How to play Hooker : Une solution d'analyse automatisée de markets Android

Georges Bossert et Dimitri Kirchner
georges.bossert@amossys.fr
dimitri.kirchner@amossys.fr

AMOSSYS

Résumé Android est aujourd'hui le système d'exploitation sur mobiles le plus répandu du marché, ce qui en a fait une cible de choix pour les applications malveillantes. Afin de parer celles-ci, l'OS de Google se base principalement sur un système de permissions : un utilisateur doit valider les permissions demandées par une application avant son installation. Ce modèle présente cependant une faiblesse particulièrement gênante : le besoin de l'utilisateur. En effet, il est difficile de refuser l'installation d'une application à partir du moment où l'utilisateur en a besoin, même si celle-ci demande des droits injustifiés.

Il existe donc *a priori* le besoin d'une base de connaissances publique permettant de comprendre si l'application que l'on souhaite installer pose un problème de sécurité. Dans cet article, nous proposons une solution d'analyse automatisée de *markets* Android : **Hooker**. Celle-ci permet d'identifier les événements sensibles intervenants sur un système Android et de les centraliser au sein d'une base de données dédiée. Le traitement de ces informations et la présentation des résultats effectuée de manière intelligente permet, au final, de réaliser des analyses tant microscopiques que macroscopiques. La première se concentre sur le fonctionnement d'une application en particulier ; la seconde sur la recherche de comportements similaires à un ensemble d'applications et donc potentiellement à un *market* entier.

1 Introduction

Le modèle de sécurité proposé par le système Android repose sur un principe simple : une application ne peut accéder qu'aux seules ressources expressément autorisées par l'utilisateur. Avec cette stratégie, le modèle de sécurité Android délègue à l'utilisateur la responsabilité d'accepter ou non une application potentiellement malveillante. Pour permettre à l'utilisateur de prendre une décision, il lui est nécessaire de disposer d'un certain nombre d'informations relatives à l'impact de l'application sur son téléphone. Android exploite pour cela la notion de permissions. Il existe

des permissions pour chaque type d'accès aux ressources du téléphone, par exemple les permissions `RECORD_AUDIO` et `SEND_SMS` qui permettent, respectivement, l'accès au micro du téléphone et l'envoi de SMS. Ainsi, lors de l'installation d'une application, l'utilisateur est informé des permissions demandées par l'application. C'est sur la base de cette information que l'utilisateur prend ensuite la décision d'accepter ou de refuser l'application.

Cependant, et malgré la pertinence et la précision des permissions proposées à l'utilisateur, celles-ci ne permettent pas de répondre à tous les questions qu'il est en droit de se poser avant de prendre sa décision. Parmi ces questions, deux sont récurrentes à savoir : 1) que fait l'application avec ces ressources et 2) l'application va-t-elle abuser de ses permissions et donc des ressources de l'utilisateur. Par exemple, dès lors qu'une application demande des droits relatifs aux communications réseau (autrement dit pour accéder à internet dans la majorité des cas), qu'est ce qui prouve que l'application n'exfiltre pas des données vers un serveur distant ?

Pour tenter de répondre à ces deux questions, plusieurs solutions existent. Celles-ci peuvent être regroupées selon le type d'analyse utilisée : 1) l'analyse statique qui repose sur l'étude du code de l'application et à l'inverse 2) l'analyse dynamique qui repose sur l'étude de l'application pendant son exécution. Ces deux types d'analyses peuvent également être combinées. Chacune de ces analyses a alors un but commun : obtenir des informations sur le comportement du programme. Dans nos travaux, nous proposons l'emploi de ces deux techniques d'analyses de manière à fournir une vision la plus complète et précise possible de l'impact d'une application sur le téléphone d'un utilisateur. Dans la suite de cet article, nous dénommons **microanalyse**, l'emploi de notre solution d'analyse pour l'étude en profondeur du fonctionnement d'une application.

Dans un second temps, nous proposons la généralisation des techniques employées pendant la microanalyse pour l'étude d'un ensemble d'applications. Le fait de reproduire une technique d'analyse à plus grande échelle permet de rechercher des motifs de comportements spécifiques au sein, non pas d'une application mais cette fois au sein d'un ensemble d'applications voire d'un *market* complet. Nous utilisons le terme de **macroanalyse** dans ce cas. À la connaissance des auteurs, ce type d'analyse à grande échelle des applications Android n'existe pas encore aujourd'hui. Toutefois, il est probable que la solution d'analyse automatique des applications soumises au *market play store* de Google exploite des techniques similaires. En effet, depuis février 2012, Google utilise cette sandbox, appelée *Bouncer*, afin de détecter la présence ou non de certaines propriétés au sein des applications publiées sur son *market*. Cette analyse est obligatoire pour que

l'application soit disponible aux utilisateurs du *market*. Si les résultats de toutes les analyses sont centralisés au sein d'une base de données proposant des fonctionnalités de *data-mining*, alors cette solution est proche de celle que nous proposons. Il est à noter qu'aucune information n'est aujourd'hui vraiment disponible sur le fonctionnement de *Bouncer* et seuls quelques articles nous laissent à penser que c'est le cas [1].

Dans cet article, nous proposons une solution d'analyse automatisée de *markets* Android baptisée **Hooker**. Cette solution est librement téléchargeable sous une licence open-source [2]. C'est notamment avec le retour de la communauté que la solution doit s'améliorer. Le but de Hooker n'est pas uniquement l'identification de certains comportements malveillants, mais surtout de centraliser et d'agréger les résultats de milliers d'analyses avec suffisamment de précision pour permettre l'emploi de techniques de *data-mining*. Nous avons donc organisé l'article de la manière suivante : nous détaillons dans la section 2, l'intérêt de notre solution par rapport aux travaux existants. Nous présentons ensuite dans la section 3, comment est réalisée l'analyse d'une application en particulier. Quelques exemples de cas d'application sont fournis afin d'illustrer le fonctionnement de Hooker. La section 4 détaille la méthodologie utilisée pour réaliser l'analyse macroscopique d'un ensemble d'applications issues d'un échantillon de près de 1000 applications téléchargées. Finalement, nous concluons cet article dans la section 5 après avoir donné quelques pistes d'améliorations.

2 État de l'art

La solution Hooker se reposant en grande partie sur de l'analyse dynamique, nous détaillons dans cette section les différentes solutions existantes, et l'intérêt de Hooker par rapport à celles-ci.

2.1 Analyse dynamique d'une application Android

L'analyse dynamique d'applications consiste à exécuter les applications à analyser dans un environnement contrôlé. Plusieurs solutions d'analyse dynamique d'applications Android ont été proposées par la communauté scientifique depuis la disponibilité du système Android en 2008.

En 2010, Enck et al. présentent la solution appelée TaintDroid [1]. Cette version modifiée du système Android permet de se mettre en coupure

1. Dissecting the Android bouncer : `https://jon.oberheide.org/blog/2012/06/21/dissecting-the-android-bouncer/`
2. Hooker : `https://github.com/AndroidHooker/hooker`

sur des API Java identifiées comme sensibles. Initialement centré sur la fuite de données personnelles, TaintDroid a ensuite été amélioré afin de proposer d'autres solutions telles que DroidBox [3]. Ce dernier repose sur l'architecture de TaintDroid et améliore l'utilisation de celui-ci en ajoutant notamment un mécanisme d'automatisation des analyses. Des événements comme les opérations de lecture et écriture sur les fichiers, les appels à des primitives cryptographiques, les ouvertures de connexions réseau, l'envoi de SMS, etc. sont ainsi détectées par DroidBox et observables au travers d'un rapport produit à la fin de l'analyse. D'autres solutions telles que DroidScope [4] reposent aussi sur une version modifiée d'Android. La principale limitation de ce type d'analyse est la nécessité de reconstruire une version d'Android adaptée à chaque nouvelle version du système.

Pour contourner cette limitation, des solutions comme APIMonitor ou Fino [2] existent aujourd'hui. APIMonitor est introduit à GSoC 2012 comme la version 2.0 de DroidBox, et repose sur la modification du bytecode de l'application afin d'y injecter le code souhaité. Cette solution permet de devenir indépendant du système d'exploitation sous-jacent, ce qui fonctionne particulièrement bien dès lors que l'utilisateur dispose d'une maîtrise sur l'application qu'il souhaite analyser. Cependant, il existe plusieurs cas où l'utilisateur ne dispose pas de cette maîtrise, par exemple lorsque l'on dispose d'une application déployée par le constructeur ou le fournisseur d'accès. En particulier, dès lors que l'application dispose de privilèges système, la disponibilité de la clé privée de signature nécessaire au déploiement de l'application dans /system devient alors problématique.

Dans le cadre de ces travaux, la méthode d'analyse dynamique proposée par Hooker repose sur le *framework* Cydia Substate. Ce dernier permet d'effectuer de l'*API hooking* en s'injectant en mémoire de chacune des applications exécutées sur le système. Cette solution a comme principal avantage de ne reposer sur aucune modification en amont, que ce soit au niveau du système comme au niveau de l'application à analyser. Par ailleurs, les modifications apportées pour détourner le flux d'exécution ne s'effectuant qu'en mémoire, il est possible de déployer Hooker sur une architecture physique comme sur un périphérique virtuel.

En décembre 2013, alors que notre projet est en cours de développement, une solution appelée *Introspy-Android* [3] est publiée sur le service d'hébergement Github et présentée à l'occasion de la Ruxcon. Cette solution développée par la société iSECPartners [4] repose elle-aussi sur le

3. `https://github.com/iSECPartners/Introspy-Android`
4. Société iSECPartners : `https://www.isecpartners.com/`

framework Cydia Substate. Le détournement du flux d'exécution est utilisé par *Introspy-Android* pour vérifier certaines propriétés sensibles telles que :

- les données liées à de la cryptographie comme les messages chiffrés, les hachés cryptographiques avant hachage, etc. ;
- les accès au système de fichiers jugés comme sensibles ;
- les communications inter-processus et autres accès aux préférences partagées et aux bases de données SQLite ;
- etc.

La vérification de ses propriétés est réalisée en temps réel et les résultats produits sont ensuites remontés à l'utilisateur d'Intropsy. Par analyse en temps réel, nous entendons qu'Introspy arrête temporairement l'application le temps de l'analyse, et dispose ainsi d'une connaissance complète du contexte d'exécution. Le principal défaut d'une analyse en temps réel, est qu'il est nécessaire de relancer une expérience afin d'effectuer de nouvelles vérifications. Par exemple, lorsqu'une connexion SSL est détournée par *Introspy*, celui-ci va récupérer les paramètres de la connexion afin d'en vérifier la validité en temps réel. Ces propriétés seront vérifiées selon certains critères. Si ceux-ci viennent à changer, une nouvelle analyse devra être réalisée avec le risque qu'elle ne soit plus possible. Dans le cas de notre Hooker, nous ne nous concentrons que sur la création d'un événement contenant toutes les informations relatives à la fonction et à son contexte. Ce n'est que dans un second temps que nous analysons les informations capturées. Cette différence d'implémentation nous permet de limiter l'impact de Hooker sur l'exécution de l'application tout en permettant la réalisation d'analyses *a posteriori*. En outre, nos travaux se sont également attachés à automatiser l'analyse sur un grand nombre d'applications, ce qu'*Introspy* ne permet pas. Il n'est donc pas possible d'utiliser Introspy tel quel pour créer une base de connaissance sur un ensemble d'applications (ou encore pour rechercher au sein de plusieurs applications des comportements similaires).

Dans la suite de cet article, nous détaillons le fonctionnement du Hooker en présentant son utilisation pour la réalisation d'analyses miscroscopique puis macroscopique.

3 Analyse microscopique

L'analyse d'une application par Hooker se décompose en plusieurs étapes représentées sur la figure 1.

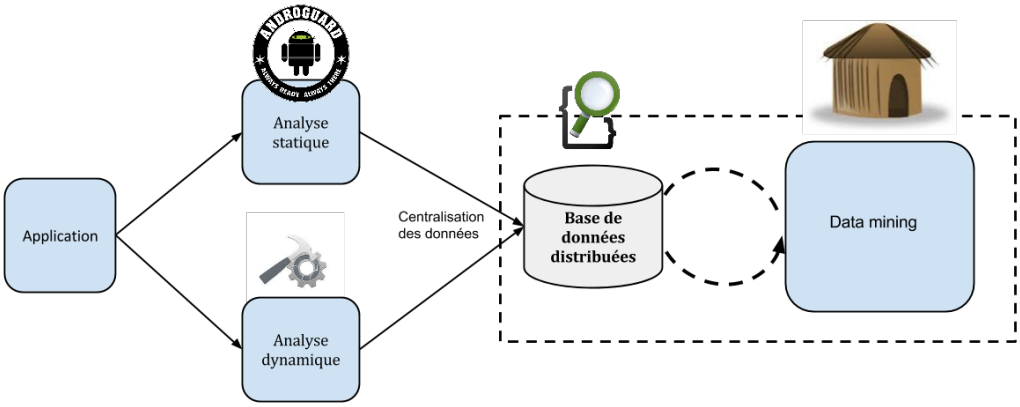

Figure 1. Principe de fonctionnement de l'analyse d'une application par Hooker.

La première étape consiste à réaliser une analyse statique préliminaire à l'aide du *framework* Androguard [5]. Cette analyse préliminaire à toute exécution sur le système permet de cibler des critères tels que les permissions demandées ou les composants utilisés par l'application. Les informations extraites de cette analyse statique permettent de disposer d'une base de connaissances simple et rapide à construire de l'application à analyser. Ces informations sont ensuite sauvegardées pour pouvoir être complétées par les données de l'analyse dynamique.

La seconde étape consiste à réaliser une analyse dynamique de l'application. Le *framework* Cydia Substrate [6] est ainsi utilisé afin de faire du détournement d'API (API Hooking en anglais) sur le système visé. En fonction de la stimulation de l'application, les événements relatifs à l'exécution de celle-ci sont alors créés et envoyés à un service en écoute sur le système. L'enregistrement des valeurs des paramètres et du retour des différentes fonctions permet alors d'acquérir un grand nombre d'informations comme les adresses IP de destination, la valeur de la clé de chiffrement utilisée, etc. Lors de cette étape, un indicateur d'indiscrétion est de plus associé à chaque type d'événement observé. Cet indicateur est présent afin de pouvoir différencier les applications les plus sensibles. Il repose sur l'hypothèse que l'accès en écriture à une ressource est souvent plus indiscret qu'un accès en lecture à celle-ci. En prenant l'exemple concret

5. Androguard : `http://code.google.com/p/androguard/`
6. Cydia Substrate : `http://www.cydiasubstrate.com`

d'un fichier, nous considérons donc dans Hooker, que l'écriture d'un fichier est davantage gênant que la lecture de celui-ci.

Enfin, la dernière étape consiste à centraliser l'ensemble de ces événements au sein d'une base de données distribuée. La combinaison des logiciels Elasticsearch [7] et Kibana [8] permet d'effectuer des recherches et de visualiser les résultats en temps réel sur les données sauvegardées.

3.1 Analyse statique

Une analyse statique de l'application est réalisée en amont de l'analyse dynamique. Pour effectuer celle-ci, le *framework* Python Androguard [9] est utilisé et permet de rassembler des informations dites *basiques* sur l'application. Le Hooker collecte donc toutes les informations relatives à la compréhension globale de l'application. Parmi celles-ci on retrouve la liste des activités, des `Providers`, des `Receivers`, des `Services`, les versions minimum et maximum du SDK compatibles avec l'application, la liste des permissions et la liste des bibliothèques natives. Toutes ces informations sont archivées dans la base de données. À ce titre, elles pourront être corrélées avec les résultats de l'analyse dynamique.

Cette étape d'analyse statique est aussi l'occasion de rassembler des informations qui seront ensuite utilisées pour l'exécution de l'application. L'activité principale de l'application (ou `Main activity`) et le nom du package sont ainsi identifiés afin que l'analyse dynamique puisse disposer de toutes les informations nécessaires au lancement de l'application.

Au final, l'analyse statique réalisée permet de rassembler les informations qui ne sont pas forcément remontées par l'analyse dynamique, et qui sont intéressantes pour la compréhension globale de l'application. Un exemple d'événement ainsi créé est donné sur le listing 1.


```
{
  "_index": "hooker_test"
  "_type": "static"
  "_id": "WzadmZt2Sha9vuioAYJUmA"
  "_version": 1
  "_score": 1
  "_source": {
    "Android Version Code": 10532
    "PackageName": "com.herocraft.game.montezuma2.lite"
    "Main Activity": "com.herocraft.game.montezuma2.lite.
        AndroidDemoStarter"
    "Libraries": [ ]
```


7. Elasticsearch : `www.elasticsearch.org`
8. Kibana : `www.elasticsearch.org/overview/kibana/`
9. Androguard : `http://www.androguard.re`

```
    "Filename": "Hooker/outputAPKs/0
        c839b890462836b06b4ec0fc143cd87a3443173.apk"
    "Activities": [
    {"Activity": "com.herocraft.game.montezuma2.lite.
        AndroidDemoStarter"}
    ]
    "Min SDK Version": 4
    "Providers": [ ]
    "Timestamp": 1397411752312
    "Receivers": [
      {"Receiver": "com.herocraft.game.montezuma2.lite.
          CommonReceiver"}
    ]
    "Max SDK Version": -1
    "Android Version Name": "1.5.32"
    "Services": [ ]
    "Permissions": [
      {"Permission": "archos.permission.FULLSCREEN.FULL"}
      {"Permission": "android.permission.INTERNET"}
      {"Permission": "android.permission.READ_CONTACTS"}
      {"Permission": "android.permission.VIBRATE"}
      {"Permission": "android.permission.SEND_SMS"}
      {"Permission": "android.permission.READ_PHONE_STATE"}
      {"Permission": "android.permission.ACCESS_NETWORK_STATE"}
    ]
  }
  "fields": {
      "_parent": "9b917a2b0cccf62d6353293e89ec6d54"
  }
}
```


Listing 1. Evénement relatif à l'analyse statique de l'application `com.herocraft.game.montezuma2.lite`.

3.2 Analyse dynamique

L'analyse dynamique d'une application repose sur le *framework* Cydia Substrate afin de s'injecter dynamiquement en mémoire de chacune des applications exécutées sur le système.

Prérequis L'utilisation de Substrate nécessite de préparer le système préalablement à une analyse. En effet, afin de disposer des fonctionnalités du *framework* de détournement, il faut d'abord disposer des droits `root` sur le système. Ces droits seront alors demandés par l'application Substrate [10] à installer sur le périphérique afin de disposer des fonctionnalités du *framework*.

10. Cydia Substrate on Google Play : `https://play.google.com/store/apps/details?id=com.saurik.substrate`

Comparés aux solutions existantes, ces prérequis sont assez faibles. En particulier, l'utilisation d'un émulateur Android ne pose pas de problème : l'utilisateur dispose nativement de droits `root` sur celui-ci. Par ailleurs, et dans le cas où l'on dispose d'un périphérique physique, l'utilisation d'exploits publics permet d'obtenir les droits nécessaires sur celui-ci.

Substrate Le déploiement de Substrate sur le système permet de s'insérer au sein du processus `Zygote`. Ce dernier étant le processus dont tous les autres sont dérivés, cette opération permet donc de s'injecter dans toutes les applications exécutées sur le système.

Le développement d'une extension Substrate (terme pour désigner le code qui est injecté dans `Zygote`) est, de prime abord, assez intuitif. L'extrait de code suivant permet d'illustrer une fonction simple interceptant le constructeur de la classe `java.net.InetSocketAddress`.

```
//Pointeur sur la classe InetSocketAddress
Class _class = Class.forName("java.net.InetSocketAddress");

//Pointeur sur le constructeur à hooker
Constructor method = _class.getConstructor(String.class, Integer.
    Type);

//Sauvegarde de la fonction avant de hooker
final MS.MethodPointer old = new MS.MethodPointer();

//On hook l'appel au constructeur
MS.hookMethod(_class, method, new MS.MethodHook() {
    public Object invoked(Object _this, Object... args) throws
        Throwable {
        // Code a executer pendant l'interception
        [...]
        // On choisit de ne pas modifier la valeur de retour
        return old.invoke (_this, args);
    }
}, old);
```

Listing 2. Fonction d'interception du constructeur de la classe `java.net.InetSocketAddress`.

L'utilisation de Substrate permet ainsi de détourner toutes les méthodes identifiées par l'utilisateur. Ces fonctions peuvent être proposées par l'API Android ou par l'application elle-même. Une fois celles-ci interceptées, il est possible de modifier les paramètres des appels de la fonction détournée mais également de modifier sa valeur de retour.

Dans le cadre du Hooker, nous utilisons les extensions Substrate pour identifier les appels à certaines méthodes. Chaque interception engendre la création d'un événement décrivant le contexte d'exécution de la méthode

surveillée. Cet événement est ensuite remonté à un service de collecte puis à l'expert. De cette manière, ce dernier peut observer toutes les interactions entre une application ciblée et son environnement d'exécution (API Android, bibliothèques tierces, *etc.*).

Notion d'événement Comme expliqué précédemment, l'objectif du Hooker est d'observer les appels à des méthodes pré-identifiées. Chaque interception engendre la création d'un événement, qui agrège toutes les informations décrivant le contexte d'appel de la méthode interceptée. Plus précisément, chaque événement agrège les informations suivantes :

- un identifiant unique à chaque événement ;
- une valeur temporelle afin de situer l'événement dans le temps ;
- le nom de l'application à l'origine de l'appel ;
- le nom de la classe dont la méthode interceptée est issue ;
- la nom de la méthode interceptée ;
- les paramètres de l'appel (type et valeur) ;
- la valeur de retour de la méthode interceptée ;
- un identifiant unique associé à l'instance de l'objet ayant généré l'interception ;
- un indicateur d'indiscrétion (ou « Intrusive Level » en anglais). Cet indicateur permet de différencier les événements les plus sensibles. En prenant comme hypothèse que l'accès en écriture à une ressource est plus indiscret qu'un accès en lecture à celle-ci, il devient possible d'identifier les applications effectuant de nombreux accès indiscrets.

Fonctions sensibles identifiées Les fonctions à intercepter doivent permettre de remonter le maximum d'informations au service de collecte. Cependant, un trop grand nombre d'informations serait pénalisant non seulement pour la rapidité d'exécution de l'application, mais aussi pour pouvoir retrouver facilement des informations sensibles dans un flot d'informations potentiellement inutiles. Il est donc primordial d'identifier précisément les différents composants que l'on souhaite cibler. Une fois ces composants identifiés, il est alors nécessaire de lister l'ensemble des fonctions sensibles.

Les informations sensibles identifiées dans Hooker sont dans un premier temps liées aux données personnelles des utilisateurs. Plus précisément :

- les accès aux données de géolocalisation ;
- les accès au carnet d'adresses ;
- les accès réseaux : SMS, appels, internet, bluetooth, NFC ;

- les accès aux données stockées localement à l'application : `SharedPreferences`, bases de données et ressources embarquées ;
- les accès au médias tel que l'appareil photo, le microphone ;
- les accès aux fichiers.

Cependant, davantage de comportements sont capturés par Hooker. Ceux-ci correspondent à des fonctions tout aussi sensibles que les accès aux données personnelles :

- l'interception des méthodes cryptographiques permet de remonter les algorithmes utilisés, les clés symétriques ou asymétriques, les messages avant chiffrement, etc. ;
- l'interception des communications inter-processus permet d'écouter les données transitant entre processus ;
- l'interception des appels à l'environnement d'Android permet par exemple de comprendre si des commandes sont exécutées sur le système ;
- l'interception du chargement de classes dynamiques permet d'observer si l'application exécute du code externe ;
- l'observation des accès au gestionnaire de paquets ;
- l'utilisation des fonctionnalités de WebView ;
- l'utilisation des fonctions de DRM, etc.

En outre, l'expert peut très facilement spécifier de nouvelles classes et méthodes à intercepter. Le *framework* développé au sein de la solution propose une spécification simple et rapide de nouveaux *hooks*. Par exemple, l'interception des accès aux données de géolocalisation par GPS se résume à la création d'une classe Java telle que celle présentée dans le listing 3. Les événements sont alors générés à chaque appel des méthodes ainsi listées. Quant aux paramètres d'appels et de retour, ils sont automatiquement sérialisés en chaîne de caractères. Pour cela, notre *framework* repose sur l'utilisation de la méthode `toString` des paramètres. Si une telle méthode n'est pas disponible, *i.e.* que sa définition correspond à celle de la classe `Object`, alors nous utilisons la réflexion Java pour produire une structure hiérarchisée de l'ensemble des attributs de la classe.

```
public class GeolocationDataHooker extends Hooker {

  public GeolocationDataHooker() {
    super("geolocation");
  }

  @Override
  public void attach() {
    String classToHook = "android.location.GpsSatellite";

    Map<String, Integer> methodsToHook;
```

```
    methodsToHook = new HashMap<String, Integer>();
    // Pour chaque methode un niveau d'indiscretion est specifie:
    // 0: pas d'acces aux donnees personnelles
    // 1: lecture des donnees personnelles
    // 2: ecriture des donnees personnelles
    methodsToHook.put("getAzimuth", 1);
    methodsToHook.put("getElevation", 1);
    methodsToHook.put("getPrn", 0);
    methodsToHook.put("getSnr", 0);
    methodsToHook.put("hasAlmanac", 0);
    methodsToHook.put("hasEphemeris", 0);

    try {
      hookMethods(classToHook, methodsToHook);
    } catch (HookerInitializationException e) {
      SubstrateMain.log("hooking android.location.GpsSatellite
          methods has failed", e);
    }

  }
}
```

Listing 3. Exemple d'une classe pour l'interception de certaines méthodes liées aux données GPS.

Les fonctions à intercepter sont donc définies selon un principe de liste blanche : il faut déclarer tout ce que l'on souhaite observer afin de pouvoir l'intercepter. L'inconvénient majeur découlant de ce principe est qu'il est possible de passer à côté de (*i.e.* d'oublier) certains événements. Si ceux-ci sont importants, l'analyse ne sera alors pas complète. Pour essayer de couvrir le maximum d'événements, la solution a été testée sur un ensemble d'applications légitimes, puis sur un ensemble de *malwares*. Nous détaillons une partie de ces résultats en section 3.4.

Parades contre l'anti-émulation L'utilisation de périphériques virtuels permet de travailler au sein d'une architecture souvent plus contrôlée que sur un périphérique réel. Bien qu'il soit impossible d'assurer une ressemblance parfaite avec un tel périphérique, il existe plusieurs solutions afin d'empêcher une détection trop aisée d'une architecture virtualisée. Nous présentons ces méthodes de détection de l'émulateur et détaillons notre solution pour s'en prémunir.

Une première technique de détection consiste à récupérer le numéro IMEI associé au périphérique. Si celui-ci est égal à une chaîne de caractères de quinze zéros, il s'agit d'un émulateur. Afin que Hooker puisse éviter ce type de détection, il est possible de détourner la méthode `getDeviceId()` de la classe `TelephonyManager` afin de renvoyer une valeur autre que zéro.

Une seconde technique consiste à récupérer les valeurs des attributs statiques de la classe `Android.os.Build`. En effet, en présence d'un émulateur, les chaînes de caractères « sdk » ou « generic » sont présentes dans de nombreux champs statiques de cette classe. Afin que Hooker puisse éviter ce type de détection, leurs valeurs sont modifiées. Pour cela, l'émulateur est initialement préparé en modifiant les propriétés par défaut.

Une troisième méthode ayant été identifiée lors de ces travaux correspond à la récupération du nom de l'opérateur réseau. En effet, si l'on se trouve en présence d'un émulateur, la valeur de celui-ci vaudra « Android ». Comme précédemment, il est alors possible d'intercepter la méthode `getNetworkOperatorName()` de la classe `TelephonyManager`.

Enfin, une technique légèrement annexe aux précédentes, correspond à une technique d'attente. Une analyse étant réalisée en temps contraint, le fait de « dormir » pendant un certain temps permet de cacher tout comportement sensible. Hooker ne permet pas de détecter ce type d'anti-émulation, cependant l'utilisation de cette méthode y est mise en valeur. De cette manière, un utilisateur de Hooker sera capable de voir qu'une application utilise un cycle d'attente pour ensuite s'intéresser à ce mécanisme si souhaité. En outre, il est envisagé de remplacer l'émulateur par un véritable téléphone pour les analyses les plus sensibles. Substrate étant également fonctionnel dans ce cadre, cette fonctionnalité fera l'objet de travaux supplémentaires permettant de réduire le risque de détection de nos analyses par l'application cible.

3.3 Collecte et visualisation des résultats

Le service sous Android Comme détaillé dans la section précédente, notre solution s'injecte dans la mémoire de toutes les applications exécutées sur le téléphone. Quelles soient liées au fonctionnement interne du téléphone ou téléchargées, ces applications produisent des événements qui doivent être remontés à l'utilisateur.

Afin de proposer une analyse en temps réel des événements capturés, le Hooker dispose d'un service dédié à la collecte des événements. Ce service, au sens Android du terme, est exécuté indépendamment du mécanisme d'interception. Pour transmettre les événements au service de collecte, nous reposons sur des `IPC` Android (`Intents`). Pour ce faire, le Hooker intercepte son propre chargement en mémoire afin de récupérer son contexte d'exécution. Celui-ci est ensuite propagé dans la mémoire de toutes les autres applications en exploitant la zone mémoire de Substrate. Comme illustré sur la figure 2, ce contexte est ensuite utilisé par les différents *Hooks* pour communiquer avec le service.

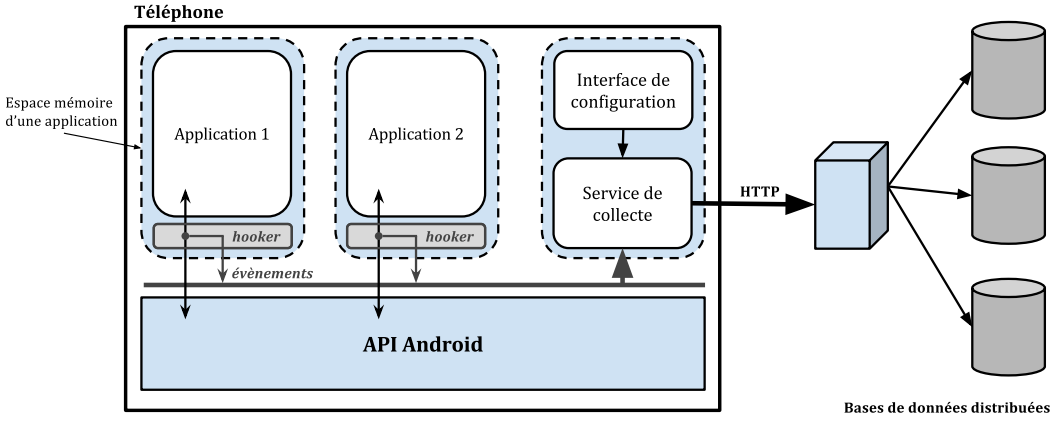

Figure 2. Schéma de propagation des événements générés lors de chaque interception.

Ce service de collecte filtre et agrège l'ensemble des événements produits par le système dans une file d'attente de type FIFO. Un mécanisme de *Pool de Thread* est ensuite chargé de vider cette file d'attente en exfiltrant les événements collectés vers une base de données externe au périphérique.

En plus de la collecte des événements, ce service assure également la communication entre l'interface graphique de configuration du Hooker et les fonctions d'interception. Cette interface de configuration permet notamment de spécifier une liste blanche et une liste noire d'applications à observer. La configuration de ces filtres permet alors de ne pas remonter les informations de certaines applications (*ex :* `com.android.phone`, `com.android.launcher`).

Bases de données distribuée L'analyse de chaque application génère un très grand nombre d'événements. Comme présenté dans la section 3.2, en plus des informations liées à l'analyse statique, l'analyse dynamique proposée par notre solution intercepte et collecte de nombreux appels à l'API Android. À titre d'exemple, l'analyse dynamique de l'application « Drum Machine » [11] génère plus de 76 429 événements. Plus globalement, l'analyse dynamique de quelques 951 applications génère près de 886 787 événements soit une moyenne de 932 événements capturés par application. Si à ce chiffre, nous ajoutons les quelques mesures produites par l'analyse statique, nous obtenons une moyenne proche de 1000 événements par analyse. En outre, d'après les différentes statistiques sur le nombre d'applications Android, la plateforme mise en place doit pouvoir supporter l'indexation et l'analyse de plusieurs milliers d'applications.

11. L'application musicale Drum Machine (package name : com.bti.dMachine) est téléchargée par plus de 100 000 utilisateurs sur le google play.

Afin d'exploiter correctement l'ensemble de ces informations, que ce soit dans le cadre d'une analyse microscopique ou macroscopique, ces événements doivent être organisés. Pour cela, nous utilisons une base de données. Cependant, la quantité de données collectées et la parallélisation des analyses (plusieurs applications sont analysées en parallèle) engendre un flux de données conséquent sur la base de données. En outre, cette base de données est également scrutée en permanence par les utilisateurs de la plateforme, et doit donc être en mesure d'absorber un très grand nombre de requêtes en lecture et en écriture. Elle doit également pouvoir aisément s'interfacer avec un moteur de recherche.

Pour ces raisons, nous avons fait le choix d'une base de données distribuée *open-source* appelée ElasticSearch. Celle-ci dispose de l'ensemble des mécanismes nécessaires à la distribution de son index sur plusieurs noeuds. Elle expose également un moteur de recherche exploitant le langage Apache Lucene. Ces technologies sont utilisées par certains acteurs du « Big Data » tels que Xinc, GitHub, Stack Overflow et Foursquare.

Visualisation des résultats L'accès aux résultats d'une ou de plusieurs analyses peut se faire de deux manières différentes. La première s'articule autour d'une interface web reposant sur le *framework* Kibana, la seconde sur une API Python dédiée à l'interrogation de notre base de données distribuée. Cette dernière étant principalement utilisée dans le cadre des macroanalyses, nous la détaillons dans la section 4.

Dans le cadre de l'interface web, l'expert dispose d'une interface graphique paramétrable pour visualiser, sous la forme de graphiques et de tableaux, l'ensemble des événements collectés. Le *framework* utilisé pour cette interface graphique permet également à l'expert de créer ses propres composants graphiques. En outre, il peut spécifier des filtres afin de se concentrer sur l'analyse des événements se rapportant à une ou plusieurs applications en particulier. En plus du filtrage sur les applications, l'expert peut rechercher l'ensemble des événements comportant un motif spécifié. Par exemple, il peut visualiser l'ensemble des applications se connectant sur la même adresse IP ou celles utilisant les mêmes clés de chiffrement. Les filtres pouvant se combiner, il est également possible de visualiser l'ensemble des événements réseaux d'une application (*e.g.* `bind`, `connect`, `recvfrom`, `send`) et en extraire les flux de communication.

Par défaut, aucun filtre n'est établi et l'ensemble des événements des analyses dynamiques et statiques sont disponibles. Quelques statistiques générales sont ainsi proposées telles que les permissions les plus courantes, les classes de l'API Android les plus utilisées et les méthodes de l'API

Android les plus exploitées. Un graphique représentant l'évolution et le type des événements capturés dans le temps est également disponible. La figure 3 est une copie d'écran de quelques graphiques par défaut de l'interface web.

En plus des composants graphiques, l'interface web propose la visualisation, dans un tableau, des événements sélectionnés après filtrage. Celui-ci peut être trié et permet d'accéder à la valeur des paramètres d'appels et de retours de chaque méthode interceptée. L'exemple représenté sur la figure 4 permet d'observer les échanges réseaux générés lors de l'analyse dynamique d'une application. La capture d'écran représentée sur la figure 5 montre le détail d'un événement se rapportant à l'initialisation d'un chiffrement par AES. Parmi les paramètres de cet événement, il est possible d'observer la valeur de la clé utilisée pour l'initialisation du chiffrement.

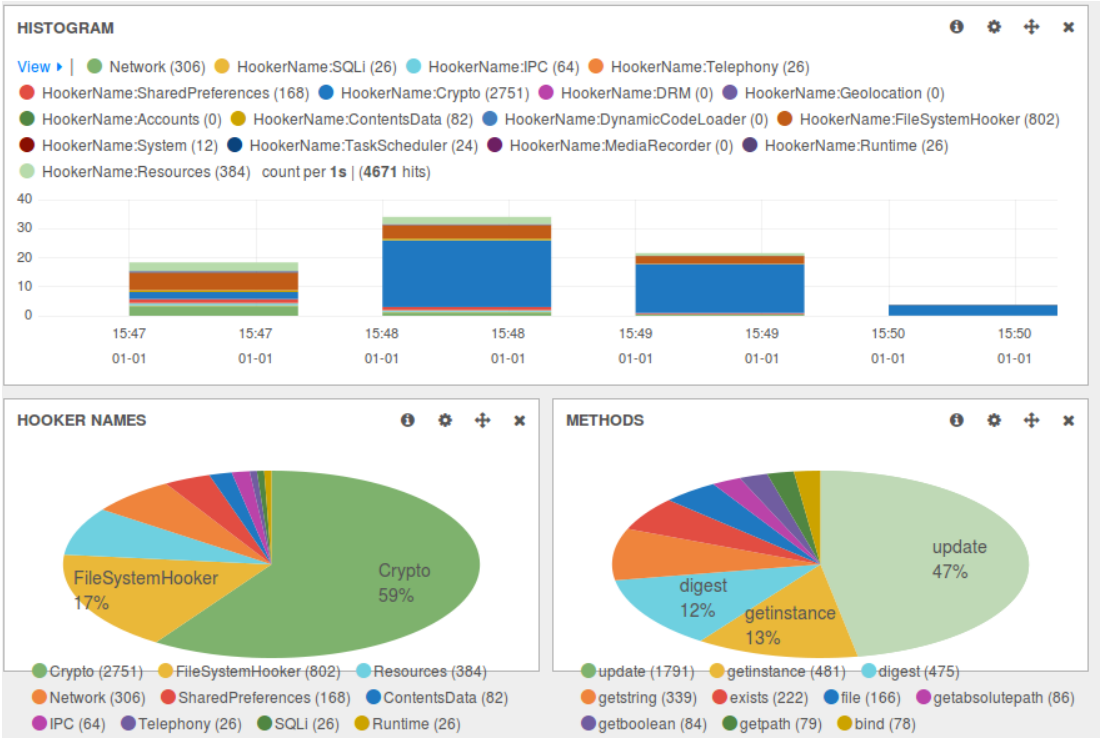

Figure 3. Page d'accueil de l'interface web de visualisation des résultats.

3.4 Exemple de cas d'applications

Dans cette section, nous expliquons quel type de résultats sont remontés par Hooker en prenant le cas concrêt de *malware* Android.

HookerName	PackageName	ClassName	MethodName	Return.ReturnValue
Network	com.melodis.midomiMusicIdentifier	java.net.URL	getHost	settings.crashlytics.com
Network	com.melodis.midomiMusicIdentifier	javax.net.ssl.HttpsURLConnection	getSSLSocketFactory	OpenSSLSocketFactoryImpl[instantiationException=&l...
Network	com.melodis.midomiMusicIdentifier	java.net.Socket	Socket	
Network	com.melodis.midomiMusicIdentifier	java.net.Socket	connect	
Network	com.melodis.midomiMusicIdentifier	javax.net.ssl.HttpsURLConnection	getSSLSocketFactory	OpenSSLSocketFactoryImpl[instantiationException=&l...
Network	com.melodis.midomiMusicIdentifier	java.net.Socket	Socket	
Network	com.melodis.midomiMusicIdentifier	javax.net.ssl.HttpsURLConnection	getHostnameVerifier	DefaultHostnameVerifier[]
Network	com.melodis.midomiMusicIdentifier	java.net.URL	getFile	/spi/v2/platforms/android/apps/com.melodis.midomiM...
Network	com.melodis.midomiMusicIdentifier	java.io.IOException	IOException	
Network	com.melodis.midomiMusicIdentifier	java.net.Socket	close	
Network	com.melodis.midomiMusicIdentifier	java.net.Socket	close	
Network	com.melodis.midomiMusicIdentifier	libcore.io.IoBridge	closeSocket	
Network	com.melodis.midomiMusicIdentifier	libcore.io.IoBridge	closeSocket	
Network	com.melodis.midomiMusicIdentifier	libcore.io.IoBridge	bind	

Figure 4. Tableau proposé par l'interface graphique pour visualiser les événements sélectionnés.

RelativeTimestamp	HookerName	PackageName	ClassName	MethodName	Return.ReturnValue
53274809	Crypto	com.melodis.midomiMusicIdentifier	javax.crypto.Cipher	init	

View: **Table** / JSON / Raw

Field	Action	Value
ClassName		javax.crypto.Cipher
Data		
HookerName		Crypto
InstanceID		1094434584
IntrusiveLevel		0
MethodName		init
PackageName		com.melodis.midomiMusicIdentifier
Parameters		{"ParameterType":"java.lang.Integer","ParameterValue":"1"}, {"ParameterType":"javax.crypto.spec.SecretKeySpec","ParameterValue":"SecretKeySpec[algorithm=AES/ECB/PKCS7Padding,key=[101,53,98,49,102,56,52,102,101,49,49,48,52,100,99,52,97,102,99,52,99,102,54,99,102,100,102,48,99,50,56]]"}
RelativeTimestamp		53274809
Timestamp		1397378485227
_id		wttEjbPRTsKJKUmaHc4JSQ
_index		hooker_test
_type		event

Figure 5. Affichage des paramètres d'appel de l'initialisation d'un chiffrement par AES.

Malware « jsmsHider » Ce malware a été téléchargé en parcourant le *market* Android Slideme [12]. Pour analyser celui-ci, nous avons stimulé manuellement l'application et avons observé les événements capturés.

Pendant l'analyse, le malware a généré 922 événements. Plus précisément, 77 événements se rapportent à des actions liées à des accès réseaux, 205 à des IPC Android, 88 à des accès à l'API de téléphonie, 189 à des accès aux `SharedPreferences` et 363 à des fonctions de cryptographie.

12. Market Android Slideme : `http://slideme.org`

La figure 6 représente, dans le temps, le nombre et le type des événements générés. On peut y observer qu'après une période d'initialisation, l'application exécute régulièrement des appels aux API cryptographique, téléphonique et réseau d'Android.

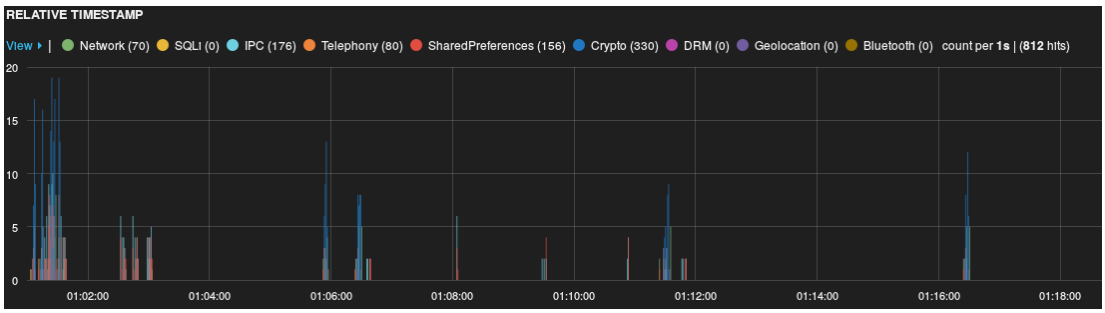

Figure 6. Observation des premiers événements générés par JSMSHider dans le temps.

L'observation des événements liés au trafic réseau fait apparaitre des connections HTTP sur l'URL `http://svr.xmstsv.com/Test/` extraite depuis ses `SharedPreferences` tel qu'illustré par l'événement du listing 4.

```
"IDXP": "42d573efa9f0dbd0a3821518150e829a",
"Data": [],
"Parameters": [
  {
    "ParameterType": "java.lang.String",
    "ParameterValue": "DOMAIN1"
  },
  {
    "ParameterType": "java.lang.String",
    "ParameterValue": "http://svr.xmstsv.com/Test/"
  }
],
"HookerName": "SharedPreferences",
"MethodName": "getString",
"PackageName": "j.SMSHider",
"Return": {
  "ReturnValue": "http://svr.xmstsv.com/Test/",
  "ReturnType": "java.lang.String"
},
"RelativeTimestamp": 89692,
"ClassName": "android.app.SharedPreferencesImpl",
"IntrusiveLevel": 0,
"Timestamp": 1391094539222
```

Listing 4. Événement capturé montrant l'extraction d'une URL depuis les `SharedPreferences`.

Par ailleurs, l'analyse montre que le malware utilise de manière répétée l'algorithme cryptographique DES. Une rapide concordance met alors en valeur que ce chiffrement est réalisé en amont d'une communication réseau.

L'événement capturé et illustré dans le listing 5 montre que le malware utilise un chiffrement DES initialisé avec la clé « `49,50,51,52,53,54,55,56` » et le générateur pseudo aléatoire `SHA1-PRNG`. On peut alors remarquer que ce dernier est instancié avec son constructeur par défaut et donc une graine arbitrairement fixée à 0. Le malware utilise ensuite cette clé pour chiffrer les données personnelles telles que le numéro et le type de téléphone, ainsi que sa géolocalisation.


```
"IDXP": "42d573efa9f0dbd0a3821518150e829a",
"Data": [],
"Parameters": [
{
  "ParameterType": "java.lang.Integer",
  "ParameterValue": "1"
},
{
  "ParameterType": "javax.crypto.spec.SecretKeySpec",
  "ParameterValue":     "SecretKeySpec[algorithm=DES,key
     ={49,50,51,52,53,54,55,56}]"
},
{
  "ParameterType": "java.security.SecureRandom",
  "ParameterValue": "SecureRandom[algorithm=SHA1PRNG,provider=Crypto
        version 1.0,secureRandomSpi=org.apache.harmony.security.
     provider.crypto.SHA1PRNG_SecureRandomImpl@412529e0,seed=0,
     nextNextGaussian=0.0,haveNextNextGaussian=false]"
}
],
"HookerName": "Crypto",
"MethodName": "init",
"PackageName": "j.SMSHider",
"Return": {},
"RelativeTimestamp": 66501,
"ClassName": "javax.crypto.Cipher",
"IntrusiveLevel": 0,
"Timestamp": 1391095199464
}
```


Listing 5. Événement capturé montrant l'initialisation d'un chiffrement DES.

Le malware disposant de nombreuses fonctionnalités, nous ne détaillerons pas davantage l'analyse de celles-ci. Cet exemple permet cependant de montrer, d'une part, que Hooker identifie précisément les informations extraites par le malware et permet d'en connaitre l'utilisation (envoi sur un serveur externe dans l'exemple) et d'autre part, que la précision des

données remontées par Hooker permet d'identifier de potentielles faiblesses d'implémentation (cas de la graine cryptographique à 0 dans l'exemple).

Bitcoin malware Récemment, un nouveau type de malware relatif au minage de Bitcoin a été référencé par les éditeurs d'antivirus. Ceux-ci se dissimulent au sein d'applications légitimes et utilisent le périphérique de l'utilisateur pour leurs opérations de *mining*. Pour analyser les résultats de Hooker sur ce type d'application, nous avons récupéré un exemplaire proposé sur le site de Contagio Minidump [13].

Une fois l'application installée et stimulée, on remarque que les principaux événements sont relatifs au système de fichiers (environ 700) et à de la cryptographie (environ 400). La figure 7 met en valeur ce résultat.

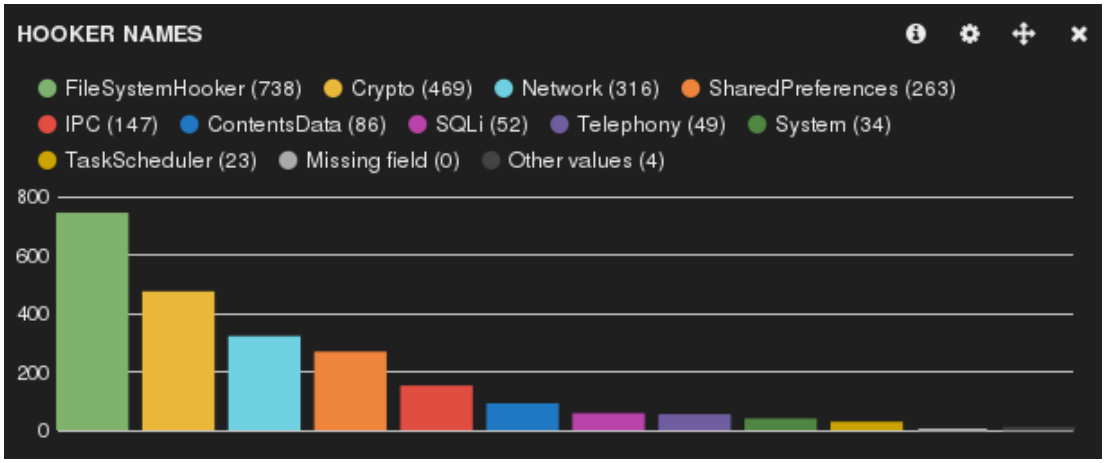

Figure 7. Type d'événements créés quelques secondes après l'installation du Bitcoin miner.

Ces événements sont dus à l'utilisation intensive des classes `java.security.MessageDigest` et `java.io.File`. Celles-ci sont utilisées par le malware pour « miner », autrement dit pour calculer des hachés cryptographiques.

Lorsque l'on quitte l'application et que le système est redémarré, on remarque alors que le même type d'opérations continue. La figure 8 illustre même que cette fois, les opérations de hachage sont désormais plus nombreuses que les accès au système de fichiers. Cela est dû à l'application légitime hébergeant le malware, celle-ci n'étant plus exécutée suite au redémarrage du système, contrairement au malware qui s'est mis en résidence pour effectuer ces opérations à l'insu de l'utilisateur.

13. Contagio minidump : `http://contagiominidump.blogspot.fr/`

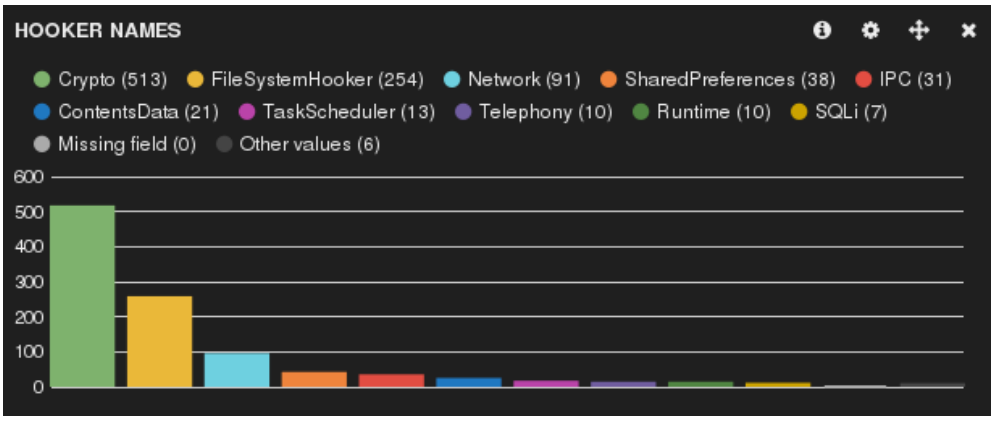

Figure 8. Type d'événements interceptés quelques secondes après redémarrage du système.

Afin de pouvoir rester le plus discret possible, le malware écoute les événements liés à la présence de l'utilisateur sur le mobile. Si celui-ci est absent, il démarre la classe `com.google.ads.Main` qui contient le code relatif à la logique de *mining*. L'événement du listing 6 illustre ce comportement.

```
"source": {
  "Data": [],
  "Parameters": [
    {
      "ParameterType": "com.google.ads.Main",
      "ParameterValue": "Main.2[mPendingResult=<null>,
          mDebugUnregister=false]"
    },
    {
      "ParameterType": "android.content.IntentFilter",
      "ParameterValue": "IntentFilter[mActions=[android.intent.
          action.USER_PRESENT],mCategories=<null>,mDataAuthorities
          =<null>,mDataPaths=<null>,mDataSchemes=<null>,mDataTypes
          =<null>,mHasPartialTypes=false,mPriority=0]"
    }
  ],
  "HookerName": "IPC",
  "MethodName": "registerReceiver",
  "PackageName": "com.melodis.midomiMusicIdentifier",
  "InstanceID": 1093725360,
  "Return": {},
  "RelativeTimestamp": 53248230,
  "ClassName": "android.content.ContextWrapper",
  "IntrusiveLevel": 0,
  "Timestamp": 1397378458650
}
```

Listing 6. Événement capturé montrant l'enregistrement du malware pour l'événement `USER_PRESENT`.

Une seconde technique utilisée par ce malware consiste à embarquer un shell *custom*. Celui-ci est proposé dans le package `com.stericson.RootTools` et offre de nombreuses *fonctionnalités* sur le système d'exploitation. Lorsque ce shell est lancé, celui-ci est donc identifié par Hooker en interceptant la classe `ProcessBuilder`, et plus spécifiquement la méthode `redirectErrorStream` (voir listing 7).


```
{
  "source": {
    "Data": [],
    "Parameters": [
      {
        "ParameterType": "java.lang.Boolean",
        "ParameterValue": "true"
      }
    ],
    "HookerName": "Runtime",
    "MethodName": "redirectErrorStream",
    "PackageName": "com.melodis.midomiMusicIdentifier",
    "InstanceID": 1093034136,
    "Return": {
      "ReturnValue": "ProcessBuilder[command=[/system/bin/sh],
          directory=<null>,environment={ANDROID_SOCKET_zygote=9,
          ANDROID_BOOTLOGO=1, EXTERNAL_STORAGE=/mnt/sdcard,
          ANDROID_ASSETS=/system/app, PATH=/sbin:/vendor/bin:/system
          /sbin:/system/bin:/system/xbin, ASEC_MOUNTPOINT=/mnt/asec,
           LOOP_MOUNTPOINT=/mnt/obb, BOOTCLASSPATH=/system/framework
          /core.jar:/system/framework/core-junit.jar:/system/
          framework/bouncycastle.jar:/system/framework/ext.jar:/
          system/framework/framework.jar:/system/framework/android.
          policy.jar:/system/framework/services.jar:/system/
          framework/apache-xml.jar, ANDROID_DATA=/data,
          LD_LIBRARY_PATH=/vendor/lib:/system/lib, ANDROID_ROOT=/
          system, ANDROID_PROPERTY_WORKSPACE=8,32768},
          redirectErrorStream=true]",
      "ReturnType": "java.lang.ProcessBuilder"
    },
    "RelativeTimestamp": 53331306,
    "ClassName": "java.lang.ProcessBuilder",
    "IntrusiveLevel": 1,
    "Timestamp": 1397378541732
  }
}
```


Listing 7. Événement capturé montrant l'utilisation d'un shell custom.

Au final, l'analyse par Hooker relève toutes les informations nécessaires à l'analyse de ce type de malware qui se cache au sein d'applications légitimes pour effectuer leurs comportements malveillants. La précision des données extraites permet alors, d'une part, d'identifier un potentiel problème, et d'autre part, d'orienter l'utilisateur vers davantage d'informations pour continuer son analyse.

4 Analyse macroscopique

Comme expliqué précédemment, le Hooker a été conçu pour mesurer l'impact d'une application sur le téléphone de l'utilisateur. Pour cela, il intercepte et qualifie une grande partie des appels à l'API Android et Java. Les événements relatifs aux interceptions sont ensuite exploités par l'analyste pour identifier les différentes fonctionnalités de l'application et caractériser l'impact de leur exécution sur le téléphone et les données qu'il héberge.

À l'inverse, l'analyse macroscopique consiste à rechercher les applications partageant des motifs spécifiques de comportements. Ce type d'analyse repose sur l'automatisation à grande échelle de l'analyse microscopique présentée jusque là. Cette automatisation vise à produire une grande quantité de données représentatives du fonctionnement des applications analysées. Comme expliqué dans la suite de cette section, l'exploitation de ces données permet d'offrir à l'analyste une vision globale de l'état des différents écosystèmes Android. Par exemple, une telle analyse permet l'identification des applications n'implémentant pas les mécanismes élémentaires de validation de certificats ainsi que de rechercher à un instant t, toutes les applications se connectant sur une page web vulnérable.

Après une courte présentation des objectifs de l'analyse macroscopique, la section 4.2 détaille son fonctionnement. Pour finir, nous proposons plusieurs cas d'applications de l'analyse macroscopique dans la section 4.3.

4.1 Objectifs

Les techniques utilisées par l'analyse macroscopique s'apparentent fortement aux techniques issues du domaine du *Data Mining* aussi appelé Extraction de Connaissances à partir de Données (ECD) en français (*sic*). Ces techniques visent à mettre en valeur des tendances et caractéristiques communes au sein d'une masse importante de données.

Pour développer cet aspect du Hooker, nous avons suivi un principe simple, très souvent associé aux travaux dans le domaine du *data mining* : nous stockons toutes les données disponibles, même les données *a priori* inutiles ou non significatives. Le coût du stockage de données n'étant plus une limitation forte à la création d'une telle infrastructure, il serait dommageable de ne pas archiver les informations pouvant participer à de futures analyses. Cette approche rend plus réalisable la recherche de précédentes traces d'exploitations d'une vulnérabilité jusque là inconnue.

4.2 Fonctionnement

Le fonctionnement de l'analyse macroscopique repose sur deux principaux blocs. Le premier assure l'exécution automatique d'une microanalyse sur de nombreuses applications. Le seconde bloc se compose de *scripts de post-analyses* permettant la recherche de motifs spécifiques dans la masse d'informations disponibles. Nous détaillons le premier aspect dans cette section et illustrons l'emploi des scripts de post-analyses dans la section 4.3.

Tout d'abord, pour automatiser l'exécution de l'analyse microscopique il est nécessaire de disposer d'applications à analyser. Dans le cadre de nos travaux, nous nous sommes intéressés aux différents *markets* Android officieux existants. Pour cela, nous avons retenu quelques *markets* types et avons développé quelques scripts permettant le téléchargement automatisé de leurs bases d'applications. Pour l'instant, seuls les *markets* Pandaap et SlideMe disposent de scripts fonctionnels. Le développement de ces scripts peut devenir en effet coûteux en raison des restrictions implémentés par ces services. Nous avons donc développé un *framework* exposant quelques méthodes récurrentes au parcours de tels sites web. Ce *framework* assure également l'ordonnancement des scripts de téléchargement et l'indexation des applications et de leurs caractéristiques au sein de la base de données. Parmi ces caractéristiques, on retrouve notamment :

- un identifiant unique attribué à chaque application ;
- l'heure du téléchargement ;
- le nom du fichier ;
- l'empreinte SHA-1 du fichier ;
- la taille du fichier ;
- le nom officiel de l'application ;
- l'URL utilisée pour télécharger l'application ;
- la version déclarée de l'application ;
- le nom du *market* auquel l'application appartient ;
- l'auteur déclaré de l'application.

D'autres informations sont également collectées telles que la catégorie de l'application et sa description. Certaines de ces informations ne sont pas toujours disponibles, dans ce cas une valeur nulle est utilisée.

Afin d'automatiser l'analyse de chaque application, un scénario d'exécution doit être défini. Celui-ci décrit l'ensemble des opérations à exécuter pendant l'analyse et qui permettent de maximiser la quantité et la qualité des informations capturés. En plus de l'analyse statique de l'application présentée dans la section 3.1, un scénario typique se découpe en cinq grandes étapes, 1) la création d'un émulateur dédié à l'analyse, 2) l'instal-

lation de l'application, 3) la stimulation de l'application avec des actions utilisateurs, 4) le redémarrage du téléphone et 5) la stimulation de l'application avec des actions extérieures au téléphone. Nous détaillons ces étapes dans la suite de cette section.

Dans un premier temps, un émulateur dédié à l'analyse est créé à partir d'un émulateur « racine ». Celui-ci est configuré avec le Hooker et est prêt pour une analyse dynamique. Une fois l'émulateur lancé, celui-ci est pré-chargé avec des informations aléatoires permettant de simuler l'utilisation normale du téléphone. On retrouve notamment l'ajout de plusieurs entrées dans le carnet d'adresses du téléphone. À partir de cet instant, le Hooker commence à intercepter l'ensemble des opérations réalisées sur le périphérique. Pour éviter d'intercepter les événements produits par les applications de base du système Android, une liste blanche d'applications à ne pas analyser est disponible.

Dans un second temps, l'application cible est installée sur l'émulateur. Après un temps de pause permettant d'observer l'impact de l'installation sur le téléphone, une stimulation de l'application est réalisée. Cette stimulation a comme objectif de parcourir les différents menus et d'exécuter les différentes fonctions de l'application. De cette manière, nous essayons de maximiser la production d'événements relatifs aux opérations réalisées par l'application. Cette stimulation exploite l'outil Monkey proposé par Google. Il permet de générer un flux pseudo-aléatoire d'événements utilisateurs tels que des clics sur les éléments graphiques, des mouvements horizontaux et verticaux, des frappes claviers ainsi que des événements systèmes (retour arrière, ouverture du menu, baisser le son du téléphone, *etc.*). La durée de cette stimulation et le nombre d'opérations qu'elle représente sont personnalisables.

Une fois l'application stimulée, nous observons les mécanismes de mise en résidence et d'exécution automatique de l'application. Pour ce faire nous redémarrons l'émulateur tout en continuant la capture des événements produits par l'application. De cette manière, il est possible d'observer une potentielle exécution automatique après un redémarrage du périphérique.

Finalement, une dernière étape de stimulation de l'application est réalisée. Contrairement à la première méthode, celle-ci repose sur la génération d'événements externes au périphérique. Par exemple, la réception de SMS, la connexion au réseau wifi ou la modification des coordonnées GPS. Cette stimulation permet d'observer le comportement des applications dont l'exécution est assujettie à l'environnement du téléphone. Pour se faire, nous utilisons un script python qui exploite les fonctionnalités exposées par l'émulateur. Par exemple, la commande Telnet `geo fix 2.33 48.89` permet

de changer la position du capteur GPS sur Paris ; et la commande `gsm call 0633416719` permet de simuler un appel entrant (avec un numéro aléatoire) sur le téléphone. La comparaison des événements capturés avec l'heure exacte de cette stimulation permet de mettre en évidence l'interception par l'application de certaines opérations. Cette stimulation est notamment adaptée à l'analyse automatique des *malwares* interceptant les SMS reçus.

Toutes ces opérations participent au scénario d'analyse des applications, qui peut être facilement adapté par l'utilisateur. Lorsque le scénario s'est correctement exécuté, un événement final est inséré dans la base de données. De cette manière, il est possible de différencier les analyses incomplètes des analyses s'étant correctement exécutées. En cas d'échec, une étude plus approfondie et manuelle de l'application peut être nécessaire.

4.3 Cas d'applications & scripts de post-analyses

Les résultats proposés par Hooker mettent en valeur de nombreux cas d'applications. En plus de l'interface graphique, ces différents cas d'applications exploitent principalement les scripts de post-analyses du Hooker. Ces derniers se connectent à la base de données et utilisent les fonctions de recherche propre à celle-ci pour identifier les applications selon des critères établies par l'utilisateur. Ces critères se rapportent aux types et aux valeurs des événements capturés lors des analyses automatiques.

Pour simplifier la rédaction des scripts, quelques fonctions sont proposées à l'utilisateur. Ces fonctions permettent, entre autres, de rechercher toutes les applications analysées, de lister les événements capturés lors de l'analyse de chaque application ou de rechercher des événements particuliers et d'obtenir la liste des applications les ayants générés. Par exemple, la fonction `getAllEvents` permet de filtrer et d'obtenir les événements capturés selon un critère précis comme la méthode Java interceptée ou la valeur des paramètres d'appels.

Nous détaillons par la suite une liste non-exhaustive de statistiques et de résultats obtenus pour l'instant à l'aide de ces scripts.

Permissions les plus courantes Cette mesure exploite les résultats de l'analyse statique. Elle est très simple et repose uniquement sur la déclaration faite par l'application dans son fichier `AndroidManifest.xml`.

Une simple requête dans la base de données permet d'identifier qu'en moyenne, une application demande 6.72 permissions différentes. Le top ten des permissions est illustré sur la figure 9.

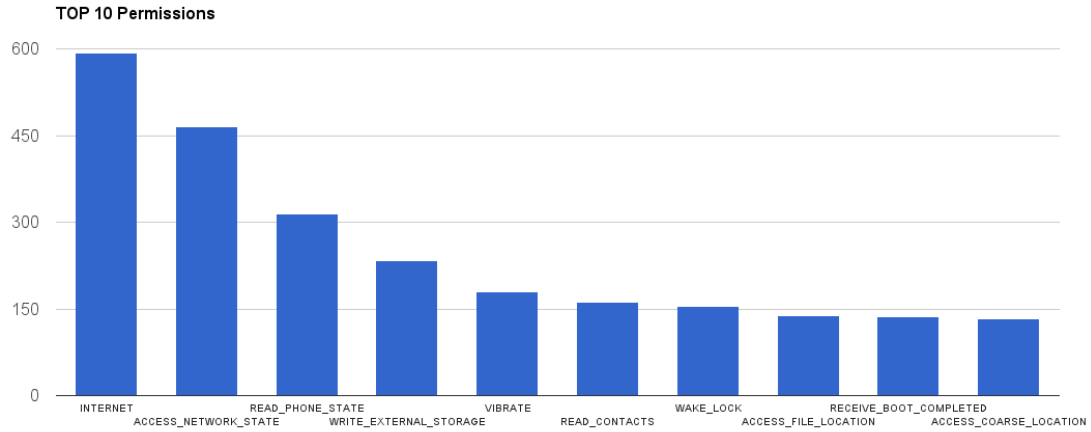

Figure 9. TOP 10 des permissions les plus courantes.

Recherche des applications géolocalisant le téléphone De la même manière, un simple script de post-analyse permet de récupérer la liste des applications demandant l'accès à la dernière position du téléphone. L'exécution de ce script au moment de la rédaction de cet article identifie 32 applications. Parmi celles-ci on retrouve des applications permettant de localiser un avocat à proximité (`com.lawinfo.lawyerlocator` [14]) ou de jouer à des petits jeux n'ayant aucun raison apparente de disposer des coordonnées de géolocalisation du téléphone (`cn.gobbin.smartsmilefree` [15]). Ce dernier exemple étant téléchargé plus de 10.000 fois sur le *market* d'Android (*sic*).

IP/Pays des connections réseaux La classe `java.net.Socket` dispose de la fonction appelée `connect` qui permet de se connecter à un serveur. Les paramètres de cette fonction sont alors particulièrement intéressants étant donné qu'ils correspondent à l'URL, l'IP et le port de destination sur lequel est effectué la requête. De cette manière, il est possible de récupérer ces informations pour chacune des applications.

Sur l'ensemble des applications analysées, le top 4 des URL contactées est le suivant :

- un total de 119 applications se connectent à `googleads.g.doubleclick.net` ;
- un total de 147 applications se connectent à `mm.admob.com` ;
- un total de 15 applications se connectent à `www.google-analytics.com` ;

14. SHA-1 : *1e67a6f204dd22dad0d69430eea5437b62c582f0*
15. SHA-1 : *8c442cc5f1d6afc5834800c1e50f0f4822986f0b*

– un total de 15 se connectent à `spodtronic.com`.

Il est à noter que certaines applications sortent parfois du lot. C'est le cas par exemple de l'application `maslov.mcalc2` qui se connecte à l'URL suivante : `14b64627f76233e07d96-4b50817e43833fe00070e2fd16eaa6e4.r56.cf2.rackcdn.com`. De plus, sur l'ensemble des applications analysées, un total de :

– 336 applications se sont connectées sur un port 80 ;
– Cinq applications se sont connectées sur un port 443 ;
– Deux applications se sont connectées sur un port 8080 (les applications `hu.javaforum.android.falldown`[16] et `de.goddchen.android.dailyquestion`[17]) ;
– Une application s'est connectée sur le port 21 d'un serveur distant (`com.saulius.mockus.solitaire`[18]).

Enfin, nous avons géolocalisé les IP où sont effectuées les requêtes. Le tableau ci-dessous montre que les destinations les plus utilisées sont en Amérique, principalement à *Los Angeles* et *New York*.

Los Angeles	New York	Berlin	Chicago	Amsterdam
40	22	6	3	2

Principaux algorithmes de cryptographie Cette mesure permet d'observer quels algorithmes cryptographiques sont utilisés par les applications. Le top ten de ce résultat est illustré sur la figure 10.

Recherche des applications exploitant le mode ECB Le script de post-analyse illustré dans le listing 8 permet d'obtenir la liste des applications utilisant ECB comme « mode » de chiffrement. Appliqué à notre base d'analyses, ce script permet d'identifier plusieurs applications exploitant ce mode. À titre d'exemple, les applications suivantes font partie des applications identifiées :

– `cn.bluesky.neatcheckers`[19]
– `com.glaa91.calucularcapacitor`[20]

```
from hooker.elasticsearch.Es import Es
from hooker.elasticsearch.EsInterrogator import EsInterrogator
```

16. SHA-1 : *5453ad675755fa17fe3abdbc5527fa730b9ab75d*
17. SHA-1 : *d6f240d7ca4be3826a2c4304afd1d875043bebeb*
18. SHA-1 : *4afb4bbd329ef1456cceda3c7a4a7d9c53f6dda1*
19. SHA-1 : *d2b2e138749b4a50c4ecdf5badf60d296f210b7c*
20. SHA-1 : *a4d2ce18d830edb2ce5fcc89a611d1c0892912bd*

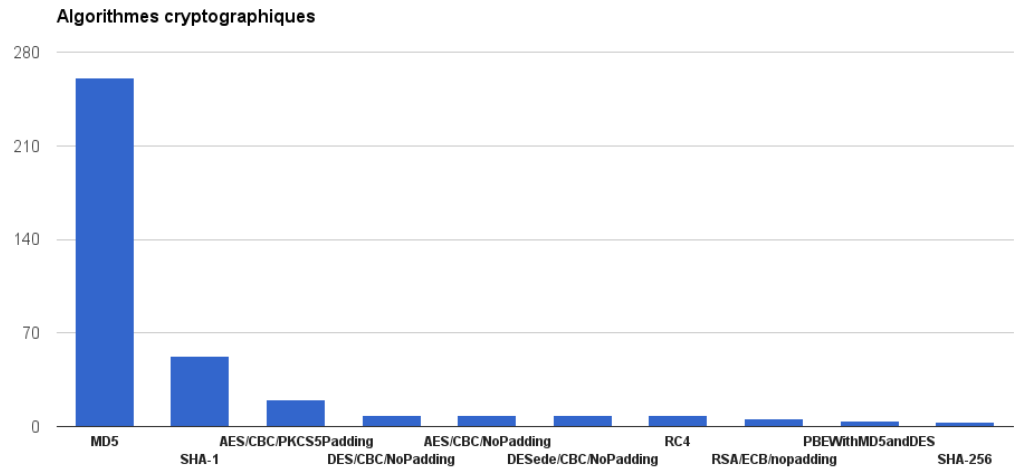

Figure 10. Principaux algorithmes de cryptographie utilisés par les applications.

```
es = Es([{"host":ES_IP, 'port':ES_PORT}])
esInterrogator = EsInterrogator(es)

# Retrieves all Crypto related events
cipherEvents = esInterrogator.getAllEvents(HookerName="Crypto",
    ClassName="javax.crypto.Cipher", MethodName="getInstance")

algosPerXP = dict()

for event in cipherEvents:
    # Stores the algorithm requested by App
    algo = event.Parameters[0]["ParameterValue"]
    if algo not in algosPerXP:
        algosPerXP[algo]=[]

    if event.Parent not in algosPerXP[algo]:
        algosPerXP[algo].append(event.Parent)

# List the Android applications for each Cipher algorithm
for algo, xps in algosPerXP.iteritems():
    apks = [esInterrogator.getAPKInXP(xp) for xp in xps]
    if "ECB" in algo:
        logger.warn("ECB detected: {0} ({1} apks): ".format(algo,
            len(xps)))
        for apk in apks:
            logger.info("\t- {0} ({1})".format(apk.Name, apk.
                Filesha1))
```

Listing 8. Exemple de script Python pour la recherche des applications utilisant ECB comme mode de chiffrement.

Vérification des certificats cryptographiques Cette mesure permet d'observer quelles sont les applications qui redéfinissent leur pro-

pre vérification de certificats X509. Certaines applications décident en effet de contourner le mécanisme de vérification en redéfinissant leur propre `TrustManager` de manière permissive. Cela n'est cependant pas recommandé étant donné que l'application se rend alors potentiellement vulnérable à une attaque de type « *Man in the middle* ». Nous expliquons ci-dessous comment extraire les informations nécessaires à la détection de cette potentielle vulnérabilité.

Afin de pouvoir contourner le mécanisme de vérification d'un certificat, une application doit d'abord définir son propre `TrustManager` et instancier un nouveau contexte SSL prenant en paramètre ce dernier. Le listing 9 illustre ces quelques étapes sous forme de pseudo code Java.

```
SSLContext sslContext = SSLContext.getInstance("SSL");
sslContext.init(KeyManager[] km, TrustManager[] trustManager,
    SecureRandom sr);
HttpsURLConnection.setDefaultSSLSocketFactory(sslContext.
    getSocketFactory());
```

Listing 9. Utilisation d'un contexte SSL se basant sur un `TrustManager` personnalisé.

Dans le cas où la méthode `init()` de la classe `javax.net.ssl.SSLContext` est interceptée, les arguments suivants sont alors récupérés :

- le manager des clés dans le cas d'une authentification du client ;
- le manager de confiance utilisé pour se connecter. C'est cet objet qui doit être modifié pour accepter des certificats dont la validité n'est pas vérifiée ;
- la source d'aléa fournie par le client pour la dérivation des clés. Il est ainsi possible d'observer si ce paramètre est à une valeur autre que `SecureRandom` ou `null` (ce qui correspond alors à l'implémentation par défaut).

À partir de ces informations, le script du listing 10 permet d'extraire le nombre d'applications disposant de leur propre `TrustManager`. Par ailleurs, ce script peut facilement être dérivé pour pouvoir vérifier si des applications utilisent une source d'aléa différente de celle par défaut, ainsi que si elles utilisent un certificat client.

```
def macroAnalyzeX509CustomVerification(esInterrogator):

  initEvents = esInterrogator.getAllEvents(HookerName="Network",
      ClassName="javax.net.ssl.SSLContext", MethodName="init")
  customTrustManagers = dict()

  for event in initEvents:
    logger.debug(eventToString(event))
    if len(event.Parameters)>=1 and len(event.Parameters)<4:
```

```
    if "ParameterType" in event.Parameters[1].keys() and event.
        Parameters[1]['ParameterType'] == "[Ljavax.net.ssl.
        TrustManager;":
      tmp_trustManager = event.Parameters[1]['ParameterValue']
      trustManager = tmp_trustManager.split('{')[1].split('}')[0].
          split('@')[0]

      if not "org.apache.harmony" in trustManager:
        if trustManager not in customTrustManagers.keys():
          customTrustManagers[trustManager] = []
        if event.Parent not in customTrustManagers[trustManager]:
          customTrustManagers[trustManager].append(event.Parent)

        logger.warn("Found custom TrustManager: {0}".format(
            eventToString(event)))
        else:
          logger.info("Found apache TrustManager: {0}".format(
              eventToString(event)))

###Results are in customTrustManagers
```

Listing 10. Détection d'un contexte SSL se basant sur un `TrustManager` personnalisé.

Sur toutes les applications analysées, deux applications ont été trouvées comme utilisant un `TrustManager` customisé. Comme expliqué précédemment, ces deux applications ne sont pas forcément vulnérables à une attaque de type MITM, et seule une analyse personnalisée permettrait de valider cette hypothèse. Les deux applications téléchargées du *market* SlideMe sont les suivantes :

- l'application `com.colapps.reminder` [21] ;
- l'application `com.dngames.websitelivewallpaper` [22].

D'autre part, ce script a mis en valeur l'utilisation du SDK appelé Flurry, et plus précisément de l'utilisation de leur `TrustManager` sur une quinzaine d'applications. Pour information, les fonctionnalités proposées par Flurry permettent de tracer le comportement des utilisateurs en utilisant le service Analytics. Il serait donc intéressant de mener une analyse plus détaillée de l'implémentation du `TrustManager` utilisé par cette bibliothèque. Une potentielle vulnérabilité au sein de celle-ci impliquerait alors que de nombreuses applications sont vulnérables.

Android Webview En fin d'année 2013, une vulnérabilité concernant les objets de type `android.webkit.WebView` a été publiée. Celle-ci permet d'exécuter du code sur le système lorsqu'une application vulnérable accède à une page web malicieuse.

21. SHA-1 : *bac4703cc94558b3bfce90445e63975dbd9bca59*
22. SHA-1 : *95a2c2f29f76a94068febc1d69bf1abb9bf40e02*

La vulnérabilité en elle-même concerne l'utilisation de la méthode `addJavascriptInterface(Object, String)`. Celle-ci permet de faire l'interface entre du code Javascript présent au sein d'une page web, et du code Java. L'exemple du listing 11 illustre l'utilisation de cette *fonctionnalité*.

```
webView.addJavascriptInterface(new JsObject(), "injectedObject");
```

Listing 11. Définition d'une interface Javascript pour la `Webview`.

Si un attaquant arrive à injecter du code Javascript dans le flux de l'application, il est alors possible d'utiliser la réflexion Java pour exécuter du code sur le système. Le listing 12 donne un exemple de code Javascript où la commande passée en paramètre est ensuite exécutée.

```
<script>
function execute(cmdArgs){
return injectedObject.getClass().forName("java.lang.Runtime").
    getMethod("getRuntime", null).invoke(null, null).exec(cmdArgs)
} </script>
```

Listing 12. Fonction Javascript utilisant l'objet précédent pour exécuter du code.

Sur l'ensemble des applications analysées, 32 applications ont été identifiées comme vulnérables. Parmi celles-ci, on peut citer les applications suivantes téléchargées du *market* SlideMe :

- `com.lawinfo.lawyerlocator` [23] ;
- `com.breakingart.worldcup` [24] ;
- `com.mob4.guesstheimagecolors` [25].

L'utilisation de cette méthode par ces différentes applications est problématique d'un point de vue sécurité, mais ne signifie pas que celles-ci sont vulnérables. Pour disposer d'un constat plus précis, il faudrait donc effectuer une analyse plus approfondie de chacune de ces applications.

Remove / bypass device locks (CVE-2013-6271) La société Curesec [26] a découvert qu'il était possible de contourner le flux d'exécution théorique de la classe `com.android.settings.ChooseLockGeneric`. De cette manière, il est possible de générer un `Intent` à destination de cette classe qui va provoquer la suppression de manière permanente ou temporaire de cette protection. Pour information, le code du listing 13 provoque le comportement souhaité.

23. SHA-1 : *1e67a6f204dd22dad0d69430eea5437b62c582f0*
24. SHA-1 : *1abf31b9890d5baa34d212cbd950d86367138a37*
25. SHA-1 : *9cbbb029418b7fb3cf9efa1957dd24d2b8c2b334*
26. Curesec : `http://www.curesec.com/de/index.html`

```
private void removeLocks() {
    Intent intent = new Intent();
    intent.setComponent(new ComponentName("com.android.settings", "
        com.android.settings.ChooseLockGeneric"));
    intent.putExtra("confirm_credentials", false);
    intent.putExtra("lockscreen.password_type",0);
    intent.setFlags(intent.FLAG_ACTIVITY_NEW_TASK);
    startActivity(intent);
}
```

Listing 13. Preuve de concept pour enlever le mécanisme de déverouillage du périphérique.

La détection des applications utilisant cette attaque pourrait facilement être réalisée. Cependant, étant donné le type d'attaque et l'absence de dissimulation une fois celle-ci jouée (l'utilisateur n'aura plus de mot de passe et sans rendra compte immédiatement), il y a dans notre contexte peu d'intérêt à détecter celle-ci. Cet exemple montre cependant qu'il est tout à fait possible de détecter des payloads d'exploits à l'aide d'un script de post-analyse.

5 Conclusion

Dans cet article nous avons proposé une solution d'analyse automatisée de *markets* Android baptisée **Hooker**. Cette solution centralise et agrége les résultats d'analyses statiques et dynamiques de milliers d'applications Android. La précision des données centralisées permet alors l'emploi de techniques dites de *data-mining* afin d'offrir une vision globale de l'état des différents écosystèmes Android. Nous avons notamment présenté quelques exemples de scripts de post-analyse permettant la recherche de vulnérabilités au sein des applications analysées.

Concernant l'analyse des applications, nos travaux se sont attachés à proposer deux approches complémentaires. La premiere, appelée micro-analyse, repose sur une analyse statique puis dynamique de l'application cible. Cette dernière consiste à intercepter l'ensemble des appels à l'API Android à l'aide du *framework* Substrate. Afin de valider cette approche, nous avons confronté la solution à des applications légitimes d'abord, puis à des applications de type *malwares*. Les résultats présentés par Hooker permettent au final d'identifier précisément les informations traitées par chacune des applications ayant été testées, ainsi que d'identifier si de potentielles faiblesses d'implémentations sont présentes au sein de celles-ci.

La seconde approche, appelée macroanalyse, repose sur l'automatisation de la microanalyse et permet la création d'une base de connaissance

caractérisant le fonctionnement de milliers d'applications. Pour cela, nous avons présenté une architecture simple permettant le traitement de grandes quantités de données. Afin d'illustrer le type de résultats obtenus, nous avons donc récupéré et automatisé l'analyse de près de 1000 applications sur des *markets* Android officieux. Le développement de scripts permet alors d'interroger cette base de connaissance. Nous avons notamment illustré l'emploi de ces scripts pour réaliser des statistiques globales, mais aussi pour identifier plusieurs applications potentiellement vulnérables à des attaques connues et publiques. L'identification de ces faiblesses ne signifie cependant pas que les applications soient réellement vulnérables. Pour vérifier cela, une analyse complémentaire est nécessaire, ce qui n'est pas le but de ces travaux.

Finalement, c'est avec le retour de la communauté scientifique que la solution que nous proposons doit s'améliorer. Pour cela, la solution **Hooker** est librement téléchargeable sous licence open-source [27].

Références

1. William Enck, Peter Gilbert, Byung-Gon Chunn, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid : An information-flow tracking system for realtime privacy monitoring on smartphones. *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.
2. Pierre Jaury and Damien Cauquil. Available online at https ://github.com/sysdream/fino ; visited on february 3rd 2014. 2013.
3. Patrik Lantz. Available online at http ://code.google.com/p/droidbox/ ; visited on february 3rd 2014. 2010.
4. Lok Kwong Yan and Heng Yin. Droidscope : Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. *Proceedings of the 21st USENIX Security Symposium*, 2012.

27. Hooker : `https://github.com/AndroidHooker/hooker`