DÉTECTION D'ACTIVITÉS MALICIEUSES DANS LES APPLICATIONS ANDROID

THÈSE PRÉSENTÉE À L'UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

EN VUE DE L'OBTENTION DU GRADE DE PHILOSOPHIÆ

DOCTOR (PH.D.) EN INFORMATIQUE

QUÉBEC, CANADA

MEMBRES DU JURY

Pr. Raphael Khoury
Département d'informatique et d'ingénierie - université de Québec à Outaouais
Directeur de recherche.

Pr. Kamel Adi
Département d'informatique et d'ingénierie - université de Québec à Outaouais
Président du jury.

Pr. Omer Landry Nguena Timo
Département d'informatique et d'ingénierie - université de Québec à Outaouais
Membre interne.

Pr. Nadia Tawbi
Département d'informatique et de génie logiciel - université Laval
Membre externe.

RÉSUMÉ

Android est le système d'exploitation le plus populaire et le plus utilisé au monde pour les téléphones intelligents. L'une des raisons de cette popularité est l'accessibilité gratuite aux applications. Malheureusement, cette flexibilité d'installation et d'utilisation de toute application surtout celle créée par des tiers a conduit à l'augmentation de propagation d'applications malveillantes qui visent à nuire aux utilisateurs et à leur vie privée.

Dans cette thèse, différentes méthodes sont présentées et expérimentées dans l'intention de résoudre le problème de la détection d'applications malveillantes Android. L'analyse des données d'un système de détection en type hybride (statique et dynamique) est développée.

L'exploration de l'efficacité des modèles utilisant deux types de fonctionnalités différentes, à savoir les permissions et les appels systéme, mènera, pour commencer, à la réduction de celles indésirables grâce à une phase d'abstraction. Ensuite, différents modèles de classification sont utilisés pour déterminer si une application donnée est infectée ou bénigne. Les métriques de performance des différents modèles sont ensuite comparées pour identifier la technique qui offre les meilleurs résultats à cet effet de détection des logiciels malveillants.

On conclut pour finir que l'approche basée sur l'analyse hybride est plus efficace que l'analyse statique ou dynamique séparée.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

TABLE DES FIGURES

REMERCIEMENT

Je tiens à remercier dans un premier temps, Monsieur Raphaël Khoury pour avoir accepté de m'encadrer. En effet, sa collaboration, sa persévérance, ses conseils précieux, la clarté et la richesse de son savoir, son esprit critique et son raisonnement empreint de précision m'ont toujours inspiré et aidé. Permettez-moi Monsieur de vous exprimer ma plus profonde gratitude.

À mes parents les plus chers au monde pour tous les sacrifices qu'ils ont faits pour moi, toute la confiance qu'ils m'accordent et tout l'amour dont ils m'entourent. Une reconnaissance éblouissante et particulière à mon cher mari qui a voulu me voir réussir et n'a pas cessé de me fournir son soutien. Merci à ma fille "Sofines" de me permettre de vivre une vie très bien animée.

J'ai une pensée pour mes ami(e)s rencontrés à Chicoutimi BelGacem, Massiva, Imène, Souhail, Maissa, Yessine, Ahmed, Amine, Rihana, Fadwa, Dounia et Nesrine pour avoir animé mes années d'étude. Je voudrais aussi me souvenir de mon amie de la France Sabrine.

Pour finir, je remercie toutes les personnes intéressées par ce projet, en espérant qu'elles puissent trouver dans cette thèse des explications utiles pour leurs propres travaux.

INTRODUCTION

Le directeur de la recherche et du développement de Motorola aux États-Unis, Martin Cooper, a inventé en 1973 le premier téléphone portable, ((34)). L'usage de ces appareils a subi une croissance fanatique qui en a fait l'une des technologies les plus consommables. En 2016 le nombre total d'abonnements à la téléphonie mobile atteignait environ 7,5 milliards et augmentait d'environ 3% l'année d'après, ((99)), et près de 80% de la population mondiale utilisaient des réseaux GSM terrestres.

Avec les progrès technologiques, les téléphones mobiles d'aujourd'hui ont évolué en « téléphones intelligents ou *Smartphone* » avec des fonctionnalités plus sophistiquées qu'auparavant. Le premier téléphone intelligent était *IBM Simon*, lancé en 1993 aux États-Unis, ((37)). Et puis Nokia a présenté son premier téléphone intelligent appelé *Nokia Communicator* en 1996, ((70)).

Aujourd'hui, les plateformes logicielles utilisées dans les téléphones intelligents sont diverses. Parmi les plus utilisées on trouve les suivantes :

— **Android** qui a été fondé à Palo Alto, en Californie, en octobre 2003, (68), par Andy Rubin, Rich Miner, Nick Sear et Chris White. Il est décrit comme un système d'exploitation mobile et il a été vendu à Google en 2005.
En 2014 un milliard de téléphones intelligents sont équipés par Android. Gartner [1] estimait que 60% des appareils mobiles se servaient des systèmes d'exploitation Android en 2015.Ce système d'exploitation est mis à disposition de nombreux fabricants de téléphones (Samsung, Acer, HTC, LG, Sony et Huawei) qui doivent respecter certaines conditions liées aux services de Google.

— **iOS d'Apple**. Le premier iPhone OS a été publié le 29 juin 2007 par Apple, ((19)). L'iOS a la deuxième place après Android dans les systèmes d'exploitation mobile les plus consommés dans le monde, avec environ 239,2 millions de téléphones intelligents

1. *Société de recherche et de conseil en technologie de l'information basée à Stamford, Connecticut US.*

équipés en 2021, ((33)) . Cependant, il fonctionne uniquement sur les propres produits d'Apple. Il faut toujours passer par l'Apple Store pour installer sur un iPhone ou iPad les applications désirées.

- **Chrome OS** a été crée par Google en 2010, ((90)). Au début de sa création, il a été destiné pour la navigation web seulement et totalement distinct d'Android mais ces dernières années certaines applications Android deviennent compatibles avec lui.
- **Windows Phone** de Microsoft. Le 21 octobre 2010, Microsoft a lancé son premier système d'exploitation mobile ((108)), avec un design basé sur le design de Windows 8. *Windows Phone* est remplacé en 2015 par *Windows 10 Mobile*.

Sur l'ensemble du marché des systèmes d'exploitation de téléphone intelligent (OS), Android a encore renforcé son avance sur iOS d'Apple en septembre de 2022, en assurant une part de marché de 71,55% à 27,8% pour iOS, ((114)). Ce qui démontre bien l'accroissement exponentiel de l'utilisation de ce type d'appareil.

CONTEXTE

La nécessité d'un téléphone dépasse le fait de téléphoner et envoyer des messages. Les téléphones intelligents permettent de faire bien d'autres choses grâce à des applications : échanger des photos, se connecter aux réseaux sociaux, enregistrer et réaliser de la vidéo et de l'audio, dessiner, accéder à Internet et bien plus encore. La variété et le besoin d'utilisation du téléphone mobile sont infinis, d'où la croissance de nombre d'applications de téléphone intelligent. Non seulement les places de marché officielles d'applications[2] , comme *Google Play* et *App Store* les plus connus, mais plusieurs autres (*F-Droid, AppChina, AppBrain, YAAM Market*, etc.) disposent des applications pour téléphones intelligents, ce qui augmente la popularité et l'utilisation. Mais une telle célébrité attire aussi les cybercriminels qui lancent des applications malveillantes sur le marché afin de voler des données personnelles. Le téléphone intelligent devient un nouvel abri pour les logiciels malveillants faisant des activités malveillantes. Le *Symantec Internet Security Threat Report* (ISTR) indique que 38% des utilisateurs de téléphones intelligents ont été victimes de cybercriminalité en 2013, ((120)).

VULNÉRABILITÉS DES APPLICATIONS ANDROID

Il existe plusieurs programmes indésirables qui portent le nom de « malware ». Le terme « malware » est une combinaison du mot malveillant (malicious) et logiciel (software). Un malware pourrait se définir comme étant un logiciel parasite présent sur un appareil, se propageant rapidement grâce à une faille d'un autre logiciel ou une erreur de l'utilisateur.

2. *"Magasin d'applications ou app store, est une plateforme en ligne d'applications destinées à des systèmes d'exploitation informatiques public."*

L'histoire des logiciels malveillants pour les téléphones intelligents commence dès 2004 avec l'apparition du premier ver, **Cabir**. Cabir a été développé à l'origine comme preuve de concept par un groupe des professionnels connu sous le nom 29A, ((74)). Il ciblait les appareils fonctionnants sous Symbian OS. Une fois l'appareil est infecté, le mot « Caribe » était affiché sur l'écran de l'appareil. Si Bluetooth était activé, Cabir se propulserait sur d'autres appareils, incitant les utilisateurs à accepter de le télécharger. Il était un virus de preuve de concept, conçu pour démontrer que le code malveillant pourrait être créé pour Symbian afin d'améliorer la sécurité de n'importe quel système que leur création cible. Le code source du ver a cependant été publié sur Internet, ce qui a entraîné que des gens ayant une intention plus espiègle l'ont utilisés. À cause de cela, Cabir a commencé à infecter petit à petit des téléphones partout dans le monde.

Pour Android, l'un des premiers virus est apparu en 2011 avec *DroidDream*[3]. Ce logiciel malveillant peut obtenir un accès racine aux appareils fonctionnant sous Android à des informations sensibles telles que les numéros d'identification uniques de l'appareil, l'identité internationale d'abonné mobile (*IMSI*) et l'identité internationale d'équipement mobile (*IMEI*), ainsi que la langue de l'appareil, le modèle de téléphone et dans certains cas l'ID utilisateur. Cela signifie que le logiciel pourrait potentiellement prendre le contrôle de tout l'appareil et des données qui y sont stockées. Cette éventualité s'est produite au courant de la même année, avec la découverte du malware *DroidKungfu*, ((59)) par des chercheurs américains. Il peut prendre le contrôle de l'appareils et ses fonctions, y compris les données personnelles de l'utilisateur.

Les logiciels malveillants mobiles sont en plein évolution. Un logiciel malveillant cible le vol des données et aide à gagner de l'argent. Les activités courantes incluent le suivi de l'emplacement de notre appareil, l'utilisation d'audio et de vidéo pour nous surveiller, le détournement de textes de notre banque, la facturation de frais sur notre téléphone, la messagerie de nos contacts, la collecte d'informations sur l'appareil et l'installation d'applications et de fichiers afin de contrôler notre appareil.

L'un des types de logiciel malveillant le plus connu et qui affecte le plus des personnes dans le monde est la fraude par SMS, (46). Les achats à distance avec des cartes bancaires falsifiées sont également à la hausse. Les détails de la carte peuvent être obtenus illégalement par des courriels non sollicités, des appels téléphoniques ou des attaques numériques, tels que des programmes malveillants et des piratages de données, puis utilisés pour effectuer des achats malhonnêtes sur Internet ou par téléphone.

3. `https://www.webopedia.com/TERM/D/droiddream.html`

Afin de distribuer un logiciel malveillant sur un téléphone intelligent, les cybercriminels se servent des applications les plus connues. Ils infiltrent un code malveillant dans une application et la proposent en téléchargement sur les magasins de téléchargement. Ce code servira à installer des comportements malveillants ou exploiter des données sensibles, personnelles ou professionnelles sur l'appareil. Prenons l'exemple du malware *MysteryBot* qui a ciblé les applications Outlook, Skype, Messenger, Viber, WhatsApp, Yahoo Mail et plus d'autres applications, ((121)). Une fois la fausse version de l'application est installée, MysteryBot sera capable d'enregistrer ce que l'utilisateur écrit, d'envoyer des SMS malveillants à ses contacts, de transférer tous ses appels vers un autre numéro et le plus scandaleux est d'obtenir de l'argent en échange de l'accès aux photos et vidéos de l'utilisateur après avoir encrypté le contenu de l'appareil.

Parmi les techniques courantes que les logiciels malveillants utilisent pour s'installer, on trouve :

— Le **reconditionnement** (repackaging). Les créateurs des logiciels malveillants choisissent généralement des applications populaires à reconditionner ou à infecter, ce qui augmente les chances que les victimes téléchargent leur version non autorisée. Pour ce faire, le hacker télécharge l'application originale, ajoute des nouvelles instructions ou classes java pour écouter et manipuler les informations échangées entre deux téléphones; sms, services de surveillance etc. et publie la version modifiée. Les applications reconditionnées se trouvent généralement dans des magasins d'applications tiers. Ces magasins ont tendance à définir des limites d'acceptation inférieures à celles de Google.
Des chercheurs allemands ont réussi à construire une base d'applications reconditionnées qui a été publiée sur leur site *Androzoo*[4].

— Les **mises à jour**. Même caractéristique que la technique de «repackaging», mais en plus le hacker cache les éléments malveillants pour éviter la détection par l'ajout de quelques instructions permettent de télécharger du code supplémentaires au moment de l'exécution de l'application. L'algorithme 1 donné par (82), est un exemple du code qui permet de créer un nouveau fichier "AppSMS.apk" et l'utilise comme paramètre pour le programme d'installation du package Android sur un périphérique.

— La **fausse application** consiste à imiter l'application originale en utilisant les mêmes ressources : images ou pages html. Lors de son installation sur l'appareil et lorsque l'utilisateur s'identifie pour accéder à l'application, le programme malveillant ajouté dans cette application imitée, envoie les informations à un serveur et affiche un message d'erreur à l'utilisateur comme illustre l'algorithme 2 de (82). Cet algorithme montre comment obtenir le login et le mot de passe des utilisateurs, envoyer des données à un serveur et afficher un message d'erreur.

4. `https://androzoo.uni.lu/`

Algorithm 1 Exemple d’une fonction de mises à jour

```
private void FUpdates()
{
File localFile = new File (getFilesDir(), "AppSMS.apk");
if localFile.exists() then
    Uri localUri = Uri.fromFile (localFile);
    Intent localUri = new Intent ("android.intent.action.VIEW", localUri);
    localIntent.setData(localUri);
    localIntent.setClassName("com.PackageInstallerActivity",
    "com.android.packageinstaller.PackageInstallerActivity");
    startActivity(localIntent);
    finish();
end if
}
```

Algorithm 2 Exemple d’un code qui récupére les informations de l’utilisateur

```
String str1= ((EditText) findViewById(2131099663)) . getText().toString().trim();
String str2= ((EditText) findViewById(2131099660)) . getText().toString().trim();
DefaultHttpClient localDefaultHttpClient = new DefaultHttpClient();
HttpPost localHttpPost = new HttpPost("http ://erofolio.no-ip.biz/login.php");
Dialog localDialog = new Dialog (this);
localDialog.setContentView (2130837505);
localDialog.setTitle("Your Android TV is not supported");
localDialog.setCancelable(true);
(...);
ArrayList localArrayList = new ArrayList(2);
localArrayList.add(new BasicNameValuePair("email"),str2);
localArrayList.add(new BasicNameValuePair("pass"),str1);
localHttpPost.setEntity (new UrlEncodedFormEntity(localArrayList));
localDefaultHttpClient.execute(localHttpPost);
}
```

— La **publicité malveillante** consiste à insérer des logiciels malveillants dans des réseaux publicitaires en ligne. Les annonces semblent parfaitement normales et apparaissent sur un large éventail d'applications et de pages Web. Une fois que l'utilisateur a cliqué sur la publicité, son appareil est immédiatement infecté par le logiciel malveillant. Certaines attaques malveillantes occupent tout l'écran de l'appareil lorsque l'utilisateur touche l'écran le logiciel malveillant se déclenche.
— Les **les escroqueries** («scams») sont des outils couramment utilisés par les pirates pour infecter les appareils mobiles avec des logiciels malveillants. Ils consistent à rediriger un utilisateur vers une page Web malintentionnée, via une redirection Web ou un écran contextuel. Dans des cas plus ciblés, un lien vers la page infectée est envoyé directement à une personne dans un courrier électronique ou un message texte. Une fois que l'utilisateur est amené sur le site infecté, le code de la page déclenche automatiquement le téléchargement du logiciel malveillant.
— **Directement à l'appareil**. Lorsque les téléphones sont laissés sans surveillance. Le pirate doit réellement toucher le téléphone pour pouvoir installer le logiciel malveillant. Cela implique généralement de brancher l'appareil sur un ordinateur et de télécharger directement le logiciel malveillant sur celui-ci.

Avec la multiplication de nombre des malwares et les méthodes d'installation. La protection des téléphones intelligents contre les attaques est devenue indispensable. C'est pourquoi beaucoup de recherches ont été faites pour implémenter des techniques de protection des téléphones intelligents contre toute intrusion et pour la détection des malwares s'ils auront lieu.

Malgré la présence de contrôle de sécurité des applications dans les magasins de téléchargement des applications, il reste difficile de savoir quelle application a des variantes malicieuses. Partant de ce fait, il devient essentiel de comprendre le fonctionnement de ces malwares afin de les détecter plus tard. C'est pourquoi plusieurs travaux reliés à la sécurité des téléphones intelligents Android se concentraient sur l'analyse et la classification des malwares. Les techniques d'analyse existantes se distinguent selon les fonctionnalités d'Android choisies. Thomas Bläsing et al. (21) ont proposé par exemple une méthode appelée *AASandbox* qui repose sur une approche d'essai et d'erreur pour identifier les modèles suspects dans le code source. Ce type de méthode a une principale limitation qui est l'utilisation de code source d'une application. Généralement le code source n'est pas disponible pour être analysé ou utilisé pour la détection. Pour cette raison d'autres chercheurs ont observé la diversité des logiciels en vérifiant l'exécution de l'application. La vérification de l'exécution est une approche qui consiste à l'analyse et l'extraction d'informations d'un système en cours de fonctionnement et leur utilisation pour détecter les variantes malicieuses.

Par ce projet, nous répondrons à la question scientifique suivante : *existe-t-il des critères permettant de distinguer une application Android malicieuse d'une application bénigne à partir d'enregistrements de leurs traces d'exécution et la récupération de leurs données* *statiques ?*

Dans ce contexte et dans le cadre de ce projet, nous avons adopté une méthode d'analyse des applications Android qui ne nécessite pas l'accès au code source original des applications. Nous avons basé notre étude sur l'observation de la diversité des logiciels en vérifiant les données statiques ainsi que les traces d'exécution des applications Android pour extraire leurs fonctions d'appels système et les comparer avec celles de logiciels malveillants. Nous avons choisi d'apprendre comment un malware modifie le comportement d'une application en analysant directement le malware et se servir de la base obtenue durant l'apprentissage afin de détecter le malware par la suite.

PRINCIPE DE DIVERSITÉ LOGICIELLE

La diversité signifie la création de multiples instances distinctes d'un programme donné. Pour les humains le concept de diversité concerne la race, le sexe, le statut socio-économique, l'âge, les capacités physiques, les croyances religieuses, les convictions politiques ou d'autres idéologies. Dans notre projet de recherche, nous proposons d'explorer les avantages de tirer parti de la diversité pour créer des systèmes plus sûrs.

Étant donné que les logiciels malveillants ciblent généralement une vulnérabilité spécifique dans le code, en prenant deux logiciels développés indépendamment qui implémentent la même fonctionnalité, il est fort probable que l'un d'entre eux présente un comportement malveillant. En conséquence, le comportement observable des deux programmes divergerait à un degré plus élevé qu'ils ne le feraient si les deux fonctionnent sans aucune intrusion. La détection de cette déviation pourrait servir d'indicateur efficace de l'infection.

DONNÉES STATIQUES

Chaque application Android possède un ensemble des permissions qui sont définies dans son fichier manifeste, représentées comme suit, (83) :

```
<permission
android:name="com.eni.android.permission.MA_PERMISSION"
android:label="@string/label_permission"
android:description="@string/desc_permission">
</permission>
```

Les permissions sont classées en trois catégories en fonction de leur niveau de sécurité : normal, dangereux et signature, (36). Nous avons choisi dans nos recherches de conserver les permissions à risque des catégories "dangereux" et "signature", qui peuvent affecter la vie privée de l'utilisateur, le fonctionnement d'autres applications ou les performances de l'appareil.

Si l'application répertorie les permissions normales dans son manifeste, autrement dit, celles qui ne présentent pas de risque important pour la vie privée de l'utilisateur ou le fonctionnement de l'appareil, le système accorde automatiquement ces autorisations. Si l'application répertorie des permissions susceptibles d'affecter la confidentialité de l'utilisateur ou son fonctionnement normal, comme l'accès aux contacts, les galeries, ou internet, l'utilisateur doit approuver ces permissions par lui même.

Dans notre projet, nous allons extraire les permissions ainsi que les ressources demandées par l'application et vérifier leurs influences sur le comportement de l'application.
Les ressources sont définies par <uses-features> dans le fichier AndroidManifest.xml de chaque application Android. Certaines fonctionnalités de la liste des autorisations et des ressources ont une forte corrélation les unes avec les autres. C'est souvent le cas car une certaine autorisation est requise pour accéder à une ressource donnée. Par exemple, la ressource *(android.hardware.camera)* et la permission (*android.permission.CAMERA*) sont corrélées à 99%.

TRACE D'EXÉCUTION

La trace d'un programme est une représentation de l'exécution de ce même programme, c'est un « compte-rendu » de son exécution.
Le traçage est une technique utilisée pour comprendre ce qui se passe dans un système afin d'enlever les erreurs du système («déboguer») ou de le surveiller. Tracer un système consiste à enregistrer des historiques d'exécution reflétant les événements qui se sont produits dans le système pendant son exécution.
Un événement peut être défini comme survenant à un instant précis. Il est donc caractérisé par son type et le moment auquel il a été généré.

Une trace d'exécution liste les adresses de toutes les instructions exécutées par le processeur. Il s'agit d'une information exhaustive, tel qu'on peut retracer la vie d'un programme dans sa totalité. Enregistrer un tel volume de données a néanmoins un coût important en termes de dégradation de performance et de quantité de trace à stocker, ce qui nous mène à la phase d'abstraction des traces par la suite.

LE TRACEUR

Pour enregistrer les événements désirés lors d'une exécution, il est important d'avoir un traceur avec un impact minimal afin que la collecte préserve les performances du systèmes et conserve les informations intactes.

Dans notre projet on a choisi le traceur **Strace**. Il est très mature et préinstallé sur tous les systèmes Unix. Il possède une logique de décodage très complète qui interprète les détails de chaque appel système, même les plus cachés.

La commande strace [5] sur Linux permet de tracer et de suivre les appels système d'un processus lors de son exécution. ((71)). Voici sa syntaxe :

```
# strace [OPTIONS] commande
```

Avec strace on peut intercepter et enregistrer les appels système, le nom de chaque appel, suivi par ses arguments et la valeur de retour qui seront imprimés en erreur standard ou dans un fichier spécifié. Le plus important c'est qu'il fournit la plupart des informations nécessaires afin de comprendre le comportement du système.

LES APPELS SYSTÈME

Le noyau Linux est la couche la plus basse de l'architecture Android. Les appels système (souvent appelés *syscalls*) représentent l'interface fondamentale entre une application et le noyau Linux. Ils fournissent des fonctions aux applications, telles que les opérations sur les fichiers (*open, read, write, etc.*), les opérations réseau (*connect, send, receive, etc.*) ou les opérations sur les processus (*create, kill, etc.*).

Les fonctions peuvent définir zéro, un ou plusieurs arguments en entrées et peuvent fournir une valeur de retour de type *long* qui signifie succès ou erreur. Habituellement, une valeur de retour négative dénote une erreur. Une valeur de retour nulle est généralement un signe de réussite.

Lorsqu'une application exécute un appel système, le noyau s'exécute pour le compte de l'application. En outre, l'application est censée exécuter un appel système dans l'espace noyau, et le noyau s'exécute dans un contexte de processus. Cette relation est la manière fondamentale dont les applications fonctionnent.

Il existe cinq types d'appels système dans un système d'exploitation.

5. `https://strace.io/`

1. **Contrôle de processus** : Parfois, les programmes et les processus restent bloqués ou doivent être fermés. Les appels système de contrôle de processus permettent de créer, quitter ou attendre un processus. Une décharge de données peut être affichée afin de vérifier l'erreur lorsqu'un processus se termine de manière anormale.
2. **Gestion de fichiers** : Il s'agit de l'un des appels système les plus utilisés, car presque toutes les applications ouvrent, lisent et enregistrent un fichier. Le développeur d'applications ne devrait pas avoir à écrire de code élaboré pour ouvrir des fichiers. Au lieu de cela, un simple appel système le lui permet de travailler avec des fichiers.
3. **Gestion d'appareils** : Ces appels système sont responsables de la manipulation d'un périphérique, telle que la lecture et l'écriture dans les tampons de périphérique, etc.
4. **Maintenance de l'information** : Ces appels système traitent les informations et leur transfert entre le système d'exploitation et le programme utilisateur.
5. **La communication** : Ces appels système sont utiles pour la communication interprocessus. Ils procèdent également à la création et la suppression d'une connexion de communication.

Le traçage est un moyen pour surveiller le comportement d'une application pendant son exécution. La présence d'un malware peut être détectée dans une application choisie en analysant les données enregistrées grâce à la trace.

MÉTHODOLOGIE

Tout d'abord, une recherche bibliographique détaillée a été effectuée. La plupart des techniques ont été classées en méthodes statiques, qui s'efforcent de détecter les logiciels malveillants avant leur exécution, ou comme des analyses dynamiques qui observent l'exécution d'une application potentiellement malveillante et réagissent à une violation d'une politique de sécurité, généralement en mettant fin à l'exécution. Basé sur les observations de l'état de la technique effectuée lors de l'étude, plusieurs recommandations sont détaillées et passées en revue en plus d'entamer une discussion sur des pistes de recherches futures. Des remarques conclusives sont données et un tableau récapitulatif de toutes les méthodes étudiées est fourni.

La présentation d'une nouvelle base de données, TwinDroid, dont nous mettons à disposition un jeu de données de traces Android était la deuxième partie de ce projet. TwinDroid est rendu idéal pour la recherche sur la détection dynamique de malware par le fait qu'il est, en grande partie, composé de traces provenant de paires d'applications identiques, une bénigne et une malveillante, dont la dernière inclut un malware. Les applications tracées couvrent une variété de catégories d'applications courantes et sont publiées dans des différentes années. Pour chaque application, le jeu de données contient plusieurs traces de longueurs variables. En plus de l'ensemble de données des traces existantes, TwinDroid présente un pipeline entièrement automatisé et accessible au public pour générer de nouvelles traces à partir d'applications existantes. De plus, tout chercheur ayant besoin de traces d'applications peut utiliser ce pipeline pour générer des traces adaptées à ses besoins spécifiques. Cette facilité d'extension est un objectif de conception crucial car elle permet à la base de données de rester à jour avec les logiciels malveillants émergents. TwinDroid contient actuellement 15 000 traces de 9000 applications différentes.

Nous reproduisons, en troisième partie, des études précédentes sur la détection dynamique à l'aide de notre ensemble de données, permettant la comparaison entre plusieurs détections de logiciels malveillants différentes sur un pied d'égalité. Nous montrons que l'inclusion d'une étape d'abstraction de trace avant d'effectuer une détection automatisée apporte des améliorations significatives au processus de classification. En nous appuyant sur une inspection de notre ensemble de données, nous proposons une nouvelle stratégie de détection des logiciels malveillants qui se base sur l'emplacement de malware dans la trace et montrons qu'elle se compare favorablement aux stratégies présentes dans la littérature.

Finalement, nous comparons l'efficacité des méthodes de détection statiques, dynamiques et hybrides, lorsqu'elles fonctionnent sur un ensemble de données commun. En nous appuyant sur une observation manuelle du jeu de données, nous montrons que l'extraction de caractéristiques à partir des paramètres basés sur ces observations améliore les performances de la classification. Nous avons constaté que l'analyse dynamique est meilleure que l'analyse statique et que l'analyse hybride surpasse les deux. Nous avons également obtenu de nouveaux résultats liés à la sélection de caractéristiques dans l'analyse statique et à l'utilisation de paramètres d'appel système dans l'analyse dynamique.

Cette thèse est écrite selon l'exigence d'une thèse par article. L'ensemble des résultats sont donnés dans les chapitres 2, 3 et 4 sous forme d'articles scientifiques.

STRUCTURE DE LA THÈSE

Dans cette thèse, le premier chapitre présente une revue de littérature des différentes études qui sont liées à la détection des logiciels malveillants pour les applications Android. Nous avons identifié les avantages et les limites de chaque étude présentée et proposé 15 recommandations qui, selon nous, permettront aux chercheurs de développer des outils de détection de logiciels malveillants plus efficaces. Nous avons trouvé difficile de comparer les méthodes de détection car elles utilisaient souvent des ensembles de données de test différents, rapportaient différents types de valeurs (exactitude, précision, rappel, etc.). De plus, la nécessité de codes sources pour appliquer les techniques rendait ardue la reproduction de celles-ci.

Le deuxième chapitre est consacré à l'introduction d'un nouveau processus d'abstraction qui améliore le processus de classification en reproduisant plusieurs techniques de détection de logiciels malveillants de la littérature. Nous proposons dans ce chapitre une nouvelle méthode de classification, basée sur notre observation selon laquelle les logiciels malveillants déclenchent des appels système spécifiques à des moments différents dans des programmes bénins. Chaque trace a été divisée en un nombre donné (k) de segments. Nous avons testé les valeurs de k=5, 10, 30, 50 et 100 segments. Ensuite, nous avons enregistré dans quel segment se trouve la première occurrence de chaque appel système dans notre base de données. La taille de chaque segment diffère naturellement d'une trace à l'autre puisque les tailles des traces sont variables. Pour vérifier l'emplacement des logiciels malveillants dans les traces divisées, nous avons étudié les appels système présents dans les segments. Lors de l'analyse de ces appels, nous avons observé que pour une paire de traces, les appels système représentant le malware sont présents pour la première fois au début de la trace infectée et à la fin de sa version bénigne.

Au troisième chapitre nous introduisons TwinDroid, un ensemble des traces d'appels système, provenant d'applications Android bénignes et infectées. Une grande partie des applications utilisées pour créer le jeu de données provient de paires d'applications bénignes-malveillantes, identiques à l'exception de l'inclusion de logiciels malveillants dans ces dernières. Cela fait de TwinDroid une base idéale pour la recherche en sécurité. En plus d'un ensemble de données de traces, TwinDroid inclut un pipeline de génération de traces entièrement automatisé, qui permet aux utilisateurs de générer de nouvelles traces de manière standardisée de manière transparente. Ce pipeline permettra à l'ensemble de données de rester à jour et pertinent malgré le rythme effréné des changements qui caractérise la sécurité Android.

La taille de TwinDroid a été augmentée de 15 000 traces au chapitre 4 où nous avons utilisé trois des fonctionnalités les plus couramment utilisées pour la détection des logiciels malveillants, à savoir : les autorisations, les données de ressources (données matérielles et logicielles) et les appels système pour détecter automatiquement les comportements malveillants dans les applications. Nous utilisons ces fonctionnalités pour vérifier en quoi les applications bénignes et malveillantes diffèrent par rapport aux données statiques et dynamiques (par rapport aux appels système dans la trace). Notre étude mène à vérifier si les données recueillies statiquement et dynamiquement (hybride) peuvent être utilisées ensemble pour améliorer la détection des logiciels malveillants.

Enfin, dans le dernier chapitre les conclusions sur les différentes méthodes d'analyses et expériences ont été sorties. Dans ce même chapitre, quelques recommandations sont mentionnées comme travaux futurs afin de proposer de travaux futurs.

Au cours de cette thèse et de la réalisation des objectifs fixés, trois articles scientifiques ont été publiés et un quatrième soumis pour publication. Ces articles représentent le corps de cette thèse, de chapitre 1, 2, 3 et 4. Les références complètes de ces articles sont données ci-dessous.

1. **A survey of malware detection in android apps : Recommendations and perspectives for future research** (98)

Asma Razgallah, Raphaël Khoury, Sylvain Hallé and Kobra Khanmohammadi.Computer Science Review, 2021, vol. 39, p. 100358.

2. **Behavioral classification of android applications** (97)
   Asma Razgallah and Raphaël Khoury. 2021b. using system calls ». In 2021 28th Asia-Pacific Software Engineering Conference (APSEC), p. 43–52. IEEE.
3. **Twindroid : A dataset of android app system call traces and trace generation pipeline** (96)
   Asma Razgallah, Raphaël Khoury and Jean-Baptiste Poulet. In 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR), p. 591–595.
4. **Comparing the effectiveness of Static, Dynamic and Hybrid Malware detection on a Common Dataset**
   Asma Razgallah, Raphaël Khoury, Kobra Khanmohammadi and Christophe Pere. *Article soumis pour publication*, 2023.

En plus de ces quatres articles, mes recherches doctorales ont contribué à la rédaction de deux autres articles.

1. **An analysis of the use of CVEs by IoT malware** (66)
   Raphaël Khoury, Benjamin Vignau, Sylvain Hallé, Abdelwahab Hamou-Lhadj and Asma Razgallah. The 13th International Symposium on Foundations Practice of Security. Montréal Canada. December 2020.
2. **Mining Trends of IoT System Vulnerabilities over a 10 Year Period**
   Raphaël Khoury, Fehmi Jaafar and Asma Razgallah. *Article soumis pour publication*, 2023.

CHAPTER 1

ÉTAT DE L'ACCEPTATION : VERSION FINALE PUBLIÉE

JOURNAL : ELSEVIER

DOI :10.1016/J.COSREV.2020.100358

CITATIONS : 40 FOIS

TITRE EN FRANÇAIS

Révue de littérature sur la détection des logiciels malveillants dans les applications Android : recommandations et perspectives pour les recherches futures

RÉSUMÉ

Android a dominé le marché des téléphones intelligents et est devenu le système d'exploitation le plus populaire pour les appareils mobiles. Cependant, les menaces de sécurité dans les applications Android ont également augmenté parallèlement avec le succès d'Android. Plus de 3 millions de nouveaux échantillons de logiciels malveillants ciblant le système d'exploitation Android ont été découverts en 2017. Bien que des efforts de recherche persistants aient été déployés pour lutter contre ces menaces et que plusieurs techniques et outils de détection aient été développés en conséquence, ils présentent tous des limitations distinctes telles qu'aucune solution ne peut prétendre résoudre le problème des logiciels malveillants Android. Dans cet article, nous passons en revue les principaux mécanismes et approches de détection des logiciels malveillants dans les applications Android. Nous identifions les avantages et les limites de chacun et proposons des pistes de recherche pour faire avancer les connaissances à cet égard.

MOTS CLÉS

Détection des logiciels malveillants, Sécurité Android, Sécurité.

CHAPTER 1

A SURVEY OF MALWARE DETECTION IN ANDROID APPS : RECOMMENDATIONS AND PERSPECTIVES FOR FUTURE RESEARCH

Asma Razgallah*, Raphaël Khoury*, Sylvain Hallé*, Kobra Khanmohammadi§
* Department of Computer Science and Mathematics, Université du Québec à Chicoutimi, Canada
§ Department Electrical and Computer Engineering, Concordia University Montreal, Canada

Android has dominated the smartphone market and has become the most popular operating system for mobile devices. However, security threats in Android applications have also increased in lockstep with Android's success. More than 3 million new malware samples, targeting the Android operating system were discovered in 2017. Although persistent research efforts have been to address these threats and several detection techniques and tools have been developed as a result, they all exhibit distinct limitations such that no single solution can claim to solve the Android malware problem. In this paper, we survey the main mechanisms and approaches for malware detection in Android applications. We identify the advantages and limitations of each and suggest avenues of research to advance knowledge in this regard.

Index terms : malware detection ; android security ; security

1.1 INTRODUCTION

It is a common truism of computer security that the user often inadvertently abets the malware running on his device. To aid in protecting the user against himself, Android's architecture is largely concordant with the *principle of least privilege*, stated by Saltzer and Schroeder in their seminal 1975 paper (103), and imposes that an application possesses only the most restrictive set of permissions possible that can still allow it to perform its intended task. However, it is ultimately up to each individual user to decide whether or not to install an application, and to determine which permissions will be granted to each application. Google's commentary on this issue is as follows (89) :

> “When installing an application, users see a screen that explains clearly what information and system resources the application has permission to access, such as a phone’s GPS location. Users must explicitly approve this access in order to continue with the installation, and they may uninstall applications at any time. They can also view ratings and reviews to help decide which applications they choose to install. We consistently advise users to only install apps they trust.”

Nonetheless, when choosing to install an app on their device, the user is often constrained to act more on intuition than on a fact-based decision process. While the user does have access to the permissions requested by the app, he may not be cognizant of the myriad ways in which permissions could be misused to compromise the confidentiality, integrity and availability of his data. The signature, which identifies the publisher of the code, is of little use if this publisher is unknown to the end-user. The user who opts to obtain apps from third-party stores is exposed to even more risks, since these sites frequently contain repackaged apps.

Antivirus software remain the first line of defence for most users. The German security firm AV-Comparatives [1] periodically evaluates antivirus software for Windows, Mac OS, Android and Linux. In January 2019, AV-Comparatives tested 250 anti-virus tools on more than two thousand Android apps, with bleak results (17). Only 80 of them detected a paltry 30% of malicious apps. Over sixty others relied upon a pre-set *white list* of permitted app names, and did not even perform an elementary scan of the app beyond checking its name from that list. In fact, some of the anti-viruses tested failed to block a single malicious app from the testing dataset.

Even worse, anti-viruses themselves can carry malware or exploitable vulnerabilities. Indeed, two entries in the Drebin malware database are antivirus apps (15). In this context, it is useful to remember that anti-viruses run continuously on the user’s devices, often with elevated privileges. They thus form an ideal vector to gather data about a user surreptitiously.

Even when they do perform as intended, only 23 of the 250 tools AV-Comparatives examined achieved 100% detection rates, which shows that anti-viruses have limitations. More particularly, any approach based on signatures is inherently reactive, and cannot provide proactive protection of emergent threats. Note that the dataset used by AV-Comparatives consisted of the “2,000 most common Android malware threats of 2018”. As the authors themselves observe, with such a benchmark, detection rates between 90% and 100% “should be easily achieved”.

Because of their limitations, anti-viruses should be supplemented by methods based on static analysis and dynamic monitoring of the code. These are tools that will analyse an app to determine if it’s *behavior* conforms with a security policy, rather than rely on a signature or a blacklist. The development of these methods is an important current topic of academic and industrial research.

1. https ://www.av-test.org/en/

In this paper, we survey the current state of the art of academic research on the topic of malware detection in Android apps, focusing particularly on the more recent developments. Most techniques can broadly be categorized as *static* methods, which endeavor to detect malware before it is executed, or *dynamic* analysis that observes the execution of a potentially malicious application and reacts to a violation of a security policy —usually by terminating the execution. Because of its ubiquity, we adopt this classification here.

Inclusion Criteria The large volume of these apps does not allow for a comprehensive listing nor for a rational discussion. To illustrate, AV-Comparatives recensed more than 200 security apps in a single category, namely anti-virus software. In order to have a meaningful analysis, we will limit the scope of the paper to the developments of the last ten years, as they are more likely to impact the immediate future. Within this time limit, rather than present an exhaustive survey, we have opted for a sample of models that span a cross-section of current thought on the topic. In doing so, we seek to capture the range of variability that exists with respect to the following questions :

— Which methods are employed to perform malware detection on Android systems ?
— On which features or aspects of the Android app is the detection process based ?
— On which dataset is the method tested ?

Particular attention was given to the way in which the accuracy of methods is tested and measured. We found that datasets and metrics differed widely, making comparison on an equal footing difficult. We briefly describe each sampled method, focusing on its advantages and drawbacks, and put forth recommendations to guide further research on the topic.

We will only review academic research, published in peer-reviewed journals and conferences, to the exclusion of non-peer reviewed industrial works. Furthermore, we focus exclusively on studies that tackle the problem of malware in Android apps, and exclude studies on the broader problem of malware in general. We also limit ourselves to a ten year horizon, and exclude any work that predates 2009. Since our objective is to present a sample of current thought on the topic, we excluded papers whose method was closely similar to other, already included papers.

The papers were drawn from the following 4 online libraries, which provide a comprehensive coverage of major academic publication venues in the field of software security.

1. IEEE (`https://www.ieee.org/`).
2. USENIX (`https://www.usenix.org/`).
3. ACM Digital Library (`https://dl.acm.org/`).
4. Springer Link (`https://link.springer.com/`).

Related work Several researchers have recently surveyed malware detection methods and techniques for Android applications. Naway and LI (84) presented a survey of static, dynamic and hybrid analysis using deep learning techniques to detect the Android malware. They detailed the techniques and specified their strengths and weaknesses. Another survey done by Alzahrani and Alghazzawi (7) on the detection Android malware. They studied eight research papers focused on deep learning for Android Ransomware [2] detection and deep learning for Android malware detection. Although Rubiya and Radhamani (118) have analyzed nine Android malware detection approaches, specifying their strengths and weaknesses. Yan and Zheng in their paper (140) surveyed dynamic mobile malware detection. They analyzed, synthesized and compared previous studies on detecting malware in smartphone. In another survey Arshad *et al.* (16), analyzed the static and dynamic techniques for detection and protection from of Android malware. The techniques analyzed are classified according to the detection mechanism used.

TABLEAU 1.1 – Comparison of surveys

Database	**Dynamic**	**Static**	**Hybrid**	**Number of article studied**	**Years**	**Analysis**
Naway and LI(84)	X	X	X	25	2014–2018	benefits and limitations
Alzahrani and Al-ghazzawi(7)	X	X		8	2014–2019	benefits and limitations
Rubiya and Radhamani(118)	X	X	-	9	2009-2014	benefits and limitations
Yan and Zheng(140)	X	-	-	29	2011–2017	benefits and limitations
Arshad *et al.*(16)	X	X	-	20	2009-2013	benefits and limitations
Our paper	X	X	X	22	2009–2020	benefits, limitations and recommendations

As can be seen in Table 1.1, our survey focuses on static, dynamic and hybrid Android malware detection methods. It tracks the evolution of malware detection during an eleven year time span, (between 2009 and 2020), a time span that is considerably longer than that considered by other similar studies (84), (7), (118), (140), (16) . In addition, to describing the strengths and weaknesses of the methods, we propose recommendations to guide future research in this topic.

The remainder of this paper is organized as follows : in Sections 1.2 and 1.3, we examine

2. Ransomware is a type of malware that prevents users from accessing their system and personal data. This malware encrypts data and demands payment before a can retrieve it.

malware detection mechanisms based on static analysis and dynamic analysis respectively. We also make several recommendations based on our observations of the current state of the art. In Section 1.4, we review these recommendations and discuss avenues for future research. Concluding remarks are given in Section 1.5, and a recapitulative table of all surveyed methods is provided in a closing Appendix.

1.2 STATIC ANALYSIS

Static analysis encompasses a broad range of methods that seek to discern the runtime behavior of a software prior to its execution. In a security context, the purpose is naturally to weed out potentially malicious apps before they are installed and executed. Static analysis is considered as *coarse*, since it flags an app as malicious according to an over-approximation of its possible runtime behavior. As a consequence, any static analysis method must maximize effective detection while minimizing the risk of false positives.

In the ten-year horizon under study, a large number of tools have been developed to address the problem of malware detection using a static analysis approach. For the purposes of this analysis, we broadly categorize these tools in three categories, namely :

1. tools that rely primarily on code analysis, such as bytecode analysis of decompiled code ;
2. tools that rely primarily on API calls and permissions ;
3. other methods that combine multiple factors for detection.

We stress that this categorization is somewhat coarse, and serves merely to organise the survey, rather as a methodological ontology of the field. Indeed, most malware detection mechanisms draw upon multiple factors and resist an easy categorisation. Nonetheless, by presenting a sample of each group, we believe we can highlight the evolution of thought on the topic under consideration over the last ten years.

1.2.1 METHODS BASED ON CODE ANALYSIS

A first category of works concentrates on the analysis of an app's code, either at the source level or at the bytecode level. In the following, we enumerate and discuss the most representative works following this approach.

TinyDroid

TinyDroid(31) is a static malware detection system for Android Apps that relies on a two-step process of first abstracting machine instructions, followed by a machine learning phase.

TinyDroid divides all apps into one of two sets : malware or benign. The APK file of each app is first decompiled into *Smali code* using a system called Apktool [3]. Smali can be seen as a higher-lever explanation of Dalvik bytecode, which is in turn be further abstracted to symbolic instructions by TinyDroid. This detection system then computes the *n*-grams of abstract instructions occurring in the code, and uses this information as the basis for its classification. Hence, a set of *n*-grams is computed for each app under consideration, and compared to the set of *n*-grams extracted from apps that are known to be either benign or malicious. If an app is declared malicious, the set of *n*-grams that characterizes its behavior will be added to TinyDroid database of malicious apps *n*-grams.

Experimental results have shown that TinyDroid exhibits a high level of accuracy. Indeed, while several anti-virus software exhibit a detection rate that falls below 50%, TinyDroid's detection rate (recall) can be as high as 95.6%, which exceeds the performance of 7 of the 9 anti-virus software to which it was compared.

> **Recommendation 1**
> The tool reported in this publication, as is the case for a large majority of the approaches listed in this paper, is not publicly available. Moreover, the benign applications have reportedly been randomly collected from the Google Play Store, but the actual contents of the sample are not disclosed. This makes it impossible to establish a comparison between this approach and any other technique developed in the future. Whenever possible, the research artifacts used in an experimental analysis should be made accessible to third parties.

Malware Detection Using Code Clone Detection Tools

Chen *et al.* (30) studied the use of a code clone detector designed to identify known malicious Android software. They used static analysis to examine the source code of the applications.

The authors first used *dex2jar* to convert the Dalvik virtual machine bytecode to JVM bytecode. The Java bytecode was then subsequently decompiled using the Java decompiler *JD-CORE*.

3. `https://apktool.en.lo4d.com/windows`

This allowed clone detection to be performed on higher-level code. The authors used *NiCad*, an open source program that detects similar segments of codes (functions, classes, blocks etc.) in sets of code files, and clusters these code files according to syntactic similarity. Using a training set consisting of known malicious and benign apps, the authors successfully trained NiCad to perform malware detection effectively.

This approach allows malicious applications belonging to certain malware families to be located efficiently and reliably. Indeed, using a dataset of 1170 malicious apps from 19 distinct malware families, 95% of previously known malware was detected.

Recommendation 2
The method described above was tested using a dataset that contained only malware. Ideally, detection methods should be tested using datasets that contain both malicious as well as benign apps, so as to provide meaningful information on both precision and recall.

Detection of Plagiarized Applications

The work of Potharaju *et al.* (92) intends to detect repackaged applications (which they refer to as plagiarized applications) containing malware, under different levels of obfuscation. The purpose of an attacker who plagiarizes an application is to take advantage of its popularity and collect sensitive information. To this end, the attacker starts with the download of the application and the recovery of its `.dex` file. He then adds his own bytecode in the application and repackages it into a new APK package.

To detect such applications, the authors devised the following three schemes :

- *Symbol-Coverage* The first scheme is applicable to non-obfuscated applications. The coverage of an application A_i by another application A is calculated as the number of classes and methods in A_i that also exist in A, divided by the total number of classes and methods in A_i. If the application A_i is highly covered by A (above a threshold), then it is considered the plagiarized version of A_i.
- *AST Distance* If the attacker has obfuscated the symbol table, meaning that the names of methods, classes, variables, and other identifiers have been changed, an alternate coverage method builds from the Abstract Sytax Tree (AST) of the app. The AST is a data structure that captures basic information about a method, including its number of parameters, the list of methods it calls, and static code metrics such as assignments, conditionals and loops, thus creating a fingerprint of the app. The authors then use Euclidean distance to compare the ASTs of two apps in order to detect possible

repackaged apps.

— *AST-Coverage* If the application is further obfuscated, possibly through the use of random methods with no added functionality, a final coverage metric, AST-Coverage, is employed. These methods proceed by constructing the AST for each method of every app in the market, as well as for the app of interest A. Comparison is then performed on a method by method basis. The app A is identified as a potential repackaging of another app A_i from the app store if A_i's coverage of A exceeds that of any other app, and exceeds a pre-set threshold as well.

Tests showed that AST-coverage outperformed the other methods, detecting all plagiarism instances from a set of actual malware incidents with 0.5% false positives, from a database of 7,600 apps.

Recommendation 3
The method described above, as with many others presented in this paper, depends on an external parameter —in this case a numerical threshold value. The favorable precision and recall of such approaches is therefore highly linked to the precise value given to these parameters, and this (manual) choice is often not discussed. From a methodological standpoint, there is a risk of overfitting parameters to the specific dataset under study.

NSDroid

Drawing on the intuition that malware families share a high level of code similarity, Liu et al. (73) propose a tool, called NSDroid, that aims to detect malware by detecting the similarity of apps with known malware thought an analysis of the apps' call graphs.

The tool first extracts the call graph from the apps using androgexf (129). NSDroid then further abstracts this information by creating a *signature* for each app. This signature is created as follows :

First, from the function graph, NSDroid creates a label for each method (each node of the function graph), that identifies which sensitive API calls are called by that function. This label records only sensitive API calls, and further records only the type of API calls called by the function, using a predefined list of 15 sensitive API call types. This information is thus recorded with a single bit, and each node is labeled with a vector for 15 bits. Finally, the label of each node is XORed with that of each of its neighbors (callers and callees), to create the signature for this node. It is this label that forms the basis for code similarity detection.

Classification is then performed on 4 different datasets of malware, totaling 32,190 apps, with three different classifiers, Random Forest, Decision Tree, and SVM, the latter of which showed optimal results. This scheme benefits in that it is highly efficient, performing it's analysis on 32,190 apps in little over 90s. The method is also highly effective, reaching 100 % accuracy, precision and recall for several malware families. On average, for all malware families, NSDroid exhibits accuracy precision and recall values of 0.959, 0.966 and 0.959, respectively.

DroidMOSS

Zhou *et al.* (146) sought to systematically detect and analyze repackaged apps. They implemented an application similarity measurement framework called DroidMOSS that applies a fuzzy hash technique to effectively locate and detect changes in an application's behavior. Unlike several of the methods seen in this section, it operates directly on the Dalvik bytecode, without requiring access to the source code.

DroidMOSS operates in three main steps. The first step is to extract the instruction set from each application, together with information about its author. These two features make it possible to identify each application in a unique way. The second step is to generate a fingerprint for each application, significantly condensing it into a much shorter sequence. Finally, based on the application fingerprint, the third step identifies the source of the applications, either from the official Android Market or third party markets, and measures the similarity between pairs of application from the same market in order to detect repackaged applications.

DroidMOSS relies upon the existence of the corresponding original applications in the data set. If the database used for testing is incomplete, (e.g., if it only includes free applications and does not include paid applications from the official Android Market, as is the case for several datasets used in academia), DroidMOSS may miss some repackaged applications. The prototype uses a white-list approach that may not detect possible malicious changes in advertising SDKs[4] or shared libraries.

Finally, DroidMoss's analysis is based on the totality of the code originating in every component of the app, as do several other mechanisms listed in this section. Android components are of 4 types : activities, services, content providers and broadcast receivers. While this may appear thorough, emerging research seems to indicate that malware developers prefer to inject malicious code in service components, which run in the background of the application (63). More research is needed to determine if a clustering method, such as the ones proposed in the paper mentioned above, can be made more effective by disregarding inputs that originate in other components.

4. "An advertising SDK is an extract of code provided to application publishers by a mobile advertising network. It helps developers integrate advertising ads into their applications" (18).

Recommendation 4
Drawing upon recent research indicating that malware is more likely to reside in the service components of repackaged apps, detection mechanisms should focus their attention on program behaviors that occur in these components.

1.2.2 METHODS BASED ON API CALLS AND PERMISSIONS

The second category of static approaches is concerned with the analysis of the permissions requested by the application, and the various API calls that occur in its source code.

DroidSieve

The DroidSieve method was proposed by Suarez-Tangil *et al.* (115). It examines several syntactical characteristics of the apps in order to detect and classify Android malware. These purely static features include the list of API calls occurring in the code, the permissions it requests, and the set of all application components. They collectively constitute the basis of an in-depth inspection of the application to identify discriminating characteristics. This data is then fed into the following classification algorithms :

— Extra Trees, an algorithm used to build a set of decision or regression trees that are not set according to the classic top-down procedure (49) ;
— support vector machines (SVM), a linear algorithm that solves the specific classification problem of determining the class to which an individual belongs among *two* possible choices (125) ;
— eXtreme Gradient Boost (XGBoost), a powerful and fast automatic learning library used for supervised learning problems (123).

DroidSieve was evaluated on over 100,000 benign and malicious apps, achieving a detection rate of 99.44%, with zero false positives. DroidSeive is also capable of classifying malware with high accuracy. However, since DroidSieve performs malware detection by looking for patterns in the app's code, it may not be robust against mimicry attacks, app cloning, or adware.

Analysis of Permissions and API Function Calls

Qiao *et al.* (93) proposed a malware detection method based on automated learning of the permissions and API function calls present in Android Apps. To begin with, this approach examines the `AndroidManifest.xml` file in order to obtain the set of permissions used by the application. However, the authors note that this may actually be an over-approximation of the permissions actually used by the app, since some applications request excess permissions. As a consequence, the authors decompile the `.dex` bytecode to Java source code, and create a list of API calls that require permissions, and that actually occur in the app's code. The persimissions used in the code, as well as the API used in the code, are then organized in feature vectors, and the classification proceeds using three different machine-learning algorithms : Support Vector Machines (*SVMs*) (125), Random Forest (22) and Artificial Neural Networks (*RNA*) (50).

Experimental results on a dataset containing 6260 applications show that detection based on the API method calls outperforms detection based on the permissions alone, with the former method achieving the highest accuracy (81.68%–94,41%) depending on the machine-learning algorithm used), but at the cost of a greater computational overhead.

Recommendation 5
The previous two methods relied principally on API calls, and other elements of behavior. However, a particular class of malware, namely Adware, exhibits only minimal differences between benign and malicious apps (63), while clones are semantically identical. It is thus important that detection methods specifically crafted to detect these types of malware be developed. Testing sets should also include some portions of adware and cloned apps.

DroidMat

Jie Wu *et al.* (134) proposed a system called DroidMat that draws upon multiple elements of static information, including permissions, intents (messaging objects that contain information about other components), and API calls to characterize the behavior of Android apps. With respect to API calls, their model includes not only the API calls themselves, but also the type of component (service, activity) in which the API is called.

From this data, DroidMat builds a features vector for each app, and applies several machine-learning algorithms to distinguish benign and malicious app. The authors found that a combination of K-Means (127) and k-nearest neighbors (KNN) (6) applies the K-means algorithm (127) to distinguish benign and with $k = 1$ to classify provides optimal results.

DroidMat uses API calls as features to determine which operations the application seeks to execute. This technique considers not only the API calls themselves, but also the type of component where these calls occur, since the same API used in different components may reflect different intentions on the part of the developer. However, for most Android malware families, DroidMat possesses only one sample of malware, a fact that limits DroidMat's ability to infer the behavior of the malware.

Interestingly, the authors stress that DroidMat is unable to detect a specific type of malware, namely malware that extracts the malicious payload from external sources at runtime, rather than preserve it in the application's code itself, a process called *dynamic loading*. This problem actually exists for all static analysis approaches ; since the payload is not present in the code, no static analysis can be expected to detect this class of malware. Another strategy often employed by malware developers, *reflection calls*, achieves the same goal. A reflection call is a Java feature that allows a program to examine and invoke an object's method at runtime. While this feature can be quite useful in some contexts, it makes it impossible to build the program's call graph statically.

Recommendation 6
Static analysis may be insufficient to detect malware in the presence of dynamic loading and reflection. Hence, it should be supplemented with a dynamic analysis, especially since, in a survey of malware found in Android apps, Khanmohammadi *et al.* (65) found that dynamic loading was frequently incorporated in the app during the repackaging process, possibly to hide the presence of malware.

Recommendation 7
For the same reason, it is important that the datasets used to test the effectiveness of malware detecting tools contain some portion of malware that exhibits reflection and dynamic loading, so as to provide a realistic dataset. In the latter case, since code is loaded from a distant location, particular care must be taken to ensure this code remains available when the app is used for testing purposes. Khanmohammadi *et al.* (65) studied Androzoo, a widely used dataset of Android malware and found that in most of the samples using dynamic loading, the network address was no longer available.

Detection Based on Risk Signals

In order to improve the existing detection mechanism based on permissions, Sarma *et al.* (106), developed an alarm system that takes into account both the permissions requested by the app, the category and sub-category of the app (as stated in the Google Store) as well as the permissions requested by other apps belonging to the same category. This allows users to make a more informed decision about the security trade-off made each time a new app is installed : if a permission requested by an app is also requested by most apps with similar functionality, it follows that the permission is probably essential for the desired functionality, and its presence in the Android manifest file does not unduly arouse suspicion. However, if the permissions requested by an app are unusual for apps in its category, the risk of installing the app is higher.

An interesting point raised by the authors is that Android deliberately attempts to limit the number of different permissions, in order to limit the mental burden on users who may not be familiar with the inner workings of the Android security architecture. However, a more granular set of permissions would actually improve the effectiveness of the approach under consideration, and lead to more meaningful alert messages. The authors have achieved a detection rate (recall) of 80.99% by applying a classification using SVMs on a dataset consisting of 158062 Android apps collected from the Android market and 121 malicious apps from the Contagio[5] malware dump repository.

Recommendation 8
Contagio is a public repository of Android malicious apps. However, this repository contains more apps than the 121 used in the paper, and the description provided in the paper does not allow the reader to exactly identify which are the 121 that have been retained. In order to improve reproducibility and future comparisons, experimental studies should provide minimal information on the apps being used (name, version, MD5 checksum) in some publicly available reference.

Taking a similar approach, Peng *et al.* (88) use a probabilistic model to assign a risk score to Android applications according to the permissions it requests and their category. Instead of a detection process that classifies each application as either malicious or benign, Peng *et al.* seek to provide an informative rating to the user which captures the likelihood of the app being malicious, with apps exhibiting a higher score if they are more likely to be malicious. Each user can then make an informed decision about the risk-return trade-off of installing the app.

The risk score is computed in such a way that the more permissions an app requests, the higher its score will be. Particularly sensitive permissions are more heavily weighted in the calculation. The risk score is computed using several models, one of which, namely Naive Bayes with Informative Priors, seems to perform optimally.

Since the user often has the ability to choose any one of several applications to perform a

5. http ://www.contagiodump.blogspot.com

given task, the ability to rank apps on security rather than flagging certain apps as completely malicious or benign is a highly actionable information to the user. The system could be made even more informative by taking into account the varied types of malware, and the different levels of damage each can make if it successfully infects the user (for instance, an adware is less damaging than a spyware). The authors also point to an incidental benefit of this fact : the scheme will encourage developers to reduce the number of permissions their apps request, thus reducing the attack surface of the end user's device.

Kirin

Enck *et al.* (41) proposed a system called Kirin, which examines the permissions requested by an app in order to determine if it meets a higher-level security policy. This provides users with a valuable added information when determining whether it is safe to run an app of unknown origin they have just installed.

Kirin first extracts the permissions from the manifest file. It then compares these permissions to nine rules, defined by the authors, that conservatively overestimate templates of undesirable security properties needed by several types of common malware. If the configuration fails to pass all rules, the installation program can reject the application ; otherwise, the user may decide to proceed with the installation anyway, if he considers the risk to be worthwhile.

Kirin was tested on 311 apps downloaded from the official Android market, ten of which failed to adhere to every rule. Of these, the authors believe that 5 apps implemented dangerous functionalities unnecessarily, while the other 5 apps operated within reasonable parameters.

Recommendation 9
Sarma et al. (106) observed that more granular permissions would have the undesirable side effect of making it more difficult for users to understand if the permissions requested by a given app are dangerous. However, if such a fine permission system were combined with a higher-level security policy, such as the one implemented by Kirin, this negative side effect would be mitigated.

DroidAPIMiner

Aafer *et al.* (2) proposed a method called *DroidAPIMiner* to extract Android malware features at the API level by focusing on critical API calls. The authors first examined a large number of malware, in order to understand the features that characterize malicious behavior. During the first phase, DroidAPI miner extracts from the app under consideration the API calls and their package-level information, as well as the requested permissions of the apps. Then, during the feature refinement phase, DroidAPI miner removes from this information the API calls that are exclusively invoked by third-party packages such as advertisement packages. The feature set is further reduced to include only those APIs whose support in the malware set is significantly higher than in the benign set. For those APIs which were frequent in both sets, a data flow analysis is performed to recover their parameter values. An API is included only if it invokes dangerous values.

DroidAPIMiner performs classification using four commonly used classification algorithms namely : ID5 DT, C4.5 DT, KNN and SVMs. Optimal results of 99% accuracy and $2,2\%$ false positive rate were obtained when using the KNN classifier.

1.2.3 OTHER METHODS

Finally, we list in a third category static methods that do not squarely fall into API or source code analysis.

DroidRanger

The DroidRanger tool (150) detects the characteristic behaviors present in malware from several malicious families. It relies on a crawler to collect Android applications from existing Android markets and stores them in a local repository. For each application collected, DroidRanger extracts the fundamental properties associated with each application (requested permissions, author information, etc.) and organizes them into a central database.

DroidRanger performs two distinct detection processes. The first, for known malware, is based on a permission-based behavioral footprint. The second, for previously unknown malware, is based on a heuristic analysis of the app's behavior, as reconstructed from the bytecode and the manifest file. Suspicious applications are then executed and monitored to verify if they actually display malicious behavior at runtime. If this is the case, the associated behavioral fingerprint will be extracted and included in the first detection process's database.

This study was tested on the most popular applications of the year 2011, and yielded positive results. However, DroidRanger only covers free applications and only five Android markets, with a false negative rate of 4.2%.

DREBIN

Arp *et al.* created DREBIN (15), a tool that performs malware detection on the results of a static analysis of the applications. DREBIN's feature set appears to be one of the most thorough of all the works we have surveyed. In all, they create 8 feature sets for each app, using data from the Android manifest file (including permissions, components and requested hardware), and from the decompiled `.dex` file (including selected API calls and network addresses). The entire feature set is constructed in linear time, without necessitating complex static analysis such as data flow analysis.

Detection is then performed using SVMs. In order to maintain a lightweight footprint on the end-users' device, training is not performed on the smartphone itself. Instead, the classifier is trained offline, and the only resulting model is passed to the user. In order to provide explanations for its results, DREBIN's classifier is trained not only to detect, but also to identify the features that lead to the application being flagged as malware. From these, DREBIN constructs a parametrized sentence that explain the reason of the verdict to the user.

DREBIN was tested using 131611 benign apps coming from the GooglePlay Store, as well as two other markets (one Chinese and one Russian), and 5560 malware samples from the Android Malware Genome Project (147). It obtained a detection rate of 93%, with only 1% of false-positives, outperforming several anti-virus software on the same dataset.

1.3 DYNAMIC ANALYSIS

Dynamic analysis is an alternative approach to malware detection, which requires running the program to study its behavior and its effects on its environment. Unlike static analysis, it is *late* in that it only detects a violation right at the moment when it is about to occur. It also suffers from coverage limitations, since it only considers a single execution, rather than all possible program executions.

As we did in the previous section, we organize dynamic tools in four broad categories, according to the element relied upon for detection. These four categories are :

1. methods that rely primarily on system calls ;
2. methods that rely on other system-level information, such as CPU usage or network communications ;
3. methods that rely upon user-space level information, such as API calls ;
4. other methods.

1.3.1 SYSTEM CALL MONITORING

The first category of works is related to the observation of system calls. We identify three main lines of work in this category.

Natural Language Processing

Xiao *et al.* (137) proposed a detection method based on processing system calls from an Android application. Drawing upon the *Long Short-Term Memory* model (LSTM (**?**)), a type of neural network model used in the processing of natural languages, they train two classifiers, one using sequences of system calls from benign applications, and the second, using sequences of system calls from malicious apps. The use of the LSTM model allows the classifiers to draw upon the complete history of the sequence up to a given call, as opposed to the commonly used *n*-grams that only consider subsequences of length *n*. In their model, a system call is considered as a "word", and a system call sequence as a natural language sentence. LSTM assigns a probability to the occurrence of a sentence (i.e. the sequence of system calls of the application being monitored) in both the valid and the malicious model. An execution is then pegged as being malicious if is it more likely to occur in the malicious model.

Testing the model under different conditions, including varying the lengths of system call sequences from 50 to 50000, the authors achieved an accuracy rate of $93,7\%$ with a false positive rate of $9,3\%$.

System call logs

Sanya *et al.* (29) proposed an approach to detect malicious behavior at runtime. They used a dataset of 66 benign and malicious apps. They first executed the apps in a controlled environment for a fixed period of time, and recorded the system call occurring during this time. After discarding the less statistically significant system calls, each app was associated with a Boolean vector that indicates if each of 18 more relevant system calls is present or absent during it's execution. This data is then fed to a machine learning algorithm. The authors used three learning methods : the Naive Bayes algorithm, the Random Forest algorithm and the stochastic descent gradient algorithm. Finally, they used this data set to classify an unknown application as malicious or benign. This approach had a detection rate of $95,5\%$ for malware detection with a false positive rate of 8%.

It should be noted that a malware could potentially evade this detection scheme if the malicious behavior does not occur during the training period, possibly because the malware detected it was being emulated, or possibly simply because it was programmed to only occur after a specific logical condition (such as a time bomb) was validated.

Canfora *et al.* (26) also relied upon system calls to perform malware detection. Their method draws upon the fact that malware tends to evolve through an iterative process of borrowing and modifying code from other malware. As a result, malware samples are likely to share common behavioral features. A similar idea underpins the static methods of Potharaju (92) who looked for similar code segments in multiple apps. A dynamic approach benefits from the fact that the same behavior can be encoded in a number of different but semantically identical ways.

The authors developed a method to automatically select, from the very large number of possible system call sequences, those that are most predictive for malware detection. Given the frequency of the sequences of the selected system calls, they classified the execution traces as malware or not. With this method, the authors obtained a detection accuracy of rate 97%, on a dataset containing 1,000 benign apps and 1,000 malicious ones. Interestingly, they collected multiple execution traces for each app, thus providing a better reflection to the varied possible behaviors that can be exhibited by each app.

Crowdroid

Iker *et al.* (25) have developed an application called Crowdroid, which draws upon the benefits of crowdsourcing to detect malware in repackaged apps. On the end-users' devices, Crowdroid uses the tracing tool Strace (available on most Linux distributions) to monitor system calls to the Linux kernel of running applications. This information is then sent to a centralized server. The latter creates a feature vector for each pair of user and application. This feature vector is a listing of the number of times each of Linux's 250 system calls is called, by a given application, as run by a given user. Clustering is then performed on this data using the k-means algorithm to differentiate applications that, while having the same name and identifier, exhibit differences in behaviors. Naturally, the more users are using Crowdroid, the more data will be provided to the server, and consequently, the more accurate and precise detection will be. Crowdroid was tested on 3 malware, one of which was written by the authors of the paper for this purpose, and achieved between 85% and 100% detection rates.

1.3.2 MONITORING OF SYSTEM-LEVEL BEHAVIORS

The second category of dynamic approaches focuses on system-level information other than system calls in order to detect malicious apps. Some of these approaches also include system calls in their analysis.

EnDroid

Feng *et al.* (47) proposed EnDroid, a malware detection system based on several types of dynamic behavior at the system level. EnDroid adopts a feature selection algorithm to eliminate irrelevant features and extract critical features from the behavior. EnDroid proceeds in two phases : the learning and the detection phase.

The learning phase consists of extracting the dynamic behavioral characteristics of a given application by monitoring input/output operations. The authors monitored ten types of application actions (cryptographic operations, network operation, file operation, information leaks, SMS messages sent, telephone calls, receiver actions, receiver startup, `.dex` class loading and system calls). Each of these functionalities is treated as a distinct feature, for the purpose of the creation of a feature vector. EnDroid ten takes as input the feature vectors generated by benign and malicious applications and trains a large number of basic classifiers. Based on the forecast probabilities of these basic classifiers for each application, it forms a final classification model by adopting a meta-classifier. This classification model is then transmitted to the detection phase.

In the subsequent detection phase, EnDroid extracts the dynamic behavioral characteristics of an unknown application and generates its feature vector. Based on this vector, the classification model is able to determine whether the application is benign or malicious. Experimental results show that this approach successfully detected $97,97\%$ of malware with $1,85\%$ of false positives.

Andromaly

Andromaly, (110), is an application that continuously monitors various system measures to detect suspicious activity by applying supervised anomaly detection techniques. Andromaly's architecture is composed of four main component groups, detailed below.

The *Main Service* component synchronizes feature collection, malware detection and the alerting process. The features upon which Andromaly relies include CPU consumption, the number of packets sent over the network, the number of running processes and battery usage, among other elements.

Feature Extractors communicate with various components of the Android structure, including the Linux kernel and the Application Framework layer, to collect feature values. These are then sent to the Main Service. *Processors* are anomaly detectors based on rules, knowledge and classifiers, derived from automatic learning methods. Processors and external components can thus be added, removed and replaced. Finally, a Graphical User Interface (GUI) provides the user with the means to configure the application settings.

Under this approach, the malware detector continuously monitors the different features and events of the system, and then applies standard Machine Learning classifiers to classify the collected observations as either healthy or malicious. In order to minimize false-positives, the authors suggest that an alarm only be raised if the anomalous behavior persists. However, as a result, the proposed approach would only be effective at detecting continuous long-lasting attacks, such as DoS attacks, and less effective for abrupt, instantaneous attacks. A natural solution, suggested by the authors, is thus to combine Andromaly with other malware detection mechanisms, such as static analysis.

Andromaly was trained using four custom made malicious apps, designed by the tool's creators. It was then tested on a dataset consisting of these 4 malicious applications and 40 benign apps (20 games and 20 tools) using several different detectors. In some cases, the accuracy reached 100% and the false positive rate 0%. Interestingly, the authors report that the system found it easier to distinguish benign applications from malware when the benign application was a game, rather than a tool. The authors hypothesised that this is because of the unique behavioral features of games. This is an interesting finding that should be investigated further.

Recommendation 10
Different classes of apps (tools, games, web browsers, social media apps etc.) exhibit different runtime behaviors, a fact that could be used for malware detection. For example, it would be easy to compel app developers to include a label identifying the purpose and general functioning of an app in natural language in the AndroidManifest.xml file. The label would then narrow the range of permissible behaviors for the app and the detection mechanisms would refer to this label when performing malware detection. For instance, a clustering algorithm would compare an app labeled as a game with other benign game apps to decide if the former behaves in an abnormal manner. Indeed, Sarma *et al.* (106) relied upon the app's category in the app store to create risk signals (see Section 1.2.2). The one-out-of-k access policy (39), an early access control policy for Java, was also based on a similar principle.

1.3.3 MONITORING OF USER-SPACE LEVEL BEHAVIORS

A third category of works uses information gathered at the user-space level to detect malicious applications. This typically includes call information at the API (rather than system) level.

RepassDroid

The tool RepassDroid (138) combines semantic and syntactic analysis to automatically detect malicious Android programs. RepassDroid analyzes the Android application by synthesizing the API used in the application as a semantic function and the essential permissions as a syntactic function. Then, it uses learning to automatically determine whether an application is benign or malicious.

RepassDroid's architecture proceeds from two main components :

— The feature extraction module. From a given Android application, the feature extraction module first generates a call graph of each application using FlowDroid[6]. Then, it extracts the application's features (APIs and permissions) from this graph to form feature vectors.

— The Classifier Module. The authors used the feature vectors to form the classification model using the Weka library[7]. Previously unknown applications are aggregated into the model after being classified as either benign or malicious.

RepassDroid was tested on a dataset containing 24288 apps, of which half were benign and the other half malicious. These apps were drawn from multiple sources including AndroZoo (4), Android Malgenome Project (148), VirusShare (126) and DREBIN (15). Using multiple classifiers, RepassDroid reached $97,7\%$ accuracy, outperforming 52 out of 57 detection tools to which it was compared using the same dataset.

Malware detection based on machine learning of dynamically generated data

Wen *et al.* (132) proposed a malware detection scheme for Android devices based on the SVM automatic learning classifier (support vector machine). Their system operates directly on the smartphone of the user, and is optimized for this purpose.

The tool is divided in two main modules. The *client* module contains a database of known infected app (identified by their MD5 hash) which is checked anytime a new app is downloaded. Thus, users are warned if they attempt to install an infected app. Otherwise, the app will be submitted to the server for further processing.

On the *server* module, the application's features are extracted in the feature extraction module using a combination of static and dynamic analysis. The static features include permissions, intents, uses-feature, application and API. Then, the app is run in a virtual environment, and monitored to obtain a second set of features, including CPU consumption, battery consump-

6. FlowDroid statically calculates the data flows in Android applications and Java programs. `https://github.com/secure-software-engineering/FlowDroid`

7. The WEKA project provides a complete collection of automatic learning algorithms and data prepossessing tools (52).

tion, the number of running processes and the number of messages as the dynamic features. The features are then sent to a feature selection module that filters out redundant features. Finally, the authors constructed a classification model using SVM and evaluated the Android application by classifying it as malware or benign.

This method is particularly innovative since it uniquely combines static and dynamic analysis to perform classification. Experimental results show that this system yields an accuracy rate of 95.2% and a false positive rate of up to 13.3%.

Recommendation 11
Any malware detection will either be applied by the app store's manager as apps are uploaded on the app store, or by the end user himself after he installs the app on his devices. This distinction often goes unmentioned in the documentation that proposes a given approach. This is unfortunate since the choice of where the app is applied bears considerable consequence.

A malware detection mechanism applied at the level of the app store must be able to scale to a considerable level — the Google store, for instance, currently has over 2.7 million apps. It is simply not possible to perform a pair-wise comparison between all apps in the store, as is sometimes suggested to detect clones. An initial clustering of apps may offer a path to minimize the number of comparisons required, and initial research has been performed in this direction (63). Furthermore, a detection mechanism applied at the store level is arguably more sensitive to concerns of false positives, as rejecting a benign app may expose the stores to legal consequences. The calculation of tolerating uncertainty with respect to whether or not an app is benign or malicious is obviously more significant for the user who puts his own data at risk when choosing to install an app.

On the other hand, any detection scheme applied by the user must be lightweight enough to execute itself with the limited computational capabilities of mobile devices. Such methods must also be sufficiently intuitive and user friendly to be usable by individuals who may not be tech-savvy. It is unfortunate that very few of the methods we surveyed included usability studies. It is also important to stress that software developers and technology experts are often poor judges of which mechanisms are intuitive or user-friendly for the general public.

Recommendation 12

Any detection mechanism that end-users is required to apply themselves will necessarily have to provide a cogent, easy to understand explanation of its verdict. This aspect is often overlooked, but it is not uncommon for users to disregard security warnings if the reasoning that led to the warnings is not explained to them. This problem is particularly salient in the context of machine learning or clustering, where explainability can be a major challenge. Research is also needed to understand how to best communicate the reason for which an app was flagged as potentially malicious to users who may lack the vocabulary commonly used by security professionals to express these ideas.

XManDroid

Bugiel *et al.* (24) proposed a security tool called XManDroid (eXtended Monitoring on Android), which dynamically analyzes application permission usage to detect and prevent privilege escalation attacks at runtime. This type of attack occurs when an application indirectly invokes another application's code, thus abusing that application's privileges. Because this attack type draws upon the cooperative effort of multiple apps, it can avoid detection by most of the other tools presented in this paper, since these evaluate the security profile of each app in isolation.

XManDroid monitors Inter-Component Communications (ICC), *i.e.* the communications that occur between different components, in order to detect possible privilege escalation attacks and determine if a given ICC call can potentially be part of an escalation attack, after consulting the device's policy. Each time a call occurs between two different components, XManDroid updates its internal states, which records all ICC calls that have previously taken place on the device, and determines if the pattern of ICC calls that have appeared so far is indicative of a privilege escalation attack. This determination is based according to a predefined policy that captures commonly observed attack patterns.

XManDroid was tested using a custom-made suite of 7 malware, all of which were successfully detected by XManDroid. A further test using 50 benign apps taken from the Android market showed that XManDroid exhibits a low false positive rate : 3% out of 3824 ICC calls were flagged as false positives.

Recommendation 13
In testing XManDroid, Bugiel *et al.* performed two separate tests : an initial test using a number of malicious apps to test for accuracy and a second test, using only benign apps to measure the false positive rate. Since XManDroid operates by observing the interaction between different apps, a test comprising both benign and malicious apps

could have yielded a more realistic measure. More generally, benchmarks should contain realistic proportions of both benign and malicious apps, if we expect the accuracy and false-positive rates measured in testing to reflect the actual performance of the tool in practice.

Recommendation 14
Likewise, it is important to stress that this mechanism targets a specific class of malware, namely those that perform privilege escalation attacks. Several of the mechanisms that we examined are similarly targeted to a specific class of malware, which allows such mechanisms to achieve higher detection rates. More research is needed to understand how multiple targeted mechanisms can be combined to achieve complete protection against all types of malware. For the same reason, it is important that the benchmarks used in testing security tools include a wide variety of malware, ideally in proportion similar to those found in the wild. The reader is referred to the work of Zhou *et al.* (149) for a thorough classification of Android malware by type.

1.3.4 OTHER METHODS

In this last category, we classify all methods that perform a dynamic observation of an app's behavior using other measurements than system calls or user-level information.

Paranoid Android

Portokalidis *et al.* (91) developed a security tool, *Paranoid Android* that simultaneously performs multiple attack detection techniques on remote servers hosting an exact replica of the user's device.

Paranoid Android distinguishes itself in that it's architecture is divided into a client side, executed on the user's device and an server side, operating on the cloud.

On the user's device, a tracer records all information necessary to accurately reproduce its execution. This information includes user input as well as events originating in the Kernel, such as system calls. The trace is then uploaded to the cloud via an encrypted channel, where a replica of the phone is executed on an emulator. On the cloud, a replayer receives the trace and replays the execution in the emulator. Inbound traffic is also recorded and stored in a proxy, which the replayer can access on demand.

Paranoid Android is compatible with several detection methods, including dynamic analysis, anti-viruses, memory scanners and system call anomaly detectors. The most interesting benefit of Paranoid Android is it's capacity to perform multiple detection mechanisms simultaneously, and an interesting avenue of research is to see how these mechanisms could be combined to achieve better detection or a lower false positive rate. Unfortunately, this tool exhibits a considerable overhead, and can reduce battery life by up to 30%.

Recommendation 15
Some malware samples use emulator detection systems to evade monitoring of code using an emulator (86). Emulator detection systems examine a selected set of features of the underlying device at run-time to ensure that the app is running in a real device and adapt the execution accordingly. Further research is needed to detect the presence of emulation detection code in apps, and pinpoint such apps as possible malware.

Policy Enforcement

In the "other" category, we also list a few works whose purpose is not to detect that an application is malicious, but rather aim at controlling the execution of the application to make sure that some security policies are being enforced.

To this end, Falcone *et al.* have created RV-Droid (**?**), a monitoring tool for Android applications, whose purpose is to monitor and enforce properties written in formal logic. The user may select these properties from a repository provided by RV-Droid, or write their own. The user may select a different property for each application. RV-Droid will then synthesize a monitor for this property, and weave it into the app's code using AspectJ, an aspect-oriented framework for Java. The authors test several common specification requirements and is shown to have only a small overhead —often below 1%.

Recommendation 16
All techniques, especially dynamic ones, require additional time and memory in order to perform their analysis. Yet, this work is one of the few that includes a discussion of performance impact; most often, experimental results are solely focused on recognition rate and false positives. Each work should minimally justify what is considered accep-

table performance or scalability, and provide experimental measurements indicating to what extent these requirements are met by the proposed approach.

Elish *et al.* (40) have applied a set of classification policies to these communicating applications in the existing Android malware collusion detection solution, in which two or more applications interact to perform malicious tasks. They showed that there are technical challenges associated with the classification of inter-component communication flows. Their results show that permission-based classification policies trigger a large number of false alarms in the pairs of applications that interact with each other.

1.4 DISCUSSION

Researchers have long realized that traditional malware detection techniques, such as signature-based anti-viruses, are inadequate to provide effective protection against new malware. Consequently, in recent years, several techniques and tools based on behavioral analysis (static or dynamic) have been at the core of malware identification. Table 1.5 summarizes the existing approaches surveyed in the previous two sections, and Table 1.2 gives a summary of the recommendations listed throughout the paper.

Open Challenges Several challenges arise when designing malware detection tools. First, the code that potentially contains malware may be obfuscated through any one of a number of different techniques, hindering program comprehension and static analysis. Furthermore, other apps may contain encrypted code, which is likewise unavailable to the detection mechanisms. This was the case for the malware *DroidKungFu1* which uses a different key for each program instance. Metamorphic malware can modify its code by rewriting itself with each infection. For example, it can alter the names of the methods and classes in the application. The TinyDroid technique (31) failed to detect this type of malware.

In addition, several tools (e.g. (115) and (30)) require access the source code of an application as the basis for analysis. This naturally limits the applicability of the tools since the source code of apps code is often unavailable. Other tools, such as DroidMat (134) and DroidRanger (150), build models using only a limited number of malware samples; indeed, the latter has only been tested on free applications. The absence (or the paucity) of malware in a detection method's training environment can affect its effectiveness. One of our principal recommendations is that malware be tested on realistic datasets, containing both benign and malicious apps, in proportion to their occurrence in the wild. Likewise, the datasets should include malware of different types (clones, adware, data-miners, etc.), since mechanisms can be quite effective against one type of malware, but less effective against others. It would be

1. Make the tools and techniques publicly available for reproducibility and ulterior comparison.
2. Test on datasets that contain both benign and malicious apps.
3. Document the influence of external parameter values on the detection procedure, and describe how such values are chosen.
4. Focus on behavior that resides in service components of repackaged apps.
5. Develop techniques specific to adware and include adware in testing datasets.
6. Address the issue of dynamic loading in the development of detection techniques.
7. Provide testing datasets that include dynamic loading, and make sure they refer to extant network locations.
8. Provide detailed information about the exact applications being included in a testing dataset.
9. Use fine-grained permissions whenever possible.
10. Correlate observed behaviors with the app's intended functionality in order to get a more precise definition of what is suspicious.
11. Clearly define when and where in an app's lifecycle the detection technique is expected to be applied.
12. Provide meaningful and understandable feedback to end users.
13. Create testing datasets with realistic proportions of both benign and malicious apps.
14. Include malware of various kinds in a testing dataset.
15. Address the issue of emulator detection when testing in a simulated environment.
16. Define scalability or performance requirements and measure them experimentally.

TABLEAU 1.2 – A summary of the recommendations listed in the paper.

worthwhile for security researchers to report a breakdown of the accuracy and precision levels of their detection mechanism by malware type, rather than an aggregate value for the entire testing set. Finally, the datasets should also include both free and paid apps, since the Android ecosystem contains both kinds, and since free and paid apps are likely to differ substantially with respect to the code features they contain.

Another aspect that is largely neglected in the current state is the emerging threat of *Malware collusion*. Malware collusion occurs when multiple apps, possibly from the same malicious developers, interact with each other to perform malicious operations. Since the malicious behavior is spread across multiple apps, current solutions, which assume that malware consists of a standalone malicious application, may be unable to detect it. Elish *et al.* (40) performed initial work on this problem, by creating a graph that captures the interaction between several apps running on the same device. Their data structure could form the basis of an effective detection procedure for this class of malware.

Testing datasets The fact that different projects use different datasets for testing makes it challenging to compare the effectiveness, accuracy and false positive rates of different mechanisms on an equal footing. For this reason, the values provided in Table 1.5 must be interpreted with caution. We recommend that researchers either make their datasets available, or make use of one of the already publicly available datasets of Android apps, such a Androzoo or Drebin. To provide even greater replicability, security researchers could make the entire evaluation process available. Tools such as LabPal (53) allow researchers to bundle an entire experimental setup, including data and code, into single, runnable JAR file, making it easy for anybody to download and re-run, or even alter the experiments.

In table 1.3, we summarize the testing datasets used in the papers we surveyed. In two cases, (29) and (91), the testing set was unavailable, hindering reproducibility of the experiment. In all other cases, we report the year of publication of the dataset, it's creator, it's size, whether or not the dataset is updated regularly and whether or not it is publicly available . We also indicate whether the database only contains malicious apps (M), benign apps (B) or both (M/B).

The datasets used in the papers are as follows :

— Google play : the official app store, operated and developed by Google. It allows users to browse and download applications developed to run on the Android operating system.
— Drebin dataset : the dataset contains 5 560 malware samples from 179 different malware families. The samples have been collected in the period between August 2010 to October 2012.
— Genome dataset : The Genome repository contains 1226 malware apps categorized in 49 families of malware. This repository includes malwares dating back from 2012.
— AndroZoo dataset : AndroZoo is one of the largest datasets of Android apps. It was

TABLEAU 1.3 – The datasets used in the papers we surveyed

Database	Type	Creator	Date	Size	Updated	Available
Google Play (51)	B	Google LLC	2008	2,9M	yes	yes
Drebin Dataset (15)	M	Arp et al.	2012	5,560	no	yes
Malware Genome Project (148)	M	Zhou et al.	2012	1,200	no	no
Androzoo (4)	M/B	Li et al.	2016	10,528,559	yes	yes
Contagio Malware Dump (87)	M	Mila Parkour	2008	11,960	yes	yes
VirusShare (126)	M	Cylance Inc	2011	34,564,348	yes	yes
McAfee (115)	M/B	McAfee Inc	2012	21,330	yes	–
AppChina (143)	**B**	STC Bing search technology centre	2010	600,000	yes	yes
PRAGuard (75)	M	Univ. of Cagliari Italy	2015	$> 10,479$	no	yes
Gfan (144)	B	Business unit	2007	$> 100,000$	yes	yes

collected from multiple sources, including from the official Google Play app market. AndroZoo currently contains more than 5 million different APKs. Each app is scanned by at least ten different anti-virus products and the results of these scans are reported in the dataset.

— Contagio : Contagio is a collection of the malware samples, threats, observations, and analyses.
— VirusShare : VirusShare is a repository of malware samples hosted and maintained by Corves Forensics. The firm also provides commercial services to support the specific needs of larger organizations including enhanced API access, data feeds, and specialized searches of VirusShare's data.
— Mcafee : This dataset contains 8 041 benign apps and 13 289 malware. The apps were collected between from 2012 to 2016 and use advanced coding technique.
— Appchina : AppChina is a Chinese app that allows Android users outside of China to download Chinese apps. It allows users to find free applications and games. In 2014 it contains more than 600 000 Android apps. It has more than 30 million users and considered like the most popular Android app store in China.
— PRAGuard : The dataset contains 10479 malware samples, obtained by obfuscating the MalGenome and Contagio datasets with seven different obfuscation techniques.
— Gfan : Gfan is one of the largest Android app stores in China. It contains more than 100 000 android apps. Only members are allowed to download apps from Gfan.

Regularly updating the benchmarks will help counter the *concept drift* problem whereby a predictive model becomes less and less accurate as time passes, because the features upon which it relies have become outdated or obsolete. This problem is particularly salient for security models due to the highly dynamic nature of malware. In this respect also, tools such as LabPal can help security researchers stay one step ahead of malware developers, by making it easy to re-test security tools with newer datasets.

When using dynamic detection, the overhead processing costs are higher since they contribute directly to the deterioration of process execution time. This can have serious consequences, especially for critical real-time processes. Detecting malware while it is running is a challenge, as any delay could lead to an infected system, and perhaps even to an unrecoverable situation. Another important research challenge is to ensure that the detection process is sufficiently lightweight to take place on the end-user's device. Furthermore, dynamic detection only relates to behavior that is ongoing at the time of analysis and thus has a more limited coverage, since it only explores one execution path at a time.

The machine learning (ML) algorithms A variety of ML algorithms have been used in developing malware detection frameworks. Figure 1.1 indicates the number of publications that make use each ML present in the literature, distinguishing between static dynamic and hybrid methods. The x-axis represents the number of publications and the y-axis represents the ML used to classify the applications. We can see that the ML algorithm the most used for static techniques is the SVM (Support Vector Machines) in seven publications, followed by the KNN with three publications. However, for dynamic techniques, we find the Naive Bayes, Decision Tree and Random Forest algorithms in three publications. The only algorithm used in hybrid methods is the SVM. A few studies have used other algorithms such as : Histogram, Logistic Regression, LSTM, etc.

The choice of the algorithm depends on the type of data to be processed and even the number of samples. Table 1.4 presents the advantages and disadvantages of the top 5 algorithms used in the articles we studied.

Comparison between methods The detection techniques proposed differ with respect to their respective false alarm rate. A malware detection system should ideally have a low false alarm rate, otherwise it can block the execution of valuable services that the user requires. False alarms that are too frequent can also lead a user to turn-off or disregard his security tools, exposing him to further danger. Tools and techniques that were developed using particularly large test datasets that contain both benign and malicious apps seem to exhibit more favorable results. In particular, EnDroid (47) and (88) both exhibit distinctly low false positive rates of 1.85% and 0.58% respectively, in part due to the large number of applications used for the construction of the detection method.

As can be seen in Table 1.5, the detection process (permissions, API calls, opcodes, etc...) is founded on a multiplicity of features. Given the fact that the studies conducted so far do not use a consistent dataset of malware and benign apps, it is difficult to affirm with confidence

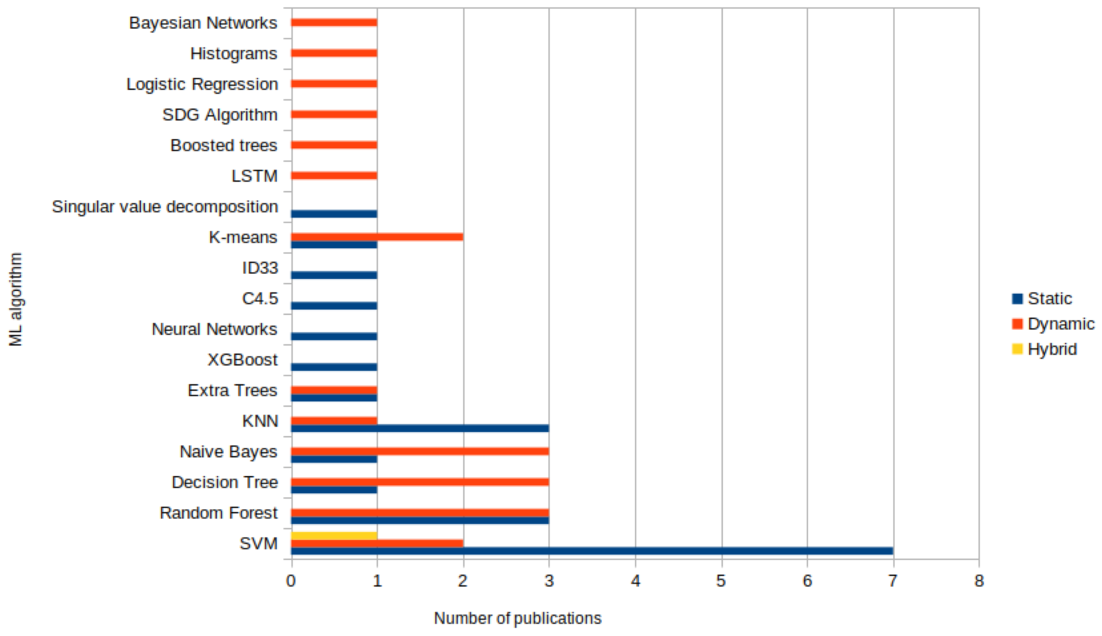


FIGURE 1.1 – ML used by Publications

that a given feature is more informative than another for the purpose of malware detection. Nonetheless, it can be intuitively inferred that using multiple features would improve the accuracy and precision of the detection process. Indeed, the results reported by a study that performs the same experiment with different feature sets (134) point in that direction. In that study, the same classification process was repeated four times : first by considering only the set of API calls and other statically available information obtained from the AndroidManifest file, then secondly by adding intents, a third time by adding permissions, and finally by adding both intents and permissions. The more complete feature sets yielded a more precise detection (though the second feature set yielded a better recall). This could mean that different classes of malware are sensitive to different features. Indeed, as mentioned earlier in this paper, some classes of malware, such as adware, do not require additional permissions to run.

Some features are used in both static and dynamic detection mechanisms. For example, API calls occur in 3 “static” and 3 “dynamic” entries of table1.5. It is important to stress in this regard that these features express the fundamentally different nature of the information involved. Statically obtained API calls are limited to the list of API calls existing in the code. The fact that some of these calls may be present in the code, but not called in a given execution, and the fact that some calls can be obfuscated through the use of reflection, means that a statically computed set of API calls is necessarily an approximation. On the other hand, dynamic analysis not only provides a precise listing of the API calls occurring during a given execution, but also informs about the number of occurrences of each call and their relative

TABLEAU 1.4 – Advantages and disadvantages of the top five ML algorithms used in literature

ML	Advantages	Disadvantages
SVM	- Its high prediction accuracy. - Works well on smaller data sets.	- Not suitable for larger datasets. - Less efficient on datasets containing noise.
Random Forest	- Flexible and easy to use. - Handle large data sets with higher dimensionality.	- The large number of trees can make the algorithm too slow. -It is difficult to interpret.
Decision Tree	- Simple and fast to use. - Handles large data. - Support incremental learning.	- It requires a long training time. - May have a more complex representation for some concepts.
Naive Bayes	- Easy to implement. - It is very fast. - Handles large volumes of data.	- Less accurate compare to other classifier.
KNN	- Simple to interpret. - Give a high accuracy. - Robust to noisy training data.	- Can be slow with large data. - Need high memory.

ordering. Naturally, one would expect that this wealth of information would result in a more accurate and precise detection process, and some of the results reported in table 1.5 do indicate that such is indeed the case, even though, as mentioned above, the paucity of controlled studies using the same dataset makes it difficult to reach definitive conclusions.

Another aspect that hinders comparisons between different methods is the fact that few papers systematically report values for false positives, accuracy, precision and recall. This is another reason for which we advocate the use of experiment management tools such as Labpal. If an experiment using such a tool is made available, it is a simple task for another researcher to modify it in order to measure a different metric.

The Human Element An important area of research that is often neglected is the human element of security. Rarely do the developers of security tools perform usability testing to ensure that end-users will be able to effectively utilize the tools they develop. An issue of particular concern is the manner in which the verdict of the detection process is communicated to the end-user, since they may ignore a security warning if they do not understand the reasoning that underlies it. This is of particular concern for methods based upon static analysis and clustering, as the explanation why an app was flagged as malicious can be difficult to articulate. More research is needed to determine how craft convincing and intelligible security alert that allows even the less tech savvy users to understand *why* app was flagged.

In this regard, it is also useful to recall that Schneier observed that users are more likely to dismiss or underestimate a given risk if the risk also confers to them some benefits. Likewise they are likely to underestimate a risk that arises from a course of action that they have chosen to undertake (109). In the context of Android security, these observations indicate that users will underestimate the risk associated with installing apps of unknown origin, since they choose to do, and presumably do so because they seek to make use of the functionalities provided by the app.

Static vs Dynamic Detection Overall, most of the techniques presented in the Sections 1.2 and 1.3 share some weaknesses. At the database level, researchers often used a limited number of applications in order to build a model of malicious and benign behavior, as few as five applications in one case.

Other malware detection tools are designed to target a specific type of malware to the detriment of other categories, as for instance, Andromaly (110) which does not detect instant attacks. Malware that deliberately reproduce a set of features similar to those produced by legitimate applications, pose a significant challenge for security researchers. For example, if a dynamic detection scheme monitors the configuration of API calls made by processes, a malicious process may attempt to scatter its own API calls into sets of benign API calls to avoid detection, (111). DroidSielver (115) for example missed the detection of such mimicry attacks.

As mentioned above, static and dynamic approaches can be seen as having mirrored advantages and drawbacks, the former being early, approximate and applied to the entire program, while the latter is late, precise and specific to a given execution. It is thus natural to consider combining both static and dynamic analysis in hybrid detection systems, as several authors have raised this possibility.

Static and dynamic analysis could work in tandem to perform a more accurate and effective detection mechanism. In addition, such a hybrid approach could potentially yield the following benefits :

- Static analysis can help reduce the overhead incurred by dynamic analysis, by ruling out certain apps, or even certain components of apps as safe, so that only potentially malicious code is monitored.
- Along the same line, dynamic analysis can be used to reduce the rate of false positives associated with static analysis : if the evaluation of an app yields a borderline score between benign and malicious, it can be allowed to run, but in a highly monitored sandbox environment.
- A hybrid method could also aid in addressing the usability problem highlighted above.

For instance, a monitor could rely upon the result of a static analysis to paint a more complete picture of a program's behavior. Then, if the monitor aborts the execution because it is potentially malicious, it could draw upon it's knowledge of the target program's probable subsequent behavior to offer a more complete explanation to the user.

1.5 CONCLUSION

In this paper, we survey malware detection methods for Android, focusing on the advantages and drawbacks of each and made recommendations for future research on the topic.

Despite the fact that a large number of solutions that have been proposed, several challenges remains to be addressed, especially because of the rapidly evolving nature of malware. We cite difficulties related to code obfuscation, the unavailability of source code and the emerging problem of malware collusion as problems that require particular attention in the near future. From our analysis of the papers we surveyed, we drew 15 recommendations that we believe will enable researchers to develop more effective malware detection tools. We especially recommend that researchers make available the testing datasets that they used, as well as the experiments they perform. Indeed, we found it particularly difficult to compare detection methods since they often used different testing datasets, are reported different types of values (accuracy, precision, recall etc.) as result.

We also argue that static and dynamic analysis may be combined in order to develop more effective enforcement mechanisms. The recommendations that we offer may help guide future research and address these challenges.

APPENDIX : RECAPITULATIVE TABLE

We provide a recapitulative table of every malware detection method we studied. For each method or tool, we provide :

— the name of the tool and its reference ;
— its year of publication ;
— the classification method used : static (S), dynamic (D) or hybrid (H) ;
— the rate of false positives (FP), accuracy (A) , precision (P) and recall (R), when provided ; for the sake of space, all values have been truncated to integer percentages, with the usual definition of these terms, namely :
accuracy = $\frac{tp+tn}{tp+fp+fp+fn}$
precision= $\frac{tp}{tp+fp}$

recall (detection rate)= $\frac{tp}{tp+fn}$
rate of false positives= $\frac{fp}{fp+tn}$;

— the size (Nb.) and origin of the testing sets ;
— the technique used to analyze the application ;
— for methods based on machine learning, the type of features used, the machine learning algorithms employed and indicate any pre-processing performed on the data ;

We used the values for accuracy, precision and false positive rate as reported, and only included these values when they were directly reported in the paper. We deliberately avoided *inferring* these values from other experimental data in the paper, out of an abundance of caution. In a few cases, the rate of false positives is not explicitly mentioned, but the paper claims it is "low", which has been indicated by an L in the corresponding table cell.

TABLEAU 1.5 – Recapitulative table

Tool	Year	Type	Rate				Database		Technique	Features	M. Learning
			FP	A	P	R	Nb.	From			
NSDroid (73)	2020	S	$< 1\%$	95%	96%	95%	32190	Drebin, (131), (105), (45)	Calculate the similarity between apps	Function call graphs	SVM, Random Forest, Decision Tree
TinyDroid (31)	2018	S	5%	95%	92%	95%	2400	Google Play, Drebin dataset	Disassembled APK file into smali codes	Opcodes	Random Forest, Naive Bayes, SVM, kNN
DroidSieve (115)	2017	S	L	99%	99%	99%	100000	Malgenome Project, Drebin dataset, PRAGuard dataset, McAfee Goodware	Static analysis of resources	Metadata of the app, syntactic features	Extra Trees, SVM, Random Forests, XGBoost
Qiao et al. (93)	2016	S	L	94%	–	–	6260	Google Play, Malgenome Project	Permissions and API calls	Source code and resource files	SVMs, Random Forest, Neural Networks
Chen et al. (30)	2015	S	–	96%	95%	94%	1170	AppChina, Malgenome Project	Clone detection	Decompiled code	–
DREBIN (15)	2014	S	1%	93%	–	90%	132171	Google Play, Malgenome Project	Clustering of malware	Perm., API calls, components, etc.	SVM
DroidAPI-Miner (2)	2013	S	2%	99%	–	–	20000	Google Play, McAfee, Malgenome Project	Extract a malware sig. from perm. and API calls	API calls, perm.	KNN, SVM, C4.5, ID3 (94)

TABLEAU 1.5 – Recapitulative table

Tool	Year	Type	Rate				Database		Technique	Features	M. Learning
			FP	A	P	R	Nb.	From			
DroidMat (134)	2012	S	–	97%	96%	87%	1738	Google Play, Contagio mobile	Clustering of benign and malicious apps according to static features	Perm., components (Activity, Service, Receiver), Intent message, API	kNN, K-means, Singular Value Decomposition
Potharaju et al. (92)	2012	S	$< 1\%$	–	–	–	7600	Google Play	Repackaging detection through detecting code reuse	Source code	–
DroidMOSS (146)	2012	S	–	–	–	–	2400	Free apps	Similarity measure	Dalvik bytecode	–
DroidRanger (150)	2012	S	–	–	–	–	204040	Google Play, eoeMarket, alcatelclub, gfan, mmoovv (free apps only)	Behavioral footprint based on perm.	Permissions	–
Sarma et al. (106)	2012	S	–	—	–	80%	158183	Google Play, Contagio Malware Dump	attribute risk levels to apps according to perm.	Permissions	SVM
Kirin (41)	2009	S	–	–	–	–	311	Google Play	verify perm. upon installation	Permissions	–
Xiao et al. (137)	2019	D	9%	93%	91%	96%	7130	Google Play, Drebin dataset	System call trace	System calls	LSTM (Long Short-Term Memory)

TABLEAU 1.5 – Recapitulative table

Tool	Year	Type	Rate				Database		Technique	Features	M. Learning
			FP	A	P	R	Nb.	From			
RepassDroid (138)	2018	D	< 1%	97%	99%	96%	24288	Google Play, Androzoo, Malgenome Project, VirusShare, Drebin dataset	Combines API calls and semantic information	API, permissions	Decision Tree, Random Forest, SVM, kNN, Naive Bayes
EnDroid (47)	2018	D	1%	97%	95%	97%	14019	Google Play, AndroZoo, Drebin dataset	classification based on system level informaton	System calls input, output (SMS, calls, network operation)	Decision Tree, SVM, Extra Trees, Random Forest, Boosted Trees
Sanya et al. (29)	2017	D	8%	–	95%	95%	66	–	runtime monitoring in a controled environment to construct a features vector of relevant system calls	System call trace	Naive Bayes, Random Forest, Stochastic Descent Gradient Algorithm
Wen et al. (132)	2017	H	13%	95%	–	–	2000	Google Play, Drebin dataset, Malgenome Project	monitoring in a virtual environment	Perm., intentions, API, Intent, Precesses, battery usage	SVM
Andromaly (110)	2012	D	12%	87%	–	–	44	Google Play	continuously monitors multiple system elements	88 functionalities (related to messaging, call phone, API)	*k*-Means, Logistic Regression, Histograms, Decision Tree, Bayesian Networks, Naïve Bayes

TABLEAU 1.5 – Recapitulative table

Tool	Year	Type	Rate				Database		Technique	Features	M. Learning
			FP	A	P	R	Nb.	From			
Crowdroid (25)	2011	D	–	100%	–	100%	5	Google Play	Sytem call monitoring	System calls	k-means
XManDroid (24)	2011	D	3%	–	–	–	50	Google Play	Monitoring of perm. usage	Perm., Reference Monitor, Decision Maker, System View, System Policy and Decisions	–
Paranoid Android (91)	2010	D	–	–	–	–	–	–	Simulates multiple attacks on a remote server running a replica	system calls trace	–

CHAPTER 2

ÉTAT DE L'ACCEPTATION : VERSION FINALE PUBLIÉE

CONFERENCE : 2021 28TH ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE (APSEC)

DOI :10.1109/APSEC53868.2021.00012

CITATIONS : 2 FOIS

TITRE EN FRANÇAIS

Classification comportementale des applications Android à l'aide d'appels système

RÉSUMÉ

L'augmentation exponentielle du nombre d'applications Android sur le marché s'est accompagnée d'une croissance correspondante des applications malveillantes. Le risque de reconditionnement d'applications, un processus par lequel les cybercriminels téléchargent, modifient et republient une application qui existe déjà sur le magasin avec l'ajout de code malveillant, est particulièrement préoccupant. La détection dynamique dans les traces d'appels système, fondée sur des modèles d'apprentissage automatique, est apparue comme une solution prometteuse. Dans cet article, nous introduisons un nouveau processus d'abstraction et démontrons qu'il améliore le processus de classification en reproduisant plusieurs techniques de détection de logiciels malveillants de la littérature. Nous proposons en outre une nouvelle méthode de classification, reposée sur notre observation selon laquelle les logiciels malveillants déclenchent des appels système spécifiques à des moments différents des programmes bénins. Nous mettons en outre notre base de données à la disposition des futurs chercheurs.

MOTS CLÉS

Android, Appels système, Classification, Sécurité.

CHAPTER 2

BEHAVIORAL CLASSIFICATION OF ANDROID APPLICATIONS USING SYSTEM CALLS

Asma Razgallah*, Raphaël Khoury*
* Department of Computer Science and Mathematics, Université du Québec à Chicoutimi, Canada

The exponential growth in the number of Android applications on the market has been matching with a corresponding growth in malicious application. Of particular concern is the risk of application *repackaging*, a process by which cybercriminals downloads, modifies and republishes an application that already exists on the store with the addition of malicious code. Dynamic detection in system call traces, based on machine learning models has emerged as a promising solution. In this paper, we introduce a novel abstraction process, and demonstrate that it improves the classification process by replicating multiples malware detection techniques from the literature. We further propose a novel classification method, based on our observation that malware triggers specific system calls at different points than benign programs. We further make our dataset available for future researchers.

Index terms : Android, system calls, classification, security

2.1 INTRODUCTION

Security concerns remain omnipresent when downloading Android applications (apps), despite the presence of security checks in the app stores, from which users obtain these apps. This is in part because app stores often contains benign and malicious variants of the same apps. Often an app will be downloaded from a store, decompiled, manipulated to include some form of malware, and then recompiled and uploaded to the app store, a process termed *repackaging*.

Detection mechanisms performed by the app store at the time an app is being loaded have seen limited success at eradicating this threat, in part because these methods are largely based on static code analysis. However, the code modifications performed as part of the repackaging process are often comparatively small, making static detection difficult. The problem is made more difficult by the presence of code obfuscations and encryption, which can render static analysis ineffective. Repackage app often require the same permissions as their original counterpart, which hinder permission-based malware detection schemes.

Furthermore, users often cannot distinguish between legitimate apps and their repackaged counterparts containing malware, when both versions of the same application are found on the same app stores.

Dynamic analysis thus remains as an essential last line of defence. It consists in runtime monitoring of the apps's behavior, in order to detect and prevent malicious behavior. This is a topic of active research (98), and several techniques based on machine learning have been proposed to automatically detect malicious behavior.

While, monitoring may take place at any execution layer (user-space, kernel, etc.), we have chosen to analyze Android applications at its lowest level, by studying the interaction between the application and the system in real time, since this strategy makes it most likely to detect attacks static analysis failed to uncover.

In this paper, we perform multiple experiments on malware detection in system calls of Android traces using a new dataset of system call traces from Android apps. We replicate multiple previous experiments from the scientific literature, using a single dataset, which allows comparison of the methods on an equal footing. Furthermore, we present a novel trace abstraction process, which is performed prior to the classification and show experimentally that using this abstraction improves the effective of the classifiers, and propose a new trace classification method derived from the analysis of the traces in our dataset.

This paper makes the following contributions :
(1) We present a new dataset, TwinDroid, which we make available dataset of Android traces. TwinDroid is composed in large part of traces from pairs of application, one benign, one malicious, but identical but for inclusion of the malware in the latter, which makes it ideal for research on dynamic malware detection. Our dataset is publicly availble on the author's repository [1]. The dataset currently contains 400 traces from 151 different apps and is being expanded.
(2) We replicate previous studies on dynamic detection using this dataset, allowing the comparison between several different malware detection on equal footing.
(3) We show that the inclusion of a trace abstraction step prior of performing automated detection yields meaningful improvements to the classification process.
(4) Drawing upon an inspection of our dataset, we propose a novel malware detection strategy and show that it compares favorably existing strategies present in the literature.

1. *https://github.com/AsmaLif/TwinDroid-dataset*

The remainder of this paper is organized as follows : in Section 2.2 we present our dataset TwinDroid and our abstraction strategies. Sections 2.3, 2.4 and 2.5 report the replication of three experiments with their classification results. We tested a new experiment on the pairs of traces in 2.6. Concluding remarks are given in Section 2.8.

2.2 DATASET CREATION

2.2.1 DATA COLLECTION

In order to obtain a benchmark of applications to be tested, we selected 151 applications between October and November 2020. These applications were installed and executed on a Google Nexus 5 mobile running Android 4.1 of API 16.

We first selected 92 infected applications [2] from the *Drebin Dataset*. The applications vary as to their size and purpose, and include a cross-section of app types including games, web browse, calendars, etc. A breakdown of the app types in our dataset is given in Table 2.2. We also selected apps that are infected with a variety of different malware families. Table 2.1 lists the malware families present in our dataset and indicates the number of apps infected by each malware family. As can seen we include a wide variety of common malware families. Finally, we endeavored to select infected version of the most popular Android apps, as well as apps that have widely repackaged by cybercrimianals. In addition, the apps we have selected are also drawn from a large variety of app types, as shown in Table 2.2.

We then obtained the benign applications from official Android application market (*Google Play*) (51) or from APK Pure (14). In the creation of our database, we excluded any application that meet one of the following conditions :

— the application is not free ;
— it does not run on our emulator because of version incompatibility or other reasons ;
— another version of the same app was already in our dataset. For example, the *Battery Super Charger* application which repeats more than ten times in *Drebin Dataset*, we keep only one copy.

We obtained benign pairs for 22 of the 92 malicious apps, and additionally, we enriched the dataset with 39 addition benign versions, which do not have a matched malicious version.

2.2.2 TRACING PROCESS

We installed the applications on the android emulator using Genymotion (1). We then simulated a real human using of the application with the Monkey tool (12) which simulates the interaction

2. availableat: https: // www. kaggle. com/ razgallah/ apps-base

Malware family	**Apps**	**Malware family**	**Apps**
Raden	2	DroidSheep	1
Opfake	7	lmlog	1
Plankton	24	SMSreg	1
DroidDream	7	GinMaster	2
Copycat	1	Adrd	1
FakeDoc	2	FakePlayer	1
Xsider	1	TrojanSMSDenofow	1
FoCobers	1	FakeRun	7
DroidKungFu	6	Yzhc	1
Glodream	2	Boxer	1
GingerMaster	1	Moghava	1
FakeRegSMS	2	SpyHasb	1
Dialer	1	Fakengry	1
FakeInstaller	6	Stealer	1
Iconosys	3	ExploitLinuxLotoor	1
SendPay	1	Adrd	1
Fidall	1	Fatakr	1

TABLEAU 2.1 – Summary of the Malware families used in this study

App category	**Apps**	
	Benign	Infected
Vehicle	2	1
Communication	9	15
Kids	2	2
Education	1	2
Food	3	1
Games	13	34
Books	3	1
Medicine	1	1
Weather	1	-
Music	8	11
Tools	4	10
Pictures	8	6
Productivity	3	4
Lifestyle	1	4

TABLEAU 2.2 – Summary of the app's categories used in this study

```
1605918097.832207 mprotect( 0x9f659000  ,  8192,
PROT_READ|PROT_WRITE) = 0
1605918097.832361 getpid()              = 4263
1605918097.832417 gettid()              = 4263
1605918097.832469 clock_gettime(CLOCK_MONOTONIC,
{8280, 454895458}) = 0
1605918097.832686 clock_gettime(CLOCK_MONOTONIC,
{8280, 455112943}) = 0
1605918097.832757 getpid()              = 4263
1605918097.832810 gettid()              = 4263
1605918097.832861 clock_gettime( CLOCK_MONOTONIC,
{8280, 455287104}) = 0
1605918097.833042 clock_gettime(CLOCK_MONOTONIC,
{8280, 455470088}) = 0
1605918097.833128 getpid()              = 4263
1605918097.833230 gettid()              = 4263
1605918097.833289 clock_gettime( CLOCK_MONOTONIC,
{8280, 455716680}) = 0
```

FIGURE 2.1 – Sample of an application trace

of the user in an automatic manner. This tool can be used to generate pseudo random events such as clicks which represents the number of random events which we want to generate. The number of clicks via Monkey used in our study ranged from 500 to 1000 clicks in order to maximize the functionality coverage of an application. We recorded traces of system calls via the Linux Strace tool (Android). We obtained a total of 400 traces (200 benign and 200 infected), running each app between 1 to 4 times.

Strace intercepts and records the name of each system call that occurs during the execution of the program, alongside with its arguments and the return value. A negative return value usually indicates the occurrence of an error while the return value zero generally indicates that the system call completed successfully. The example in Figure 3.3 shows the trace of the execution of a popular game app. Each line of this trace contains the system call along with its parameters and return value, a timestamp and the system call's execution time.

2.2.3 ABSTRACTION PROCESS

The large volume of information contained in these traces can made detection arduous. To streamline the process, we adopt the following four step abstraction process.

First, we consider only the system call names in the trace, thus ignoring other information contained in trace such as the execution time, and parameters and return values of each system call. This step is commonly made in the context of system call trace analysis, and every one of the experiments we replicate in the next sections also performs this abstraction. The subsequent three steps are original to our study.

Second, after studying the different types and categories of system calls (62), (80), we deduced that several system calls are inconsequential to a proper characterization of the program's behavior. This includes, for instance, system calls used to transfer information between the application and the operating system or system calls related to memory management. These calls are : *mmap, mmap2, munmap, mremap, mlock, munlock, mlockall, munlochkall, set_mempolicy, brk, sbrk, mprotect, modify_ldr, futex, clock_gettime, clock_nanosleep, getcpu, getpagesize, olduname, gettimeofday, timerfd_settime*. These system calls tend to occur multiple times throughout the execution of both benign and malicious traces. By analogy to natural language, we can think of these system calls as stopwords or articles, that can encumber a semantic analysis of the text. The second step of the abstraction process is to elide these calls from the trace.

Thirdly, many system calls have similar or overlapping features (9), (42), (78), and can be used interchangeably to implement the same functionalities. For example, the system call *fstat* has almost the same functionality as *asstat*, except that the former takes a file descriptor *fd* as input while the latter takes a path *asstat*. To compare traces from different systems on equal footing, we thus defined a list of semantic equivalencies between system calls, shown in Table 2.3. Treating such system calls as distinct would artificially heighten the differences between program that may in practice be quite similar. Thus, an important part of the abstraction process is to uniformize equivalent system call names.

Finally, unsuccessful system calls have no impact on application's behavior and can safely be removed from the trace as part of the abstraction process.

We posit that by eliding extraneous information, this abstraction process allows for a more precise classification of benign and malicious traces. In the next sections, we demonstrate this by replicating 4 studies on malware trace classification, and showing that the use of our abstraction improves the results of the classification process.

The abstraction process also considerably reduces the size of the traces, which in turn has benefits for classification, and for any other trace manipulation that we endeavor to perform. The entire corpus (400 traces) consists of 1 778 167 occurrences of 98 system calls. This is reduced to 954 660 occurrences of 62 different system calls, a reduction of 46,31%. Of these, 8 system calls were deleted in phase 2 and 28 replaced in phase 3. For some traces, the reduction in the number of system calls is even more substantial, reaching over 94%.

The reduction in size is generally independent to the length of the trace. The smallest reduction in size, (3.76%) occurs in the trace of a calendar app, which with the smallest reduction in size (3.76%) occurring in the shortest trace of the dataset.

System call	Equivalences	System call	Equivalences
fchmodat	fchmod, chmod	getpid	getppid
lseek	llseek, _llseek	getdents	getdents64
fdatasync	fdatasync	access	faccessat
write	writev, pwrite, pwrite64	truncate	ftruncate , ftruncate64 , truncate
getuid	geteuid, geteuid32, getuid32	accept	accept4
send	sendto, sendmsg, sendmmsg	fchown32	fchown , chown, lchown
fstat	stat, stat64, statfs64, asstat, lstat, lstat64	add_key	request_key , keyctl , keyutils , keyrings
open	open, openat, create, dup	adjtimex	ntp_adjtime
read	readv,pread , pread64 , readlink,readlinkat	fcntl	fcntl64
recv	recvfrom, recvmsg	epoll_wait	epoll_pwait
rename	renameat, renameat2	poll	epoll
exec	execve		

TABLEAU 2.3 – System calls and their equivalences

When considering the size in kb, the reduction is even more pronounced. The total size of the dataset passes from 143444 kb to 8822kb, a reduction of 93.84%.

2.3 EXPERIMENT 1 : SYSTEM CALLS

In this initial experiment, we consider a vector with the number of occurrences of each system call in the execution trace of an application.

2.3.1 ANALYSIS OF THE DATA

Bhatia et. al.,(20) constructed a dataset by running the Monkey tool on 50 benign apps and 50 malicious apps, generating 500 clicks for each app. They argue that the system calls present in the trace constitute an important feature set to analyze the behavior of unknown applications, as well as to distinguish benign and malicious applications. Table 2.4 contains the top ten system calls in terms of their occurrence in both benign and malicious traces, in Bhatia et. al.'s dataset and in ours. Observe that several of the system calls which they found to be amongst

Benign apps		Infected apps	
in (20)	in our dataset	in (20)	in our dataset
clock_gettime	ioctl	read	recv
read	recv	clock_gettime	send
ioctl	getuid	_llseek	ioctl
epoll_pwait	send	ioctl	getuid
rt_sigprocmask	write	pread64	read
getuid32	epoll_wait	recvfrom	write
recvfrom	read	epoll_pwait	fstat
futex	fstat	write	lseek
gettimeofday	prctl	getuid32	close
write	close	close	fcntl

TABLEAU 2.4 – Top 10 system calls in terms of their occurrence according to Bhatia et al. (20) and in our dataset

the most frequent are absent from our data because of the abstraction process described in section 2.2.3. For instance *clock_gettime* and *gettimeofday* system calls which have been deleted from our dataset.

For the infected version, the system calls which are present in our database but not in Bhatia et. al.'s analysis are :
— *fcntl*; manipulate file descriptor
— *fstat*; get the file status
— *send*; send a short message on a socket.
In addition, two system calls appear in the top 10 most frequent list compiled by Bhatia et. al. and not in our dataset, namely *pread64* which we replaced by *read* during the abstraction process and *epoll_pwait* which synchronises input/output events.

As can be seen in Table 2.4, the system calls *fcntl* and *lseek* are among the most frequent system calls in infected traces but are absent from benign traces. These system calls are related to writing and reading data from files stored on the phone and SD memory, a behavior exhibited by multiple malware families such as *FakeInstaller*, *Opfake* and *Plankton*. This observation hints at the possibility of effective malware detection through system call analysis.

We also found that a number of system calls are very common in both malware and benign apps, a fact that poses an obstacle to effective detection. Table 2.5 presents the frequency of the most common system calls in both benign and infected traces. We also observe that malware generates the *recv*, *send*, *read* and *close* system calls more frequently than benign apps. These system calls serve to connect to a socket and request access to sensitive resources, a common behavior of malicious apps.

system call	benign traces	infected traces
ioctl	19.85%	18.63%
recv	15.88%	27.47%
getuid	14.43%	9.99%
send	13.89%	20.67%
write	12.94%	5.77%
read	6.63%	7.53%
fstat	2.56%	1.92%
close	1%	1.34%

TABLEAU 2.5 – Frequency ratio of some system calls

Based on these observations, we performed an automatic classification of benign and malicious traces based on the system calls they contain. This is a replication of an experiment performed by Bhatia et. al. (20) on the dataset described above.

2.3.2 FEATURE EXTRACTION

We begin by extracting a feature vector from each trace. Following Bhatia et. al., we extract from each trace a feature vector that indicates the number of occurrences of each system call in that trace. Studying the frequency of system calls can help distinguish which class an application belongs to. For example, Marko Dimjaševic et. al. in (38) claim that an increase in the use of I/O system calls may be indicative of malicious behavior. After we performed the manipulations described in section 2.2.3, the abstracted traces contain 62 different system calls.

2.3.3 CLASSIFICATION

We used the WEKA tool (133) for data classification, in order to map the input data to one of the categories benign or infected. We used 80% of our data for training and 20% for testing. We compare the effectiveness of two algorithms, *Random forest* and *J48* algorithms— the same two algorithms employed by Bhatia et. al. (20). The results are given in Table 2.6.

We have achieved 95.3% accuracy in correctly classifying the benign applications and 85% accuracy in correctly classifying the infected applications using the *Random Forest* and *J48* algorithm.

Bhatia et. al., (20) tested the same algorithms to perform the same classification. They achieved 88% accuracy using the *Random Forest* algorithm and we achieved 92.1%.

Algorithm	TP Rate	Precision	F-Measure	Class
Random Forest	95.3%	93.2%	94.3%	benign
	85%	89.5%	87.2%	infected
	92.1%	92%	92%	
J48	95.3%	93.2%	94.3%	benign
	85%	89.5%	87.2%	infected
	92.1%	92%	92%	

TABLEAU 2.6 – The classification rates of traces - System calls

We posit that the abstraction process described in section 2.2.3 is responsible for the improved accuracy of the detection process. To verify this hypothesis, we repeated the same experiment on our dataset, but without first going through the abstraction phase. As expected, the results lower, with a TP rate of 87.1% and 90.3% by applying the J48 and Random Forest algorithms respectively.

2.4 EXPERIMENT 2 : TF–IDF

2.4.1 FEATURE EXTRACTION

Another common way to analyse system call traces is to compute of the "Term Frequency–Inverse Document Frequency" (TF–IDF) weight vectors of the system calls (142; 136; 102). TF (*text frequency*) refers to the number of times a certain word occurs in a document while IDF (*inverse document frequency*) refers to the number of times the word occurs throughout the corpus (32). The value of TF-IDF is a measure of the relevance of given system call in a trace. The technique has it's origin in the analysis of texts in natural language, and is widely used in trace analysis. Following (95), we compute the TF-IDF as follows :

$$F_{TF-IDF} = (1 + log f_{t,d}) \times log \frac{N}{n_t}$$

where t is the system call name, d represents the trace, f is the frequency of the system call in each trace, N is the total number of traces and n is the number of traces where the system call t occurs.

We applied this formula on our dataset and created a matrix where each row contains the value of TF-IDF for each system call, based on the frequency of the call found in experiment 1, and each column corresponds to a distinct system call. We then proceeded with classification by using each line of this matrix as a feature vector. This strategy was used by Kumar et. al. (35).

Algorithm	**TP Rate**	**Precision**	**F-Measure**	**Class**
Random Forest	93.8%	91.4%	92.6%	benign
	79%	84.2%	81.5%	infected
	89.4%	89.3%	89.3%	
SGD	92.7%	92.3%	92.5%	benign
	81.5%	82.5%	82%	infected
	89.4%	89.4%	89.4%	
SVM	92.7%	89.1%	90.9%	benign
	72.8%	80.8%	76.6%	infected
	86.9%	86.6%	86.7%	
MLP	92.7%	92.3%	92.5%	benign
	81.5%	82.5%	82%	infected
	89.4%	89.4%	89.4%	
J48	82.9%	85.6%	84.2%	benign
	66.7%	62.1%	64.3%	infected
	78.1%	78.6%	78.3%	

TABLEAU 2.7 – The classification rates of traces - TF-IDF

2.4.2 CLASSIFICATION

We tested the *Random forest*, *SGD*, *SVM*, *J48*, and *MLP (MultiLayer Perceptron)* algorithms. For this experiment we used the 10 fold cross validation technique. We applied the same algorithms and test option tested in (35) as well as the (*Random forest*) algorithm, which game us better results. Table 2.7 summarizes the results obtained.

In their own study, Kumar et. al. (35) generated traces of different length for 534 apps using the Monkey tool, recorded the system calls using Strace and abstracted them using TF-IDF as described above. The resulting vectors where then used to place each app in a behavior category (tool, lifestyle, education, etc.). This type of research has applications in security, since previous research (107) has found that automatic malware detection techniques can be improved if behavioral observations are supplemented with knowledge about the purpose and expected behavior of the application.

With our dataset, the optimal result was 93.8% accuracy in correctly classifying the benign applications with the *Random Forest* algorithm. In addition to 81.5% in correctly classifying the infected applications with the *SGD* and *MLP* algorithms.

The variable length of execution traces can introduces bias when using TF-IDF. To palliate this issue, Chan Woo Kim (67) uses a normalized TF-IDF values using the following Euclidean norm :

Algorithm	TP Rate	Precision	F-Measure	Class
SVM	92.6%	92.6%	92.6%	benign
	85.7%	85.7%	85.7%	infected
	90.2%	90.2%	90.2%	
SGD	92.6%	89.3%	90.9%	benign
	78.6%	84.6%	81.5%	infected
	87.8%	87.7%	87.7%	
J48	92.6%	86.2%	89.3%	benign
	71.4%	83.3%	76.9%	infected
	85.4%	85.2%	85.1%	

TABLEAU 2.8 – The classification rates of traces - TF-IDF normolized

$$F_{norm} = \frac{F_{TF-IDF}}{\sqrt{F_1^2 + F_2^2 + ... + F_n^2}}$$

Where F_{TF-IDF} is the TF-IDF weight of each system call in the trace. We repeated his experiment on our dataset by applying *SVM*, *SGD* and *J48* classification algorithm. Optimal results were obtained when 85% of the data was used for training and 15% for testing, a fact that may be explained by the limited size of our dataset. The results are presented in Table 2.8.

We obtained optimal results with the *SVM* algorithm, reaching 90.2% of correctly classifying application. However Chan Woo Kim achieved a higher accuracy of 96% using the *SGD* algorithm, working on a dataset comprising 63.7% malicious apps. It may be that the cognate nature of our dataset, with malicious app being matching with benign pair that is identical in all respect except for the inclusion of the malware, makes detection more challenging. Normalized TF-IDF outperforms unnormalized, because of the variable length of traces

We replicated these experiments without first performing the abstraction described in section 2.2, and find that the use of abstraction improves the accuracy of the classification process in all but one cases. Indeed, when using the abstraction phase, the accuracy of TF-IDF classification using Random Forest improves from 88.5% to 89.4%. The same pattern is visible with SVM (from 86.2% to 86.9%) , SGD (from 85.9% to 89.4%) and MLP (from 86.2% to 89.4%). The algorithm J48 is the only outlier, and shows a slight diminution of accuracy when operating on abstracted traces.

The same result is observed in the case of normalized TF-IDF. Pre-processing the traces using the abstraction process improves the accuracy of classification by an average 5% with every classification algorithm except J48.

2.5 EXPERIMENT 3 : N-GRAMS

2.5.1 ANALYSIS OF THE DATA

In this next experiment, we apply the use of unigrams, 2-grams and 3-grams of system calls to our dataset. N-grams preserve the local ordering of system calls in the input data and are widely used form of trace abstraction. Notably, N-grams are used in a security context by Ananya A. et. al. (10). They used unigrams, 2-grams and 3-grams for malware detection in system call traces of Android apps. Igor Santos et. al. (104) demonstrated that the N-grams technique allows the detection of unknown variants of malware.

An N-gram is a substring of a given sample of trace of length N. For example, consider a part of a system calls trace after the abstraction phase :

```
{epoll_wait, read , getuid , fstat , fstat,
fstat, ioctl}
```

Figure 2.2 illustrate the corresponding 2-grams and 3-grams.

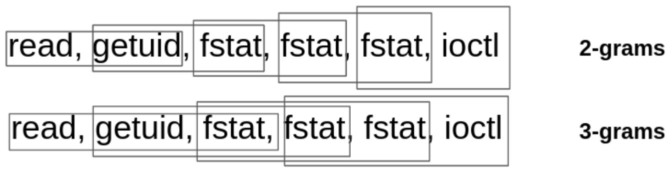

read, getuid, fstat, fstat, fstat, ioctl
2-grams
read, getuid, fstat, fstat, fstat, ioctl
3-grams

FIGURE 2.2 – 2-grams and 3-grams example

Table 2.9 shows the top 5 most used 2-grams in both types of traces. We can observe that the benign traces share its most used 2-grams with the infected traces.

The 2-grams most invoked by infected applications is *recv, send*. This combination of system call is used to send SMS to premium services owned by malware author (76). While this 2-gram can be found in benign traces, it is rather infrequent, occurring in only 70% of benign traces. It occurs in every malicious trace.

The same discrepancy was observed with several 2-grams. Furthermore, multiple 2-grams are only present only in malicious traces, but not in benign ones. Many of these 2-grams, such as *setsockopt,connect - connect,fcntl - fcntl,getsockname - getsockname,setsockopt - setsockopt,send - send,select - select,select - select,recv*, contain sequences of system calls that serve to establish a connection to a server and send information on the network.

In our dataset, the most commonly used 3-grams in infected samples are : *ioctl,ioctl,ioctl - getuid,epoll_wait,read - ioctl,ioctl,open - send,recv,send*. These calls are used to control the

	Benign traces		Infected traces
1	epoll_wait,read - getuid,epoll_wait - write,ioctl	1	recv,send
2	ioctl,ioctl	2	ioctl,ioctl - epoll_wait,recv - epoll_wait,read - getuid,epoll_wait - write,ioctl - ioctl,send - send,recv - ioctl,open
3	epoll_wait,recv	3	write,write - send,send - read,recv - recv,ioctl
4	ioctl,write - read,recv	4	fcntl,close - open,fcntl - epoll_wait,ioct - recv,write - close,epoll_ctl
5	epoll_wait,ioctl	5	ioctl,write

TABLEAU 2.9 – Top 5 2-grams system calls in term of their presence in our dataset

device and send SMS. However *getuid,epoll_wait,read* is the most 3-grams used by the benign samples. It allows reading the user ID.

2.5.2 FEATURE EXTRACTION

The dataset contains 62 distinct unigrams (system calls) ; 510 2-grams and 2188 3-grams. We create tow sets of feature vectors for each case : The first is a binary vector that indicates the presence or absence of the n-gram in each trace as in (58) and the second indicates the number of occurrences of the n-gram in the trace as in (27).

2.5.3 CLASSIFICATION

We evaluate 3 different algorithms, namely *SVM*, *SGD* and *LMT*.

For each experiment, the vector sample was randomly split into 80% for training and 20% for testing. The results of the experiments are shown in Table 2.10 and 2.11. Table 2.10 shows the result of the N-grams binary classification rates.Table 2.11 represent the classification rates of the N-grams occurrence.

From Tables 2.10 and 2.11 it can be seen that classification using 3-grams vectors of system calls outperforms that using unigrams and 2-grams vectors. When classifying using binary vectors, the *SVM* algorithm yielded optimal performance compared to that achieved by the

	SVM		SGD		LMT		Class
	TP rate	**Precision**	**TP rate**	**Precision**	**TP rate**	**Precision**	
Unigrams	81.8%	97.3%	75%	97.1%	84.1%	94.9%	benign
	90%	52.9%	90%	45%	80%	53.3%	infected
	83.3%	89.1%	77.8%	87.4%	83.3%	87.2%	
2-grams	97%	91.4%	90.9%	90.9%	84.8%	96.6%	benign
	87.5%	95.5%	87.5%	87.5%	95.8%	82.1%	infected
	93%	93.1%	89.5%	89.5%	89.5%	90.5%	
3-grams	100%	97.1%	100%	94.3%	100%	91.7%	benign
	95.8%	100%	91.7%	100%	87.5%	100%	infected
	98.2%	98.3%	96.5%	96.7%	94.7%	95.2%	

TABLEAU 2.10 – The classification rates of the N-grams binary

	SVM		SGD		LMT		Class
	TP rate	**Precision**	**TP rate**	**Precision**	**TP rate**	**Precision**	
Unigrams	91.7%	91.7%	86.1%	96.9%	88.9%	94.1%	benign
	84.2%	84.2%	94.7%	78.3%	89.5%	81%	infected
	89.1%	89.1%	89.1%	90.4%	89.1%	89.6%	
2-grams	84.8%	100%	90.9%	96.8%	97%	94.1%	benign
	100%	82.8%	95.8%	88.5%	91.7%	95.7%	infected
	91.2%	92.7%	93%	93.3%	94.7%	94.8%	
3-grams	87.9%	96.7%	87.9%	96.7%	100%	94.3%	benign
	95.8%	85.2%	95.8%	85.2%	91.7%	100%	infected
	91.2%	91.8%	91.2%	91.8%	96.5%	96.7%	

TABLEAU 2.11 – The classification rates of the N-grams occurrence

SGD and *LMT* algorithms. Indeed, the classifier reached 100% and 95.8% accuracy to classify benign and infected instances respectively. However, for unigrams the accuracy does not exceed 90% for either class. In (58) the authors calculated the F-measure. For the 3-gram they obtained a rate of 97.5%, with our dataset we obtained 98.2%.

The classification rates of the N-grams occurrence get the best TP rate with 96.5%, and *LMT* performed with the best TP when n is 3 compared to 89.1% for unigrams and 94.7% for the 2-grams. For the correct classification of infected traces in 2-grams experiment, the *SVM* algorithm gives the best rates with 100% True Positive rate. For the 3-grams, the authors in (27) obtained a result of 97% of accuracy, or our method achieves 91.8%.

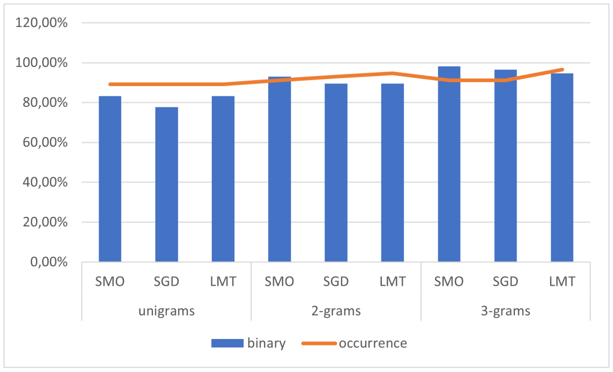


FIGURE 2.3 – Histogram of TP rates of N-grams

The Figure 2.3 shows the results of classification using N-grams. We can observe that for every classifier, results improve as the value of N increases. For example, the number of correctly classified instances of binary vectors with the *SVM* algorithm increases by 10% between unigrams and 2-grams, and by a further 5% between 2-grams and 3-grams. Also, we can observe the same thing for the correctly classified instances with the vector of occurrence. The result augmented by 5% between unigrams and 2-grams, and 2.5% between 2-grams and 3-grams. For a better classification of unigrams, the use of the occurrence was the best method, unlike 2-grams and 3-grams whose classification of its binary vectors gave the best result. So we can conclude that the use of the sequences occurrences is not enough to discriminate malicious from benign traces.

2.6 EXPERIMENT 4 : RELATIVE ORDERING

2.6.1 ANALYSIS OF THE DATA

By studying the pairs of benign and infected traces in our dataset, we noticed that the attack often takes place at the onset of the execution. In this final experiment, we propose a novel scheme for malware detection in traces, premised on this observation.

We divided each trace into a given number (k) of segments. We tested k with the values 5, 10, 30, 50, and 100 segments. Then we recorded in which segment is the first occurrence of each system call in our database. The size of each segment naturally differs from one trace to another since the sizes of the traces are varies.

To verify the location of malware in the divided traces, we have studied the system calls present in the segments. During the analysis of these calls, we observed that for a pair of the traces, the system calls represent the malware are present for the first time at the start of the infected trace and the end of its benign version.

For example, the system calls *recv* and *send* allow to steals information from the device and sends them to a remote server. We can find these two calls in the pairs traces of the "Battery charger" application. In the infected trace, these two calls are present for the first time in segment 1 for k = 5 and k = 10, and between segments 2 and 11 for k = 30, 50 and 100. However, the call *recv* is first found in benign traces between segments 2 and 37, and the call *send* between segments 5 and 90 for different values of k.

Another example of the pairs of traces, those of the application "Kobe Bryant". In the infected version of the trace, we can find the *fstat*, *open* and *close* calls which appear for the first time in segment 1 for any value of k. As well as the *fcntl* call which appears for the first time in segments 1, 3, 5 and 9 for k=5, k=10, k=30 ,k=50 and k=100 respectively. However, these calls first appeared between segments 5 and 82 in benign traces divided into k segments. These four system calls are responsible for reading the data stored in the device and SD memory.

2.6.2 CLASSIFICATION

After dividing the traces into segments, we have constructed a vector with the number of segments where each system calls appears for the first time or a 0 if this system calls is not in the trace. The size of each vector corresponds to the number of system calls in our database, namely 62.

We tested the classification of the five types of segments presented above. For this experiment, we used 66% of the vectors as a base and 34% for the test and compared several classification algorithms. Optimal results were obtained by the *LMT*, *J48*, and *3NN* algorithms. Consider the J48 algorithm, for the classification of our data, we obtained 79.31% of TP with 5 segments, 72.41% for the 10 segments, 96% for the 30 segments, 93% for the 50 segments, and 75.86% for the 100 segments. Therefore the best classification of benign and infected traces was obtained by dividing the traces into 30 segments. The results of the classification rates for the 30 segments were presented in Table 2.12.

Algorithm	TP Rate	Precision	F-Measure	Class
LMT	91.3%	100%	95.5%	benign
	100%	75%	85.7%	infected
	93.1%	94.8%	93.4%	
J48	95.7%	100%	97.8%	benign
	100%	85.7%	92.3%	infected
	96.6%	97%	96.6%	
3NN	100%	92%	95.8%	benign
	66.7%	100%	80%	infected
	93.1%	93.7%	92.6%	

TABLEAU 2.12 – The classification rates of traces - Experiment 5

We can observe that algorithms *LMT* and *J48* have successfully classified every infected traces with a recall of 100%. Then the *3NN* algorithm succeeded in classifying benign traces also with a recall of 100%.

The optimal result was given by the J48 algorithm, with a TP rate of 96.6% with a precision of 97%.

2.7 RELATED WORK

Addressing the security risks associated with Android apps is an urgent problem and a number of solutions have been suggested. A thorough survey of every proposed solution would be out of the scope of this paper and the interested reader is referred to the surveys of Razgallah et al. (98), Naway and Li (84), Alzahrani and Alghazzawi (7), Rubiya and Radhamani (118) Yan and Zheng (140) or Arshad et al. (16).

It is noteworthy to stress that static approaches, based on permissions, code analysis or other features obtainable without running the code, outnumber dynamic solutions. The former have the advantages of early detection and of minimal runtime overhead, while the latter can circumvent strategies used by malicious adversaries to avoid detection, such as code obfuscation, dynamic loading or encryption.

System calls are widely used by dynamic analysers, since they reveal the code low-level behavior, and can be obtained even when the application's source code is unavailable. However, due to the large volume of data present in a system call trace, most of these approaches involve an abstraction phase that relate the raw system call trace to a higher-level behavior.

For instance, Hamou-Lhadj et al. compare and correlate the system call traces of two different programs offering the same functionalities, beforming the same high-level behavior on two different operating systems (54). In order to reveal the commonality of behavior, they first elide extraneous information such as pid, parameters and return values from the trace. They further remove sequential repetitions of the same system call. Finally, they replace sequences of system calls with higher-level abstract behaviors using a pre-established pattern library. This process allows them to uncover the presence of malicious behavior in one of the two instance traces.

N-grams (subsequences of fixed length) are widely used for the comprehension and analysis of system call traces (28). Li et al. instead opt to employ Markov Chain to extract variable length subsequence from system call traces. These sequences are then fed to a neural network to distinguish malware from benign traces.

Drawing an analogy to natural language, Xiao *et al.* (137) proposed a detection method that employs the *Long Short-Term Memory* model (LSTM (56)), a type of neural network model used in the processing of natural languages. They treat a trace of system calls as a sentence, with each system call being an individual word, and train two classifiers, one for benign traces and one for malicious ones. An execution is then pegged as being malicious if it is more likely to occur in the malicious model.

Canfora *et al.* (26) also relied upon system calls to perform malware detection. Their method draws upon the fact that malware tends to evolve through an iterative process of borrowing and modifying code from other malware. The trace is first abstracted into a vector that records the number of occurrences of subsequences of a given length, and detection is then performed using a supervised learning algorithm.

Sanya *et al.* (29) proposed an approach to detect malicious behavior at runtime. They first executed apps in a controlled environment for a fixed period of time, and recorded the system call occurring during this time. After discarding the less statistically significant system calls, each app was associated with a Boolean vector that indicates if each of 18 more relevant system calls is present or absent during its execution. This data is then fed to a machine learning algorithm.

2.8 DISCUSSION AND CONCLUSION

In this paper we analyzed system calls traces of Android. We first propose a novel abstraction process that highlights the features that most readily distinguish benign and malicious features, and perform experiments that indicate that this abstraction process improves the performance of classifiers. We further replicate 5 studies that employed system call traces for malware detection, confirming the results of these studies. The Replication of multiple malware detection techniques on the same dataset allows comparison on equal footing.

Furthermore, we propose a novel malware detection technique, based on the intuition that benign and malicious traces differ with respect to the relative ordering of the first occurrence of each system call in the trace. This technique shows promising results, specially if it is used in conjunction with one of the other methods replicated in this paper.

Finally, we make our data available to other researchers in the field.

The classification algorithms tested in each experiment provide us with insights about the effectiveness of each abstraction when applied to the problem of classifying traces as benign or infected. By testing each experiment using the same database of traces, we observed that optimal results were obtained when considering the 3-grams binary of system calls and traces divided by segments. In this later two cases, precision rate varied between 97% and 98.3% and TP rate between 96.6% and 98.2%.

Figure 2.4 displays the best TP rates and ROC area for each experiment. The ROC (Receiver Operator Characteristics) area gives the rate of true positives as a function of the rate of false positives. If the value of the ROC of an algorithm is larger, then the classification is better.

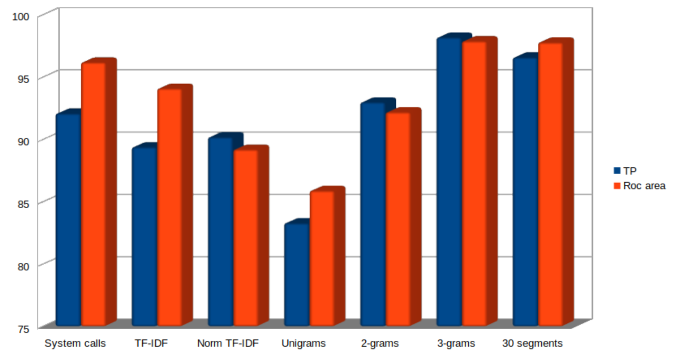


FIGURE 2.4 – The best TP rates and ROC area of our expirements

The experimental results in our experiments show that the proposed abstraction phase allow to have high classification accuracy. Because of the removal of irrelevant information which can give false information about the behavior of an application and increase the rate of false positive classification.

CHAPTER 3

ÉTAT DE L'ACCEPTATION : VERSION FINALE PUBLIÉE

CONFERENCE : 2022 IEEE/ACM 19TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR)AT : PITTSBURGH, PA, USA

DOI :10.1145/3524842.3528502

CITATIONS : 1 FOIS

TITRE EN FRANÇAIS

TwinDroid : une base de données de traces d'appels système des applications Android et de pipeline de génération de traces

RÉSUMÉ

Les suivis d'appels système sont une source inestimable d'informations sur le comportement d'exécution d'un programme et sont particulièrement utiles pour la détection de logiciels malveillants dans les applications Android. Cependant, la rareté des ensembles de données de haute qualité accessibles au public entrave le développement du domaine. Dans cet article, nous présentons *TwinDroid*, une base de données de plus de 1 000 traces d'appels système, provenant d'applications Android bénignes et infectées. Une grande partie des applications utilisées pour créer la base de données provient de paires d'applications bénignes-malveillantes, identiques à l'exception de l'inclusion de logiciels malveillants dans ces dernières. Cela fait de

TwinDroid une base idéale pour la recherche en sécurité, et une version antérieure de *TwinDroid* a déjà été utilisée à cette fin. En plus d'un ensemble de données de traces, *TwinDroid* comprend un pipeline de génération de traces entièrement automatisé, qui permet aux utilisateurs de générer de nouvelles traces de manière standardisée et transparente. Ce pipeline permettra à l'ensemble de données de rester à jour et pertinent malgré le rythme rapide des changements qui caractérisent la sécurité Android.

MOTS CLÉS

Base de données, Applications Android, Sécurité, Suivi des appels système.

CHAPTER 3

TWINDROID : A DATASET OF ANDROID APP SYSTEM CALL TRACES AND TRACE GENERATION PIPELINE

Asma Razgallah*, Raphaël Khoury*, Jean-Baptiste Poulet*
* Department of Computer Science and Mathematics, Université du Québec à Chicoutimi, Canada

System call traces are an invaluable source of information about a program's runtime behavior and be particularly useful for malware detection in Android apps. However, the paucity of publicly available high-quality datasets hinders the development of the field. In this paper, we introduce *TwinDroid*, a dataset of over 1000 system calls traces, from both benign and infected Android apps. A large part of the apps used to create the dataset is from benign-malicious app pairs, identical apart from the inclusion of malware in the latter. This makes *TwinDroid* an ideal basis for security research, and an earlier version of *TwinDroid* has already been used for this purpose. In addition to a dataset of traces, *TwinDroid* includes a fully automated traces generation pipeline, which allows users to generate new traces in a standardized manner seamlessly. This pipeline will enable the dataset to remain up-to-date and relevant despite the rapid pace of change that characterizes Android security.

Index terms : dataset, Android apps, security, System call traces

3.1 INTRODUCTION

System call traces reveal a wealth of information about the execution of the underlying system and serve as the basis for a variety of analyses, including malware detection, optimization, feature enhancement, and performance analysis. System call traces have been shown to be particularly useful in the case of malware detection in Android apps, and a number of security solutions that rely upon system calls have been proposed (57), (3), (55), (113), (122).

However, the paucity of publicly available datasets of Android traces— often decried by researchers, forces each research team to rely upon its custom-generated dataset, with the consequence that it is not easy to compare the effectiveness of these methods on equal footing. The considerable time required to generate an adequate volume of traces is also an obstacle to developing security solutions. Many studies on the topic are tested on trace sets whose size seems insufficient to draw general conclusions.

To address these issues, we present *TwinDroid*, a dataset of Android system call traces. At present, *TwinDroid* contains 10,859 traces. A large part (42%) of the traces present in the dataset has been generated from pairs of benign and infected (repackaged) versions of the apps. The presence of apps that are identical but for the inclusion of malware makes this dataset ideal for research on dynamic malware detection. We intend to continue to expand the size and variety of the dataset in the near future.

Furthermore, *TwinDroid* repository also includes a well-controlled and reproducible pipeline, which automates the process of generating new traces. This allows users to seamlessly generate new traces that are tailored to their particular needs. For example, a researcher who wishes to study the evolution of Android apps over a period of time may generate traces from both earlier and more recent apps.

In addition to traces, *TwinDroid* also extracts permissions and features from each app. These datums are highly predictive of malicious behavior and repackaging (93), (150),(41).

While specially designed for security purposes, *TwinDroid* is sufficiently versatile to serve in research in several areas of software engineering, mainly because of the ability to generate additional traces on-demand with minimal effort and allows the dataset to remain up to date.

TwinDroid offers several advantages, namely :

— The dataset consists of traces of apps drawn from multiple sources and contains both benign apps and infected apps with various malware. An important part of the dataset consists of traces from pairs of applications, one benign, one malicious, but identical but for inclusion of the malware in the latter. This makes TwinDroid uniquely suited for security research.

— The apps traced span a variety of common app categories and have been published over ten years. For each app, the dataset contains multiple traces of varying lengths.

— In addition to each trace, the database records the random seed and event stream that created the trace. This provides a large degree of reproducibility and a path for research on explainability if a trace exhibits interesting or unusual behavior.

— In addition to the dataset of existing trace, *TwinDroid* presents a fully automated pipeline to generate new traces from existing apps. Furthermore, any researcher in need of app traces may use this pipeline to generate traces suitable to their specific needs. This ease of extension is a crucial design goal as it allows the database to remain up-to-date with emerging malware.

An earlier version of the *TwinDroid* dataset was used to perform malware detection on abstracted traces (97).

Both the dataset and the trace generation pipeline are public and freely available for research purposes [1] [2].

The remainder of this paper is organized as follows : Section 3.2 describes the dataset, while Section 3.3 details the associated trace generation pipeline. Section 3.4 discussed some limitations of the dataset, and Section 3.5 presents some avenues of research that TwinDroid opens.

3.2 THE TWINDROID DATASET

The *TwinDroid* dataset includes over 10,859 traces, from 773 benign and 4,715 infected apps. Each trace is generated by using the Monkey tool [3] to run the app on random inputs for a pre-determined amount of time (the dataset includes traces of differing lengths) and tracing the execution using Strace [4]. All apps are run in an emulated environment.

TwinDroid's database schema is given in Figure 3.1.

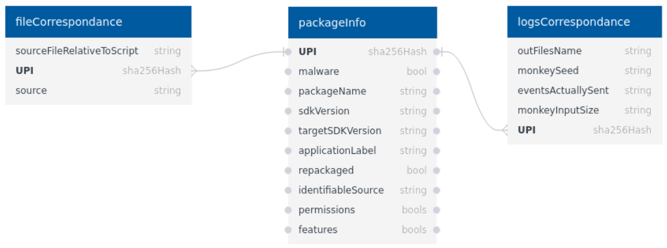


FIGURE 3.1 – Database Schema

A Unique Package Identifier (UPI), computed by applying the SHA-256 hash to each app, serves as a primary key (field `Unique Package Identifier`). The fileCorrespondance table additionally contains the relative path of the apk on the host system, including its filename (field `sourceFileRelativeToSCript`) and the source of the app, in case apps are selected from multiple datasets of apps.

Table packageInfo contains a variety of information about the apps. Notably, this table indicates the file name of the apk (field `packageName`), the version of the SDK (field `sdkVersion`)

1. `https://github.com/RaphaelKhoury/automated-apk-tracing`
2. `https://zenodo.org/record/6259612#.YioFmxvCpH4`
3. `https://developer.android.com/studio/test/monkey`
4. `https://strace.io/`

and the target SDK version (field `TargetSDKVersion`). this information is extracted from the manifest file, as is the name of the app (field `ApplicationLabel`). The presence or absence of malware is indicated by the Boolean attribute `malware`. In many cases, the exact name of the malware infecting an apk can be obtained from the database from which the apk was downloaded.

The `permissions` and `features` fields respectively contain the permissions and features declared in the manifest file of the apk. This information was included in the database because of its relevance to malware detection and other security-relevant applications. These fields are constructed by comparing the permissions and features declared in the manifest file with lists contained in two files, "features.txt" and "permissions.txt", which include all standard permissions and features listed on the official Android developer documentation at the time of the creation of TwinDroid. This scheme was adopted to avoid cluttering the database with user-defined permissions, which are not informative. When generating additional traces, the user may update these files to reflect any changes in the standard permission and feature sets.

Finally, the logsCorrespondance table holds data resulting from the actual tracing. For a given UPI, multiple entries will exist, one for each trace test performed on the apk. The field `OutFilesName` holds the base name of the two files generated for each trace : a `.monkdata` file that contains the output of the Monkey script and a `.trace` file that contains the trace proper. To improve reproducibility, we also include the seed passed as a parameter to the Monkey tool (field `MonkeySeed`), to generate the random input as well as the number of input that was intended to be sent by Monkey to the app (field `MonkeySizeInput`). The field `eventActuallySent` contains the number of events that the Monkey tool was actually able to send before the application failed or execution was terminated. Such failures are usually caused by a crash in the application. This is not surprising considering that the script performs fuzzing, which much ill-developed software can't handle. Such failures do not hinder the execution of the tracing process : the abnormal execution is recorded in the database, and the tracing proceeds. If `monkeySizeInput` and `eventActuallySent` are equal it can be assumed that the test was successfully completed.

TABLEAU 3.1 – Content of the *TwinDroid* dataset.

Dataset	Benign apps	Malicious apps	Pairs	Benign traces	Malicious traces	Years
Androzoo	704	1601	1601	1313	3201	2016 – 2021
Google Play	40	0	22	80	0	2008 – 2022
APKpure	29	0	0	38	0	2014 – 2022
Drebin	0	3114	22	0	6227	2010 – 2012
Total	773	4715	1623	1431	9428	

Each trace consists of two files, a trace file, and a .monkeydata file.

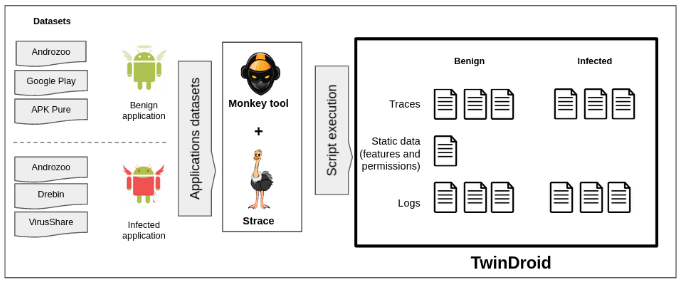


FIGURE 3.2 – Overview of the Trace Generation Pipeline.

```
[pid  3103] 1641327580.279530 prctl(PR_SET_VMA, PR_SET_VMA_
ANON_NAME, 0x713d39b38000, 16384, "stack_and_tls:3103") = 0
[pid  3103] 1641327580.279611 mmap(NULL, 36864, PROT_READ|
PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x713d39aad000
[pid  3103] 1641327580.279688 mprotect(0x713d39aad000, 4096,
PROT_NONE) = 0
[pid  3103] 1641327580.279750 sigaltstack({ss_sp=0x713d39aae000,
ss_flags=0, ss_size=32768}, NULL) = 0
[pid  3103] 1641327580.279820 prctl(PR_SET_VMA, PR_SET_VMA_ANON_
NAME, 0x713d39aae000, 32768, "thread signal stack") = 0
[pid  3103] 1641327580.280081 rt_sigprocmask(SIG_SETMASK, [QUIT
USR1 PIPE RTMIN], NULL, 8) = 0
[pid  3103] 1641327580.280175 mprotect(0x713a96c04000, 4096,
PROT_NONE) = 0
```

FIGURE 3.3 – Sample of an application trace

The trace file is the output of Strace, with parameters : -f -ttt. These parameters can be modified if additional traces are generated.

Each line of the trace file corresponds to a single system call and indicates a timestamp, followed by the system call itself, including parameters and return values. If more than one thread or process is traced (because the initial process forked), then the PID of the process that performs the system call is prefixed to each line. A sample of a trace is shown in Figure 3.3.

The .monkdata file is the output of the Monkey tool on stdout. It contains a wealth of data that aids in understanding the underlying behavior of the trace, notably a listing of the random events sent to the app. We refer the reader to the Monkey tool's documentation for more information on this data.

Each trace is on average *4kB*, for a total of *7GB* for the entire dataset.

Origin of the apps The TwinDroid dataset currently contains over 10,859 traces from 4,715 malicious and 773 benign apps. Of these, 2,327 apps occur in matching pairs of otherwise identical benign and malicious apps. The make-up of the dataset is detailed in Table 3.1.

We opted to include samples of traces from several different sources, both to increase the diversity of app types and malware types present in the dataset, as well as to illustrate the ease with which new traces can be added using the trace generation pipeline. As such, we selected benign apps from the Google play (51), and APKpure (13) datasets, and malicious apps from the Drebin, (15) Andozoo (4) and Google play datasets. The Androzoo dataset identifies 15,000 repackaged variants of 3,000 benign apps in the Androzoo dataset. An additional 22 pairs were manually identified by comparing the Google Play and Drebin datasets. We used the online tool VirusTotal[5] to determine if the applications of Google Play and Pure APK are benign.

All of these datasets are widely used in academic research. Also, note that while Androzoo and APKpure contain more recent apps appropriate for security research, those from Google play and Drebin span a more extended period, allowing longitudinal studies on software engineering.

3.3 TRACE GENERATION PIPELINE

In addition to the traces already present, TwinDroid includes a trace generation pipeline that automates the process of tracing Android apps and of generating new trace data. The apps are run on an emulator, which is run inside a container. This design choice is motivated by providing identical settings for all trances. Using a container avoids the time-consuming process of installing and configuring the emulator since the container was created from a system on which the emulator is already installed. The use of containers provides additional benefits, including :

- Additional security and isolation of the malware from the underlying system;
- Ease of deployment, avoiding the installation of the emulator, and providing a common emulator setup for every execution;
- Possible parallelization.

The pipeline creates a fresh container for each app and then proceeds to install the apk via adb. Duplicate apks are seamlessly detected (by way of the hash) and discarded.

The pipeline first extracts static information from the apk (features and permissions) before tracing. Finally, each trace is added to the database upon completion.

If a trace of a given apk and input size already exists in the database, the script skips to the next input size or apk. This ensures that the script can be stopped at any point during tracing

5. https ://www.virustotal.com/gui/home/upload

and then restarted, skipping over the work already done.

Implementation Considerations To generate new traces, the user only has to update the configuration file and run script.sh. More specifically, the user must specify : (1) the number of traces to generate for each app, and the number of events that the Monkey tool should generate in each case ; (2) the paths of the folders containing the benign and infected traces ; (3) and optionally, a list of pairs of apps, identified by their hash values, in case the dataset includes matching pairs of benign and infected versions of the same app. The apps can be copied directly in the benign and infected folders. However, if the apps come from multiple sources and the user wishes to keep track of each app's origin, a distinct sub-folder for each app source should be created in these folders. The name of this sub-folder will populate the `source` field in the database.

As mentioned above, both the Monkey tool and the emulator are run inside an emulator. Unfortunately, Monkey launches the app internally and does not offer the possibility of simultaneously invoking the Strace. To circumvent this problem, it was decided to trace the Monkey process from the onset alongside all of its sub-processes, encapsulating the application to test. This solution is perhaps sub-optimal since it implies that one of the processes being traced is the Monkey tool rather than the app. Still, the only plausible alternative, starting Strace after launching the app, would omit the beginning of the execution from the trace. Such a situation would be particularly undesirable since malware often performs malicious actions at the onset of the execution. In any case, it is straightforward to identify the system calls made by the Monkey tool, rather than by the underlying application, by their PID [6].

Some security risks are unavoidable when running apps infected with malware, even if this is done only in a simulated environment. In fact, emulators may themselves contain security vulnerabilities that could be exploited by the malware present in the underlying app (139).

The added layer of security provided by the container should thwart such attempts. The docker container runs without administrator privilege on the host machine, and it incorporates multiple built-in security features, which should ensure that the host system remains unaffected if the container is compromised. No security measure is perfect, but the attack surface should be limited to adb and the two exposed ports on the container.

3.4 LIMITATIONS

An important limitation of TwinDroid is the fact that the apps are run in an emulated environment. Some malware may be able to detect the presence of the emulator and will not perform

6. Note that Strace only records the PID after the first fork occurs. Previous system calls, for which a PID is not listed, are also performed by the Monkey tool.

malicious actions. The collected trace is thus not an accurate reflection of the actual behavior of the underlying malicious app, with a consequence for its suitability in malware detection.

In a small number of cases, the emulator is unable to run the app, with consequences for the completeness of the dataset if we require traces from every app in a sample. It is important to stress that this situation is comparatively rare, occurring in less than 5% of the apps we attempted to trace.

As discussed above, malicious apps are prone to crash in fuzzing, which reduces the utility of the traces. However, since our database contains, for each trace, the number of events that the Monkey tool intended to send, as well as the number of events that were sent, it is easy to exclude such misbehaving traces if they are unwanted.

3.5 PREVIOUS AND FUTURE USE

In our previous survey of malware detection in Android apps (98), we found that system calls were one of the preferred sources of information to profile malicious behavior in apps. However, the heterogeneity of the datasets commonly used in research makes comparing the effectiveness of different methods difficult. The considerable time required to generate a dataset of suitable size is also in hindrance to the development of the field. The use of *TwinDroid* can help address these issues.

In our subsequent research (97), we employed an early version of *TwinDroid* for malware detection in system call traces. An analysis of the pairs of benign and malicious traces present in *TwinDroid* allowed us to develop a trace abstraction process that highlights the differences between the two categories of traces. Performing malware detection on the abstracted traces was shown to improve detection accuracy.

As mentioned above, the fact that a large portion of TwinDroid consists of traces from pairs of otherwise identical benign and malicious apps gives us the means to study the low-level behavior of malware with a higher degree of precision than was previously possible. In particular, this dataset will make it possible to verify that malware detection algorithms really are detecting malware behaviors and are not simply sorting different apps based on other features.

Aside from security purposes, system traces are widely used for several other software engineering tasks, and TwinDroid is a suitable basis in this regard. In this regard, TwinDroid's trace generation pipeline will allow the dataset to remain up-to-date and relevant on a forward-going basis.

Amongst possible usages, we note the creation of pattern libraries that relate sequences of system calls to higher-order events (e.g., Opening a file or sending data to the network) (44). Such methods allow users to extract more information from the traces and open the door to further avenues of research, thus increasing the value of the dataset.

Karan et al. (35) show that the power consumption of a device can be estimated from the system calls performed, while Tian et al. rely upon patterns of system calls to detect code plagiarism (**?**). Karn et al. (61) showed that an analysis of the system calls performed by an IoT device can reveal the presence of crypto mining.

Syed et al. (119) use traces of system calls for remote attestation, the process of ensuring the trustworthiness of clients running software locally. Ezzati-Jivan (43) shows how system call traces can reveal the root cause of performance degradation in distributed systems.

Furthermore, the large time span of the apps traced in the dataset allows us to perform longitudinal studies on the evolution of these aspects of app development over the last several years.

CHAPTER 4

ÉTAT DE L'ACCEPTATION : ACCEPTÉE

CONFÉRENCE : IEEE SMC2023 HAWAII, ÉTATS-UNIS

TITRE EN FRANÇAIS

Comparaison de l'efficacité de la détection des logiciels malveillants statiques, dynamiques et hybrides sur un ensemble de données communes.

RÉSUMÉ

La détection des applications Android malveillantes est un enjeu de sécurité majeur. Un certain nombre de techniques fondées sur l'apprentissage automatique ont été proposées, et certaines d'entre elles ont atteint une grande précision. Cependant, la diversité des applications et la fréquence à laquelle de nouvelles familles de logiciels malveillants sont découvertes signifient que le problème reste non résolu. Cet article est basé sur nos recherches précédentes dans lesquelles nous avons présenté TwinDroid, un nouvel ensemble de données de traces d'exécution. TwinDroid, contient plus de 13 000 traces d'appels système provenant d'applications bénignes et malveillantes. Dans cet article, nous utilisons à la fois des analyses statiques, dynamiques et hybrides pour classer automatiquement les applications Android comme bénignes ou infectées. Nous avons constaté que l'analyse hybride présente les meilleurs résultats.

MOTS CLÉS

Détection de logiciels malveillants, Sécurité Android, Sécurité

CHAPTER 4

COMPARING THE EFFECTIVENESS OF STATIC, DYNAMIC AND HYBRID MALWARE DETECTION ON A COMMON DATASET

Asma Razgallah*, Raphaël Khoury*, Kobra Khanmohammadi, Christophe Pere*****
* Department of Computer Science and Engineering, Université du Québec en Outaouais, Canada
** Department Electrical and Computer Engineering, Concordia University Montreal, Canada
*** Department of Computer Science and Software Engineering, Université Laval Canada

The detection of malicious Android applications is a major security challenge. A number of machine learning-based techniques have been put forth, and some of them have attained great accuracy. However, the diversity of apps and frequency at which new malware families are found means that the issue remains unresolved. In this paper, we use both static, dynamic and hybrid analysis to automatically classify Android apps as benign or infected. We compare all three approaches on a common dataset — the TwinDroid dataset which contains over over 15,000 system call traces from over 9,000 benign and infected app, which allows comparison on equal footing. We make further contributions on the topic of feature selection and trace abstraction.

Index terms : malware detection, android security, security

4.1 INTRODUCTION

Nowadays, the number of smartphone users is constantly increasing worldwide. This increases the risk of attacks, especially on less secure systems like Android, even though it is the most widely used system in the world. According to Proofpoint (81) teams, attempted malware attacks on smartphones have increased by 500%. Phishing via SMS attacks were declining at the end of 2021, but have been steadily increasing since early 2022, it reaches $2,649$ billion SMS sent per week in April 2022[1]. Malware is spread via fake apps downloaded from

1. https://earthweb.com/smishing-statistics/

application stores. The Android Play Store is particularly targeted by cybercriminals because third-party apps can be downloaded without verification.

Apps are the basis of using the Android system. These apps can be infected via various methods, the most popular being installation via repackaging, update attacks, and adware. In the case of repackaging, the attacker downloads a popular app, disassembles it, adds malicious code to the app, recompiles it and published the malicious version on the app store. In an update attack, the .apk is disassembled and attached to a component that downloads a malicious update during execution. Adware is a particular class of malware in which the advertisement seen by the user of an app are controlled by an adversary in order to do malicious operations. These operations may be limited to showing Advertisements in a way that is not concordant with the Advertisement Network policies (e.g., AdMob[2]) or may include other malicious operations such as stealing users' confidential information (e.g., the device IMEI, the user account list and the device's).

Therefore, it is important to develop a method for detecting malicious applications. While there are several research papers ((101), (116), (141), (110), (48), etc.) on Android malware detection, attackers keep applying new methods to evade security. The most common approaches for malware detection can be broadly classified into three main categories : static, dynamic, and hybrid.

All research relies on apk datasets as initial sources. Researchers use these apks to extract the desired data and test their methods and processing techniques. However, for some research (such as dynamic analysis that relies on execution traces), this means spending additional time if the desired data is not already in a dataset. TwinDroid (**?**) fills this gap by providing a dataset with more than $15,000$ of system calls traces, from more than $9,000$ of apps, alongside with static data from these apps. The variety of apps used (benign, infected, pairs, old or new versions of apps) is an excellent basis for any research in Android smartphone security.

In our research, we used three of the most commonly used features for malware detection (98), namely : permissions, resource data (hardware and software data), and system calls to automatically detect malicious behavior in apps. In this article, we use these features to answer the following research questions :

— **RQ.1** : How do benign and malicious apps differ with respect to static data ?
— **RQ.2** : How do benign and malicious apps differ with respect to the system calls in the trace (dynamic analysis) ?
— **RQ.3** : Can statically and dynamically gathered data be used in tandem to improve the detection of malware ?

The main contributions of this paper are as follow :

— We compare the effectiveness of static, dynamic and hybrid detection methods, when operating on a common dataset.

2. `https://support.google.com/admob/answer/6128543?hl=en`

— Drawing upon a manual observation of the dataset, we show that extracting features from the parameters of system calls (which are normally ignored when performing dynamic analysis) enhances the classification's performance.
— We demonstrate the use of TwinDroid as tool for scientific investigation of malware. A large portion of our dataset is composed of benign\infected application pairs, which increases its value for research.

The remainder of this paper is organized as follows. In Section 4.2 we presented the TwinDroid dataset and examine case studies related to the data collected from our dataset. Our static, dynamic, and hybrid detection methods are developed in Sections 4.3, 4.4, and 4.5, respectively. The related works are discussed in Section 4.6. Concluding remarks are given in Section 4.7 .

4.2 DATASET DESCRIPTION

The TwinDroid dataset [3] is a publicly available dataset that contains execution traces of both benign and infected Android apps as well as a static data vector for each app traced. It also includes an automated pipeline for generating new traces from existing apps [4]. Apps were selected from four sources. Malicious apps were taken from the Androzoo database(4), which contains the largest number of available pair apps (original / repackaged), and from the Drebin database (15). We supplemented this selection with more recent benign apps from APK Pure [5] and the Google Store [6]. For each app, the dataset contains multiple traces of varying length. TwinDroid currently contains 15,000 traces.

The apps used to create TwinDroid are divided into two groups : the first consists in *app pairs*, i.e. pairs of otherwise identical benign and malicious apps. The dataset currently contains $2,288$ such pairs, consisting $2,288$ infected versions of 405 benign apps from Androzoo dataset. These pairs consist of benign and repackaged apps obtained by editing an app's original code via reverse engineering, adding malicious code, repackaging, and republishing the app. These apps were collected from the source repositories between 2016 to 2022. Note that a benign app may have multiple infected counterparts. The remainder of the dataset consists in unpaired recent 773 benign apps from Google Store, APK Pure and Androzoo, and $5,560$ infected apps, collected between 2012 and 2014 from drebin dataset.

Before using the apps, we performed an additional verification of its status as benign or malware, instead of relying on the benign\infected label provided by the source dataset from which the app came. This verification was performed using VirusTotal [7] a tool that includes several antivirus providers, such as McAfee, Symantec, Avast, etc.

3. `https://zenodo.org/record/6563328#.YpEDpajMLIU`
4. `https://github.com/RaphaelKhoury/automated-apk-tracing`
5. `https://apkpure.com/fr/`
6. `https://play.google.com/store/games?hl=fr&tab=w8`
7. `https://www.virustotal.com/gui/home/upload`

As shown in Table 4.1, VirusTotal correctly labels the apps from drebin as malicious apps, and the apps from APK Pure and Google Store as benign. However, out of 8,700 pairs from Androzoo, only 2,288 pairs are correctly identified, with the original app labeled as benign and the repackaged version is detected as malware, as we would expect. In 5,450 cases, both the original and repackaged apps are pinpointed as malware by VirusTotal. An additional 897 pairs appear to consist in two benign variants. Finally, in 65 cases, VirusTotal provides the counterintuitive result that the original app is malicious while the repackaged version is benign. An app is considered infected if it is detected as malicious by any one of VirusTotal's 13 antivirus programs. These results highlight the limitations of relying exclusively on antivirus solutions, and the need for the type of feature-based solutions explored in this paper.

TABLEAU 4.1 – Result of applying the VirusTotal scan to the apps in the TwinDroid dataset

Dataset	Number of pairs	class	benign	infected
Androzoo	2288	benign	X	
		infected		X
	5450	benign		X
		infected		X
	897	benign	X	
		infected	X	
	65	benign		X
		infected	X	
Dataset	**Number of apps**	**class**	**benign**	**infected**
Drebin	5560	infected		X
APK Pure	29	benign	X	
Google store	40	benign	X	

The most common malware type for the pairs in our dataset is adware. This is a particularly difficult type of malware to detect because of minimal amount of changes to the app's code that occur during the infection process, compared with other classes of malware (64). Apps infected with adware make up 78% of the infected apps, followed by the Trojan with a 14% share, as well as spyware with 4% of the samples, and the remaining 4% are types of worm, riskware, etc. This diversity allows users of the dataset to evaluate the effectiveness of the malware detection mechanisms they propose in a variety of contexts.

4.3 STATIC ANALYSIS

The static approach is based on features obtained from reverse engineering and studying the Android app's package file. This allows the extraction of information such as the intent, activities, the permissions requested during installation and the list of hardware resources used by the app. Static approaches are widely used in the literature (**?**). In this section, we perform malware detection on apps using static data extracted from the apps in the Twindroid dataset. More specifically, we perform classification based on the permissions and resources requested by the app. Following our previous research, we first perform a data abstraction phase, which we found increases the effectiveness of the classification process (97). This classification will serve as a baseline to evaluate the effectiveness of dynamic and hybrid classification in the next sections.

4.3.1 ABSTRACTION PROCESS

In the abstraction phase, we eliminate redundant and irrelevant static data by selecting a subset of the original, eliding datum that do not bear on the determination of an app's status as benign or malicious.

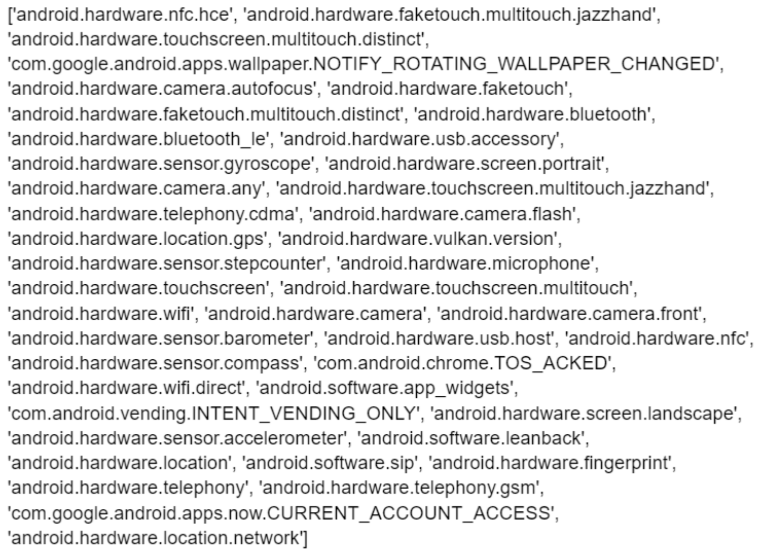
In this analysis, we were guided by the examination of the malware in our dataset as well as by the official Android documentation (36). Permissions are classified into three categories depending on their security sensitivity : normal, dangerous, and signature. After completing this phase, we chose to retain 162 of the 811 permissions, namely permissions at risk of "dangerous" and "signature" categories, which may affect the user's privacy, the operation of other applications, or the performance of the device.

We also extracted a list of resources for the apps. The resources are defined by <uses-features> in the AndroidManifest.xml file of every android app. Some features in the list of permissions and resources have a high correlation with each other. This is often the case because a certain permission is required to access a given resource. For example, the resource *(android.hardware.camera)* and the permission (*android.permission.CAMERA*) are 99% correlated. We need to consider only one of them to remove the redundancy. We thus start by identifying such highly correlated resources and permissions and keep only one of them. Note that these features are removed from the feature set even if they are highly correlated with the classification class. This is because the presence of a permission that strongly correlates with the resource is sufficient for the classification process.

All the features including the permissions and resources with their correlation are available in the author's repository [8].

We elided from the feature set 22 resources that had over over 50% correlation with a permission. This left a feature set of 162 permissions and 31 resources. The list of resource features that have less than 50% correlation with the rest of the resources is shown in Figure 4.1. It depicts that resources are not always correlated with permissions and including them as features will improve the classification model.

8. `https://github.com/AsmaLif/Permissions-and-resources-with-their-correlation`



```
['android.hardware.nfc.hce', 'android.hardware.faketouch.multitouch.jazzhand',
'android.hardware.touchscreen.multitouch.distinct',
'com.google.android.apps.wallpaper.NOTIFY_ROTATING_WALLPAPER_CHANGED',
'android.hardware.camera.autofocus', 'android.hardware.faketouch',
'android.hardware.faketouch.multitouch.distinct', 'android.hardware.bluetooth',
'android.hardware.bluetooth_le', 'android.hardware.usb.accessory',
'android.hardware.sensor.gyroscope', 'android.hardware.screen.portrait',
'android.hardware.camera.any', 'android.hardware.touchscreen.multitouch.jazzhand',
'android.hardware.telephony.cdma', 'android.hardware.camera.flash',
'android.hardware.location.gps', 'android.hardware.vulkan.version',
'android.hardware.sensor.stepcounter', 'android.hardware.microphone',
'android.hardware.touchscreen', 'android.hardware.touchscreen.multitouch',
'android.hardware.wifi', 'android.hardware.camera', 'android.hardware.camera.front',
'android.hardware.sensor.barometer', 'android.hardware.usb.host', 'android.hardware.nfc',
'android.hardware.sensor.compass', 'com.android.chrome.TOS_ACKED',
'android.hardware.wifi.direct', 'android.software.app_widgets',
'com.android.vending.INTENT_VENDING_ONLY', 'android.hardware.screen.landscape',
'android.hardware.sensor.accelerometer', 'android.software.leanback',
'android.hardware.location', 'android.software.sip', 'android.hardware.fingerprint',
'android.hardware.telephony', 'android.hardware.telephony.gsm',
'com.google.android.apps.now.CURRENT_ACCOUNT_ACCESS',
'android.hardware.location.network']
```

FIGURE 4.1 – List of the resource features that does not have above 50% correlation with any permission

We applied *mutual_info_classif()* from the *scikit-learn* library in Python to determine the correlation of features with the class of samples. Note that the class is the category of malware or benign samples. Figure 4.2 shows the top 20 features correlated with the class of samples. These features are the ones playing a key role in creating a model to detect malware versus benign applications. As it is shown in 4.1, some of these features related to screen usage, which is mostly available in benign apps, and not in malware samples. That is an original aspect of our classification approach. In most previous research (72), the features were selected by only examining malware samples. Contrary to them, we use features that identify benign apps as well.

4.3.2 CLASSIFICATION

The TwinDroid dataset contains a static data vector for each app that is constructed. Each position of the vector corresponds to permission or resource. A "1" indicates that the app uses the corresponding permission or resource while a "0" indicates that the static datum is not utilized by the app.

To measure the performance of our approach, we used the common 10-fold cross-validation method (23). We divided the data randomly into 10 subsets of approximately equal size. Setting aside one for testing, the remaining 9 subsets are combined into a training set (79). The calculation of the accuracy (ACC) rate or error of the classifier is estimated for the reserved subset. The overall accuracy is the weighted average of the 10 accuracy estimates. Following

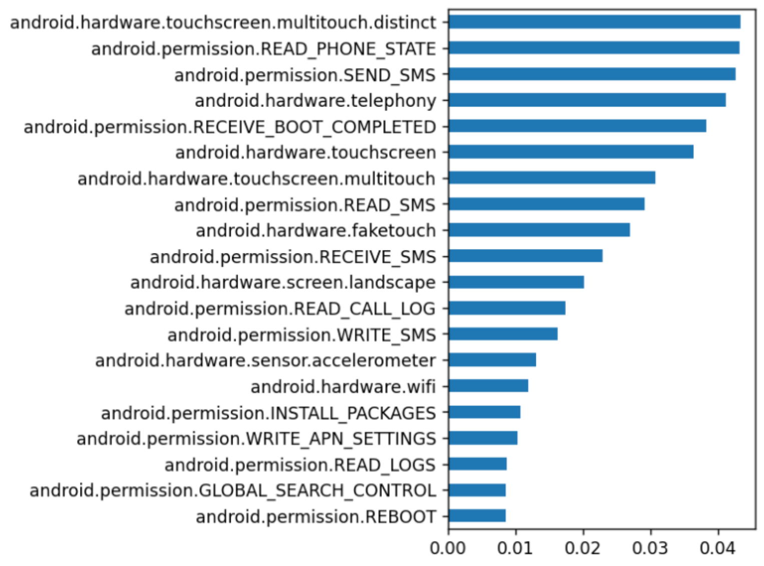


FIGURE 4.2 – The top 20 features correlated with the class of the samples, malware or benign class

common usage, the accuracy is the percentage of correctly identified apps (145). We also calculated the F-score to measure the accuracy of our model on our dataset and the Recall, a metric that indicates the number of correct positive predictions among all possible positive predictions.

We tested two algorithms SGD and RandomForest algorithms separately to classify the apps. We used the WEKA (133) tool to assign input data to one of the categories 'benign' or 'infected'. The results are shown in the table 4.2.

Algorithm	**ACC**		**F-score**		**Recall**		**Error**	
	yes abs.	No abs.	yes abs.	No abs.	yes abs.	No abs.	yes abs.	No abs.
SGD	92.65%	89.01%	92.70%	89%	92.70%	89%	7.35%	10.99%
RandomForest	93.44%	90.75%	93.40%	91%	93.40%	91%	6.56%	9.25%

TABLEAU 4.2 – Classification using statically gathered data of apps with (yes) and without (No) abstraction phase (abs.)

To validate the abstraction process used for this analysis, we tested app classification with and without the abstraction phase, in Table 4.2. With the abstraction phase, we achieved an accuracy of more than 3.37% and more than 2.07% compared to classifications without the abstraction phase using the SGD and RandomForest algorithms, respectively.These results are in line with those obtained by other similar studies (98), which provides evidence of the validity of using the TwinDroid dataset for malware classification research.

RQ 1 : How do benign and malicious apps differ with respect to the static data ?

Our findings in the abstraction phase show that not all the resources are highly correlated with permissions. Therefore we used the resources alongside with permissions as the feature set for the classification of benign and malware apps. Earlier research studies select the features by studying malware behaviors, while we used both benign and malware characteristics for identifying features. Our results compared favorably with that of previous research earlier studies (e.g., (101), (141), (5), (128), (145)).

4.4 DYNAMIC ANALYSIS

Dynamic analysis relies on the execution behavior of an application in real time in order to detect the ongoing execution of malware. Data collected during the execution, usually traces of system calls or API calls, forms an execution trace, that captures behavior of the program at runtime, and may reveal the presence of malware.

In this section, we present an abstraction method applicable to system call traces. It takes as input a fairly large raw execution trace, which makes direct analysis impossible, and returns a trace consisting of a sequence of smaller, more descriptive and expressive high-level operations. The abstraction phase is based on observations obtained by comparing pairs of traces. In addition, in this section we perform malware detection in traces and compare the classification of traces before and after abstraction.

4.4.1 ABSTRACTION PROCESS

In our previous study (97) we introduced a novel trace abstraction procedure that improves the detection of malicious behavior in traces. The abstraction process consists of eliding from the trace certain system calls that are not related to the behavior of known malware, and unifying equivalent system calls under a common name. For example, the system call ‘fstat’, can be found in various forms such as ‘stat’, ‘stat64’, ‘statfs64’, ‘asstat’, ‘lstat’, ‘lstat64’. We remove all of these forms and replace them with the canonical name ‘fstat’. In that study, trace classification was performed only on the basis of system call names, ignoring other information present in the trace such as parameters and return values. This is the strategy that is commonly used in dynamic analysis. In this section, we extend the previous study to include information derived from an analysis of the parameters of system calls.

One of the goals of the abstraction process is to reduce the size of the traces while maintaining the accuracy of the classification process. The abstraction phase allowing a 70% reduction in the size of the data (from 3.6 GB to 1.08 GB).

4.4.2 TRACE ANALYSIS

The parameters of system calls carry a number of highly specific information datum, such as file names when processing files, e.g. accessing files (open, stat), or memory addresses when writing to memory. When malicious behavior occurs, it is common for it to leave tell-tale signs in the system call parameters (69), a fact that has until now been ignored in the context of dynamic malware detection.

Our own investigation of the TwinDroid dataset provides evidence of possibility of relying upon system calls parameters to improve the detection of malware.

As an example, consider a pair of applications whose infected version belongs to the *fakeInstaller* malware family, a malware that sends SMS messages to premium rate numbers (100). According to a study by Malik et al. (77), the following system calls are associated with this malware family : *sendto, recvfrom, write, read, ioctl, stat, open*. We examined the parameters of these system calls traces from the TwinDroid dataset and made the following observations :

— The system call *(read)* attempts to read from file descriptor into a *buffer*. It was used only five times in the benign version of the application, to read the same file each time, and its parameter denoting the number of bytes to be read is small. However, in the infected version, the same system call is used 3332 times to read different files of different sizes. In addition, one can also observe a difference in the names of the memory buffer that appear in the parameters, with malicious traces sometimes using long and unreadable buffer names that contain special characters, such as the name “$\}\%227/211?4\S\&2222 = d\xi20222261(61E$”.

— The system call *(recvfrom)* allows the application to receive data from a socket. In the benign version, the buffer names in which the data is stored are generally readable, but in the infected version, most buffers names contain special characters. Otherwise, several buffers in the infected version have names that start with *“HTTP/1.1"*, which indicates that they relate to the outside environment. The same address was used in the *(sendto)* system call.

— The *(writev)* system call writes data into multiple buffers. The data parameters of the infected version indicate that the requested file was blocked and written to the stream (”*System.err*”). This does not occur in traces from benign apps.

— The *(stat)* system call retrieves information about the file pointed by *pathname* parameter. It appears in the infected version, but not in the benign version. It returns the status of a file. We further observed that in the trace of the malicious app that *(stat)* is requested to obtain information about .apk files different than the current app.

— The *(open)* system call opens the file specified by *pathname* parameter. In the benign version of the application opens a file in the /dev directory. However, in the infected version, the files that are requested are in the /sdcard directory. Thus, the malware consults files stored on the device’s SD card.

As a second example, consider a pair of apps, one of which is infected with malware from the

Feature	Description
WWW.	sends data to a remote server.
HTTP/1.1	allows the use of external data.
System.err	reading the requested file was blocked and data was written to the stream (”*System.err*”).
/sdcard	consults files stored on the device’s SD card.
com.	name of a new application to be installed on the device.
special characters	unreadable buffer names.

TABLEAU 4.3 – List of parameters used for classification as supplementary features.

Plankton malware family. This malware uses *(recvfrom)* and *(sendTo)* system calls for sending SMS to premium services owned by malware author. The comparison of the two versions at the parameter level revealed the following information :

— When the Plankton malware makes use of the *(recvfrom)* system call, it stores data in buffers whose names contain unreadable special characters. This system call is also twice as common in traces from malicious apps.

— The *(sendto)* system call sends a message over a socket in connection mode. The second parameter of this system call specifies the name of the *buffer* containing the message to be sent. In the trace of the infected version of the app, the parameter of this system call contains names that start with *WWW*, which indicates sending information from the device to the remote server.

These observations allowed us to conclude that the malware can present itself not only through the system call, but also through the parameters. Therefore, and based on these observations, we add features derived from an analysis of these parameters along with the system calls themselves to classify the application as benign or infected. Table 4.3 lists these new features.

4.4.3 CLASSIFICATION

Based on the data about system calls and parameters, we created a binary vector for each application. In each vector, the value ‘1’ means that this system call or parameter is present in the application, otherwise ‘0’.

To classify the apps, we used the same rate estimation method as for the static data presented above and the same classification algorithms. Otherwise, we used 10-fold cross-validation with the SGD and RandomForest algorithms.

The results obtained with classification in Table 4.4 compare favorably with those obtained with static data. When using the RandomForest algorithm, the accuracy does not exceed 97.70%. This result is consistent with previous studies (e.g., (85)) that found that SGD outperforms the

Algorithm	**ACC**		**F-score**		**Recall**		**Error**	
	yes par.	No par.	yes par.	No par.	yes par.	No par.	yes par.	No par.
SGD	98.49%	95.12%	98.50%	95.20%	98.50%	95.20%	1.51%	4.8%
RandomForest	97.70%	92.63%	97.70%	92.71%	97.70%	92.71%	2.30%	7.4%

TABLEAU 4.4 – The classification rates for dynamic data of apps with (yes) and without (No) parameters (par.)

Random Forest if the features are sufficiently tuned. Our experiments also confirm that SGD provides better accuracy where we select the features based on a study over the correlation between the features and samples' class.

In order to determine the impact of the parameter-derived features, we performed the same experiment, on the the same dataset, using only the occurrence of canonized system calls in the trace as a feature. The results are reproduced Table 4.4. Without the parameter-derived features, the accuracy was 3% lower when using the SGD algorithm and 5% with the RandomForest algorithm.

RQ 2 : How do benign and malicious apps differ with respect to the system calls in the trace ?

Dynamic malware detection, using the system calls present in the trace outperforms detection based on statically obtainable features, such as permissions and resource usage. We further found that security-relevant information can be derived from an analysis of the parameters of the system call, which all previous dynamic detection mechanisms tended to omit from their investigations.

4.5 HYBRID ANALYSIS

In this section, we present a novel hybrid method that combines the features of static and dynamic analysis from the previous two sections to provide more accurate and effective detection. In practice, a malware detection method based on this paradigm would first perform a static analysis of a newly downloaded app at the moment it is installed. The information it obtains during this phase will then supplement a runtime analysis of the program, thus providing a more revealing picture of the underlying program. Alternatively, a static detection method could run an app in a simulated environment and aggregate static data with data derived from monitoring the simulated execution before making a decision with respect to whether or not the app should be allowed to be installed on the host system.

In this section, we present the result of performing malware detection on the TwinDroid dataset, using a feature vector built from a combination of the static features described in

Section 4.3 and the dynamic features described in section 4.4.

Classification

We once again used the SGD and RandomForest algorithms. Results are given in Table 4.5. The classification of applications reaches 99.85% with an error of 0.15% with the SGD algorithm, thus outperforming both static and dynamic analysis. This shows the complementary between the type of information obtained through the two types of analyses. Hybrid analysis is the least commonly used type of malware detection in Android (98) and this results strongly hints at the necessity to pursue this line of enquiry.

Algorithm	**ACC**	**F-score**	**Recall**	**Error**
SGD	99.85%	99.80%	99.80%	0.15%
RandomForest	98.87%	98.87%	98.87%	1.13%

TABLEAU 4.5 – Accuracy and precision of hybrid classification.

RQ 3 : Can statically and dynamically gathered data be used in tandem to improve the detection of malware ?

Hybrid analysis outperforms both static and dynamic analysis for malware detection, which indicates that the two types of analyses gather complementary information. Further research is needed to determine the optimal way to incorporate static and dynamic data to achieve a synergistic detection of malware.

4.6 RELATED WORKS

Several studies have drawn upon diverse types of static analysis to detecting malware for Android applications. These methods differ in part with respect to the features, or statically computed element, which is relied upon for classification. Şahın *et al.* (101) used a linear regression model for Android malware detection based on permissions.Muhammed *et al.* (116) extracted Android permissions and applied the Chi-Square test algorithms and Fisher's exact test to rank and filter features. They then used machine learning algorithms to detect malicious apps.An extensible prototype of a feature selection algorithm has been presented in (135). The functionalities include permissions, Intent, and Opcode.Kang *et al.* (60) used the 10-gram opcode features and machine learning to identify and categorize Android malware.

There are several malware detection studies based on dynamic analysis. R. Surendran *et al.* (117) proposed a detection mechanism based on the occurrence of malicious system calls in the system call sequence of an Android app. A detection mechanism by analyzing the frequency of system calls was proposed by Amamra *et al.* (8). They used a binary machine learning classifier trained with the frequency of system calls generated by known malware and software applications. The classifier is designed to predict whether the frequency of system calls generated by an unknown application corresponds to malware or not. S. Shakya and M. Dave (112) executed the malicious application in a monitored and controlled environment. The system calls collected are analyzed and fed to various machine learning models for malware family detection and classification. A new method with dynamic analysis was proposed by A. Razgallah and R. Khoury (97). They divided each trace of system calls into k segments and then determined in which segment the first occurrence of each system call could be found. The authors found that the system calls consistent with the presence of malware occurred for the first time at the onset of the infected trace and at the end of the benign version.

A smaller number of studies have suggested hybrid detection methods. Wen *et al.* (132) combine static and dynamic analysis, relying on a variety of features including battery usage, processes, API, intentions, and permissions features. Tong and Yan (124) suggested another hybrid approach. Android apps generated patterns of system calls related to file and network accesses. To evaluate the unidentified app, they then compared with both the infected and benign pattern sets. Wang *et al.* (130) proposed the hybrid method called FGL_Droid, which merges the dynamic API call sequence into a function call graph and the extracted permission request features to perform accurate malware detection.

4.7 CONCLUSION

In this study, we use the TwinDroid dataset as a common basis for the comparison of the effectiveness of static, dynamic and hybrid analysis for malware detection on Android apps. We found that dynamic analysis seems to outperform static analysis, and that hybrid analysis outperform both. We also obtained novel results related to feature selection in static analysis and the use of system call parameters in dynamic analysis.

CONCLUSION ET RECOMMANDATIONS

CONCLUSION

L'adoption des Smartphones, des appareils qui passent d'appareils de communication simples à des appareils «intelligents» et multifonctionnels, ne cesse d'augmenter. C'est pourquoi il devient de plus en plus une cible de diverses attaques, qui causent la fuite de la vie privée de l'utilisateur, les frais de service supplémentaires et l'épuisement de la puissance de la batterie.

Dans notre projet, nous avons choisi la plateforme **Android** puisqu'elle connaît une croissance trois fois supérieure à celle d'Apple, qui occupe la deuxième place dans le marché des Smartphones. De plus, contrairement à Apple qui peut vérifier chaque application disponible manuellement par des experts en sécurité logicielle, Android manque un processus complet d'inspection des applications avant que les applications ne soient publiées sur le marché. Google adopte un mécanisme passif fondé sur les autorisations. Chaque fois que l'utilisateur installe une nouvelle application, il doit lui-même d'approuver ou rejeter toutes les autorisations demandées par l'application. Dès qu'une application sera installée, elle aura accès aux ressources approuvées. Dans ce cas le système n'aura pas le contrôle sur l'usage des ressources. Aussi si une application est signalée comme étant un logiciel malveillant par les utilisateurs, elle sera supprimée.

Pour ces raisons, plusieurs recherches ont été faites dans le but de faciliter la détection des comportements malveillantes pour les applications Android, afin d'éviter les vulnérabilités face à des menaces telles que les accès aux informations personnelles et les modifications auprès de systèmes non autorisés.
Dans cette thèse tout d'abord, on a passé en revue les solutions de sécurité actuelles pour les téléphones intelligents Android en se concentrant sur les mécanismes et les approches existants. Pour le reste de la thèse les principaux résultats obtenus au cours de ce projet sont :

— La création d'une base de données, «TwinDroid», présente un pipeline entièrement automatisé pour générer de nouvelles traces à partir d'applications tirées de plusieurs sources. Tout chercheur ayant besoin de traces d'application peut utiliser ce pipeline pour générer des traces adaptées à ses besoins spécifiques. Cette facilité d'extension permet à la base de données de rester à jour avec les logiciels malveillants émergents.

— Un nouveau processus d'abstraction proposé met en évidence les fonctionnalités qui distinguent le plus facilement les fonctionnalités bénignes et malveillantes. Ce processus est responsable de l'amélioration de la précision du processus de détection. Nous avons testé son efficacité en classifiant nos applications avec et sans ce processus. Les résultats obtenus après la phase d'abstraction sont toujours plus élevés que la classification sans l'abstraction.

— La réplication de plusieurs techniques de détection de logiciels malveillants sur le même ensemble de données permet une comparaison sur un pied d'égalité. En testant chaque expérience, nous avons observé que des résultats optimaux étaient obtenus en considérant le 3 grammes des appels système.

— En étudiant les paires de traces bénignes et infectées dans notre base de données, nous avons trouvé que l'attaque a souvent lieu au début de l'exécution. Nous avons proposé une méthode de détection de logiciels malveillants dans les traces, fondée sur cette observation. Nous avons divisé chaque trace en un nombre donné (k) de segments. Nous avons testé k avec les valeurs 5, 10, 30, 50 et 100 segments. Ensuite, nous avons enregistré dans quel segment se trouve la première occurrence de chaque appel système dans notre base de données. Nous avons utilisé plusieurs algorithmes de classification pour vérifier notre modèle, la meilleure classification des traces bénignes et infectées a été obtenue en divisant les traces en 30 segments. Les algorithmes LMT et J48 ont réussi à classer toutes les traces infectées avec un rappel de 100%.

— La détection dynamique des logiciels malveillants, à l'aide des appels système présents dans la trace, surpasse la détection basée sur des fonctionnalités pouvant être obtenues de manière statique, telles que les autorisations et l'utilisation des ressources. Lors de la classification avec des données dynamiques, l'ACC le plus bas était de 97,70%, contre 92,65% lors de la classification avec des données statiques. Nous avons en outre découvert que les informations relatives à la sécurité peuvent être dérivées d'une analyse des paramètres de l'appel système, que tous les mécanismes de détection dynamique précédents avaient tendance à omettre de leurs investigations.

— L'analyse hybride surpasse à la fois l'analyse statique et dynamique pour la détection des logiciels malveillants, ce qui indique que les deux types d'analyses recueillent des informations complémentaires.

Pour finir, dans cette thèse, l'ensemble des objectifs fixés ont été atteints avec succès. Ce projet a démontré l'importance de l'utilisation des données statique et dynamique combinées pour avoir plus de performance à la détection de malware dans les applications Android. De plus, l'analyse des traces d'exécution nous a permet de détecter l'existence des comportements malveillants ce qui démontre l'utilité de l'utilisation de ses traces dans nos expérimentations.

RECOMMANDATIONS

Pour donner suite à la réalisation de cette étude, une série de recommandations pour des travaux futures est établie ci-dessous :

- Implémenter des applications pratiques pour notre travail, telles que l'intégration de notre approche dans les outils de sécurité des applications.
- Définir quand et où dans le cycle de vie d'une application la technique de détection doit être appliquée.
- Corréler les comportements observés avec la fonctionnalité prévue de l'application afin d'obtenir une définition plus précise de ce qui est suspect.
- Résoudre le problème de la détection de l'émulateur lors des tests dans un environnement simulé.
- Expérimenter d'autres fonctionnalités au-delà des appels système et des permissions pour la détection de malware dans une application Android en utilisant TwinDroid.

BIBLIOGRAPHIE

[1] 2014. Genymotion android emulator. `https://www.genymotion.com/`.

[2] Aafer, Y., W. Du, et H. Yin. 2013. « DroidAPIMiner : Mining api-level features for robust malware detection in android ». In *Security and Privacy in Communication Networks - 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013, Revised Selected Papers*, p. 86–103.

[3] Aggarwal, K., C. Zhang, J. C. Campbell, A. Hindle, et E. Stroulia. 2014. « The power of system call traces : predicting the software energy consumption impact of changes. ». In *CASCON*. T. 14, p. 219–233.

[4] Allix, K., T. F. Bissyandé, J. Klein, et Y. L. Traon. 2016. « Androzoo : Collecting millions of Android apps for the research community ». In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, p. 468–471.

[5] Alshahrani, H., H. Mansourt, S. Thorn, A. Alshehri, A. Alzahrani, et H. Fu. 2018. « Ddefender : Android application threat detection using static and dynamic analysis ». In *2018 IEEE Inter. Conf. on Consumer Electronics (ICCE)*, p. 1–6. IEEE.

[6] Altman, N. S. 1992. « An introduction to kernel and nearest-neighbor nonparametric regression », *j-AMER-STAT*, vol. 46, no. 3, p. 175–185.

[7] Alzahrani, N. et D. Alghazzawi. 2019. « A review on android ransomware detection using deep learning techniques ». In *Proceedings of the 11th International Conference on Management of Digital EcoSystems*, p. 330–335.

[8] Amamra, A., J.-M. Robert, A. Abraham, et C. Talhi. 2016. « Generative versus discriminative classifiers for android anomaly-based detection system using system calls filtering and abstraction process », *Security and Communication Networks*, vol. 9, no. 16, p. 3483–3495.

[9] Amamra, A., J.-M. Robert, et C. Talhi. 2015. « Enhancing malware detection for android systems using a system call filtering and abstraction process », *Security and Comm. Networks*, vol. 8, p. 1179–1192.

[10] Ananya, A., A. Aswathy, T. Amal, P. Swathy, P. Vinod, et S. Mohammad. 2020. « Sysdroid : a dynamic ml-based android malware analyzer using system call traces », *Cluster Computing*, p. 1–20.

[Android] Android, D. strace - linux syscall tracer. `https://strace.io/`.

[12] ———. 2020. Ui/application exerciser monkey. `https://developer.android.com/studio/test/monkey`.

[13] APKPure. 2014-2022a. `https://apkpure.com/`.

[14] ———. 2022b. Apkpure. `https://apkpure.com/fr/`.

[15] Arp, D., M. Spreitzenbarth, M. Hübner, H. Gascon, et K. Rieck. 2014. « DREBIN : effective and explainable detection of android malware in your pocket ». In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society.

[16] Arshad, S., M. A. Shah, A. Khan, et M. Ahmed. 2016. « Android malware detection & protection : a survey », *International Journal of Advanced Computer Science and Applications*, vol. 7, no. 2, p. 463–475.

[17] AV-Comparatives. 2019. Android test 2019 – 250 apps. `https://www.av-comparatives.org/tests/android-test-2019-250-apps/`.

[18] Bathelot, B. 2018. Définition : Sdk publicitaire. `https://www.definitions-marketing.com/definition/sdk-publicitaire/`.

[19] Beky, A. 2007. L'iphone sera officiellement lancé le 29 juin 2007. `https://www.clubic.com/actualite-74561-iphone-officiellement-29-2007.html`.

[20] Bhatia, T. et R. Kaushal. 2017. « Malware detection in android based on dynamic analysis ». In *2017 International Conference on Cyber Security And Protection Of Digital Services (Cyber Security)*, p. 1–6. IEEE.

[21] Bläsing, T., L. Batyuk, A. Schmidt, S. A. Çamtepe, et S. Albayrak. 2010. « An Android application sandbox system for suspicious software detection ». In *5th International Conference on Malicious and Unwanted Software, MALWARE 2010, Nancy, France, October 19-20, 2010*, p. 55–62.

[22] Breiman, L. 2001. « Random forests », *Mach. Learn.*, vol. 45, no. 1, p. 5–32.

[23] Breiman, L., J. H. Friedman, R. A. Olshen, et C. J. Stone. 2017. *Classification and regression trees*. Routledge.

[24] Bugiel, S., L. Davi, A. Dmitrienko, T. Fischer, et A.-R. Sadeghi. 2011. XManDroid : A new Android evolution to mitigate privilege escalation attacks. Rapport no. TR-2011-04, Technische Universität Darmstadt.

[25] Burguera, I., U. Zurutuza, et S. Nadjm-Tehrani. 2011. « Crowdroid : behavior-based malware detection system for android ». In *SPSM'11, Proceedings of the 1st ACM Workshop Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2011, October 17, 2011, Chicago, IL, USA*, p. 15–26.

[26] Canfora, G., E. Medvet, F. Mercaldo, et C. A. Visaggio. 2015a. « Detecting android malware using sequences of system calls ». In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, p. 13–20.

[27] ———. 2015b. « Detecting android malware using sequences of system calls ». In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, p. 13–20.

[28] Cavnar, W. B. et J. M. Trenkle. 1994. « N-gram-based text categorization ». In *Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, p. 161–175, Las Vegas, US.

[29] Chaba, S., R. Kumar, R. Pant, et M. Dave. 2017. « Malware detection approach for Android systems using system call logs », *CoRR*, vol. abs/1709.08805.

[30] Chen, J., M. H. Alalfi, T. R. Dean, et Y. Zou. 2015. « Detecting Android malware using clone detection », *J. Comput. Sci. Technol.*, vol. 30, no. 5, p. 942–956.

[31] Chen, T., Q. Mao, Y. Yang, M. Lv, et J. Zhu. 2018. « Tinydroid : A lightweight and efficient model for Android malware detection and classification », *Mobile Information Systems*, vol. 2018, p. 4157156 :1–4157156 :9.

[32] Christian, H., M. P. Agus, et D. Suhartono. 2016. « Single document automatic text summarization using term frequency-inverse document frequency (tf-idf) », *ComTech : Computer, Mathematics and Engineering Applications*, vol. 7, no. 4, p. 285–294.

[33] Clover, J. 2019. Apple shipped an estimated 36.4 million iphones worldwide in q1 2019, a 30% year-over-year decline. `https://www.macrumors.com/2019/04/30/apple-36-million-iphones-shipped-q1-2019/`.

[34] Dantzer, A. 2019. Le téléphone portable ordinateur nokia 9000 communicator de 1996. `https://www.thevintagenews.com/2019/02/13/first-cell-phone/`.

[35] Das, P. K., A. Joshi, et T. Finin. 2017. « App behavioral analysis using system calls ». In *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, p. 487–492. IEEE.

[36] Developers, G. 2022. Documentation for app developers. `https://developer.Android.com/docs`.

[37] Dimitri, T. 2014. Le premier smartphone a 20 ans (oui, déjà !). `https://www.generation-nt.com/premier-smartphone-ibm-angler-simon-aout-1994-actualite-1905520.html`.

[38] Dimjašević, M., S. Atzeni, I. Ugrina, et Z. Rakamaric. 2016. « Evaluation of android malware detection based on system calls ». In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, p. 1–8.

[39] Edjlali, G., A. Acharya, et V. Chaudhary. 1999. « History-based access control for mobile code ». In *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*, p. 413–431.

[40] Elish, K. O., D. Yao, et B. G. Ryder. 2015. « On the need of precise inter-app icc classification for detecting Android malware collusions ». In *Proceedings of IEEE mobile security technologies (MoST), in conjunction with the IEEE symposium on security and privacy*.

[41] Enck, W., M. Ongtang, et P. McDaniel. 2009. « On lightweight mobile phone application certification ». In *Proceedings of the 16th ACM Conference on Computer and Communications Security*. Coll. « CCS '09 », p. 235–245, New York, NY, USA. ACM.

[42] Ezzati-Jivan, N. et M. R. Dagenais. 2017. « Multi-scale navigation of large trace data : A survey », *Concurrency and Computation : Practice and Experience*, vol. 29, no. 10, p. e4068.

[43] Ezzati-Jivan, N., H. Daoud, et M. R. Dagenais. 2021. « Debugging of performance degradation in distributed requests handling using multilevel trace analysis », *Wireless Communications and Mobile Computing*, vol. 2021.

[44] Fadel, W. 2010. Techniques for the abstraction of system call traces to facilitate the understanding of the behavioural aspects of the linux kernel. `https://users.encs.concordia.ca/~abdelw/papers/Fadel_MASc_S2011.pdf`.

[45] Fan, M., J. Liu, X. Luo, K. Chen, T. Chen, Z. Tian, X. Zhang, Q. Zheng, et T. Liu. 2016. « Frequent subgraph based familial classification of android malware », *27th International Symposium on Software Reliability Engineering*, p. 24–35.

[46] Felt, A., M. Finifter, E. Chin, S. Hanna, et D. Wagner. 2011. « A survey of mobile malware in the wild », *Proceedings of the ACM Conference on Computer and Communications Security*.

[47] Feng, P., J. Ma, C. Sun, X. Xu, et Y. Ma. 2018a. « A novel dynamic Android malware detection system with ensemble learning », *IEEE Access*, vol. 6, p. 30996–31011.

[48] ———. 2018b. « A novel dynamic android malware detection system with ensemble learning », *IEEE Access*, vol. 6, p. 30996–31011.

[49] Geurts, P., D. Ernst, et L. Wehenkel. 2006. « Extremely randomized trees », *Machine Learning*, vol. 63, no. 1, p. 3–42.

[50] Goodfellow, I., Y. Bengio, et A. Courville. 2016. *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

[51] Google. 2008. Google play. `https://play.google.com/store/apps?hl=fr`.

[52] Hall, M., E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, et I. H. Witten. 2009. « The weka data mining software : an update », *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, p. 10–18.

[53] Hallé, S., R. Khoury, et M. Awesso. 2018. « Streamlining the inclusion of computer experiments in a research paper », *IEEE Computer*, vol. 51, no. 11, p. 78–89.

[54] Hamou-Lhadj, A., S. S. Murtaza, W. Fadel, A. Mehrabian, M. Couture, et R. Khoury. 2013. « Software behaviour correlation in a redundant and diverse environment using the concept of trace abstraction ». In *2013 International Conference on Reliable And Convergent Systems (ACM RACS 2013)*.

[55] Han, L., Q. Zhou, J. Zhang, X. Yang, R. Zhou, et J. Tang. 2020. « Polymorphism and consistency : Complex network based on execution trace of system calls in linux kernels », *International Journal of Modern Physics C*, vol. 31, no. 09, p. 2050126.

[56] Hochreiter, S. et J. Schmidhuber. 1997. « Long short-term memory », *Neural computation*, vol. 9, p. 1735–80.

[57] Hofmeyr, S. A., S. Forrest, et A. Somayaji. 1998. « Intrusion detection using sequences of system calls », *Journal of computer security*, vol. 6, no. 3, p. 151–180.

[58] Jerome, Q., K. Allix, R. State, et T. Engel. 2014. « Using opcode-sequences to detect malicious android applications ». In *2014 IEEE International Conference on Communications (ICC)*, p. 914–919. IEEE.

[59] Jiang, X. 2011. Security alert : New sophisticated Android malware droidkungfu found in alternative chinese app markets. `https://www.csc2.ncsu.edu/faculty/xjiang4/DroidKungFu.html`.

[60] Kang, B., S. Y. Yerima, S. Sezer, et K. McLaughlin. 2016. « N-gram opcode analysis for android malware detection », *arXiv preprint arXiv :1612.01445*.

[61] Karn, R. R., P. Kudva, H. Huang, S. Suneja, et I. M. Elfadel. 2021. « Cryptomining detection in container clouds using system calls and explainable machine learning », *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, p. 674–691.

[62] Kerrisk, M. 2019. Linux man pages project. `http://man7.org/linux/man-pages/dir_section_2.html`.

[63] Khanmohammadi, K., N. Ebrahimi, A. Hamou-Lhadj, et R. Khoury. 2019a. « Empirical study of android repackaged applications », *Empirical Software Engineering*.

[64] ———. 2019b. « Empirical study of android repackaged applications », *Empirical Software Engineering*, vol. 24, no. 6, p. 3587–3629.

[65] Khanmohammadi, K., R. Khoury, et A. Hamou-Lhadj. 2019c. « On the use of api calls to detect repackaged malware apps : Challenges and ideas ». In *The 30th International Symposium on Software Reliability Engineering (ISSRE 2019)*.

[66] Khoury, R., B. Vignau, S. Hallé, A. Hamou-Lhadj, et A. Razgallah. 2021. « An analysis of the use of cves by iot malware ». In *Foundations and Practice of Security : 13th International Symposium, FPS 2020, Montreal, QC, Canada, December 1–3, 2020, Revised Selected Papers 13*, p. 47–62. Springer.

[67] Kim, C. W. 2018. « NtMalDetect : A machine learning approach to malware detection using native api system calls », *arXiv preprint arXiv :1802.05412*.

[68] Krajci, I. et D. Cummings. 2013. *History and Evolution of the Android OS*. Apress, Berkeley, CA.

[69] Kruegel, C., D. Mutz, F. Valeur, et G. Vigna. 2003. « On the detection of anomalous system call arguments ». In *European Symposium on Research in Computer Security*, p. 326–343. Springer.

[70] lilian MORER. 2015. Le téléphone portable ordinateur nokia 9000 communicator de 1996. `http://www.mobilophiles.com/pages/Le_telephone_portable_ordinateur_Nokia_9000_communicator_de_1996-3711919.html`.

[71] Lin, Y., Y. Lai, C. Chen, et H. Tsai. 2013. « Identifying android malicious repackaged applications by thread-grained system call sequences », *Computers & Security*, vol. 39, p. 340–350.

[72] Liu, K., S. Xu, G. Xu, M. Zhang, D. Sun, et H. Liu. 2020a. « A review of android malware detection approaches based on machine learning », *IEEE Access*, vol. 8, p. 124579–124607.

[73] Liu, P., W. Wang, X. Luo, H. Wang, et C. Liu. 2020b. « Nsdroid : efficient multi-classification of android malware using neighborhood signature in local function call graphs », *International Journal of Information Security*.

[74] Louvet, J.-P. et B. Chervet. 2004. Cabir, premier virus pour téléphone portable se propageant par réseau. `https://www.futura-sciences.com/tech/actualites/tech-cabir-premier-virus-telephone-portable-propageant-reseau-3874/`.

[75] Maiorca, D., D. Ariu, I. Corona, M. Aresu, et G. Giacinto. 2015. « Stealth attacks : An extended insight into the obfuscation effects on android malware », *Computers Security*, vol. 51, p. 16 – 31.

[76] Malik, S. et K. Khatter. 2016a. « System call analysis of android malware families », *Indian Journal of Science and Technology*, vol. 9, no. 21.

[77] ———. 2016b. « System call analysis of android malware families », *Indian Journal of Science and Technology*, vol. 9, no. 21, p. 1–13.

[78] man pages, L. 2020. Linux/unix system programming training. `https://man7.org/linux/man-pages/dir_section_2.html`.

[79] Martin, J. K. et D. S. Hirschberg. 1996. « Small sample statistics for classification error rates i : Error rate measurements ».

[80] Mauerer, W. 2008. *Professional Linux Kernel Architecture*. Birmingham, UK, UK : Wrox Press Ltd.

[81] McNeil, A. et W. S. Jones. 2022. Mobile malware is surging in europe : A look at the biggest threats. `https://www.proofpoint.com/us/blog/email-and-cloud-threats/mobile-malware-surging-europe-look-biggest-threats`.

[82] Medina, L. V. M. et S. J. Rueda. 2014. « Identifying Android malware instructions ». In *IEEE Latin-America Conference on Communications, LATINCOM 2014, Cartagena de Indias, Colombia, November 5-7, 2014*, p. 1–7.

[83] Meier, R. 2010. *Développement d'application professionelles avec Android 2*. Pearson France.

[84] Naway, A. et Y. Li. 2018. « A review on the use of deep learning in android malware detection », *arXiv preprint arXiv :1812.10360*.

[85] Nhat-Duc, H. et T. Van-Duc. 2023. « Comparison of histogram-based gradient boosting classification machine, random forest, and deep convolutional neural network for pavement raveling severity classification », *Automation in Construction*, vol. 148, p. 104767.

[86] Pan, X., X. Wang, Y. Duan, X. Wang, et H. Yin. 2017. « Dark hazard : Learning-based, large-scale discovery of hidden sensitive operations in Android apps ».

[87] Parkour, M. 2008. Contagio mobile. `http://contagiodump.blogspot.com/`.

[88] Peng, H., C. S. Gates, B. P. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, et I. Molloy. 2012. « Using probabilistic generative models for ranking risks of android apps ». In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, p. 241–252.

[89] Perez, S. 2010. Tap snake game in Android market is actually spy app (update). `readwrite.com/2010/08/17/tap_snake_game_in_android_market_is_actually_spy_app`.

[90] Pichai, S. 2009. Introducing the google chrome os. `https://googleblog.blogspot.com/2009/07/introducing-google-chrome-os.html`.

[91] Portokalidis, G., P. Homburg, K. Anagnostakis, et H. Bos. 2010. « Paranoid android : versatile protection for smartphones ». In *Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6-10 December 2010*, p. 347–356.

[92] Potharaju, R., A. Newell, C. Nita-Rotaru, et X. Zhang. 2012. « Plagiarizing smartphone applications : Attack strategies and defense techniques ». In *Engineering Secure Software and Systems - 4th International Symposium, ESSoS 2012, Eindhoven, The Netherlands, February, 16-17, 2012. Proceedings*, p. 106–120.

[93] Qiao, M., A. H. Sung, et Q. Liu. 2016. « Merging permission and API features for Android malware detection ». In *5th IIAI International Congress on Advanced Applied Informatics, IIAI-AAI 2016, Kumamoto, Japan, July 10-14, 2016*, p. 566–571.

[94] Quinlan, J. R. 1986. « Induction of decision trees », *Machine learning*, vol. 1, p. 81–106.

[95] Ramos, J. et al. 2003. « Using tf-idf to determine word relevance in document queries ». In *Proceedings of the first instructional conference on machine learning*. T. 242, p. 29–48. Citeseer.

[96] Razagallah, A., R. Khoury, et J.-B. Poulet. 2022. « Twindroid : A dataset of android app system call traces and trace generation pipeline ». In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, p. 591–595.

[97] Razgallah, A. et R. Khoury. 2021. « Behavioral classification of android applications using system calls », *Proceedings of the 28th Asia-Pacific Software Engineering Conference, December 6-9, 2021*, p. 73.

[98] Razgallah, A., R. Khoury, S. Hallé, et K. Khanmohammadi. 2021. « A survey of malware detection in android apps : Recommendations and perspectives for future research », *Computer Science Review*, vol. 39, p. 100358.

[99] Report, E. M. 2016. 5g subscriptions to reach half a billion in 2022 : Ericsson mobility report. https://www.ericsson.com/en/press-releases/2016/11/5g-subscriptions-to-reach-half-a-billion-in-2022-ericsson-mobility-report.

[100] Ruiz, F. 2012. ‘fakeinstaller’ leads the attack on android phones. https://www.mcafee.com/blogs/mobile-security/fakeinstaller-leads-the-attack-on-Android-phones/.

[101] Şahın, D. Ö., S. Akleylek, et E. Kiliç. 2022. « Linregdroid : Detection of android malware using multiple linear regression models-based classifiers », *IEEE Access*, vol. 10, p. 14246–14259.

[102] Salah, A., E. Shalabi, et W. Khedr. 2020. « A lightweight android malware classifier using novel feature selection methods », *Symmetry*, vol. 12, no. 5, p. 858.

[103] Saltzer, J. H. et M. D. Schroeder. 1974. « The protection of information in computer systems », *Communications of the ACM*, vol. 17.

[104] Santos, I., Y. K. Penya, J. Devesa, et P. G. Bringas. 2009. « N-grams-based file signatures for malware detection. », *ICEIS (2)*, vol. 9, p. 317–320.

[105] Saracino, A., D. Sgandurra, G. Dini, et F. Martinelli. 2016. « Madam : Effective and efficient behavior-based android malware detection and prevention », *IEEE Transactions on Dependable and Secure Computing*, p. 1–14.

[106] Sarma, B. P., N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, et I. Molloy. 2012a. « Android permissions : A perspective combining risks and benefits ». In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*. Coll. « SACMAT ’12 », p. 13–22, New York, NY, USA. ACM.

[107] ———. 2012b. « Android permissions : A perspective combining risks and benefits ». In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*. Coll. « SACMAT ’12 », p. 13–22, New York, NY, USA. Association for Computing Machinery.

[108] Savov, V. 2017. Windows phone was a glorious failure. https://www.theverge.com/2017/10/10/16452162/windows-phone-history-glorious-failure.

[109] Schneier, B. 2007. « The psychology of security », *Commun. ACM*, vol. 50, no. 5, p. 128.

[110] Shabtai, A., U. Kanonov, Y. Elovici, C. Glezer, et Y. Weiss. 2012. « Andromaly : a behavioral malware detection framework for android devices », *J. Intell. Inf. Syst.*, vol. 38, no. 1, p. 161–190.

[111] Shahzad, F., A. Akbar, et M. Farooq. 2012. « A survey on recent advances in malicious applications analysis and detection techniques for smartphones ».

[112] Shakya, S. et M. Dave. 2022. « Analysis, detection, and classification of android malware using system calls », *arXiv preprint arXiv :2208.06130*.

[113] Simonsson, J., L. Zhang, B. Morin, B. Baudry, et M. Monperrus. 2021. « Observability and chaos engineering on system calls for containerized applications in docker », *Future Generation Computer Systems*, vol. 122, p. 117–129.

[114] Statcounter. 2018. Mobile operating system market share worldwide - sept 2021 - sept 2022. `https://gs.statcounter.com/os-market-share/mobile/worldwide`.

[115] Suarez-Tangil, G., S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, et L. Cavallaro. 2017. « Droidsieve : Fast and accurate classification of obfuscated android malware ». In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, March 22-24, 2017*, p. 309–320.

[116] Suleman Saleem, M., J. Mišić, et V. B. Mišić. 2022. « Android malware detection using feature ranking of permissions », *arXiv e-prints*, p. arXiv–2201.

[117] Surendran, R., T. Thomas, et S. Emmanuel. 2020. « On existence of common malicious system call codes in android malware families », *IEEE Transactions on Reliability*, vol. 70, no. 1, p. 248–260.

[118] Sweetlin, R. et A.S.Radhamani. 2016. « Survey on detection of malware in android », *International Journal of Latest Trends in Engineering and Technology*, vol. 11, p. 44–47.

[119] Syed, T. A., S. Jan, S. Musa, et J. Ali. 2016. « Providing efficient, scalable and privacy preserved verification mechanism in remote attestation ». In *2016 International Conference on Information and Communication Technology (ICICTM)*, p. 236–245. IEEE.

[120] Symantec. 2013. « Global smb it confidence index ».

[121] ThreatFabric. 2018. Mysterybot; a new android banking trojan ready for android 7 and 8. `https://www.threatfabric.com/blogs/mysterybot__a_new_android_banking_trojan_ready_for_android_7_and_8.html`.

[122] Tian, Z., T. Liu, Q. Zheng, M. Fan, E. Zhuang, et Z. Yang. 2016. « Exploiting thread-related system calls for plagiarism detection of multithreaded programs », *Journal of Systems and Software*, vol. 119, p. 136–148.

[123] Tianqi Chen, T. H. 2019. « Xgboost : extreme gradient boosting », *R package version 0.82.1*, p. 1–4.

[124] Tong, F. et Z. Yan. 2017. « A hybrid approach of mobile malware detection in android », *Journal of Parallel and Distributed computing*, vol. 103, p. 22–31.

[125] Tong, S. et E. Chang. 2001. « Support vector machine active learning for image retrieval ». In *Proceedings of the Ninth ACM International Conference on Multimedia*. Coll. « MULTIMEDIA '01 », p. 107–118, New York, NY, USA. ACM.

[126] VirusShare. 2011. Virusshare.com - because sharing is caring. `https://virusshare.com/`.

[127] Wagstaff, K., C. Cardie, S. Rogers, et S. Schrödl. 2001. « Constrained k-means clustering with background knowledge ». In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML 2001), Williams College, Williamstown, MA, USA, June 28 - July 1, 2001*, p. 577–584.

[128] Wang, C., Q. Xu, X. Lin, et S. Liu. 2019. « Research on data mining of permissions mode for android malware detection », *Cluster Computing*, vol. 22, no. 6, p. 13337–13350.

[129] Wang, W., Z. Gao, M. Zhao, Y. Li, J. Liu, et X. Zhang. 2018. « Droidensemble : Detecting android malicious applications with ensemble of string and structural static features », *IEEE Access*, vol. 6, p. 31798–31807.

[130] Wang, W., C. Ren, H. Song, S. Zhang, P. Liu, et al. 2022. « Fgl_droid : An efficient android malware detection method based on hybrid analysis », *Security and Communication Networks*, vol. 2022.

[131] Wei, F., Y. Li, S. Roy, X. Ou, et W. Zhou. 2017. « Deep ground truth analysis of current android malware ». In *DIMVA*.

[132] Wen, L. et H. Yu. 2017. « An Android malware detection system based on machine learning ». In *AIP Conference Proceedings*. T. 1864, p. 020136. AIP Publishing.

[133] Witten, I. H. et E. Frank. 2002. « Data mining : practical machine learning tools and techniques with java implementations », *Acm Sigmod Record*, vol. 31, no. 1, p. 76–77.

[134] Wu, D., C. Mao, T. Wei, H. Lee, et K. Wu. 2012. « Droidmat : Android malware detection through manifest and API calls tracing ». In *Seventh Asia Joint Conference on Information Security, AsiaJCIS 2012, Kaohsiung, Taiwan, August 9-10, 2012*, p. 62–69.

[135] Wu, Y., M. Li, J. Wang, Z. Fang, Q. Zeng, T. Yang, et L. Cheng. 2022. « Droidrl : Reinforcement learning driven feature selection for android malware detection », *arXiv preprint arXiv :2203.02719*.

[136] Xiao, F., Y. Sun, D. Du, X. Li, et M. Luo. 2020. « A novel malware classification method based on crucial behavior », *Mathematical Problems in Engineering*, vol. 2020.

[137] Xiao, X., S. Zhang, F. Mercaldo, G. Hu, et A. K. Sangaiah. 2019. « Android malware detection based on system call sequences and lstm », *Multimedia Tools and Applications*, vol. 78, no. 4, p. 3979–3999.

[138] Xie, N., F. Zeng, X. Qin, Y. Zhang, M. Zhou, et C. Lv. 2018. « Repassdroid : Automatic detection of Android malware based on essential permissions and semantic features of sensitive apis ». In *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, p. 52–59. IEEE.

[139] Xu, F., S. Shen, W. Diao, Z. Li, Y. Chen, R. Li, et K. Zhang. 2021. « Android on pc : On the security of end-user android emulators ». In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, p. 1566–1580.

[140] Yan, P. et Z. Yan. 2018. « A survey on dynamic mobile malware detection », *Software Quality Journal*, vol. 26, no. 3, p. 891–919.

[141] Yerima, S. Y., S. Sezer, et I. Muttik. 2015. « High accuracy android malware detection using ensemble learning », *IET Information Security*, vol. 9, no. 6, p. 313–320.

[142] Yuan, H., Y. Tang, W. Sun, et L. Liu. 2020. « A detection method for android application security based on tf-idf and machine learning », *Plos one*, vol. 15, no. 9, p. e0238694.

[143] Zangief. 2014. Appchina is the best android app store alternative. `http://appcakefans.com/appchina-is-the-best-android-app-store-alternative/`.

[144] ———. 2017. Gfan provides you free android apps and games. `http://appcakefans.com/gfan-provides-you-free-android-apps-and-games/`.

[145] Zarni Aung, W. Z. 2013. « Permission-based android malware detection », *International Journal of Scientific & Technology Research*, vol. 2, no. 3, p. 228–234.

[146] Zhou, W., Y. Zhou, X. Jiang, et P. Ning. 2012a. « Detecting repackaged smartphone applications in third-party android marketplaces ». In *Second ACM Conference on Data and Application Security and Privacy, CODASPY 2012, San Antonio, TX, USA, February 7-9, 2012*, p. 317–326.

[147] Zhou, Y. et X. Jiang. 2012. « Dissecting Android malware : Characterization and evolution ». In *2012 IEEE Symposium on Security and Privacy*, p. 95–109.

[148] Zhou, Y. et X. Jiang. 2012a. « Dissecting android malware : Characterization and evolution ». In *2012 IEEE symposium on security and privacy*, p. 95–109. IEEE.

[149] Zhou, Y. et X. Jiang. 2012b. « Dissecting Android malware : Characterization and evolution ». In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, p. 95–109.

[150] Zhou, Y., Z. Wang, W. Zhou, et X. Jiang. 2012b. « Hey, you, get off of my market : Detecting malicious apps in official and alternative Android markets ». In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*.