de notre premier modèle.

Une fois tous les taux d'anomalies récupérés, on peut calculer l'anomalie moyenne de la trace en faisant la moyenne de tous les taux d'anomalies des *n-grams* qui le constituent. Cette anomalie moyenne peut ensuite être comparée à un seuil afin de déterminer si la trace est anormale.

Nous avons également implémenté une seconde version de l'arbre de *n-grams* dans laquelle nous avons apporté quelques modifications à l'algorithme. Cette amélioration part d'un constat simple : en pratique, comme on peut le voir sur la figure 5, ce sont les mêmes séquences qui apparaissent 90% du temps. En effet, cette figure ne montre que les 400 *n-grams* les plus fréquents. Au total, plusieurs milliers apparaissent.

Dans notre version améliorée, nous avons donc décidé de classer l'arbre par nombre d'apparition. Chaque nœud voit ses fils rangés par nombre d'apparition décroissant. Ainsi, lors du parcours de l'arbre en profondeur, on vérifie toujours en priorité les fils qui apparaissent le plus souvent, c'est à dire ceux qui ont le plus de chances d'être celui que l'on cherche.

En pratique, cette amélioration ne modifie que peu le code, puisque l'on construit les arbres de proche en proche en parcourant les traces. L'ensemble des fils de chaque nœud prend la forme d'une liste, il nous suffit donc de regarder à chaque fois que l'on incrémente le nombre d'apparition d'un nœud si on doit l'inverser avec son prédécesseur dans la liste où il se situe.

Ceci permet d'avoir en permanence des listes triées par fréquence décroissante au coût de quelques opérations supplémentaires à chaque insertion. Comparé à l'utilisation d'un algorithme de tri une fois la liste crée dont le coût est au minimum, pour une longueur de liste n, de $nlog(\sqrt{n})$, le gain en performance nous semble intéressant.

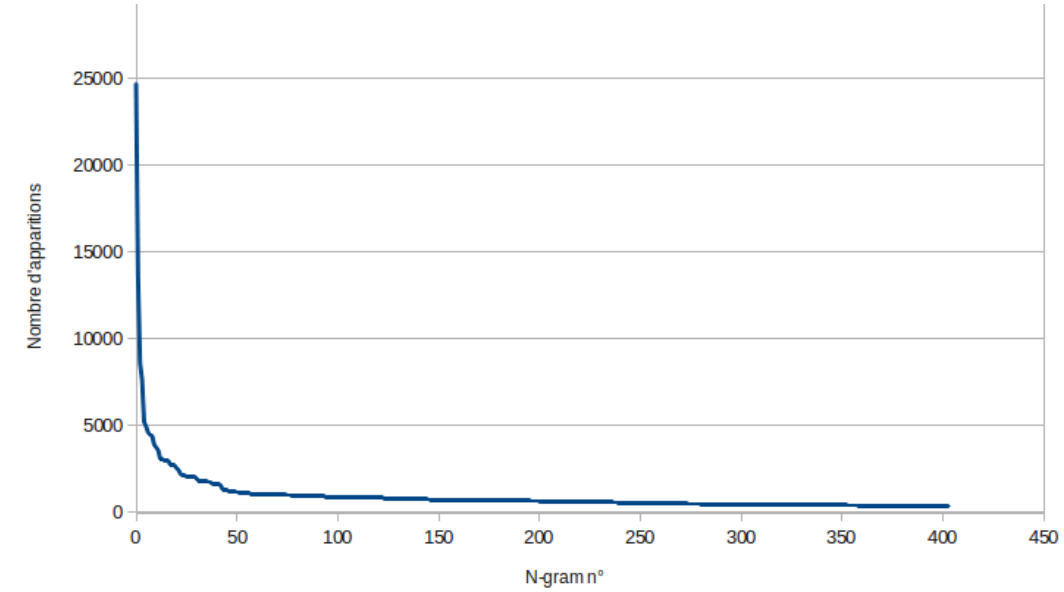


FIGURE 5 – Nombre d'apparition des 400 *n-grams* les plus fréquents pour une exécution du jeu Angrybirds dans un modèle allant jusqu'aux 6-grams.

4 Tests effectués

Nous avons effectué plusieurs jeux de tests, dans un premier temps pour valider notre approche, puis pour la mettre à l'épreuve face à de réels *malwares*. En effet, l'acquisition d'échantillons de logiciels malveillants et d'un téléphone dédié à effectuer des tests demande un certain investissement : il nous semblait nécessaire de vérifier que notre approche pouvait fonctionner avant de nous y lancer.

4.1 Conditions d'expérimentation

Nous avons utilisé deux environnements pour nos tests. Dans un premier temps, nous avons utilisé l'émulateur fourni par le SDK (*software development kit*) fourni par Android afin de tester et valider les composants du prototype les moins simple à installer, notamment *strace*. Cet émulateur s'est néanmoins avéré assez lent et peu propice à des tests réels d'application.

Nous avons donc par la suite utilisé un téléphone de type Samsung Galaxy S3 qui nous a permis d'effectuer des tests bien plus efficacement. Ce dernier tourne sous la version la version 4.1.1 d'Android et possède 2Gb de mémoire vive ainsi qu'un CPU dual core tournant à 1.5 GHz.

Lancer le *monitorage* dans un environnement réel nous a permis d'être sûr que les programmes surveillés se comportaient comme ils se comportent sur n'importe quel téléphone et ne risquaient pas de détecter un environnement

de type *sandbox*. En effet, certains *malwares* sont connus pour chercher à détecter de tels environnements et éviter d'exprimer leur comportement malicieux si ils y parviennent. Nous nous sommes donc servi de ce téléphone pour récupérer toutes les traces utilisées lors de nos tests de détection.

Pour des raisons pratiques, nous avons effectué la plupart des analyses de trace sur ordinateur. Une fois les traces extraites par un vrai téléphone, les algorithmes se comportent de manière totalement déterministe et donnent les mêmes résultats quelle que soit la plate-forme sur laquelle ils sont lancés. Nous avons donc opté pour un ordinateur pour pouvoir effectuer facilement le plus de tests possibles et pouvoir exploiter les résultats plus facilement.

Par ailleurs, nous avons effectué sur le téléphone nos tests de surconsommation. De tels tests nous semblaient évidemment bien plus pertinents effectués sur un vrai téléphone plutôt que sur un émulateur. Nous avons donc lancé nos algorithmes de détection en temps réel dessus afin de déterminer leur coût en ressources. Les résultats de ces tests sont présentés en partie 4.4

4.2 Validation de l'approche

4.2.1 Motivations

Notre premier jeu de tests a pour but de vérifier la possibilité de détecter un changement de comportement via les appels système. Par changement de comportement, nous entendons des appels à des fonctions qu'un programme n'a pas l'habitude d'appeler.

Pour cela, nous avons utilisé une application open-source. Dans un premier temps, nous avons construit un modèle de comportement normal en *monitorant* l'application telle quelle. Une fois ceci fait, nous avons modifié le code source de l'application afin d'y ajouter des appels à des fonctions que l'application n'utilisait pas au préalable. Lancer le scan de l'application en utilisant le modèle construit auparavant comme modèle normal devrait permettre d'observer des taux d'anomalie supérieurs dans les parties du code où les appels de fonction supplémentaires ont lieu.

Notre principal objectif dans cette partie est de connaître la sensibilité de notre approche, c'est à dire le nombre d'appels fonctions supplémentaire qu'il nous faut ajouter au code source avant de pouvoir détecter une anomalie de manière fiable.

4.2.2 Résultats

Dans un premier temps nous avons extrait des traces du programme non modifié afin de construire notre modèle normal. Afin d'être certain de traiter les deux types de modèle dans les mêmes conditions, nous avons construit un modèle de chaque type à partir de ces mêmes traces. Une fois les modèle normaux construits, nous avons injecté un seul appel de fonction supplémentaire à un endroit précis du code. Nous avons ensuite lancé le

monitorage de l'application et extrait les traces, que nous avons soumises à nos deux détecteurs. Au cours de nos essais, la longueur de trace qui nous est apparu offrir de meilleurs résultats est de 1200 appel système par trace. Ceci dit la différence de performances ne s'est pas montrée flagrante et un nombre d'appels système moins important peut tout à fait être utilisé.

Les figures 6 et 7 montrent respectivement les taux d'anomalie obtenus pour le modèle de type paires et le modèle de type arbre de *n-gram*, pour différentes profondeurs d'analyse (taille des *n-grams* et nombre de paires).

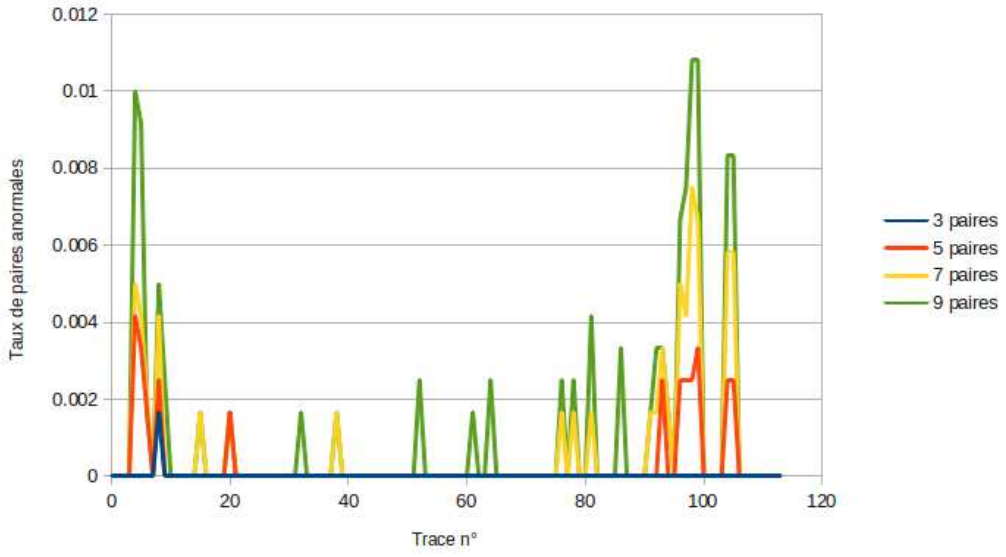


FIGURE 6 – Taux d'anomalie d'un appel de fonction supplémentaire, modèle de type paires

Lors de l'enregistrement de nos traces, nous avons effectué l'appel de fonction supplémentaire entre les traces 94 et 100. On observe un pic du taux d'anomalie à cet endroit dans tous nos modèles quelle que soit leur configuration, excepté pour le modèle de paire dans sa configuration minimale de 3 paires par appel système. La détection semble bien être possible.

En revanche, un second pic apparaît sur nos deux figures entre les traces 4 et 10 sans que nous n'y ayons fait appel à du code ajouté. Ce second pic a donc lieu lors d'un fonctionnement tout à fait normal de l'application. Dans la plupart de nos configurations, ce pic est même plus élevé que le pic qui correspond au réel comportement anormal : nous n'aurions aucun moyen de lever une alerte au bon moment sans lever également un faux positif. Seul le modèle de paires dans ses configurations à 7 ou 9 paires affiche un taux d'anomalie plus bas pour ce pic que pour le pic anormal. Ceci dit, l'écart entre les deux reste faible, et nous pensons donc qu'un seul appel de fonction

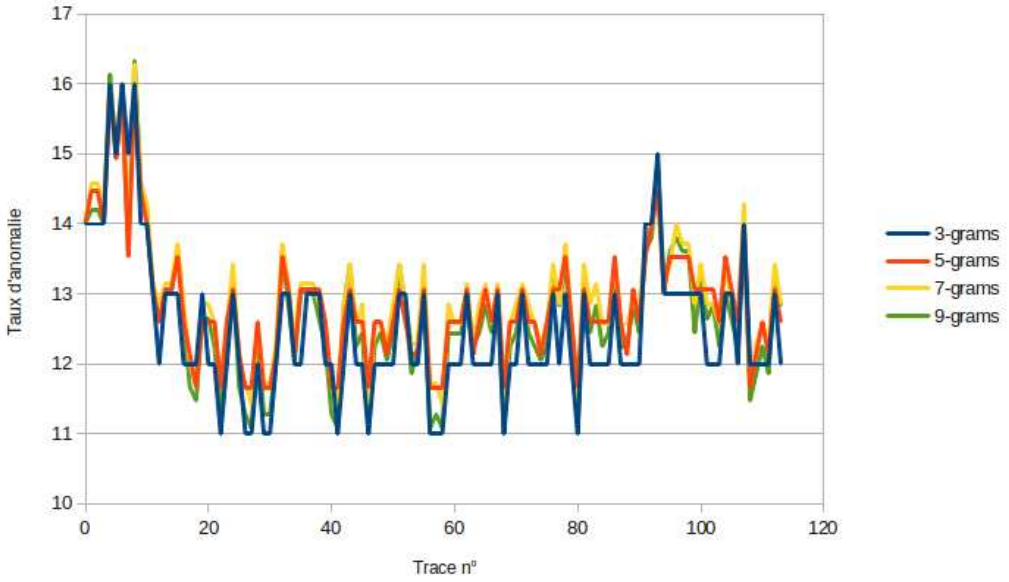


FIGURE 7 – Taux d'anomalie d'un appel de fonction supplémentaire, modèle de type arbre

n'est pas détectable de manière fiable.

Nous avons donc décidé d'augmenter le nombre d'actions anormales et d'ajouter deux appels de fonctions supplémentaire au même point du programme, ce qui porte notre « activité anormale »à 3 appels de fonction anormaux consécutifs. Nous avons effectué exactement les mêmes expériences dans ces conditions. Les figures 8 et 9 montrent respectivement les taux d'anomalie obtenus pour le modèle de type paires et le modèle de type arbre de *n-gram*, pour différentes profondeurs d'analyse.

Dans cette expérience, l'activité anormale se situe entre les traces 43 et 48. Le pic y est bien plus prononcé que dans la première expérience, quel que soit le modèle. Comme précédemment, on observe également un pic d'anomalie durant l'activité normale au sein des premières traces. Encore une fois, le modèle de type paires se montre plus performant et affiche un pic beaucoup plus marqué lors de l'anomalie réelle que le modèle de type arbre. Avec ce nombre d'appels fonctions supplémentaires, même sa configuration la plus faible de 3 paires peut détecter l'anomalie.

Le modèle de type arbre se montre moins performant. Si le pic correspondant à l'activité anormale est plus marqué que dans l'expérience précédente, il ne suffit toujours pas a différencier l'activité anormale de l'activité normale, puisqu'il ne dépasse pas le pic affiché pour les premières traces, et ce pour n'importe quelle longueur de *n-grams*.

Continuer ces expériences en augmentant encore le nombre d'appels de fonctions anormaux suit la même tendance pour le modèle de type paire : le

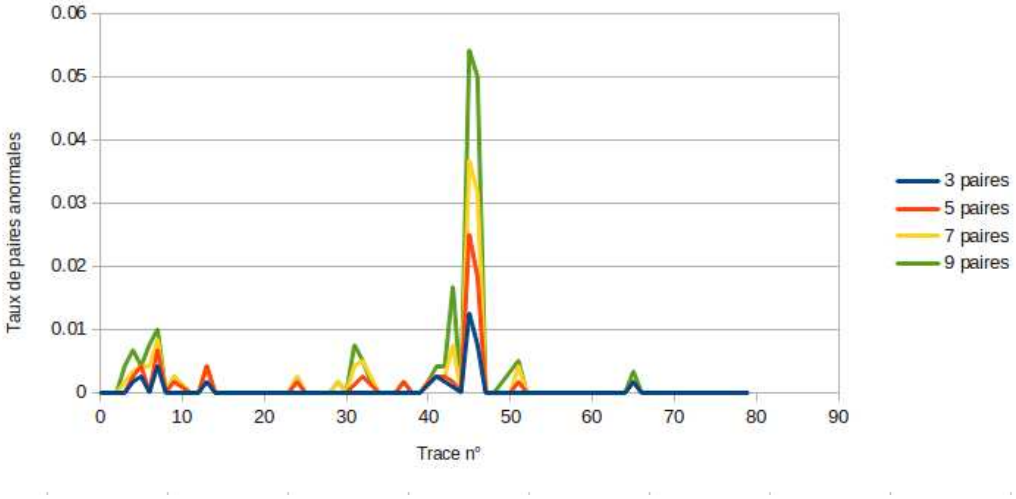


FIGURE 8 – Taux d'anomalie de trois appels de fonction supplémentaire, modèle de type paires

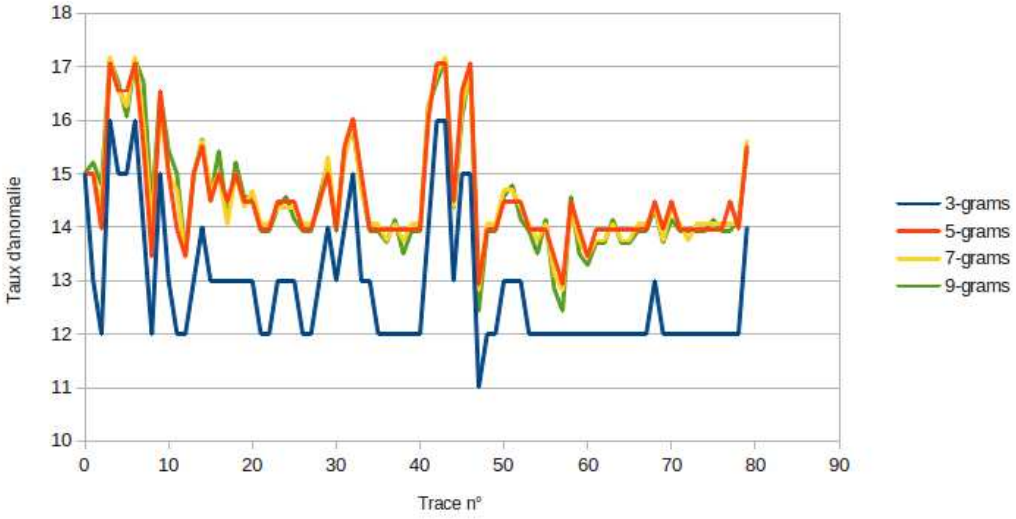


FIGURE 9 – Taux d'anomalie de trois appels de fonction supplémentaire, modèle de type arbre

pic d'anomalie correspondant aux fonctions ajoutées continue d'augmenter. Ce modèle semble donc assez performant pour détecter des anomalies de seulement quelques appels de fonctions, la performance augmentant avec la profondeur de l'analyse : il s'agit bien de ce qu'on l'on espérait obtenir.

En revanche, pour le modèle de type arbre, augmenter le nombre d'ap-

pels de fonction n'a pas permis d'obtenir un pic qui se démarque vraiment du pic des premières traces. Nous avons pu faire se démarquer ce pic en augmentant le score d'anomalie associé aux *n-grams* inconnus (voire partie 3.3.2). Avec cette modification, pour un grand nombre d'appels de fonction supplémentaires, de l'ordre de 40, nous pouvions différencier le pic dû à l'activité anormale. Cependant, de tels résultats ne nous semblent pas utiles. En effet, un tel nombre d'appels de fonction supplémentaires nous semble induire bien plus d'activité anormale qu'un *malware*, qui cherche à minimiser la longueur de sa charge utile. De plus, augmenter de la sorte le score associé aux *n-grams* inconnus tend à rendre le modèle d'arbre similaire au modèle de paires, dans le sens où on ne garde plus que l'information sur l'apparition ou non de séquences dans le modèle, sans tenir compte de leur fréquence. Dans ce cas, autant utiliser le modèle de type paires qui est plus performant et moins gourmand en ressources (voire partie 4.4).

Ceci étant dit, nous avons effectué nos tests sur de réels logiciels malveillants avec nos deux modèles. En effet, un logiciel malveillant a probablement un comportement plus complexe que le logiciel que nous avons modifié à la main. Le modèle de type arbre présentera peut être des avantages dans des conditions réelles qui ne sont pas observables sur notre exemple simple.

4.3 Tests sur des logiciels malveillants

Nous avons effectué des tests sur une version LeNa du *malware* DroidKungFu. Nous sommes conscients qu'il serait préférable d'effectuer des tests à plus grande échelle sur plusieurs logiciels malveillants et bénins, mais nous manquons de temps pour une telle étude. En effet, pour chaque logiciel que nous testons, ils nous faut construire un modèle sur une version normale d'un logiciel, puis confronter ce modèle à d'autres versions légitimes (programme bénins) et non légitimes (programmes contaminés). Chaque application testée demande donc un travail fastidieux :

- Trouver une application contaminée par un *malware* effectuant une attaque par mise à jour.
- Construire un modèle normal de cette application avant la mise à jour.
- Déclencher la fausse mise à jour puis analyser le comportement de l'application.
- Trouver une mise à jour saine de cette même application à analyser.

Nous avons effectué ce travail avec une version contaminée de l'application angrybirds. Les résultats obtenus sont visibles sur la figure 10 pour le modèle de type paire et sur la figure 11 pour le modèle de type arbre.

Les résultats pour le modèle de type arbre présentent une série de mesures supplémentaires : des traces du logiciel d'origine (sans mise à jour, ni bénigne ni malicieuse) sont confrontées au modèle normal. Une telle expérience avec le modèle de paire n'a donné que des scores nuls aussi nous n'avons pas jugé nécessaire d'en montrer le tracé.

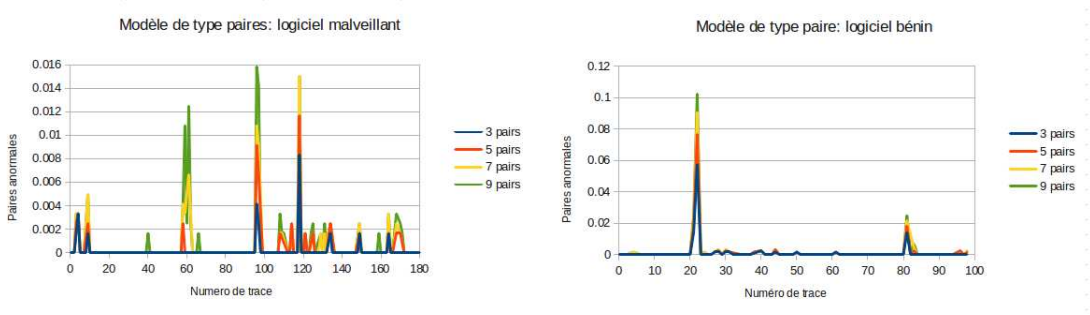


FIGURE 10 – Valeurs d'anomalies lors de tests réels, modèle de type paires

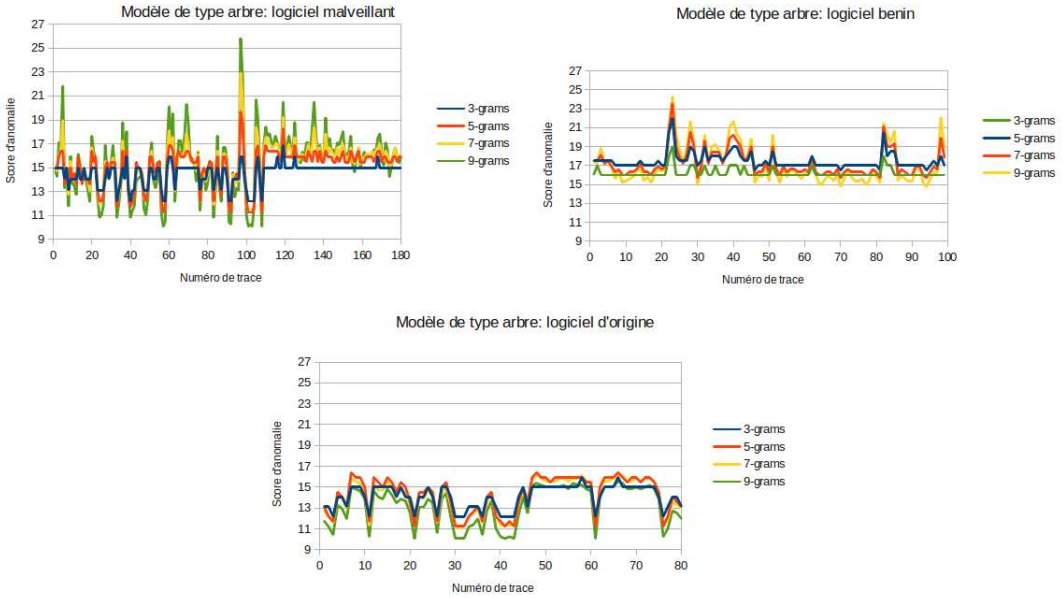


FIGURE 11 – Valeurs d'anomalies lors de tests réels, modèle de type arbre

Pour que les figures soient lisibles, nous n'avons pas mis les deux courbes des résultats du modèle de type paire à la même échelle : on peut voir qu'en réalité le pic visible dans les résultats obtenus pour le logiciel bénin est beaucoup plus important que le pic obtenu pour le logiciel malveillant. Les courbes des résultats du modèle de type arbre sont, en revanche, toutes les trois à la même échelle.

Les deux modèles affichent des pics d'anomalie quelle que soit la mise à jour effectuée. La mise à jour bénigne apporte de nouvelles fonctionnalité et il est donc normal d'observer un changement de comportement lors de celle-ci. En somme, comme nous le supposions dans la partie 2.4, nous arrivons bien à détecter des changements de comportement induits par des mises à

jour et ce avec nos deux modèles. Si des tests sont nécessaires avec d'autres programmes pour pouvoir tirer des conclusions générales, il nous semble donc possible de détecter par cette méthode des mises à jour effectuées à l'insu de l'utilisateur.

Le nombre de paire utilisées pour le modèle de type paires semble avoir une certaine influence sur l'allure des courbes. La courbe tracée à l'aide de trois paires pour un programme malicieux sur la figure 10 est plus plate que les courbes tracées avec un nombre de paires supérieures. Au delà, les pics d'anomalies sont très semblables entre les courbes tracées à l'aide de 5 paires ou plus. Nous pensons donc que nous limiter à 5 ou 6 paires permet d'obtenir sensiblement les mêmes résultats que l'utilisation d'un nombre plus élevé.

La longueur maximale des *n-grams* utilisés semble avoir une plus grande importance. En effet, en se limitant aux *3-grams*, la courbe tracée pour un programme malicieux sur la figure 11 ne permet même pas de voir la différence avec la courbe tracée pour le programme qui a servi à construire le modèle : pour les deux cas le score d'anomalie ne présente pas de pic marqué au dessus de 15. La différence commence à être observable à partir d'une longueur maximale de 5.

Il est impossible pour l'instant de faire la différence entre une mise à jour normale et une mise à jour malicieuse. En effet, le modèle de paire montre des taux d'anomalies bien plus importants pour la mise à jour normale. Le modèle de type arbre, quand à lui, montre un pic d'anomalie légèrement plus élevé pour la mise à jour malveillante. Cependant la différence avec les résultats obtenus pour la mise à jour normale ne nous semble pas assez marquée pour pouvoir effectuer un tri systématique. De plus, rien ne permet de dire que de tels résultats seront observés systématiquement.

Par conséquent, les deux modèles semblent dans un cas réel se valoir au niveau des performances. Dans la partie suivante, nous allons comparer leur coût en ressources.

4.4 Tests de surconsommation

Nous avons effectué des tests de surconsommation en utilisant une fois encore l'application angrybird, très répandue parmi les utilisateurs de smarpthones. Ces tests ont pour but de déterminer la surconsommation engendrée par nos deux algorithmes en fonction du nombre de paires ou de la taille des *n-grams* utilisés.

Nous avons effectué deux jeux de tests. Dans le premier, nous avons lancé le scan de l'application dans des conditions normales d'utilisation, c'est à dire en l'utilisant manuellement. Nous avons récupéré l'utilisation moyenne du processeur engendrée par notre prototype pour chaque nombre de paires et longueur de *n-grams*. Les résultats de ce premier jeu de tests sont montrés sur la figure 12.

Ces résultats montrent que les modifications que nous avons apporté à

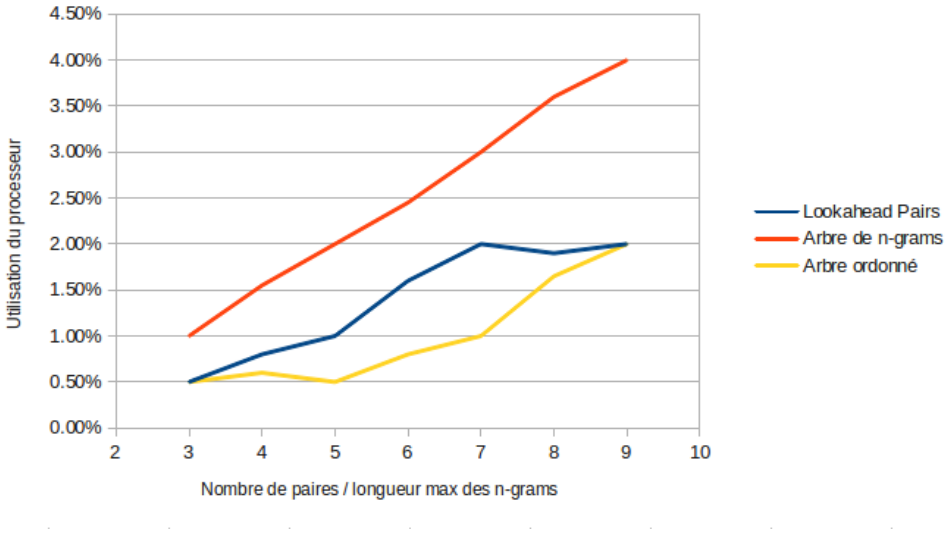


FIGURE 12 – Consommation CPU pour une utilisation normale

l'algorithme de type arbre semblent améliorer sensiblement ses performances. Pour les deux modèles on peut facilement rester sous la barre des 2% d'utilisation du CPU. On peut même descendre en dessous de 1% d'utilisation du CPU en diminuant suffisamment la profondeur de l'analyse, ce qui revient à diminuer de moitié la surconsommation. À l'utilisation, aucun ralentissement n'est observable, même dans le pire cas de ce graphe. Même lors de l'utilisation d'applications gourmandes en CPU, une surconsommation de 0.5% ne nous semble pas si problématique. Les surconsommations plus élevées engendrées par les analyses plus poussées ne devraient pas poser de problème à un téléphone en veille branché sur le secteur.

Ces mesures, bien que donnant une idée des consommations respectives des algorithmes, ne sont pas bien précises. En effet, à de faibles valeurs de consommation il nous est difficile d'obtenir des valeurs stables. La principale chose à retenir de cette première expérience est l'ordre de grandeur de la consommation CPU des algorithmes, de l'ordre de 1%, plus qu'une comparaison de leurs performances respectives.

Nous avons effectué un deuxième jeu de tests pour compléter ce dernier point. Afin de comparer les performances de nos algorithmes de manière plus précises, nous avons utilisé l'outil *monkey* fourni par le SDK d'Android pour générer automatiquement une forte sollicitation du programme. Ceci a pour effet de lui faire générer rapidement un très grand nombre d'appels système. Les algorithmes consomment dès lors beaucoup plus de CPU que lors d'une utilisation normale, ce qui permet de plus facilement comparer leurs performances respectives en diminuant les imprécisions de notre outil

de mesure de la consommation CPU. Les résultats de cette expérience sont représentés sur la figure 13.

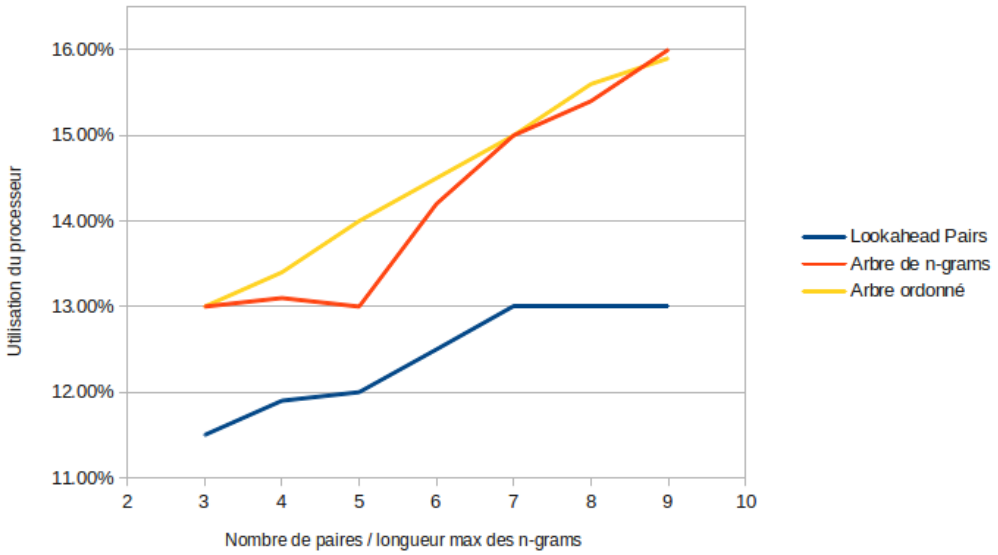


FIGURE 13 – Consommation CPU pour une forte sollicitation

Si la consommation engendrée semble forte, il faut garder en mémoire que cette expérience n'est pas faite dans des conditions d'utilisation normale et a pour but de comparer les performances respectives des algorithmes et non de donner un taux de surconsommation.

Les améliorations que nous avons apportées à l'algorithme de type arbre se montrent totalement inefficaces dans cette expérience. Ceci est selon nous dû à la nature de l'outil *monkey*. En effet, celui-ci génère des actions de l'utilisateur de manière totalement aléatoire, tandis que nos améliorations du modèle de type arbre se basent sur une connaissance des séquences les plus probables. Ils nous semble fort possible que l'outil *monkey* aie donc autant de chances de générer des actions peu probables dans le cadre d'une utilisation normale que les actions que l'utilisateur lui même effectue le plus. Par conséquent, l'idée de chercher en priorité les séquences les plus probables dans le modèle perd toute son efficacité. Les performances mesurées ici ne nous semblent donc pas représentatives des performances réelles de notre algorithme, et nous les considérerons comme un pire cas jamais atteint dans une utilisation normale.

Le modèle de type paire nous paraît donc comme plus fiable, dans le sens où il assure une faible consommation de ressources, tandis que le modèle de type arbre consomme moins de ressources que celui-ci dans la grande majorité des cas, mais peut être amené à en consommer plus dans le cas

d'une utilisation atypique.

4.5 Taille des modèles

Un autre élément important pour comparer les ressources utilisées par les modèles est la taille de ceux-ci. Comme nous l'avons évoqué, les applications Android se limitent en général à quelques Mo pour l'ensemble de leurs données, et il nous faut donc être capable de rester dans le même ordre de grandeur.

Le premier indice qui nous permet de comparer les deux modèles est le nombre d'éléments qu'ils contiennent. La figure 14 indique les nombres de *n-grams* et de paires contenus dans deux modèles construits à partir du même jeu de traces produite à partir de l'application angrybird.

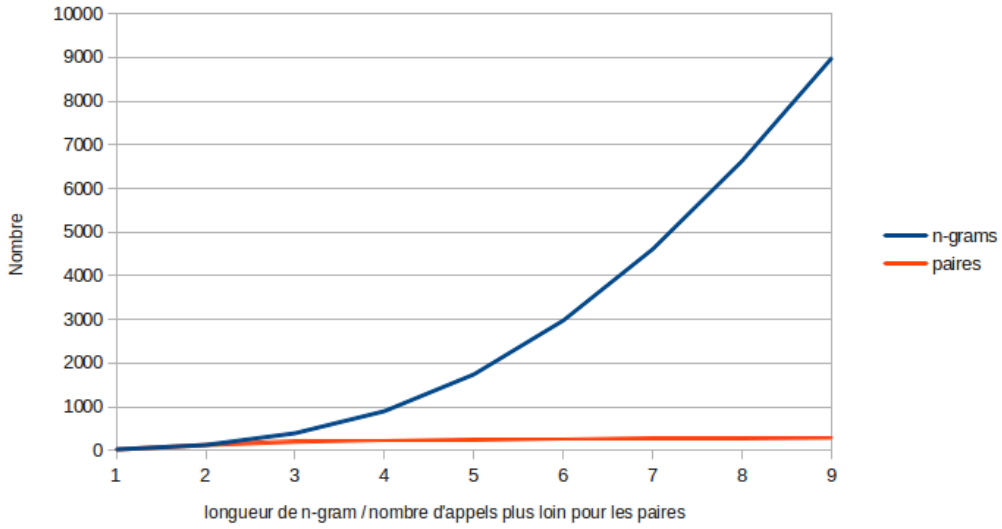


FIGURE 14 – Nombre d'éléments de différentes tailles pour deux types de modèles

Le modèle de type paires apparaît comme bien plus économe que le modèle de type arbre. Le nombre de paires possibles se stabilise presque au delà de deux paires par appel système. Le nombre de *n-grams*, en revanche explose avec la longueur des *n-grams*. La perte d'information engendrée par le modélisation par paires permet donc d'obtenir un modèle beaucoup plus concis.

La figure 15 montre l'espace disque occupé par les modèles de différentes profondeurs qui peuvent être construits à partir de ce même jeu de trace. La taille indiquée et l'espace disque occupé par le modèle sérialisé puis compressé à l'aide de la commande *tar* de linux. La tendance est sans surprise la

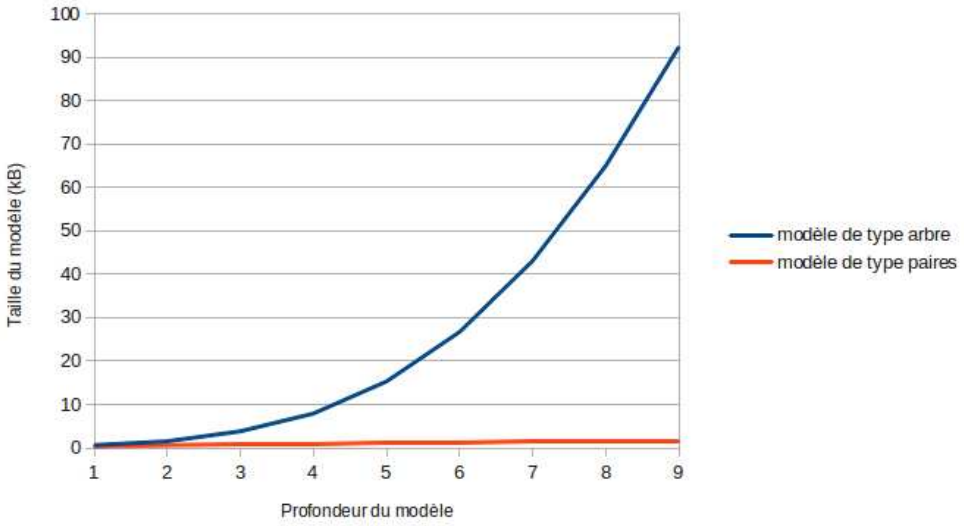


FIGURE 15 – Tailles de modèles de différentes profondeur

même que dans la figure précédente. La taille des modèle est proportionnelle aux nombre total d'éléments qu'ils contiennent, c'est à dire à l'intégrale des courbes de la figure 14. La taille d'un modèle de type paire est donc proportionnelle à la profondeur de modèle, tandis que la taille d'un modèle de type arbre semble augmenter de manière polynomiale.

Les ordre de grandeurs sont satisfaisants pour le modèle de type paires. En effet, le modèle le plus poussé que nous ayons testé, qui va jusqu'à 9 paires, occupe 1.6 kB. Ceci nous semble suffisamment bas pour permettre de stocker sans problème un modèle par application sur un téléphone, c'est à dire quelques dizaines tout au plus. Pour le modèle de type arbre, en revanche, la taille des modèles finit par devenir difficile à tolérer. En effet, si l'on stocke plusieurs dizaines de modèles de type arbre et de profondeur 9, on atteint rapidement plusieurs Mo de données. Cependant, si on se limite à un modèle contenant des séquences allant jusqu'à 5 ou 6 appels système, la taille de ceux-ci reste acceptable.

La compression est une étape cruciale. En effet, les modèles de type arbre non compressés ont vite fait d'occuper plusieurs Mo de stockage. La compression permet de réduire l'espace occupé jusqu'à un facteur 20 pour les modèles les plus volumineux : un modèle de type *n-grams* de profondeur 9 non compressé pèse 2.1 Mb, contre 92.4 kb lorsqu'il l'est.

5 Discussion

Nos deux modèles ont chacun leurs avantages et leurs inconvénients. Si leurs performances sont comparables en termes de détection de changement de comportement, ils présentent des caractéristiques différentes en ce qui concerne le coût en ressources.

Les modèles de type paires présentent l'avantage d'occuper très peu d'espace disque, même lorsqu'un grand nombre de paire par appel système est utilisé. Bien que la taille des modèles de type arbre reste acceptable grâce à la compression des données, ceci constitue un avantage certain en faveur des paires.

Par ailleurs, la consommation en CPU semble donner de meilleurs résultats pour le modèle de type arbre. Bien que la nature de nos améliorations ne garantissent pas des performances optimales pour n'importe quelle utilisation, la philosophie de notre approche consiste à détecter des changements de comportement. Par conséquent, le fait que le programme garde ses habitudes d'utilisation est une hypothèse qui est la base de notre projet et il nous semble donc logique d'effectuer nos choix en la gardant en tête. Bien que le modèle de type paire offre de meilleures performances dans les cas défavorables, nous aurions tendance à considérer ceux-ci comme suffisamment rares pour que cela ne défavorise pas réellement le modèle de type arbre.

La suite de notre travail consisterait à développer des moyens de différencier les mises à jour bénignes des mises à jour malicieuses afin d'être à même de mettre à jour le modèle si nécessaire. D'un point de vue général, notre méthode n'a en soi pas pour but de faire une telle distinction : seul le changement est détecté. Nous pensons pouvoir utiliser à cet effet le réseau social fourni par un téléphone. En effet, des utilisateurs proches ont des chances d'utiliser des mêmes applications. On peut supposer qu'un utilisateur a plus de chances de tomber sur une version bénigne d'une application que sur une version malveillante. Par conséquent, en « mettant en commun »les modèles crées par un groupe d'utilisateur, on peut imaginer que lors d'une mise à jour les changements normaux ont lieu chez la majorité des utilisateurs. À l'opposé, une mise à jour malicieuse ne devrait apporter ses changements que chez une minorité d'entre eux. Une telle approche demanderait cependant d'étudier les différences entre les modèles d'une même application pour deux téléphones différents.

Des améliorations peuvent également être apportées au niveau des ressources consommées. Nous avons construit des modèles que l'on peut exploiter plus ou moins en profondeur selon les besoins. L'étape suivante serait de mettre à profit cette adaptabilité en tenant compte de l'utilisation du processeur : augmenter la profondeur de l'analyse quand le processeur est au repos et la réduire lorsque celui-ci est fortement sollicité. Lors de nos tests nous n'avons fait des essais que sur une seule application à la fois : il serait pertinent de regarder l'influence de notre système lorsqu'il est utilisé sur

plusieurs applications simultanément. Dans de telles conditions, l'adaptation deviendrait probablement plus importante.

6 Conclusion

Ce document présente une approche de détection d'anomalie nouvelle dans le cadre des smartphones. Nous avons mis en avant la possibilité de modéliser le comportement normal d'un programme à partir de ses interactions avec le système d'exploitation et de détecter les sorties de ce comportement. Si la distinction entre mise à jour bénigne et malicieuse n'est pas faite, les mises à jour nous semblent bel et bien détectables de manière fiable via cette approche, ce qui peut permettre de détecter des actions faites à l'insu de l'utilisateur.

Ce travail constitue une approche préliminaire, puisqu'il ne permet pas encore une utilisation de notre prototype pour des détections réelles. Cependant, nos tests de performances montrent que le coût en ressources d'un tel système reste acceptable, tout comme la taille des modèles construits, ce qui est d'une importance capitale dans un environnement tel que celui d'Android. Par conséquent, avec les évolutions nécessaires, ce travail pourrait se montrer complémentaire des antivirus classiques.

Références

[1] Lookout. Mobile threat report, June 2012.

[2] Carlos A Castillo. Android malware past, present, and future. *White Paper of McAfee Mobile Security Working Group*, 2011.

[3] Google. Supported locations for distributing applications, 2013.

[4] Steven Millward. Chinese apps are bypassing google's play store, giving android apps straight to users, March 2012.

[5] Malicious mobile threats report. Technical report, Juniper Networks, 2011.

[6] Yajin Zhou and Xuxian Jiang. Dissecting android malware : Characterization and evolution. In *Security and Privacy (SP)*, pages 95–109. IEEE, 2012.

[7] Xuxian Jiang. Security alert : New stealthy android spyware – plankton – found in official android market, June 2011.

[8] Xuxian Jiang. Security alert : Anserverbot, new sophisticated android bot found in alternative android markets, September 2011.

[9] Marc Rogers. The bearer of badnews, April 2013.

[10] Dan Goodin. More »adnews « for android : New malicious apps found in google play, April 2013.

[11] Emil Ong. The continuing saga of fake app toll fraud, April 2012.

[12] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 3–14. ACM, 2011.

[13] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. A sense of self for unix processes. In *Security and Privacy*, pages 120–128, May 1996.

[14] Stephanie Forrest, Steven Hofmeyr, and Anil Somayaji. The evolution of system-call monitoring. In *Annual Computer Security Applications Conference (ACSAC)*, pages 418–430. IEEE, 2008.

[15] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls : Alternative data models. In *Symposium on Security and Privacy*, pages 133–145. IEEE, 1999.

[16] Christopher Kruegel, Darren Mutz, Fredrik Valeur, and Giovanni Vigna. On the detection of anomalous system call arguments. In *Computer Security–ESORICS*, pages 326–343. Springer, 2003.

[17] Neminath Hubballi, Santosh Biswas, and Sukumar Nandi. Sequencegram : n-gram modeling of system calls for program based anomaly detection. In *Communication Systems and Networks (COMSNETS)*, pages 1–10. IEEE, Jan 2011.

[18] Debin Gao, Michael K Reiter, and Dawn Song. Behavioral distance measurement using hidden markov models. In *Recent Advances in Intrusion Detection*, pages 19–40. Springer, 2006.

[19] Anup K Ghosh and Aaron Schwartzbard. A study in using neural networks for anomaly and misuse detection. In *USENIX Security Symposium*, volume 8, pages 141–152. Washington DC : ASME Press, 1999.

[20] David Endler. Intrusion detection. applying machine learning to solaris audit data. In *Computer Security Applications Conference*, pages 268–279. IEEE, 1998.

[21] Christopher Kruegel, Darren Mutz, William Robertson, and Fredrik Valeur. Bayesian event classification for intrusion detection. In *Computer Security Applications Conference*, pages 14–23. IEEE, 2003.

[22] Asaf Shabtai, Robert Moskovitch, Yuval Elovici, and Chanan Glezer. Detection of malicious code by applying machine learning classifiers on static features : A state-of-the-art survey. *Information Security Technical Report*, 14(1) :16–29, 2009.

[23] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 108–125. Springer, 2008.

[24] Grant A Jacoby and Nathaniel J Davis. Battery-based intrusion detection. In *Global Telecommunications Conference*, volume 4, pages 2250–2255. IEEE, 2004.

[25] Francesco Di Cerbo, Andrea Girardello, Florian Michahelles, and Svetlana Voronkova. Detection of malicious applications on android os. In *Computational Forensics*, pages 138–149. Springer, 2011.

[26] Aubrey-Derrick Schmidt, Hans-Gunther Schmidt, Jan Clausen, Kamer A Yuksel, Osman Kiraz, Ahmet Camtepe, and Sahin Albayrak. Enhancing security of linux-based android devices. In *International Linux Kongress*, 2008.

[27] Thomas Blasing, Leonid Batyuk, A-D Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection. In *Malicious and Unwanted Software (MALWARE)*, pages 55–62. IEEE, 2010.

[28] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taintdroid : An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, volume 10, pages 255–270, 2010.

[29] Lookout. Technical analysis droiddream malware, March 2011.