Etude des malware évasifs : conception, protection et détection

Cédric Herzog

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

CentraleSupélec

THESE DE DOCTORAT DE

CentraleSupelec
COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par
Cédric HERZOG

Étude des malware évasifs :

conception, protection et détection

Thèse présentée et soutenue à Rennes, le 28 janvier 2022
Unité de recherche : IRISA
Thèse N° : 2022CSUP0001

Rapporteurs avant soutenance :

Aurélien Francillon — Professeur - EURECOM
Jacques Klein — Associate Professor - Université du Luxembourg

Composition du Jury :

Président :	Vincent Nicomette	Professeur - INSA Toulouse
Examinateurs :	Marion Videau	Responsable scientifique - Quarkslab
Dir. de thèse :	Valérie Viet Triem Tong	Professeur, CentraleSupelec
Encadrant :	Pierre Wilke	Maître de conférences, CentraleSupelec

Invité(s) :

Jean-Marie Borello — Thalès

Résumé

Il y a une confrontation permanente entre les malware et les antivirus qui conduisent les deux parties à évoluer continuellement. D'un côté, les antivirus mettent en place des solutions de plus en avancées et proposent des techniques de détection complexes qui viennent s'ajouter à la classique détection par signature. Lorsqu'un nouveau malware est détecté, les antivirus conduisent des recherches plus approfondies afin de produire une signature et garder leur base de données à jour rapidement. Cette complexification conduit les antivirus à laisser des traces de leur présence sur les machines qu'ils protègent.

D'un autre côté, des auteurs de malware qui souhaitent créer des malware durables à moindre coût peuvent quant à eux utiliser de simples méthodes qui permettent d'éviter une détection et une analyse approfondie par ces antivirus. Il est possible pour un malware de rechercher la présence de traces ou d'artéfacts laissés par les antivirus pour ensuite décider d'exécuter ou non leur charge malveillante. Nous désignons de tels programmes comme étant des *malware évasifs*.

Cette thèse se concentre sur l'étude de malware évasifs qui ciblent le système d'exploitation Windows. Une première contribution vise à évaluer l'efficacité de techniques d'évasion contre un panel d'antivirus. Nous proposons par la suite une contre-mesure qui vise à stopper l'exécution de malware qui utilisent ce genre de techniques d'évasion en simulant les modifications faites par un antivirus dans le système d'exploitation. Ces leurres sont créés grâce à l'instrumentation de l'Application Programming Interface (API) Windows à l'aide de Microsoft Detours. Nous évaluons cette contre-mesure sur quelques exemples de malware évasifs récupérés dans la nature.

Une seconde contribution a pour objectif de répondre à l'absence de dataset de malware évasifs. Pour cela, nous étiquetons un dataset de malware Windows existant de deux manières. Premièrement, à l'aide d'une détection automatique qui utilise la contre-mesure issue de notre première contribution et deuxièmement, à l'aide d'un étiquetage collaboratif accessible publiquement.

Publications

[1] Valérie Viet Triem Tong et al. « Isolating malicious code in Android malware in the wild ». In : *MALCON 2019 - 14th International Conference on Malicious and Unwanted Software*. MALCON 2019. Nantucket, United States : IEEE Computer society, oct. 2019 (cf. p. 90).

[2] Cédric Herzog et al. « Evasive Windows Malware : Impact on Antiviruses and Possible Countermeasures ». In : *SECRYPT 2020 - 17th International Conference on Security and Cryptography*. Proceedings of the 17th International Joint Conference on e-Business and Telecommunications - (Volume 3). Lieusaint - Paris, France : SCITEPRESS - Science and Technology Publications, juill. 2020, p. 302-309. doi : 10.5220/0009816703020309 (cf. p. 89).

[3] Cédric Herzog et al. « Malware Windows Evasifs : Impact sur les Antivirus et Possible Contre-mesure ». In : *C&ESAR 2020 - Computer & Electronics Security Applications Rendez-vous*. Rennes, France, déc. 2020 (cf. p. 89).

Remerciements

Je souhaite tout d'abord remercier Valérie Viet Triem Tong, Pierre Wilke et Jean-Louis Lanet pour leur disponibilité et leurs conseils. Merci surtout à Valérie d'avoir supporté mes montages photos :-)

Merci à Marion Videau, Vincent Nicomette, Aurélien Francillon et Jacques Klein d'avoir accepté de faire partie de mon jury de thèse et plus particulièrement Aurélien et Jacques pour leur relecture de mon manuscrit. Merci également à Davide Balzarotti, Clémentine Maurice et Marco Balduzzi pour le temps qu'ils m'ont accordé et les conseils qu'ils m'ont donnés lors des comités de thèse.

La mise en place des expérimentations a été grandement facilitée par l'aide d'Alexandre Sanchez et Léopold Ouairy que je remercie énormément. Merci à Lydie et Myriam pour leur patience et leur connaissance de l'ingénierie administrative à laquelle j'ai dû faire face et qui était plus supportable grâce à elles.

Il était très agréable d'effectuer cette thèse dans l'équipe CIDRE. Je souhaite remercier tous ses membres pour tous les moments passés ensemble et les souvenirs que j'en garde. Je retiens particulièrement les moments passés avec Aimad Berady, Tomàs Concepción Miranda, Pierre Graux, David Lanoë, Ronny Chevalier et Benoît Fournier, notamment pour rédiger l'article *UT : Towards a new approach to cyber team building*.

Signal aura été la clef de la réussite du groupe *4272613d05021b131f* pour rendre le confinement plus supportable. Danke schön donc à Ludovic Claudepierre, Mathieu Escouteloup, Pierre-Yves Péneau et Léopold Ouairy pour toutes les idioties que l'on s'est partagées (cf. Listing 1).

Enfin, merci à ma famille qui me pousse et m'encourage depuis longtemps et grâce à qui j'ai poursuivi mes études. Merci particulièrement à Shirin qui traverse la même chose que moi et avec qui j'ai pu partager mes doutes et qui n'a jamais cessé de m'encourager pendant ces trois ans.

```
;Mais du coup IA-32 c'est du x86?
org  0x100
mov  dx, msg
mov  ah, 9
int  0x21
mov  ah, 0x4c
int  0x21
msg  db 'IDIOTS! <3', 0x0d, 0x0a, '$'
```

Listing 1: **:** Merci!

Table des matières

Table des figures

Liste des tableaux

Liste des Listings

Prologue

1 Introduction

Les ordinateurs de bureau et portables représentent 39.13% des appareils présents dans le monde d'après NetMarketShare [1]. Parmi ceux-ci, Windows dont la première version date de 1985 est de loin le système d'exploitation le plus utilisé avec une part de marché de 87.56% [2]. C'est donc logiquement que ce système est la cible privilégiée des malware et 93% de tous les malware que possède l'institut de recherche AV-TEST [3] lui sont destinés.

1: https://netmarketshare.com/device-market-share

2: https://netmarketshare.com/operating-system-market-share.aspx

3: https://portal.av-atlas.org/malware

Le nombre de malware découverts chaque année est quasiment sans cesse en augmentation depuis 2008. Cette année-là, ce sont par exemple 7 333 179 nouveaux malware qui ciblent Windows qui sont découverts contre 101 251 777 en 2021, toujours selon AV-TEST [4].

4: https://portal.av-atlas.org/malware/statistics

Des antivirus sont inventés pour détecter et stopper les malware et permettent de protéger ainsi des ordinateurs personnels, mais également des serveurs. Leur présence a poussé les malware à implémenter différentes techniques qui permettent d'éviter la détection. Les antivirus ont bien entendu répondu à cela en adaptant et inventant de nouvelles méthodes de détection. Ce bras de fer a poussé les deux parties à devenir de plus en plus complexes jusqu'à devenir les logiciels et malware que nous connaissons aujourd'hui.

1.1 Évolution de la lutte entre attaquants et défenseurs

Le premier programme informatique qui peut être considéré comme un ver informatique est Creeper [4] sorti en 1971 [5]. Il s'agit d'un programme expérimental qui cible les ordinateurs centraux PDP-10 et qui démontre la faisabilité de relocaliser dynamiquement un programme sur un autre système via l'ARPANET. La première version de ce logiciel est uniquement capable de migrer d'un système TENEX à un autre, mais une seconde version lui ajoute la possibilité de se répliquer.

[4]: Sutherland et al. (1974), « Natural Communication with Computers. Volume III. Distributed Computation Research at BBN »

5: https://en.wikipedia.org/wiki/Creeper_(program)

Creeper est contré par un programme appelé Reaper et qui parcourt lui aussi le réseau pour supprimer le ver dès qu'il le rencontre sur un système.

C'est en 1982 qu'apparaît le premier virus qui cible les ordinateurs personnels et qui ne vient pas d'un laboratoire [6]. Elk Cloner est un programme qui utilise une bombe logique qui attend de s'exécuter cinquante fois avant d'afficher un message à l'écran des ordinateurs Apple II sur lesquels il s'exécute. Son auteur le crée dans le but de faire une blague à ses camarades de lycée en attachant le virus à un jeu qu'il distribue sur une disquette.

6: https://en.wikipedia.org/wiki/Elk_Cloner

Les systèmes d'exploitation de l'époque sont contenus sur des disquettes qui, si elles sont infectées, déclenchent le virus à chaque démarrage. La solution pour se débarrasser du malware consiste alors à utiliser un outil tel que le MASTER CREATE d'Apple pour réécrire un nouveau Disk

Operating System (DOS) sur la disquette. À ce moment, aucun antivirus n'existe encore et les utilisateurs sont peu familiarisés avec le concept de malware ce qui peut expliquer en partie le succès de ce virus.

L'année 1987 marque l'arrivée des premiers antivirus produits par des entreprises telles que G Data Software, McAfee ou encore ESET. Ceux-ci se basent principalement sur de la recherche de séquences binaires appelées signatures et qui sont spécifiques à des malware précis.

Certains malware ont essayé d'éviter de se faire détecter en chiffrant leur contenu. C'est le cas de Cascade en 1987 [7] qui contient une routine de déchiffrement qui applique un XOR sur son contenu [8]. En 1988, Cascade infecte des ordinateurs au siège d'IBM qui décide alors de développer sa propre solution antivirus [9].

Cette technique de chiffrement est améliorée par la suite en faisant varier aléatoirement la routine de déchiffrement entre chaque copie du malware. C'est le cas par exemple du malware polymorphe 1260 écrit en 1990 qui ajoute du « junk code » à sa routine de déchiffrement [10]. Contrairement au virus Cascade dont le code de déchiffrement reste inchangé et peut donc être détecté par une analyse par signature, 1260 obtient une routine de chiffrement qui varie constamment grâce à l'ajout de « junk code », ce qui rend plus ardue sa détection par un antivirus.

L'analyse par signature n'est donc plus assez efficace et les antivirus doivent trouver de nouvelles techniques qui permettent de détecter ces malware [11]. Toutefois, le corps du malware polymorphe après déchiffrement reste le même à chaque itération ce qui donne une possibilité aux antivirus de le détecter par signature à ce moment-là [5].

Des malware métamorphes apparaissent alors pour pallier cela et sont capables de se reprogrammer eux-mêmes en changeant leur code à chaque itération. Ces modifications peuvent être du renommage de registres, de la permutation de code ou l'ajout de « junk code » par exemple. La sémantique du programme reste ainsi toujours la même tout en possédant un code et donc une signature différente. Le premier malware métamorphe qui utilise ce procédé est Win95/Regswap sorti en 1998 [12].

Des «sandboxes» et émulateurs apparaissent au sein des antivirus pour analyser des malware pendant leur exécution [13]. Les malware qui exécutaient toujours le même comportement à chaque exécution ont donc commencé à observer l'environnement sur lequel ils s'exécutent avant de déclencher le comportement malveillant. Un malware capable de repérer un environnement d'analyse peut ainsi s'abstenir de présenter un comportement malveillant afin d'éviter une détection et ainsi prolonger sa durée de vie.

1.2 Contributions

Nous désignons un logiciel comme évasif s'il correspond à la définition 1.2.1. Un malware évasif est un logiciel évasif ayant un but malveillant.

7: https://encyclopedia.kaspersky.fr/knowledge/year-1987/

8: https://harrisonwl.github.io/assets/courses/malware/spring2017/slides/FinalWeeks/EncryptedOligomorphic.pdf

9: https://encyclopedia.kaspersky.fr/knowledge/year-1987

10: https://userpages.umbc.edu/~dgorin1/432/example_decryptor.htm

11: https://encyclopedia.kaspersky.fr/knowledge/year-1990

[5]: You et al. (2010), « Malware Obfuscation Techniques : A Brief Survey »

12: https://arxiv.org/pdf/1406.7061.pdf

13: https://www.kaspersky.com/enterprise-security/wiki-section/products/sandbox

Définition 1.2.1 *Un logiciel évasif est un programme, conçu pour changer de comportement s'il est en présence d'un environnement précis.*

La première contribution présentée dans cette thèse a pour but d'évaluer l'efficacité de techniques d'évasion. Pour cela nous créons Nuky, un ransomware évasif configurable que nous exécutons en présence d'antivirus. Nous remarquons que Nuky est bel et bien capable de détecter et d'identifier les antivirus à l'aide de techniques d'évasion basiques.

Notre seconde contribution concernera la contre-mesure que nous avons créée afin de stopper le fonctionnement de certains malware évasifs sur une machine physique tout en assurant le bon fonctionnement des logiciels légitimes. Cette contre-mesure est conçue pour avoir un impact minimum sur le système hôte en ajoutant peu d'overhead. Nous présenterons la façon dont a été évaluée cette contre-mesure et les raisons qui nous ont poussés à créer un dataset de malware évasifs partagé publiquement.

Enfin, nous avons souhaité offrir un ensemble de malware évasifs qui permette aux chercheurs du domaine d'évaluer leurs expérimentations simplement. Pour cela, nous cherchons à construire ce dataset en introduisant le minimum de biais possible. Le dataset que nous souhaitons construire doit être composé de malware que l'on sait être évasifs, et ce avec un niveau de confiance élevé. La manière dont est construit le dataset doit être transparente afin de permettre à d'autres chercheurs de pouvoir le critiquer et de l'améliorer. Faciliter la collaboration permet d'augmenter la taille du dataset, mais également de réduire certains biais par exemple en faisant étiqueter le dataset par plusieurs personnes au lieu d'une seule comme dans certains datasets présentés dans la bibliographie. Un chercheur doit également pouvoir reconstruire un dataset à partir des informations présentées dans un article soit en donnant une liste de hash, soit en donnant la liste des critères utilisés afin de créer le dataset.

Nous commençons par décrire l'état actuel de nos connaissances des malware évasifs dans le Chapitre 2 puis évaluons l'efficacité de quelques techniques d'évasion dans le Chapitre 3. Nous détaillons ensuite la façon dont nous construisons une contre-mesure anti malware évasifs dans le Chapitre 4. Enfin, nous construisons un dataset de malware évasifs que nous décrivons dans le Chapitre 5 avant de conclure cette thèse dans le Chapitre 6.

Etat de l'art 2

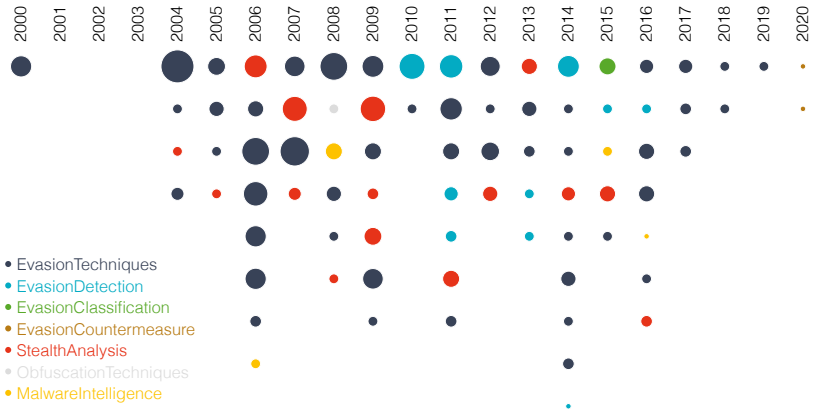


FIGURE 2.1 – Visualisation de la bibliographie.

FIGURE 2.2 – Lien pour accéder à la bibliographie interactive : https://tempo.irisa.fr/visualization/dots

Cet état de l'art repose sur une bibliographie consultable en version interactive au lien contenu dans la Figure 2.2. La Figure 2.1 représente une visualisation de la bibliographie dans laquelle chaque point est une référence. La surface de chaque point est proportionnelle au nombre de références qui les citent et une couleur est associée en fonction du sujet de recherche.

La version interactive permet de visualiser les citations entre les références en affichant des liens entre les points. Un résumé est également disponible pour chaque référence de la bibliographie.

Ce chapitre débute avec une présentation des champs de recherche présents dans l'état de l'art. Sont ensuite décrites les techniques d'évasion et leur évolution au fil des années dans une deuxième partie. L'étude de certaines de ces techniques d'évasion nous a permis de créer une contre-mesure présentée dans le Chapitre 4 qui inhibe le comportement malveillant de malware évasifs.

Enfin, nous discuterons de la construction des datasets de malware présentés dans les différents articles et les raisons qui nous ont poussés à créer un dataset de malware évasifs que nous présentons dans le Chapitre 5.

2.1 Les différents sujets de recherche

La bibliographie utilisée contient quatre-vingts références réparties en six catégories selon leur sujet d'étude. Chaque référence est liée à une seule thématique bien que certaines traitent de plusieurs sujets. Par exemple, les articles qui présentent un outil d'analyse furtive commencent souvent par présenter des techniques d'évasion. Comme le sujet principal de ces articles est l'analyse furtive, nous les lions à cette catégorie uniquement. Dans cette section nous décrivons succinctement les six sujets d'étude liés aux malware évasifs.

2.1.1 Techniques d'évasion

Définition 2.1.1 *Une technique d'évasion est un ensemble de procédés conçu pour détecter tout ou une partie d'un environnement prédéfini et retournant une information qui permet à un malware évasif de décider de la suite de son comportement.*

Le sujet de recherche le plus présent dans la littérature concerne les **techniques d'évasion**. Il peut s'agir de pages web qui décrivent une attaque précise [6] ou d'outils qui permettent d'exécuter ces attaques [7]. Les «*surveys*» sont également intégrés dans cette catégorie. Les avancées technologiques qui apparaissent au fil des années ont fait émerger de nouvelles techniques d'évasion. Cette mise à jour constante des techniques d'évasion fait que ce sujet de recherche est présent sur toute la période étudiée.

La visualisation permet de comprendre la popularité de ce sujet de recherche. L'étude des techniques d'évasion concerne quarante-huit références soit 59% de la bibliographie. Chaque référence est citée 4,6 fois en moyenne.

Nous détaillons les techniques d'évasion présentes dans notre bibliographie dans la Section 2.2 qui leur est dédiée.

[6]: Rutkowska (2004), *Red Pill. . . or how to detect VMM using (almost) one CPU instruction*

[7]: Klein (2006), *ScoopyNG : The VMware detection tool.*

2.1.2 Analyse furtive

Les environnements d'analyse peuvent chercher à être les plus discrets possibles pour éviter qu'un malware évasif ne les repère et puisse changer de comportement. Un environnement est transparent lorsqu'il est indistinguable d'une machine physique non destinée à faire de l'analyse de malware.

Après la parution des premières techniques d'évasion, les chercheurs ont essayé de cacher les outils d'analyse afin de faire de l'**analyse furtive**. Ce fut d'abord de simples astuces qui permettaient de cacher le debugger d'IDA [8] ou qui appliquaient des patchs sur des binaires d'hyperviseur [9] et sur des fichiers de configuration [10]. Sun *et al.* [11] proposent un plugin IDA permettant de localiser dynamiquement deux techniques d'évasion utilisées par les malware et de modifier les résultats qu'elles retournent de sorte à cacher VMWare. Ces procédés permettent d'analyser des malware évasifs tout en utilisant les outils habituels de rétro-ingénierie.

Par la suite, de nouveaux environnements furtifs conçus pour être capables d'exécuter des malware évasifs paraissent. Corbra [12] remplace par exemple des instructions qui peuvent trahir la présence d'un environnement d'analyse par ce que les auteurs appellent des «*stealth implants*». L'instruction ReaD Time Stamp Counter (RDTSC) est par exemple remplacée par un MOV qui stocke un compteur interne à Cobra dans le registre EAX. MAVMM [13] est un hyperviseur conçu pour résister aux attaques de type *Red Pill* [6] et ses dérivées, car il ne modifie pas la table Interrupt Descriptor Table (IDT) qui est utilisée par ces techniques d'évasion.

D'autres outils construisent des graphes de flot de contrôle et enlèvent les techniques d'évasion présentes. Par exemple, certaines sandboxes analysent les malware pendant quelques minutes puis stoppent leur

[8]: Guilfanov (2005), *2005 - Simple trick to hide IDA debugger*

[9]: kortchinsky (2004), *Counter measures to VMware fingerprinting*

[10]: Carpenter et al. (2007), « Hiding Virtualization from Attackers and Malware »

[11]: Sun et al. (2008), « An automatic anti-anti-VMware technique applicable for multi-stage packed malware »

[12]: Vasudevan et al. (2006), « Cobra : Fine-grained Malware Analysis using Stealth Localized-executions »

[13]: Nguyen et al. (2009), « MAVMM : Lightweight and Purpose Built VMM for Malware Analysis »

[6]: Rutkowska (2004), *Red Pill. . . or how to detect VMM using (almost) one CPU instruction*

exécution. Pour éviter d'être analysé, un malware peut utiliser un code qui s'exécute pendant un long moment avant de déclencher sa payload. Cette technique se rapproche d'une technique d'évasion bien qu'elle ne permette pas au malware de vérifier l'environnement d'exécution et que la payload soit activée dans tous les cas. L'outil HASTEN [14] vise à enlever les parties du graphe de flot de contrôle qui correspondent à un temps d'exécution long et ainsi permettre d'analyser le malware dans les temps prévus par les sandboxes.

[14]: Kolbitsch et al. (2011), « The Power of Procrastination : Detection and Mitigation of Execution-Stalling Malicious Code »

Une solution pour effectuer des analyses plus furtives est d'utiliser un environnement qui possède plus de privilèges que le malware à analyser. Par exemple, SPIDER [15] est un outil qui permet de placer des points d'arrêt de manière plus discrète qu'à l'accoutumée à l'aide de manipulation de pages mémoire. DRAKVUF [16] est un système de monitoring qui utilise l'hyperviseur Xen pour manipuler la pile et le Central Processing Unit (CPU) virtualisé et ainsi prendre le contrôle du processus à analyser. Pour analyser les malware, MALT [17] est un debugger bare-metal qui exécute le programme sur une machine dont le CPU est en System Management Mode (SMM). Ce mode est conçu pour être transparent du système d'exploitation et est également utilisé par HOPS [18].

[15]: Deng et al. (2013), « SPIDER : Stealthy Binary Program Instrumentation and Debugging via Hardware Virtualization »

[16]: Lengyel et al. (2014), « Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System »

[17]: Zhang et al. (2015), « Using Hardware Features for Increased Debugging Transparency »

[18]: Leach et al. (2016), « Towards Transparent Introspection »

Enfin, des outils utilisent deux environnements, le premier étant conçu pour ressembler à un environnement standard et le second équipé d'outils d'analyse. Par exemple, Kang *et al.* [19] décrivent un outil qui vise à corriger les erreurs d'émulation en comparant le comportement d'un malware sur un système de référence puis celui obtenu avec l'émulateur. Un autre outil appelé V2E [20] enregistre un log d'une exécution d'un malware à l'aide de virtualisation matérielle. Dans cette première exécution, le malware n'est pas censé s'évader. Le log est ensuite rejoué dans un environnement qui possède des outils qui permettent d'analyser le comportement en détail.

[19]: Kang et al. (2009), « Emulating Emulation-Resistant Malware »

[20]: Yan et al. (2012), « V2E : Combining Hardware Virtualization and Softwareemulation for Transparent and Extensible Malware Analysis »

Un survey publié en 2012 par Egele et. al. [21] décrit plusieurs outils utilisés pour analyser dynamiquement un malware. Certaines techniques d'évasion ainsi que des méthodes pour cacher les outils d'analyse sont présentées.

[21]: Egele et al. (2008), « A Survey on Automated Dynamic Malware-Analysis Techniques and Tools »

Dans un article de 2007, Garfinkel et. al. [22] considèrent que du fait que les machines virtuelles sont conçues pour être performantes et non transparentes, elles resteront distinguables d'un environnement physique.

[22]: Garfinkel et al. (2007), « Compatibility Is Not Transparency : VMM Detection Myths and Realities »

Ce sujet apparaît assez rapidement à la suite des premières publications de techniques évasives. Ce champ de recherche est représenté par quinze références soit 18,7% de la bibliographie. Chaque référence est citée 2,1 fois en moyenne. La transparence est également un élément important des outils de détection de malware évasifs que nous étudions dans la section suivante.

2.1.3 Détection de malware évasifs

Le flux grandissant de malware produits chaque année, rend impossible les analyses manuelles de chaque échantillon. Il peut être intéressant

pour un analyste d'avoir un premier filtre automatisé qui lui permette de sélectionner des échantillons potentiellement évasifs.

Les plateformes d'analyse furtive existantes produisent un overhead trop important et ne peuvent donc pas être utilisées sur un tel volume de malware évasifs [23]. Des méthodes de **détection de malware évasifs** automatisées sont créées dès 2010 pour répondre à ce problème.

[23]: Balzarotti et al. (2010), « Efficient Detection of Split Personalities in Malware »

Une méthode de détection de malware évasifs consiste à faire une exécution dans un environnement physique puis dans un environnement virtualisé. Le comportement du malware est enregistré dans les deux environnements pour être ensuite comparé à la recherche de divergences causées par l'évasion. On suppose alors que le comportement du malware entre les deux environnements sera différent.

[23]: Balzarotti et al. (2010), « Efficient Detection of Split Personalities in Malware »

Pour repérer ce changement de comportement, Balzarotti et. al. [23] enregistrent une trace d'appels système dans un premier environnement et la rejoue dans un deuxième environnement. Chaque exécution démarre du même état initial et reçoit les mêmes entrées. Si une déviation est détectée, alors le malware étudié est considéré comme évasif. Les auteurs mentionnent également le fait que deux exécutions différentes effectuées sur un même environnement ne produisent pas forcément deux traces identiques. C'est ce qui a motivé le fait d'enregistrer une trace afin de la rejouer dans un autre environnement ensuite et ainsi limiter les aléas générés par les optimisations du système d'exploitation par exemple. Le même principe est utilisé par Liu *et al.* [24] mais qui utilisent un algorithme différent pour localiser le point de divergence des deux comportements.

[24]: Liu et al. (2014), « Fast Discovery of VM-Sensitive Divergence Points with Basic Block Comparison »

[25]: Hsu et al. (2013), « Divergence Detector : A Fine-Grained Approach to Detecting VM-Awareness Malware »

Hsu *et al.* [25] améliore cette détection pour corriger certains cas où le comportement évasif des malware pouvait être rejoué dans l'environnement d'analyse.

[26]: Naval et al. (2014), « Environment-Reactive Malware Behavior : Detection and Categorization »

Naval *et al.* [26] représentent les comportements des malware sous forme de chaîne de Markov puis les classifient à l'aide d'un algorithme basé sur un réseau neuronal.

[27]: Sun et al. (2011), « Malware Virtualization-Resistant Behavior Detection »

Sur un principe similaire, Sun *et al.* [27] comparent deux comportements capturés avec le logiciel Process Monitor. Une distance est calculée via un algorithme utilisé pour trouver des correspondances de deux chaînes génétiques.

[28]: Kirat et al. (2014), « BareCloud : Bare-metal Analysis-based Evasive Malware Detection »

Barecloud [28] effectue la comparaison sur les opérations de fichiers et registres obtenues à partir des données brutes du disque.

[29]: Vishnani et al. (2011), « Detecting and Defeating Split Personality Malware »

Vishnani *et al.* [29] et Tan *et al.* [30] utilisent Intel Pin [1] pour monitorer les appels de fonctions de l'API Windows utilisées par les malware pour s'évader. Les malware qui appellent ces fonctions sont alors considérés comme évasifs.

[30]: Tan et al. (2016), « Detecting Malware Through Anti-analysis Signals - A Preliminary Study »

1: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html

Intel Pin est également utilisé par DMA [31] pour monitorer les processus, services, registres, connections et activités de fichiers. Le malware est considéré comme évasif dès qu'une anomalie est détectée.

[31]: Pektas et al. (2014), « A dynamic malware analyzer against virtual machine aware malicious software »

DISARM [32] représente un comportement par un ensemble de *features* telles que les créations de fichiers ou les communications réseau par exemple. Une trace est enregistrée sur les sandboxes étudiées puis une distance est calculée entre chaque environnement. Ces distances sont

[32]: Lindorfer et al. (2011), « Detecting Environment-Sensitive Malware »

ensuite utilisées pour calculer un score qui s'il dépasse un seuil signifie que le malware est évasif.

Cette thématique apparue assez tardivement dans la littérature représente 11% de la bibliographie avec sept citations en moyenne par article.

2.1.4 Classification de malware évasifs

Les malwares évasifs peuvent utiliser différentes techniques d'évasion et peuvent agir de multiples façons dès qu'un outil d'analyse est détecté. Une classification de ces malware est alors possible et permettrait d'apporter un éclairage sur la nature des connaissances que nous possédons sur les malware évasifs.

Nous avons relevé une seule référence [33] qui traite de «*clustering*» qui s'approche de la **classification de malware évasifs**. Cela pourrait s'expliquer par le fait qu'il faille tout d'abord obtenir un dataset de malware évasifs. Cet article de Kirat et. al. fait suite à une précédente publication qui traite de la détection de malware évasifs et qui leur fournisse un dataset. Pour chaque malware, une signature liée à l'évasion est créée et un algorithme de «*clustering*» hiérarchique est appliqué. Cela permet de grouper les malware qui utilisent des techniques d'évasion similaires entre eux.

[33]: Kirat et al. (2015), « MalGene : Automatic Extraction of Malware Analysis Evasion Signature »

Ce sujet de recherche représente 1,2% de la bibliographie et la seule publication publiée en 2015 est citée sept fois.

2.1.5 Malware intelligence

Certains articles détaillent l'évolution des malwares évasifs et donnent des informations sur les outils utilisés par les attaquants. Ce type de recherche que nous pouvons qualifier de **malware intelligence** permet par exemple d'observer des tendances concernant les techniques d'évasion implémentées par les malware évasifs.

Lenny Zeltser [34] indique en 2006 que plusieurs malware récoltés à ce moment-là cherchent à éviter de s'exécuter sur une machine virtuelle VMWare. L'utilisation de packers qui intègrent des techniques d'évasion pour protéger des applications légitimes est également mentionnée.

[34]: Zeltser (2006), *Virtual Machine Detection in Malware via Commercial Tools*

Lau *et al.* [35] publient en 2008 un article qui cherche à obtenir la proportion de malware qui utilise des techniques d'évasion de machines virtuelles. Sur deux millions d'échantillons que les auteurs savent malveillants, 2.8% utilisent des techniques qui leur permettent de savoir qu'ils s'exécutent sur une machine virtuelle.

[35]: Lau et al. (2010), « Measuring virtual machine detection in malware using DSD tracer »

En 2015, Graziano *et al.* [36] n'a pas pour but de présenter des informations exclusivement sur des malware évasifs. Toutefois en étudiant les malware soumis à leur sandbox, les auteurs trouvent des malware qui contiennent des techniques d'évasion qui utilisent les tables IDT, Service Description Table (SDT) et Global Descriptor Table (GDT) pour détecter une machine virtuelle. Un autre malware recherche simplement le nom de l'ordinateur et un dernier vérifie la présence d'instrumentations. D'autres échantillons qui utilisent des techniques d'évasion de sandboxes sont également

[36]: Graziano et al. (2015), « Needles in a Haystack : Mining Information from Public Dynamic Analysis Sandboxes for Malware Intelligence »

décrits. Cela permet de connaître les techniques réellement utilisées par les attaquants à cette période.

[37]: Jadhav et al. (2016), « Evolution of evasive malwares : A survey »

Jadhav *et al.* [37] présentent l'évolution des techniques d'évasion utilisées par les malware ainsi que les possibles nouvelles techniques à venir. Une évolution du nombre de malware évasifs qui proviennent de Lastline en 2014 est présentée et permet de se rendre compte de l'augmentation progressive de ce type de malware.

Ces articles sont présents sur toute la période étudiée et représentent 5% de la bibliographie et sont cités 2,25 fois en moyenne.

2.1.6 Prévention

[38]: Chen et al. (2008), « Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware »

[39]: Herzog et al. (2020), « Evasive Windows Malware : Impact on Antiviruses and Possible Countermeasures »

[40]: Zhang et al. (2020), « Scarecrow : Deactivating Evasive Malware via Its Own Evasive Logic »

L'étude de solutions de **prévention** qui vise à stopper les malware évasifs est mentionnée dans certains articles dès 2008 [38]. Nous traitons principalement du sujet dans le Chapitre 4 et nous avons présenté nos résultats dans une conférence [39] en 2020. Un article concurent paraît la même année [40]. Le principe est très similaire et consiste à charger une librairie dans les programmes pour instrumenter les fonctions de l'API Windows pour simuler la présence d'artéfacts afin de faire s'évader les malware. Tout comme pour la classification, cela demande d'avoir un dataset d'échantillons évasifs ce qui pourrait expliquer le faible nombre de références sur le sujet.

Les deux articles sur cette thématique représentent 2,25% et sont encore trop récents pour être cités.

2.2 Techniques d'évasion

Cette section décrit l'apparition et l'évolution des techniques d'évasion présentes dans la bibliographie présentée dans la Section 2.1.1. La compréhension de ces techniques est essentielle pour concevoir des contre-mesures ou des solutions de détection adaptées aux malware évasifs.

Dans cette thèse, nous nous intéressons principalement aux malware qui cherchent à s'exécuter sur des ordinateurs personnels physiques et qui évitent les analyses en environnement cloisonné. Nous étudions les techniques d'évasion utilisées par les malware qui visent principalement les outils d'analyse dynamique et les environnements virtualisés.

Les techniques d'évasion sont très liées aux environnements qu'elles visent. Un attaquant peut adapter son malware afin de cibler ou d'éviter certains systèmes d'exploitation, outils d'analyse et outils de virtualisations.

Par exemple, un malware pourrait cibler les ordinateurs personnels qui ont a priori plus de chances d'être des machines physiques plutôt que virtualisées. Un environnement virtualisé pourrait être signe de la présence d'un analyste qui observe le fonctionnement du malware. Pouvoir repérer une machine virtuelle afin de changer son comportement en évitant par exemple de déclencher sa payload permettrait au malware de prolonger sa durée de vie.

Windows est le système d'exploitation le plus utilisé pour un ordinateur [2] et est donc la cible privilégiée des créateurs de malware [3]. Nous constatons également dans la littérature que Windows ainsi que les outils utilisés sur ce système d'exploitation sont les plus étudiés. Les techniques d'évasion que nous étudions ainsi que les contributions que nous présentons sont par conséquent conçues pour fonctionner sous Windows.

2: https://gs.statcounter.com/os-market-share/desktop/worldwide/#monthly-202005-202105

3: https://portal.av-atlas.org/malware?s=eb7e441930ede3460e580911a33a0f5157bf9d5d

À notre connaissance, la première description d'une méthode d'évasion est présentée en 2004 par Joanna Rutkowska. En testant le rootkit SucKIT, Rutkowska découvre que ce malware refuse de se charger sur une Virtual Machine (VM) créée par VMWare. Cela est dû à l'instruction du CPU Store Interrupt Descriptor Table Register (SIDT) qui se comporte différemment sur un environnement virtualisé que sur un environnement physique [6]. Rutkowska décrit une méthode qui exploite cette instruction afin de détecter la présence d'un hyperviseur.

[6]: Rutkowska (2004), *Red Pill... or how to detect VMM using (almost) one CPU instruction*

Un premier article publié en 2000 discutait déjà de différences de comportement des processeurs Intel Pentium entre une machine virtuelle et une machine physique [41]. Les auteurs mentionnent des différences de disponibilité des ressources système, de timing, ainsi que des périphériques d'entrées/sorties.

[41]: Robin et al. (2000), « Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor »

Cette idée de chercher des différences de comportement entre un environnement physique et virtualisé est reprise afin de créer une multitude de techniques d'évasion. Ces techniques peuvent être communes à plusieurs logiciels de virtualisation ou spécifiques à un logiciel précis.

2.2.1 Machines virtuelles et émulateurs

Afin d'étudier le fonctionnement d'un malware, les analystes peuvent l'exécuter dans des environnements virtualisés. Ce type d'environnement a été adopté dès 1999 d'après Lau *et al.* [35].

[35]: Lau et al. (2010), « Measuring virtual machine detection in malware using DSD tracer »

L'intérêt de ce type d'outil est qu'il permet d'exécuter un malware dans un environnement cloisonné et ressemblant le plus possible à l'environnement ciblé. Il est très rapide de créer, copier, déplacer ou détruire ces machines virtuelles. Les premières VM à être utilisées ne cherchent pas forcément à cacher à la machine invitée qu'elle est virtualisée. Au contraire, certains types de virtualisation comme la paravirtualisation modifient le noyau de l'Operating System (OS) invité afin de gérer plus facilement le partage des ressources. Dans ce cas, l'OS possède l'information que l'environnement sur lequel il s'exécute est virtualisé. Il est donc très simple pour un malware de repérer ce type d'environnement qu'il souhaite éviter. Afin d'analyser un malware, un analyste préférera donc utiliser une VM purement logicielle ou assistée par le matériel. Bien qu'étant plus difficiles à détecter, elles ne sont pas entièrement furtives comme nous le verrons dans la suite de cette partie.

Les hyperviseurs

L'hyperviseur est un programme responsable de la gestion des machines virtuelles. C'est lui qui partage les ressources entre les machines et s'assure du cloisonnement.

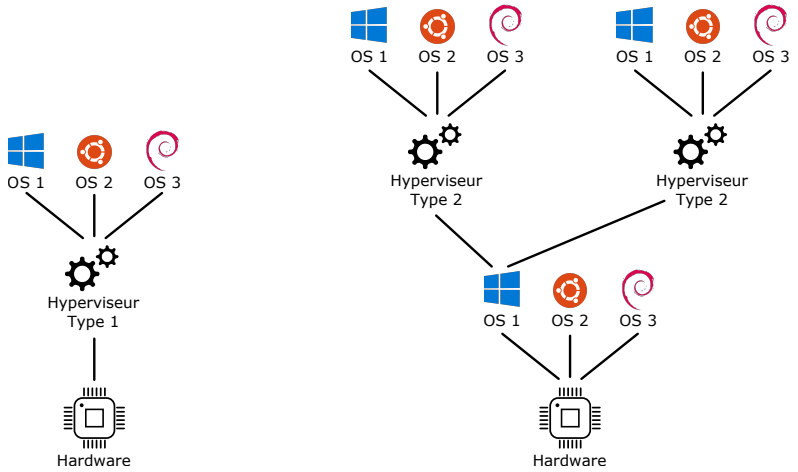


Figure 2.3 – Différences d'architecture entre un hyperviseur de type 1 et un hyperviseur de type 2.

Dans la bibliographie étudiée, les types de virtualisation les plus étudiés utilisent des hyperviseurs de type 2. C'est-à-dire que l'hyperviseur s'exécute sur un système d'exploitation et non sur le matériel directement comme présenté sur la Figure 2.3. Nous présentons donc ici ces environnements ainsi que les techniques d'évasion associées aux hyperviseurs de type 2.

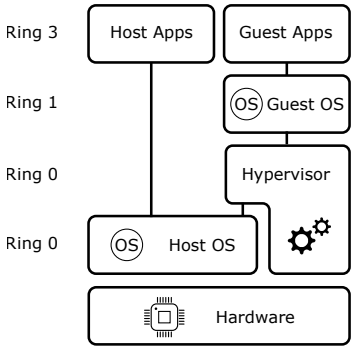


Figure 2.4 – Fonctionnement de la virtualisation purement logicielle. (Source : https://stackoverflow.com/questions/6044978/full-emulation-vs-full-virtualization)

[42]: Charette (2009), *How not to detect virtualization*

4: https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf

La virtualisation purement logicielle

Les malware ont commencé à détecter les environnements virtuels afin de prolonger leur durée de vie en évitant les analyses. Les techniques d'évasion utilisées se basent sur une recherche de différence de comportement entre un environnement virtualisé et un environnement physique. En effet, les premiers jeux d'instruction x86 n'ont pas été pensés pour être virtualisés et des contournements sont mis en place pour créer des VM sur cette architecture. Ceci a été corrigé partiellement par la suite via une extension du jeu d'instruction.

Il est donc important de remettre chaque technique dans son contexte, car elles sont extrêmement dépendantes des technologies utilisées au moment de leur création. Plusieurs techniques de détection de virtualisation publiées en 2004 n'étaient déjà plus exploitables dès 2006 [42].

Le fonctionnement des VM au début des années 2000 est différent de celui des VM des années 2020 [4]. Les premières machines virtuelles sont purement logicielles et le partage des ressources entre la machine hôte et la machine cliente doit donc être géré par un hyperviseur. Nous présentons l'architecture d'une virtualisation purement logicielle dans la Figure 2.4. Pour ce type de virtualisation, la séparation s'effectue en se basant principalement sur trois procédés.

La translation de code permet de traduire à la volée les instructions d'une architecture source vers les instructions d'une architecture de destination. Les appels système privilégiés qui viennent de la machine invitée sont traduits et mis en cache pour améliorer les performances de virtualisation. Les appels non privilégiés ne sont pas modifiés.

Les Shadow Page Table permettent de garder une copie de l'état des pages que doit retrouver la machine invitée. Ces pages sont utilisées pour

éviter à la machine invitée d'accéder aux pages mémoire de la machine hôte et donc de prendre son contrôle.

L'émulation d'Entrées/Sorties permet de créer de faux périphériques comme un clavier, une souris. Cela permet d'émuler la présence de carte réseau ou de port série par exemple.

Aucun de ces procédés qui a pour but de rendre possible la virtualisation de l'architecture x86 n'est parfait d'un point de vue transparence. Des petites différences entre un environnement virtualisé et un environnement physique subsistent. Les premières techniques qui visent à détecter des machines virtuelles chercheront ces petites différences laissées par ces procédés.

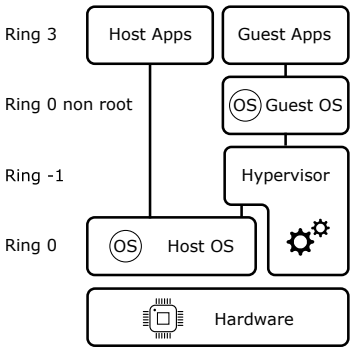


Figure 2.5 – Fonctionnement de la virtualisation assistée par le matériel. (Source : https://stackoverflow.com/questions/6044978/full-emulation-vs-full-virtualization)

La virtualisation assistée par le matériel

La virtualisation assistée par le matériel apparaît plus récemment que la virtualisation purement logicielle. Intel et AMD ont commercialisé les premiers processeurs qui incluent une extension du jeu d'instruction x86 et qui offrent un support de virtualisation respectivement en 2005 et 2006 [5].

5: https://en.wikipedia.org/wiki/X86_virtualization

Ces technologies permettent d'exécuter l'hyperviseur à un niveau plus privilégié que l'OS hôte lui-même comme présenté sur la Figure 2.5. Les appels système sont directement interceptés par l'hyperviseur, ce qui enlève de ce fait la nécessité d'utiliser la translation de code.

Les Shadow Page Tables induisent un overhead important dû à l'émulation des pages mémoire. Pour éviter cela, une technologie appelée Second Level Address Translation (SLAT) ou «*nested paging*» peut être utilisée. Un Translation Lookaside Buffer (TLB) contenant en cache des adresses mémoires virtuelles et leurs adresses mémoires physiques correspondantes est créé. Quand un hyperviseur doit traduire une adresse virtuelle en adresse physique, il peut alors faire une requête au TLB. Si l'information est présente dans le TLB, l'adresse physique est retournée sans traduction ce qui réduit l'overhead.

L'apparition de ces technologies a rendu certaines techniques d'évasion obsolètes [42], mais de nouvelles approches ont alors émergé.

[42]: Charette (2009), *How not to detect virtualization*

Les émulateurs

La dernière technique utilisée pour créer un environnement cloisonné est l'émulation. Dans ce cas, les instructions de la machine invitée ne s'exécutent pas directement sur le processeur de la machine hôte comme pour une machine virtuelle. À la place, les instructions sont exécutées sur un processeur émulé par la machine hôte et qui vise à reproduire le comportement d'un vrai CPU le plus fidèlement possible comme présenté sur la Figure 2.6. C'est le cas par exemple de QEMU qui est capable d'émuler un système entier en incluant les périphériques.

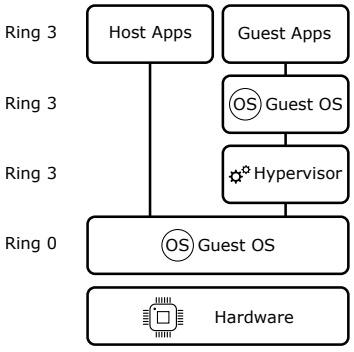


Figure 2.6 – Fonctionnement de l'émulation. (Source : https://stackoverflow.com/questions/6044978/full-emulation-vs-full-virtualization)

La Table 2.1 liste quelques technologies de virtualisation ainsi que leur année d'apparition afin de remettre les techniques d'évasion dans leur contexte. On peut imaginer que les malware évasifs ne se sont pas tout de suite adaptés dès la sortie d'une nouvelle technologie telle qu'Intel

Table 2.1 – Chronologie des outils de virtualisation.

Nom d'outils	Type d'outils de virtualisation	Année d'apparition
Bochs	Émulateur	1994
VMware Workstation	Hyperviseur (type-2)	1999
Xen	Hyperviseur (type-1)	2003
QEMU	Émulateur	2005
Intel VT-x	Extension matérielle	2005
AMD-V	Extension matérielle	2006
VirtualBox	Hyperviseur (type-2)	2007
Rapid Virtualization Indexing	Nested paging	2007
Extended Page Table	Nested paging	2008

VT-x par exemple. Les malware qui ciblent Intel VT-x ont probablement émergé une fois ces technologies adoptées par une majorité d'utilisateurs. Cette arrivée plus tardive de ces technologies peut expliquer le nombre plus faible de techniques d'évasion qui les ciblent.

2.2.2 Techniques d'évasion visant les environnements virtuels

Exemples visant le CPU

[6]: Rutkowska (2004), *Red Pill. . . or how to detect VMM using (almost) one CPU instruction*

La première technique utilisée pour repérer une VM purement logicielle a été présentée par Joanna Rutkowska [6] en 2004 et utilise l'instruction non privilégiée, SIDT. Cette instruction permet de stocker le contenu de l'Interrupt Descriptor Table Register (IDTR) à une adresse mémoire pour le lire. En virtualisation logicielle, le registre IDTR doit être partagé entre l'OS hôte et l'OS invité. Pour une machine virtuelle, il est relocalisé à une adresse assez élevée 0xffXXXXXX (VMWare), 0xe8XXXXXX (VirtualPC). Un IDTR élevé trahit donc la présence de la VM et permet à un malware de changer de comportement en évitant par exemple de déclencher sa payload.

[43]: Issa (2012), « Anti-virtual machines and emulations »

Une autre méthode possible repose sur l'observation des registres CPU dans des systèmes utilisant l'Address Space Layout Randomization (ASLR). Il est prévu que certains registres soient initialisés aléatoirement en évitant certaines valeurs. Par exemple, le registre EAX ne peut être initialisé à 0. Il est donc possible de repérer les émulateurs initialisant ce registre à 0 alors que l'ASLR devrait être activé. Cette attaque est décrite en 2012 par Anoirel Issa [43] qui repère ce comportement dans un malware appelé ZBot.

Exemples basés sur l'observation du temps

Une des attaques les plus simples qui permet de détecter une machine virtuelle consiste à repérer l'overhead induit par la virtualisation. Par exemple pour la virtualisation purement logicielle, le temps d'exécution d'appels système privilégiés est plus long que sur une machine physique.

Cela est dû au traitement des appels système privilégiés effectués dans un mode non privilégié.

Sur une machine physique, l'exception produite par le CPU est retournée au processus par le noyau du système d'exploitation.

Dans le cas d'une virtualisation logicielle, l'exception produite par le CPU est retournée au processus en passant d'abord par un hyperviseur qui émule le comportement attendu pour l'instruction privilégiée ce qui ralentit l'exécution.

Une expérimentation effectuée par Gaël Delalleau [44] démontre qu'il est possible de repérer une machine virtuelle créée à l'aide de VMWare en mesurant l'overhead présent sur les appels privilégiés.

[44]: Delalleau (2004), « Mesure locale des temps d'exécution : application au contrôle d'intégrité et au fingerprinting »

Exemples spécifiques à un logiciel de virtualisation

Chaque machine virtuelle doit être isolée du système hôte ainsi que des autres VMs. L'hyperviseur doit donc faire croire à chaque VM qu'elle possède son propre CPU, disque dur, périphérique. Lorsque VMWare émule du matériel, un nom par défaut est choisi et peut permettre à un malware de détecter qu'il s'exécute sur un environnement virtualisé. Par exemple, VMWare crée un périphérique Integrated Drive Electronics (IDE) s'appelant *VMware Virtual IDE Hard Drive* que l'on ne peut pas renommer sur les versions antérieures à la 4.5 et qui permet facilement de trahir la présence d'une VM [45]. Une autre possibilité pour cacher les VM consiste à modifier le binaire de VMWare afin de changer les noms de périphériques utilisés par défaut [9].

[45]: Holz et al. (2005), « Detecting honeypots and other suspicious environments »

[9]: kortchinsky (2004), *Counter measures to VMware fingerprinting*

2.2.3 Exemples visant les debuggers

Les machines virtuelles permettent seulement d'exécuter les malware dans un environnement cloisonné et le plus ressemblant possible à une machine physique, mais ne permettent pas de faire une analyse fine du fonctionnement du malware. Pour cela, les analystes utilisent des debuggers qui permettent d'inspecter un programme pas à pas ou en le stoppant à une instruction précise.

L'usage premier des debuggers était fait par les développeurs qui souhaitent corriger les bugs de leurs programmes. Ils seront ensuite utilisés par les analystes non pas pour déboguer des programmes, mais pour faire de la rétro-ingénierie de malware afin de comprendre leur fonctionnement.

Contrairement à une machine virtuelle qui peut être utilisée par des non-informaticiens, car elle peut permettre de résoudre des problèmes de compatibilité entre un logiciel et un système d'exploitation, un debugger est presque exclusivement utilisé par des connaisseurs.

Ce sont des outils qui ont peu de raison d'être présents sur des ordinateurs privés. Un attaquant a par conséquent intérêt à éviter de s'exécuter en présence de ce type d'outil.

Les attaques qui visent les debuggers font partie des premiers moyens de détecter une analyse en cours. Des techniques sont par exemple décrites par Holz et. al. dès 2005 [45].

[45]: Holz et al. (2005), « Detecting honeypots and other suspicious environments »

Un moyen pour un debugger de placer des points d'arrêts consiste à utiliser l'instruction `INT3` en remplaçant l'instruction où l'analyste

souhaite s'arrêter par l'opcode `0xCC` [6]. Cette modification est visible par le malware qui pourra par conséquent analyser la mémoire à la recherche de cet opcode et changer son comportement en conséquence.

L'une des méthodes les plus simples pour un malware évasif qui cible Windows de détecter la présence d'un debugger est d'utiliser la fonction de l'API Windows `IsDebuggerPresent()` qui retourne `true` si le programme qui appelle cette fonction est entrain d'être débogué [7].

2.2.4 Divers

Bien que les techniques qui visent la détection de machines virtuelles et de debuggers soient les plus présentes dans la littérature, d'autres cibles existent dans une moindre mesure. Nous détaillons ici les techniques qui ciblent les antivirus et celles qui effectuent un test de Turing inversé. Bien que moins présentes, elle sont utilisées dans des outils tels qu'Al-Khaser [8] ou Pafish [9] et qui permettent de tester plusieurs techniques d'évasion sur son environnement dont on souhaite évaluer la furtivité par exemple. Ces outils utilisent également des techniques plus classiques de détection de debuggers et de machines virtuelles.

Antivirus

Une méthode qui permet à un antivirus de surveiller l'activité d'un système consiste à rediriger certains appels système vers des fonctions qui effectueront un prétraitement et pourront par exemple lever une alerte si quelque chose de suspect est repéré. Il est notamment intéressant pour un antivirus de surveiller les créations et manipulations de fichiers sur le système.

Une ancienne méthode pour effectuer cela consiste à placer des hooks dans la System Service Dispatch Table (SSDT). Cette table, présente dans l'espace noyau et accessible en lecture seule pour les applications de l'espace utilisateur, contient un tableau de pointeurs de fonctions du noyau Windows. Afin de surveiller la création de fichiers, il suffit donc de remplacer le pointeur de `ZwCreateFile` dans la SSDT par sa propre fonction.

Tan [46] rapporte que cette technique est utilisée par des outils de sécurité au début des années 2000. Dans le même article, Tan démontre qu'il est possible de restaurer la table SSDT depuis un programme qui possède seulement les droits utilisateur. Un malware évasif pourrait donc lire le contenu de la SSDT et arrêter son exécution ou restaurer la table originale s'il repère une modification.

La modification de la table SSDT n'est cependant plus possible sur les systèmes x64 depuis la mise en place par Microsoft du KPP Kernel Patch Protection (KPP) qui empêche toute modification du noyau [10] y compris de cette table [47].

6: https://www.intel.fr/content/www/fr/fr/architecture-and-technology/64-ia-32-architectures-software-developer-vol-2a-manual.html - Vol. 2A 3-457

7: https://docs.microsoft.com/en-gb/windows/win32/api/debugapi/nf-debugapi-isdebuggerpresent?redirectedfrom=MSDN

8: https://github.com/LordNotoworthy/al-khaser

9: https://github.com/a0rtega/pafish

[46]: Tan (2004), « Defeating Kernel Native API Hookers by Direct Service Dispatch Table Restoration »

10: https://fr.wikipedia.org/wiki/Kernel_Patch_Protection

[47]: Ligh et al. (2014), *The Art of Memory Forensics : Detecting Malware and Threats in Windows, Linux, and Mac Memory*

Test de Turing

Un test de Turing vise à faire converser un humain C à l'aveugle avec un autre humain B et une machine A programmée pour se faire passer pour un humain. Si l'humain C est incapable de distinguer quel interlocuteur est la machine alors la machine réussit le test de Turing. Ce test est représenté sur la Figure 2.7.

Un moyen rapide pour un analyste d'avoir une idée du comportement d'un malware est d'utiliser une plateforme d'analyse dynamique comme AnyRun [11] ou HybridAnalysis [12]. Ce genre de plateformes peut utiliser des sandboxes pilotées de manière automatique. Un exemple d'automatisation est la manipulation de la souris qui permet d'interagir avec une interface graphique.

Une difficulté pour les plateformes d'analyse est de simuler un comportement humain crédible et donc de réussir le test de Turing. Souvent, les transitions de la souris d'un point à un autre de l'écran seront effectuées de manière trop parfaite pour être le résultat d'une action humaine. Le mouvement peut sauter directement au point suivant sans passer par des points intermédiaires, être parfaitement rectiligne ou trop rapide.

Pour s'évader, un malware pourra effectuer un test de Turing en observant par exemple les mouvements de la souris et vérifier qu'ils correspondent à ceux générés lorsqu'un humain manipule la souris. Si le malware constate que tous les mouvements sont trop rapides pour être générés par un humain, il pourra décider de ne pas déclencher sa payload [13].

11: https://any.run

12: https://www.hybrid-analysis.com

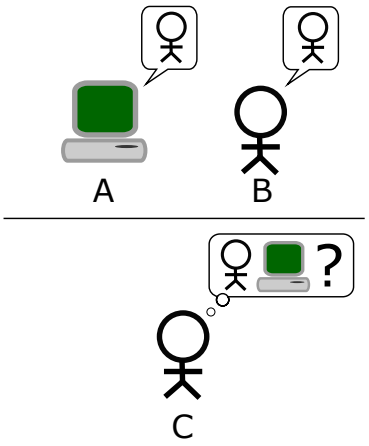


Figure 2.7 – Test de Turing. (Source : https://upload.wikimedia.org/wikipedia/commons/e/e4/Turing_Test_version_3.png)

13: https://www.fireeye.com/blog/threat-research/2014/06/turing-test-in-reverse-new-sandbox-evasion-techniques-seek-human-interaction.html

2.3 Dataset de malware évasifs existants

En effectuant de la rétro-ingénierie, il est possible de découvrir et comprendre de nouvelles méthodes d'évasion. Pour cela, construire ou récupérer un dataset de malware évasifs est essentiel.

Un dataset est également utile pour tous les champs de recherche. En ayant un nombre d'échantillons suffisamment important et varié, il est possible de construire une vérité terrain. Cette vérité terrain permettra par exemple d'évaluer des méthodes de détection ou de classification. Il est possible de tirer des informations concernant les habitudes des auteurs de malware évasifs ou sur l'évolution de la technique à partir d'un dataset qui couvre une période importante. Enfin, un dataset varié permettra de valider et évaluer le bon fonctionnement de contre-mesures et analyses furtives.

2.3.1 Description des datasets de la bibliographie

Comme aucun dataset n'est disponible publiquement, nous avons étudié comment ceux présentés dans les différents articles du domaine créaient ou récupéraient les leurs.

Nous avons relevé les méthodes de construction des datasets de malware évasifs pour tous les articles présents dans la bibliographie. Nous relevons la provenance des malware, mais également des logiciels légitimes s'il y en a. Nous étudions également les méthodes de construction des vérités

terrain utilisées. Les relations entre les articles sont également étudiées, par exemple quand un dataset est transmis d'article en article.

Source des malware

La source de provenance des malware varie beaucoup entre chaque article. Nous relevons quatre types de sources possibles parmi les références étudiées et énonçons quelques avantages et inconvénients.

Il est possible de créer un accord avec une *entreprise privée*. Les données récupérées dépendront de ce qui a été convenu avec l'entreprise. Cela peut aller d'une centaine de malware jusqu'à plusieurs millions. Certains datasets peuvent être partagés avec des métadonnées comme un attribut qui désigne la capacité d'un échantillon à s'évader.

Une source de malware fréquente est la *plateforme d'analyse publique*. Par exemple, beaucoup d'articles utilisent des malware fournis par la plateforme Anubis jusqu'à sa fermeture en 2016 [14]. Il était par exemple possible d'envoyer des exécutables Windows et de recevoir un rapport concernant les actions effectuées par le programme [15].

Une solution pour récolter des malware sans avoir à passer d'accord avec une entreprise privée ou une plateforme d'analyse consiste à visiter certains *sites web* connus pour héberger de potentiels malware. C'est le cas par exemple de GitHub [16] qui est parfois utilisé par les auteurs de malware pour stocker leurs codes et par quelques utilisateurs qui répertorient les malware.

Enfin, les auteurs peuvent également s'appuyer sur de *précédents articles* en utilisant le dataset produit par leurs expérimentations passées.

La Table 2.2 détaille les sources de malware utilisées par chaque article qui possède un dataset.

Source des goodware

De manière plus ponctuelle, certains articles ont besoin d'un dataset qui contienne des logiciels légitimes. Cela peut être le cas pour les travaux concernant la détection de malware évasifs. Dans ce cas, un dataset de logiciels légitimes peut être utilisé afin de vérifier qu'ils ne sont pas considérés à tort par la solution de détection comme étant évasifs.

Tout comme pour les malware, il est possible de récupérer des programmes légitimes sur des *sites web*. ScareCrow [40] récupère par exemple des échantillons sur le site CNET [17].

D'autres créent un dataset en récupérant les programmes présents *localement* dans le dossier `Windows/System32` [26].

14: https://assiste.com/Sandboxes_gratuites_en_ligne/Anubis.html

15: https://www.lastline.com/blog/from-anubis-and-wepawet-to-llama

16: https://github.com

[40]: Zhang et al. (2020), « Scarecrow : Deactivating Evasive Malware via Its Own Evasive Logic »

17: https://download.cnet.com

[26]: Naval et al. (2014), « Environment-Reactive Malware Behavior : Detection and Categorization »

Construction de la vérité terrain

Un élément commun à beaucoup d'articles et nous semblant important est la façon de construire la vérité terrain de malware évasifs qui est utilisée dans beaucoup de sujets de recherche. Il s'agit d'un ensemble de malware que l'on sait évasifs avec certitude. Cette vérité terrain peut servir par exemple à paramétrer une solution de détection de malware évasifs. Une autre vérité terrain permettra ensuite d'évaluer l'efficacité de cette détection.

Aucune vérité terrain de malware évasifs n'étant disponible publiquement, beaucoup d'articles en construisent une eux-mêmes. Pour cela, certains auteurs ont par exemple analysé les échantillons manuellement. Les méthodes employées pour effectuer les analyses ne sont souvent pas décrites en détail.

D'autres font confiance aux plateformes d'analyse qui leur ont distribué les malware et qui ont déjà étiqueté les malware avec leur propre méthode qui n'est souvent pas communiquée dans l'article. Certaines plateformes fournissent également des métadonnées associées à chaque malware et qui peuvent être utilisées par les chercheurs afin de définir si un malware est évasif ou non.

Il existe également des règles YARA [18] conçues pour détecter les malware évasifs et que l'on peut appliquer sur un ensemble de malware.

18: https://virustotal.github.io/yara

Enfin, des auteurs créent artificiellement des malware évasifs en appliquant un packer connu comme étant évasif sur des malware non évasifs.

Partage des datasets

La majorité des datasets de malware évasifs n'est pas partagée publiquement. Nous avons toutefois relevé des cas de partage, par exemple entre BareCloud [28], MalGene [33] et ScareCrow [40]. La Figure 2.8 représente le passage du dataset d'article en article, ainsi que les transformations effectuées à chaque passage.

Plusieurs auteurs ont mentionné l'impossibilité d'évaluer les solutions décrites dans leurs articles par manque de vérité terrain partagée publiquement.

[28]: Kirat et al. (2014), « BareCloud : Bare-metal Analysis-based Evasive Malware Detection »

[33]: Kirat et al. (2015), « MalGene : Automatic Extraction of Malware Analysis Evasion Signature »

[40]: Zhang et al. (2020), « Scarecrow : Deactivating Evasive Malware via Its Own Evasive Logic »

2.3.2 Biais liés aux vérités terrain

Nous avons listé différents biais présents dans les datasets décrits dans notre bibliographie. Ces biais peuvent avoir lieu lors de la construction d'un dataset ou de l'évaluation d'une expérimentation par exemple.

Les biais cognitifs que nous décrivons ici sont listés en détail par Buster Benson dans sa *Cognitive bias cheat list* [19].

19: https://betterhumans.pub/cognitive-bias-cheat-sheet-55a472476b18#.wru9kyadv

La Figure 2.8 est utilisée pour présenter des exemples des trois biais présentés ici.

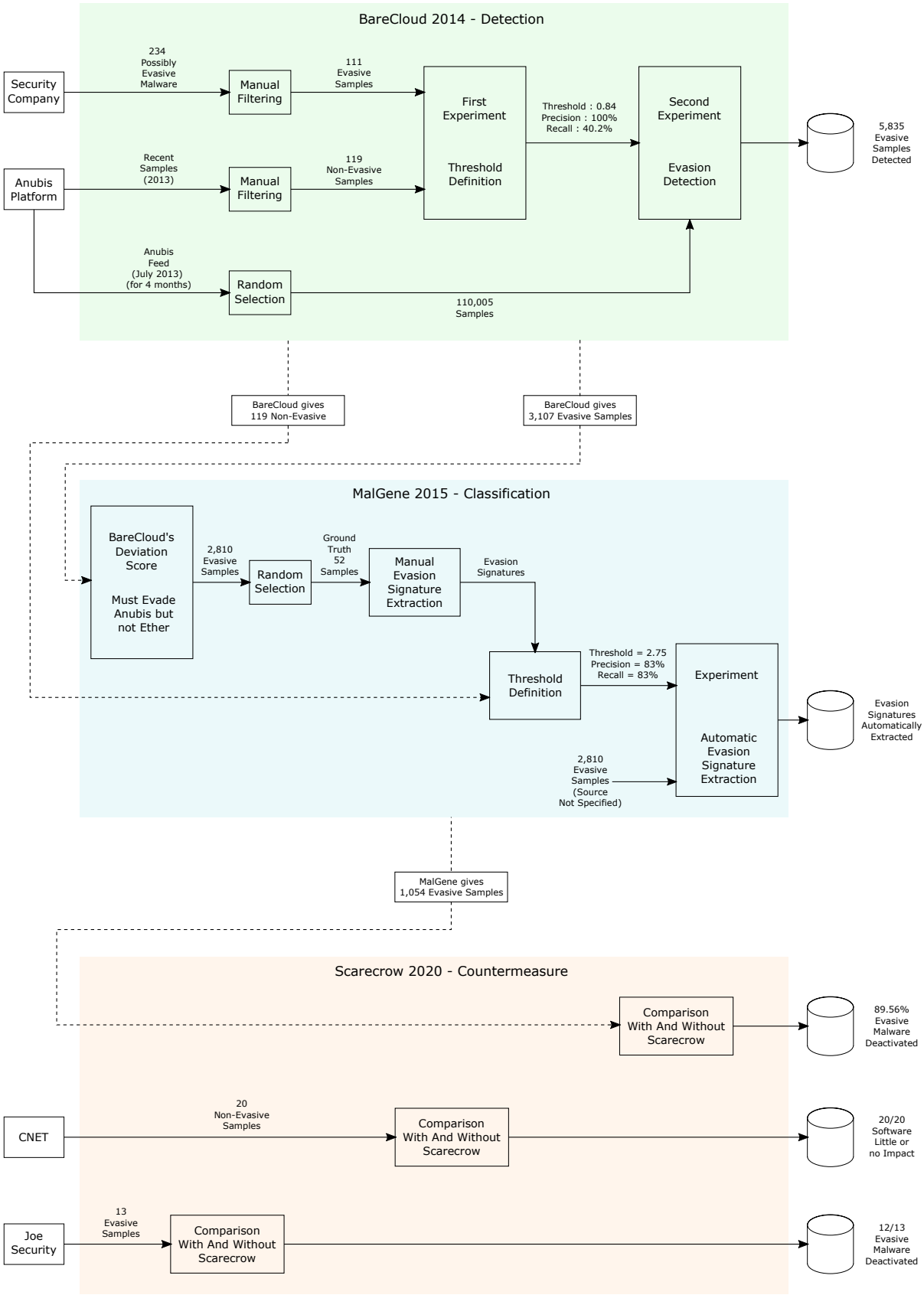


Figure 2.8 – Exemple d'un dataset utilisé dans une suite d'articles.

Effet expérimentateur

L'effet expérimentateur est un biais qui consiste pour un expérimentateur à influencer les résultats d'une expérimentation dans le but de valider son hypothèse de manière consciente ou non.

Plusieurs datasets de malware évasifs ont été construits à la main par des expérimentateurs qui ont étiqueté un ensemble de malware. C'est le cas de BareCloud qui contient une étape de filtrage manuel peu détaillé dans l'article. Il est possible que les résultats soient différents entre les expérimentateurs en fonction des critères utilisés pour étiqueter un malware comme évasif ou non.

Ces critères qui permettent de décider si un malware est évasif ou non ne sont souvent pas détaillés dans les articles. Il est donc impossible de s'assurer de la qualité de la vérité terrain, l'étiquetage étant potentiellement biaisé par ce que l'expérimentateur souhaite sélectionner.

Le problème est que la solidité du dataset dépend souvent de ce filtrage manuel. Les biais présents dans cet ensemble de malware filtrés à la main seront probablement présents dans le dataset généré à partir de ces malware.

Biais de sélection

Le biais de sélection désigne l'erreur faite dans la construction du dataset étudié lorsqu'il n'est par exemple pas randomisé ou pas représentatif du sujet étudié.

Il est important de posséder une vérité terrain qui contient des informations aussi fiables et variées que possible. Considérons par exemple une vérité terrain qui contient exclusivement des malware qui évitent VirtualBox. Un outil de détection paramétré à partir de cette vérité terrain serait probablement efficace contre les malware qui ciblent VirtualBox, mais inefficace contre les malware qui ciblent d'autres outils de virtualisation. Si le but était de pouvoir détecter un maximum de malware évasifs de tous types, cette vérité terrain n'est pas adaptée.

Nous avons vu précédemment que les techniques d'évasion ont rapidement évolué au fil des années. Ainsi, un malware créé en 2000 fonctionnera très différemment d'un malware qui date de 2020. Cela a son importance au moment de la création d'une vérité terrain. Prenons l'exemple de l'évaluation des performances d'une solution de détection à l'aide d'une vérité terrain. Si la solution vise à détecter tout type de malware depuis le début des années 2000 à nos jours, alors la vérité terrain devra être représentative de cette période. En effet, évaluer une telle solution avec une vérité terrain qui contienne majoritairement d'anciens malware permettra seulement de conclure de l'efficacité de la solution sur ces anciens malware. Il manquera une évaluation pour les malware plus récents.

Un exemple concret est le dataset utilisé par ScareCrow en 2020. Les malware évasifs utilisés en entrée ont été fournis par MalGene. L'article qui décrit MalGene précise que les malware évasifs sont sélectionnés pour s'évader d'Anubis, mais pas d'Ether. Il est probable que les malware donnés par MalGene à ScareCrow sont ces mêmes malware.

Or les malware que possède MalGene ont été récoltés par BareCloud en 2014. Scarecrow a donc été probablement évalué en utilisant des malware qui ont au minimum cinq ans et visent une plateforme fermée depuis 2016. Il est donc très difficile de généraliser les résultats de cette contre-mesure sur des malware évasifs modernes, car il est probable que leur fonctionnement soit différent de ceux qui datent de 2014 et ciblent Anubis.

Biais des survivants

Le biais du survivant consiste à surévaluer les résultats d'une méthode en se focalisant sur les éléments qui ont eu des résultats positifs sur cette méthode et en ignorant ceux qui ont des résultats négatifs.

Un exemple est présent dans l'article BareCloud où une première expérimentation permet de définir un seuil qui permet d'avoir une précision de 100% et une sensibilité de 40%. Cela veut dire qu'en appliquant ce seuil, la seconde expérimentation pourra a priori détecter seulement 40% de tous les malware présents dans le flux de malware à détecter. En revanche, les malware détectés comme évasifs seront presque exclusivement véritablement évasifs.

Une partie de ces malware est réutilisée dans un deuxième article qui décrit la solution MalGene. Le problème est que cette expérimentation utilise donc des malware qui ont des résultats positifs à l'expérimentation faite par BareCloud. Ces malware «survivants» ne sont représentatifs que de 40% des malware évasifs disponibles en entrée de BareCloud et non 100%.

De plus, seuls 3 107 malware sur les 5 835 malware évasifs détectés par BareCloud sont réutilisés dans l'expérimentation de MalGene. Aucune explication n'est donnée concernant l'utilisation partielle des malware évasifs disponibles.

MalGene donne à son tour 1 054 malware évasifs à ScareCrow sans expliquer comment est faite la sélection sur les 3 107 malware possédés par MalGene.

Il est difficile de savoir de quoi est composé le dataset «survivant» et donc de tirer des conclusions sur les résultats des expérimentations qui l'utilisent.

2.3.3 Vérité terrain non communiquée

L'absence d'échange de vérité terrain entraîne plusieurs problèmes freinant les avancées dans le domaine. Nous détaillons trois points qui nous semblent les plus importants et auxquels nous pouvons apporter une solution en fournissant un dataset public.

Vérification de la présence de biais

L'absence d'informations sur le contenu des datasets utilisés empêche d'étudier de quoi il est composé. Il est donc très difficile de repérer la présence volontaire ou non d'éventuels biais. Il est également compliqué d'évaluer si le dataset a été créé de manière à orienter les résultats d'une certaine manière.

Reproductibilité et comparaison des résultats

Une manière de valider une solution est de la faire reproduire par une autre équipe. Des reproductions avec succès renforcent la confiance que l'on peut avoir dans les résultats. Or les vérités terrain et les solutions n'étant pas communiquées, il est très difficile pour quelqu'un qui souhaite répéter l'expérimentation de le faire.

Une pratique fréquente dans le domaine de l'intelligence artificielle par exemple consiste à comparer deux solutions. Pour cela, les solutions utilisent souvent un dataset commun qui permet de comparer leurs efficacités. Cela permet d'améliorer les algorithmes au fil des publications. Aucun dataset de malware évasif public n'étant disponible, il n'est pas possible d'effectuer cette comparaison simplement.

Perte de temps pour créer une vérité terrain

La détection de malware évasifs est un sujet de recherche à part entière. Comme il n'existe pas de datasets disponibles, une personne qui souhaite créer une contre-mesure devra d'abord créer son propre dataset. Il est possible d'étudier des malware à la main pour sélectionner des malware évasifs ou de créer une expérimentation dans ce but. Il est également possible de passer un accord avec un organisme extérieur qui pourrait fournir un dataset.

Toutes ces possibilités prennent du temps et demandent un effort important. Cela pourrait expliquer pourquoi les articles étudient majoritairement les techniques d'évasion et la détection de malware évasifs.

Table 2.2 – Détail des datasets de malware présents dans la littérature.

Article	Catégorie	Sources	Échantillons
BareCloud [28]	Privée	?	224 possiblement évasifs
	Plateforme	Anubis	119 non-évasif 110 005 nature inconnue
MalGene [33]	Précédent article	BareCloud	119 non-évasifs 3 107 évasifs
ScareCrow [40]	Précédent article	MalGene	1 054 évasifs
	Plateforme	JoeSecurity	13 évasifs
Environment Reactive [26]	Web	?	1120 malware
	Privé	User agencies	
Evasive Windows Malware [39]	Web	?	18 possiblement évasifs
Detecting Environment [32]	Plateforme	Anubis	185 échantillons 1 686 échantillons
Detecting Malware [30]	Plateforme	Offensive Computing VirusShare Malwar ?	69 malware évasifs
	Web	?	22 common malware
Efficient Detection [23]	Plateforme	Anubis	10 évasifs
Detecting & Defeating [29]	Web	?	8 évasifs
Using Hardware [17]	Web	?	10 packing tools
Scalability [16]	Plateforme	ShadowServer	1000 échantillons
	Web	?	1 échantillons avec rapport 1 échantillons analysé manuellement

Contributions

Des Malware Évasifs 3

Beaucoup d'informations sont disponibles concernant les techniques d'évasion, mais nous en possédons peu concernant leur facilité d'implémentation et leur efficacité contre de réels outils d'analyse.

Dans ce chapitre, nous étudions le fonctionnement des antivirus dans la Section 3.1 et celui de quelques techniques d'évasion dans la Section 3.2. Nous implémentons ensuite ces techniques dans un malware de démonstration que nous décrivons dans la Section 3.3. Dans cette même section, nous étudions la résistance des antivirus à ce malware. Nous récoltons ensuite un petit ensemble de malware évasifs que nous détaillons dans la Section 3.4.

3.1 Le fonctionnement des antivirus

Les antivirus sont des outils qui visent à détecter et stopper l'exécution de programmes malveillants. Ils sont un agrégat de plusieurs éléments conçus pour détecter la présence de malware par différents moyens, comme décris par Koret *et al.* [48]. Ces différents éléments induisent une consommation de ressources supplémentaires qui, si trop importante, peut limiter l'usage de programmes légitimes. Afin de rester compétitifs, les éditeurs d'antivirus doivent limiter cette surcharge. Il leur est difficile pour cette raison d'utiliser des techniques qui demandent un long temps d'exécution.

[48]: Koret et al. (2015), *The Antivirus Hacker's Handbook*

La plupart des antivirus utilisent des analyses par signature appliquées sur tout ou partie de chaque fichier puis comparées à leur base. Cette base de signatures a besoin d'être mise à jour régulièrement pour détecter les malware les plus récents [1].

1: https://www.kaspersky.co.uk/blog/the-wonders-of-hashing/3629/

Une autre méthode consiste à analyser dynamiquement un malware pendant son exécution. Cependant, ces analyses ont le plus souvent lieu dans des environnements cloisonnés tels que des émulateurs, qui rajoutent une consommation de ressources supplémentaire [2].

2: https://eugene.kaspersky.com/2012/03/07/emulation-a-headache-to-develop-but-oh-so-worth-it/

Pendant leur installation, les antivirus ajoutent des fichiers et modifient le comportement du système d'exploitation. Un antivirus peut par exemple charger des modules en utilisant la fonction de l'API Windows *LoadLibrary*, pour charger un fichier Dynamic Link Library (DLL), comme décrit par Koret *et al.* [48]. Comme nous le détaillons dans la section suivante, les malware évasifs peuvent détecter ces modifications et ensuite adapter leur comportement.

[48]: Koret et al. (2015), *The Antivirus Hacker's Handbook*

Après avoir détecté un malware, les antivirus l'envoient à leurs serveurs pour effectuer des analyses plus complètes. Les analystes peuvent alors utiliser plusieurs outils comme un debugger pour arrêter le programme à une instruction particulière ou des machines virtuelles et des émulateurs pour l'analyser dans un environnement cloisonné.

Un attaquant qui souhaite créer un malware durable peut alors essayer de compliquer le travail de l'analyste en utilisant des techniques anti-debugger ou anti-VM telles que celles détaillées dans la Section 2.2 afin de ralentir son analyse.

3.2 Évasion des antivirus

Nous détaillons maintenant les techniques aisément implémentables par un attaquant pour détecter des traces d'antivirus. Koret *et al.* [49], Catalin-Valeriu *et al.* [50] et Afianian *et al.* [51] détaillent plusieurs techniques d'évasion. De plus, de multiples projets qui traitent de ces techniques de détection sont disponibles sur Github tels qu'Al-Khaser par exemple [3]. Ce projet est un logiciel qui permet d'exécuter de multiples techniques d'évasion et d'évaluer leur efficacité sur un environnement d'analyse que l'on souhaite tester par exemple.

[49]: Bulazel et al. (2017), « A Survey On Automated Dynamic Malware Analysis Evasion and Counter-Evasion : PC, Mobile, and Web »

[50]: Lita et al. (2018), « Anti-emulation trends in modern packers : a survey on the evolution of anti-emulation techniques in UPA packers »

[51]: Afianian et al. (2019), « Malware Dynamic Analysis Evasion Techniques : A Survey »

3: https://github.com/LordNotoworthy/al-khaser

Nous classons ces techniques en trois catégories en fonction de l'outil ciblé : détection de debuggers, détection d'artéfacts d'installation et d'exécution des antivirus, et détection de machines virtuelles. Dans cette section, nous détaillons ces catégories et présentons quelques artéfacts laissés par les antivirus.

3.2.1 Détection de debuggers

Un debugger peut permettre d'ajouter un point d'arrêt à une instruction spécifique et analyser un malware en détail. Pour ce faire, le programme d'analyse doit prendre le contrôle du processus debugué. Cette prise de contrôle laisse des traces visibles depuis le processus debugué. Ce processus peut être un malware évasif, qui peut alors savoir s'il subit un debugging et modifier son comportement en conséquence. Ces traces peuvent, par exemple, être l'initialisation de registres x86 spécifiques ou le lancement de processus de debugging en mémoire. L'architecture prévoit par exemple des registres Dr0 jusqu'à Dr7 dont les quatre premiers permettent de placer des points d'arrêt matériels. Un malware peut lire la valeur de ces registres et ainsi repérer une éventuelle existence de points d'arrêt qui trahissent la présence d'un debugger.

3.2.2 Détection des artéfacts d'installation et d'exécution des antivirus

L'installation d'antivirus crée des changements dans le système d'exploitation. Pour Windows, cela commence par la création d'un dossier souvent dans le dossier `Program Files`. Ce dossier contient l'exécutable de l'antivirus ainsi que les ressources dont il a besoin. Pendant l'installation d'un antivirus, des clefs de registre peuvent être ajoutées ou modifiées. Des instrumentations de certaines fonctions de l'API Windows peuvent avoir lieu dans l'espace utilisateur ou noyau.

De la même manière, l'exécution du processus principal de l'antivirus et de ses plug-ins laisse des artéfacts au sein du système d'exploitation qui sont visibles au niveau utilisateur. Une technique simple qui permet de

détecter la présence d'un antivirus est alors de chercher en mémoire les processus dont le nom correspond à un antivirus connu. Un malware évasif pourra alors chercher ces artéfacts pour savoir s'il est en présence d'un antivirus et adapter son exécution.

3.2.3 Détection de VMs, émulateurs, sandboxes

Pour finir, un antivirus peut exécuter le malware dans une sandbox, *i.e.*, un environnement de test contrôlé et contenu dans lequel le comportement de l'échantillon peut être soigneusement analysé. Ces sandboxes tentent de reproduire l'environnement ciblé le plus fidèlement possible, mais de légères différences permettent de le distinguer d'un environnement réel. Cela peut être la présence d'une interface réseau virtuelle par exemple. Un malware évasif peut alors savoir s'il s'exécute sur une machine virtuelle ou une sandbox en cherchant ces différences et changer son exécution.

3.3 Création d'un malware évasif test

Nous avons créé un ransomware évasif appelé Nuky et spécialement conçu pour s'évader en présence d'outils d'analyse. Ce malware a pour but principal de valider le bon fonctionnement de la contre-mesure présentée dans le Chapitre 4, mais nous en profitons également pour évaluer l'impact de ce type de malware sur des antivirus modernes.

3.3.1 Nuky, un malware configurable

Nuky est un ransomware évasif et configurable qui est composé de deux parties dont la première contient différentes techniques d'évasion et la seconde plusieurs payloads. Un schéma qui présente ces deux parties est visible sur la Figure 3.1.

La partie évasion est composée de trois blocs qui implémentent des techniques qui recherchent plusieurs artéfacts que nous listons dans la Table 3.1.

Le bloc qui cible les debuggers recherche des noms de processus, noms de fenêtres, valeurs de registres CPU, certaines fonctions importées ainsi que des noms et registres Windows liés à des debuggers.

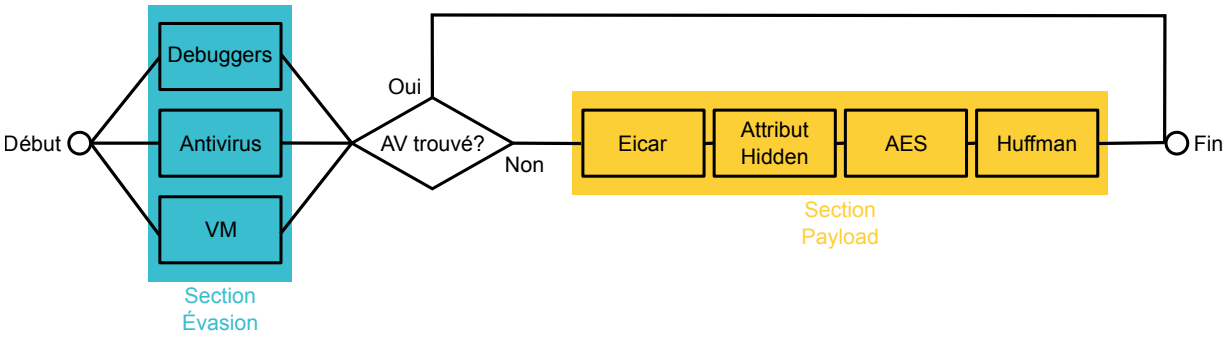


Figure 3.1 – Diagramme du fonctionnement de Nuky.

Table 3.1 – Artéfacts recherchés par Nuky dans chaque catégorie.

Artéfacts	Debugger	Antivirus	VM
Noms de processus	×	×	×
Noms des fenêtres	×		
Valeurs de registres CPU	×		
Fonctions importées	×		×
Noms et valeurs de registres Windows	×		
Noms de dossiers		×	×
Noms de .DLL			×
Noms d'utilisateurs			×
Adresses MAC			×

Table 3.2 – Liste des payloads de Nuky.

Type	Méthode
Chiffrement	AES ECB mode
Compression	Huffman
Cache	Passe l'attribut caché à vrai
Test	Dépose un fichier Eicar sur le Bureau

Le second bloc recherche des noms de processus et de dossiers qui appartiennent à des antivirus.

Le dernier bloc cherche à détecter des noms de processus, de dossiers, de librairies ou d'utilisateurs liés à des machines virtuelles. Des fonctions importées ainsi que des adresses Media Access Control (MAC) sont également recherchées.

Ces blocs peuvent être lancés ensemble ou indépendamment. Nuky peut par exemple être exécuté en activant le bloc qui cible les debuggers et celui qui recherche des machines virtuelles.

La partie payload détaillée dans la Table 3.2 possède quatre payloads conçues pour modifier les fichiers contenus dans `Mes Documents`. Ces blocs sont activables indépendamment en fonction de la configuration voulue. Nous concevons ces payloads pour être les plus voyantes possibles des antivirus.

La première payload chiffre les documents à l'aide d'un algorithme Advanced Encryption Standard (AES) [52] en mode Electronic Code Book (ECB). Ce mode de fonctionnement découpe le fichier en blocs qui sont tous chiffrés avec la même clef. Les fichiers chiffrés ainsi possèdent encore des informations sur le fichier en clair et plusieurs attaques sont possibles pour les déchiffrer. Une comparaison du chiffrement d'une image à l'aide d'AES d'abord en mode ECB puis en mode Cipher Block Chaining (CBC) est présentée sur la Figure 3.2. Cela pourrait donner des chances supplémentaires aux antivirus de repérer cette payload. Nous choisissons une implémentation dont le code est publiquement accessible sur GitHub [4] et dont les antivirus peuvent par conséquent avoir créé des signatures pour détecter cette librairie.

[52]: Standards et al. (2001), « Advanced Encryption Standard »

4: https://github.com/SergeyBel/AES/

La seconde permet de les compresser à l'aide de l'algorithme de codage de Huffman [53]. L'implémentation choisie est également disponible publiquement sur GitHub [5] afin d'augmenter les chances de détection de cette payload par les antivirus.

[53]: Huffman (1952), « A Method for the Construction of Minimum-Redundancy Codes »

5: https://github.com/nayuki/Reference-Huffman-coding

La troisième payload change les attributs «caché» à vrai ce qui permet de rendre les fichiers invisibles dans l'interface graphique. La fonction `SetFileAttributesA` [6] de l'API Windows est utilisée pour placer

6: https://docs.microsoft.com/en-gb/windows/win32/api/fileapi/nf-fileapi-setfileattributesa

(a) Image originale avant chiffrement.

(b) Image chiffrée avec AES en mode ECB.
openssl enc -aes-128-ecb -e -in logo.bmp -out ECB.bmp -K 1001011
dd conv=notrunc if=logo.bmp of=ecp.bmp bs=8 count=16

(c) Image chiffrée avec AES en mode CBC.
openssl enc -aes-128-cbc -e -in logo.bmp -out CBC.bmp -K 1001011
-iv 0010011
dd conv=notrunc if=logo.bmp of=cbc.bmp bs=8 count=16

Figure 3.2 – Comparaison du chiffrement d'une image à l'aide d'AES d'abord en mode ECB puis CBC. Nous choisissons d'utiliser le mode ECB car une partie des informations du fichier clair peut être retrouvée à partir du fichier chiffré ce qui pourrait éventuellement aider un antivirus à détecter un chiffrement.

l'attribut `FILE_ATTRIBUTE_HIDDEN` sur les fichiers à cacher.

La dernière payload dépose un fichier Eicar sur le Bureau [7]. Ce fichier est créé par l'European Institute for Computer Antivirus Research (EICAR) et la Computer Research Organization Antivirus (CARO) pour tester les antivirus sans avoir à utiliser des malware réels qui pourraient causer des dégâts. Pour obtenir un fichier EICAR, il suffit de créer un fichier texte qui contienne la chaîne de caractères présentée dans le Listing 3.1.

7: https://www.eicar.org/?page_id=3950

Listing 3.1: Contenu du fichier EICAR

```
X50!P%@AP[4\PZX54(P^)7CC)7}
$EICAR-STANDARD-ANTIVIRUS
-TEST-FILE!$H+H*
```

Dans la suite, nous configurons Nuky de manière à tester les capacités des malware à s'évader. Pour cela, nous utilisons des techniques d'évasion théoriques présentées dans la Section 3.2 et utilisant les fonctions de l'API Windows détaillées dans la Section 4.2.

3.3.2 Capacité des antivirus à détecter un malware évasif

Nous profitons de posséder un malware évasif pour tester l'efficacité de nos trois blocs de techniques d'évasion sur un panel d'antivirus. Ces antivirus que nous listons dans la Table 3.3 sont choisis, car ils figurent parmi les plus populaires du moment selon AVTest [8].

8: https://www.av-test.org/en/antivirus/home-windows/

Nous créons une image Windows 10 par antivirus pour les tester indépendamment. Ces images sont toutes dérivées d'une même image de base, la seule différence étant l'antivirus installé.

Chaque antivirus est paramétré sur la configuration la plus stricte disponible afin de déclencher le plus d'alertes possible. Si un antivirus propose un moyen spécifique anti-ransomware, nous créons une nouvelle image du système avec cette fonctionnalité activée et configurée pour surveiller

Table 3.3 – Payloads détectées par les antivirus.
✓ : L'antivirus lance une alerte
✓* : L'antivirus lance une alerte mais laisse Nuky chiffrer une partie des malware

Antivirus	AES	Huffman	Hide	Eicar
Windows Defender				✓
Windows Defender+	✓	✓	✓	✓
Immunet (Clamantivirus)				✓
Kaspersky				✓
Avast				✓
Avast+	✓	✓	✓	✓
antivirusG				✓
Avira				✓
K7 Computing	✓*	✓*	✓*	✓

`Mes Documents`. Ceux-ci sont désignés par un signe «+» dans la Table 3.3. Nous utilisons ces configurations pour donner toutes les chances aux antivirus de détecter Nuky.

Capacités de détection des Antivirus

Cette première expérimentation vise à définir quels antivirus sont capables de détecter Nuky en fonction de la payload activée et de s'assurer qu'ils repèrent tous le fichier EICAR. Ce fichier est utilisé pour tester le bon fonctionnement des antivirus, car sa signature doit déclencher des alertes pour tous les antivirus de notre panel.

Dans ce scénario d'expérimentation, Nuky n'utilise aucune méthode d'évasion et exécute chacune de ses quatre payloads. Comme ce programme est un nouveau malware, il n'est par conséquent pas présent dans les bases de signatures des antivirus. Néanmoins, ses trois premières payloads (Compression, Chiffrement et Cache) sont selon nous caractéristiques des ransomware. Un antivirus devrait donc être capable de repérer ces actions dynamiquement.

Comme décrit dans la Table 3.3, seuls les antivirus équipés d'une fonction dédiée à la détection des ransomware et que nous marquons par des «+» ont été capables de détecter les payloads de Nuky et l'empêchent de chiffrer les fichiers. Il est à noter que K7 Computing détecte Nuky mais ne le stoppe pas immédiatement et laisse une partie des fichiers se faire chiffrer. Nous suspectons K7 Computing et Windows Defender de générer de faux positifs, car certaines manipulations légitimes déclenchent des alertes.

Par exemple, K7 Computing génère une alerte lorsque nous effectuons une simple rotation d'image à l'aide du logiciel «Photos». Windows Defender quant à lui génère une alerte lorsque nous sauvegardons un fichier à l'aide du programme «LibreOffice Draw».

Toutefois, ces antivirus détectent Nuky et sont par conséquent ceux que notre malware doit pouvoir repérer et éviter pour prolonger sa durée de vie. Finalement, tous les antivirus sont capables de détecter la payload qui dépose un fichier EICAR sur le bureau.

Le fait que la majorité des antivirus ne détecte pas la compression, le chiffrement et la payload *cache* peut signifier que ceux-ci se basent encore uniquement sur la détection par signature.

Antivirus	Debugger	Antivirus	VM
Windows Defender	✓	×	✓
Immunet (ClamAV)	✓	×	✓
Kaspersky	✓	×	✓
Avast	✓	×	✓
AVG	✓	×	✓
Avira	✓	×	✓
K7 Computing	✓	×	✓

Table 3.4 – Déclenchement des alertes en fonction des blocs d'évasion activés.
✓ : L'antivirus lance une alerte
× : L'antivirus ne lance pas d'alerte

Antivirus	Nom de dossiers	Noms de processus
Windows Defender	3	1
Immunet (ClamAV)	1	1
Kaspersky	4	4
Avast	2	2
AVG	2	2
Avira	1	9
K7 Computing	2	9

Table 3.5 – Nombre d'artéfacts repérés par le bloc d'évasion antivirus.

Capacités de Nuky à s'évader

Dans cette seconde expérimentation, nous cherchons à vérifier le bon fonctionnement des blocs d'évasion implémentés dans Nuky. Pour cela, nous activons seulement la payload EICAR que nous savons repérée par tous les antivirus.

Nous exécutons Nuky trois fois (une fois par bloc d'évasion) par antivirus en activant à chaque fois un bloc d'évasion différent. Si le bloc d'évasion étudié fonctionne, alors le fichier EICAR n'est pas déposé et donc aucun antivirus ne devrait lancer d'alerte.

Dans le cas contraire, le fichier est déposé, car le bloc d'évasion ne détecte aucun outil d'analyse. L'antivirus déclenche alors une alerte ce qui nous permet de savoir que le bloc d'évasion testé a échoué à le repérer. La Table 3.4 liste pour chaque bloc si les antivirus du panel lancent une alerte.

Comme attendu, le seul bloc pour lequel nous observons une absence d'alerte est le bloc *antivirus* qui cible les artéfacts d'installation et d'exécution des antivirus. La Table 3.5 montre le nombre d'artéfacts trouvés par les deux techniques d'évasion qui ciblent les antivirus et qui composent le bloc d'évasion.

Pour les autres blocs, soit l'évasion est effectuée dans un environnement fermé pour lequel nous ne pouvons pas observer les résultats, soit les antivirus ne génèrent pas les artéfacts recherchés par Nuky. Ces outils sont plus utilisés par des analystes une fois les échantillons récupérés sur les serveurs des antivirus. Toutefois, nous souhaitions tester ces blocs, car nous pensions que des antivirus peuvent effectuer de l'analyse dynamique à l'aide de machines virtuelles ou debuggers directement sur la machine à protéger.

Le bloc d'évasion «*Antivirus*» est donc capable de détecter et identifier facilement un antivirus. Nuky peut alors décider soit de ne pas exécuter sa payload soit de lancer un exploit qui cible cet antivirus afin d'augmenter sa durée de vie. Ce malware évasif test nous permettra de valider

le bon fonctionnement de notre contre-mesure dans la Section 4.4.1. Nous souhaitons ensuite pouvoir effectuer une expérimentation sur des malware évasifs récoltés dans la nature.

3.4 Dataset de malware évasifs récoltés dans la nature

Dans la précédente section, nous avons conclu à l'aide de Nuky que les méthodes d'évasion qui ciblent les outils d'analyse sont simples à mettre en place et efficaces contre les antivirus modernes.

Nous détaillons maintenant la façon dont nous avons collecté des échantillons de malware évasifs dans la nature afin de créer un dataset qui permet de tester notre contre-mesure. Premièrement, nous décrivons notre façon de construire un filtre qui permet de garder seulement des échantillons évasifs. Deuxièmement, nous détaillons la façon dont nous avons utilisé ce filtre pour collecter les échantillons.

3.4.1 Filtrage via des règles Yara

Pour créer notre filtre, nous utilisons un programme appelé Yara [9], qui nous permet de trouver des échantillons qui correspondent à un ensemble de règles. Yara peut trouver des échantillons dont le binaire contient des chaînes de caractères spécifiques ou qui charge une librairie ou fonction Windows spécifique. Une règle conçue pour trouver des échantillons évasifs existe déjà [10], mais celle-ci retourne beaucoup d'échantillons dont une grande partie contient des malware qui ne semblent pas évasifs après rétro-ingénierie ou pas exécutable d'après nos essais. Nous avons donc modifié ces règles en rajoutant des conditions afin de récupérer le plus grand nombre possible de malware évasifs avec peu de faux positifs.

9: http://virustotal.github.io/yara/

10: https://github.com/Yara-Rules/rules/blob/master/antidebug_antivm/antidebug_antivm.yar

Nous appliquons notre filtre de manière statique, car cela nous permet d'analyser des échantillons plus rapidement qu'avec une analyse dynamique. Il serait possible d'exécuter les malware afin d'enregistrer une trace d'appels système et d'appliquer le filtre sur celles-ci. Toutefois, nous souhaitions récolter des échantillons rapidement et utilisons donc une analyse statique dans un premier temps. Nous traitons de la détection de malware évasifs à l'aide d'une méthode dynamique dans le Chapitre 5.

Notre filtre contient cinq règles chacune composée de plusieurs sous-règles et conçues pour trouver respectivement des malware qui essaient de *détecter un outil d'analyse, détecter un antivirus ou une sandbox qui s'exécute, manipuler et itérer le contenu de dossiers, trouver une adresse MAC de VM et trouver une fenêtre d'un debugger*. Un malware est considéré comme évasif s'il est détecté par au moins une des règles du filtre.

Malware qui détectent des outils d'analyse

La règle destinée à filtrer les malware qui détectent des debuggers et des outils d'analyse est composée de deux sous-règles dont les codes sources sont détaillés dans les Listings 3.2 et 3.3. Pour être capturé par cette règle, un malware doit remplir les conditions décrites dans ces deux

sous-règles. La première vise à détecter les malware qui contiennent au moins dix chaînes de caractères qui correspondent à des noms d'outils d'analyse parmi une liste de vingt-cinq dans leur binaire. Yara contient un module spécifique aux *PE Files* qui est un format d'exécutable Windows [11]. Ce module permet de vérifier les imports déclarés dans l'entête du *PE File*. La règle présentée dans le Listing 3.2 est conçue pour détecter les échantillons qui importent les fonctions CreateToolhelp32Snapshot, Process32First et Process32Next présentes dans la librairie kernel32.dll.

11: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format

```
private rule Check_Running_Debugger
{
    meta:
        Author = "Cedric HERZOG"
    Description = "Checks for debuggers and analysis tools process names"
    strings:
        $name1 = "ollydbg" wide nocase ascii fullword
        $name2 = "processhacker" wide nocase ascii fullword
        $name3 = "tcpview" wide nocase ascii fullword
        $name4 = "autoruns" wide nocase ascii fullword
        $name5 = "autorunsc" wide nocase ascii fullword
        $name6 = "filemon" wide nocase ascii fullword
        $name7 = "procmon" wide nocase ascii fullword
        $name8 = "idaq" wide nocase ascii fullword
        $name9 = "immunitydebugger" wide nocase ascii fullword
        $name10 = "wireshark" wide nocase ascii fullword
        $name11 = "dumpcap" wide nocase ascii fullword
        $name12 = "hookexplorer" wide nocase ascii fullword
        $name13 = "importrec" wide nocase ascii fullword
        $name14 = "petools" wide nocase ascii fullword
        $name15 = "lordpe" wide nocase ascii fullword
        $name16 = "sysinspector" wide nocase ascii fullword
        $name17 = "proc_analyzer" wide nocase ascii fullword
        $name18 = "sysanalyzer" wide nocase ascii fullword
        $name19 = "sniff_hit" wide nocase ascii fullword
        $name20 = "windbg" wide nocase ascii fullword
        $name21 = "joeboxcontrol" wide nocase ascii fullword
        $name22 = "joeboxserver" wide nocase ascii fullword
        $name23 = "netmon" wide nocase ascii fullword
        $name24 = "x64dbg" wide nocase ascii fullword
        $name25 = "x32dbg" wide nocase ascii fullword
    condition:
        10 of ($name*) and
        pe.imports("kernel32.dll", "CreateToolhelp32Snapshot") and
        pe.imports("kernel32.dll", "Process32First") and
        pe.imports("kernel32.dll", "Process32Next")
}
```

Listing 3.2: Code de la sous-règle qui permet de détecter des malware qui cherchent à repérer un debugger en cours d'exécution.

Lors de nos tests, beaucoup d'échantillons remplissaient cette condition. Nous avons donc ajouté une deuxième règle conçue pour retenir les échantillons qui implémentent une attaque connue contre OllyDbg et qui utilise la fonction `OutputDebugString()` [12]. Cette fonction est très ciblée, mais est décrite dans plusieurs forums et projets qui traitent de l'évasion. Il est donc possible qu'elle soit utilisée par beaucoup de malware évasifs.

12: https://www.cvedetails.com/cve/CVE-2004-0733/

```
private rule Check_Debug_String
{
    meta:
        Author = "Cedric HERZOG"
    Description = "Checks for anti debugger attack"
    strings:
        $a = "%s%s%s%s" wide nocase ascii fullword
    condition:
        pe.imports("kernel32.dll", "SetLastError") and
```

Listing 3.3: Code de la sous-règle qui permet de détecter des malware qui utilisent une attaque contre un debugger connu.

```
        pe.imports("kernel32.dll", "OutputDebugStringA") and
        pe.imports("kernel32.dll", "GetLastError") and $a
}
```

Malware qui détectent des antivirus

Cette seconde règle vise à repérer les malware qui détectent les antivirus et les machines virtuelles. Elle est composée de quatre sous-règles décrites dans les Listings 3.4, 3.5, 3.6 et 3.7.

Une technique d'évasion consiste à vérifier la présence de noms de processus qui s'exécutent à l'aide de la fonction `CreateToolhelp32Snapshot()`. Cette règle vise à repérer les malware qui utilisent cette technique en retenant ceux qui contiennent au moins quatre (chiffre choisi de manière arbitraire) noms d'antivirus sur onze et importent les fonctions `CreateToolhelp32Snapshot()`,`Process32First()` et `Process32Next()`.

Listing 3.4: Code de la sous-règle qui permet de détecter des malware qui cherchent à repérer un antivirus en cours d'exécution.

```
private rule Check_Running_AV
{
    meta:
        Author = "Cedric HERZOG"
    Description = "Checks for antivirus process names"
    strings:
        $name1 = "aswidagent" wide nocase ascii fullword
        $name2 = "Avast" wide nocase ascii fullword
        $name3 = "avgnt" wide nocase ascii fullword
        $name4 = "avshadow" wide nocase ascii fullword
        $name5 = "Avira" wide nocase ascii fullword
        $name6 = "klnagent" wide nocase ascii fullword
        $name7 = "avp" wide nocase ascii fullword
        $name8 = "ksde" wide nocase ascii fullword
        $name9 = "mbam" wide nocase ascii fullword
        $name10 = "MBAMService" wide nocase ascii fullword
        $name11 = "mbamtray" wide nocase ascii fullword
    condition:
        4 of ($name*) and
        pe.imports("kernel32.dll", "CreateToolhelp32Snapshot") and
        pe.imports("kernel32.dll", "Process32First") and
        pe.imports("kernel32.dll", "Process32Next")
}
```

Sur le même principe, nous créons une règle vérifiant la présence de "vmware" ou "vbox" et des fonctions permettant de faire un snapshot de processus dans les malware.

Listing 3.5: Code de la sous-règle qui permet de détecter des malware qui cherchent à repérer une sandbox en cours d'exécution.

```
private rule Check_Running_VM
{
    meta:
        Author = "Cedric HERZOG"
    Description = "Checks for common sandbox process names"
    strings:
        $name1 = "vmware" wide nocase ascii fullword
        $name2 = "vbox" wide nocase ascii fullword
    condition:
        any of ($name*) and
        pe.imports("kernel32.dll", "CreateToolhelp32Snapshot") and
        pe.imports("kernel32.dll", "Process32First") and
        pe.imports("kernel32.dll", "Process32Next")
}
```

Certaines sandbox utilisent toujours les mêmes noms d'utilisateurs que les auteurs de malware connaissent et peuvent lire. La règle qui permet de récupérer ce genre de malware recherche la présence dans

l'échantillon d'au moins trois noms sur cinq ainsi que la présence des chaînes "getenv_s" et "USERNAME".

```
private rule Check_Username
{
    meta:
        Author = "Cedric HERZOG"
    Description = "Checks for common sandbox usernames"
    strings:
        $name1 = "sandbox" wide nocase ascii fullword
        $name2 = "maltest" wide nocase ascii fullword
        $name3 = "malware" wide nocase ascii fullword
        $name4 = "virus" wide nocase ascii fullword
        $name5 = "sample" wide nocase ascii fullword
        $a = "getenv_s" wide nocase ascii fullword
        $b = "USERNAME" wide ascii fullword
    condition:
        3 of ($name*) and $a and $b
}
```

Listing 3.6: Code de la sous-règle qui permet de détecter des malware qui cherchent à repérer un nom d'utilisateur fréquemment utilisé par des sandboxes.

Nous réutilisons également une règle ayant pour but de détecter les malware qui cherchent des DLL liées à des sandboxes.

```
private rule Check_Dlls
{
  meta:
    Author = "Nick Hoffman"
    Description = "Checks for common sandbox dlls"
    Sample = "de1af0e97e94859d372be7fcf3a5daa5"
  strings:
    $dll1 = "sbiedll.dll" wide nocase ascii fullword
    $dll2 = "dbghelp.dll" wide nocase ascii fullword
    $dll3 = "api_log.dll" wide nocase ascii fullword
    $dll4 = "dir_watch.dll" wide nocase ascii fullword
    $dll5 = "pstorec.dll" wide nocase ascii fullword
    $dll6 = "vmcheck.dll" wide nocase ascii fullword
    $dll7 = "wpespy.dll" wide nocase ascii fullword
  condition:
    3 of them
}
```

Listing 3.7: Code de la sous-règle qui permet de détecter des malware qui cherchent à repérer la présence de DLL propres à des sandboxes.

Chacune de ces sous-règles détecte beaucoup de malware, c'est pourquoi nous les regroupons dans une règle unique.

Malware qui détectent des manipulation de dossiers

Cette règle est composée de deux sous-règles destinées à détecter des manipulations liées à des dossiers. Lors de leur installation, les antivirus créent des dossiers portant leur nom. La sous-règle visible dans le Listing 3.8 est conçue pour trouver des malware qui peuvent tenter de détecter ces dossiers en vérifiant la présence d'au moins trois noms d'antivirus dans une liste de cinq.

```
private rule Check_Installation_Folder
{
    meta:
        Author = "Cedric HERZOG"
    Description = "Checks for antivirus directories"
    strings:
        $name1 = "avast" wide nocase ascii fullword
        $name2 = "avira" wide nocase ascii fullword
        $name3 = "kaspersky" wide nocase ascii fullword
        $name4 = "malwarebytes" wide nocase ascii fullword
```

Listing 3.8: Code de la sous-règle qui permet de détecter des malware qui listent des dossiers qui appartiennent à des antivirus.

```
        $name5 = "avg" wide nocase ascii fullword
        $a = "directory_iterator" wide nocase ascii fullword
    condition:
        3 of ($name*) and $a
}
```

Une seconde sous-règle présentée dans le Listing 3.9 se base sur le même principe mais cherche à détecter les malware qui sont à la recherche de dossiers souvent présents dans des sandboxes.

Listing 3.9: Code de la sous-règle qui permet de détecter des malware qui listent des dossiers qui appartiennent à des sandboxes.

```
private rule Check_Folder
{
    meta:
        Author = "Cedric HERZOG"
    Description = "Checks for common sandbox directories"
    strings:
        $name1 = "sandbox" wide nocase ascii fullword
        $name2 = "maltest" wide nocase ascii fullword
        $name3 = "malware" wide nocase ascii fullword
        $name4 = "virus" wide nocase ascii fullword
        $name5 = "sample" wide nocase ascii fullword
        $a = "directory_iterator" wide nocase ascii fullword
    condition:
        3 of ($name*) and $a
}
```

Malware qui détectent des adresses MAC spécifiques

La règle décrite dans le Listing 3.10 vise à détecter les malware qui ont pour but de lister les interfaces réseau à la recherche de celles utilisées par des sandboxes. Cette règle vérifie la présence d'au moins six de ces adresses MAC parmi une liste de quinze.

Listing 3.10: Code de la règle qui permet de détecter des malware qui recherchent des adresses MAC fréquemment utilisées par des sandboxes.

```
rule Check_MAC_Address
{
    meta:
        Author = "Cedric HERZOG"
    Description = "Checks for common sandbox MAC addresses"
    strings:
        $name1 = "00-05-69" wide nocase ascii fullword
        $name2 = "00-0C-29" wide nocase ascii fullword
        $name3 = "00-1C-14" wide nocase ascii fullword
        $name4 = "00-50-56" wide nocase ascii fullword
        $name5 = "08-00-27" wide nocase ascii fullword
        $name6 = "00-1C-42" wide nocase ascii fullword
        $name7 = "00-0F-4B" wide nocase ascii fullword
        $name8 = "00-16-3E" wide nocase ascii fullword
        $name9 = "00-03-FF" wide nocase ascii fullword
        $name10 = "00-15-5D" wide nocase ascii fullword
        $name11 = "00-0D-3A" wide nocase ascii fullword
        $name12 = "00-12-5A" wide nocase ascii fullword
        $name13 = "00-17-FA" wide nocase ascii fullword
        $name14 = "00-50-F2" wide nocase ascii fullword
        $name15 = "00-1D-D8" wide nocase ascii fullword
    condition:
        6 of ($name*) and
        pe.imports("Iphlpapi.dll", "GetAdaptersInfo")
}
```

Malware qui détectent des fenêtres d'outils d'analyse

La dernière règle est présentée dans le Listing 3.11 et cherche à repérer les malware qui utilisent la fonction FindWindow pour chercher des fenêtres portant un nom de debuggers.

```
rule Check_Window
{
    meta:
        Author = "Cedric HERZOG"
    Description = "Checks for FindWindow evasion"
    strings:
        $name1 = "OLLYDBG" wide nocase ascii fullword
        $name2 = "WinDbgFrameClass" wide nocase ascii fullword
        $name3 = "Immunity Debugger" wide nocase ascii fullword
        $name4 = "Zeta Debugger" wide nocase ascii fullword
        $name5 = "Rock Debugger" wide nocase ascii fullword
        $name6 = "ObsidianGUI" wide nocase ascii fullword
        $name7 = "ID" wide nocase ascii fullword
        $name8 = "x32dbg" wide nocase ascii fullword
        $name9 = "x64dbg" wide nocase ascii fullword
    condition:
        4 of ($name*) and
        pe.imports("User32.dll", "FindWindow")
}
```

Listing 3.11: Code de la règle qui permet de détecter des malware qui recherchent des noms fenêtres qui correspondent à des debuggers connus.

3.4.2 Collecte d'échantillons

La difficulté de la collecte d'échantillons réside dans le fait que par définition, ces malware évitent de se faire détecter. Nous avons manuellement parcouru le web dans des datasets publics tels qu'Any Run [13] ainsi que des répertoires publics tels que theZoo [14], par exemple. La collecte couvre une période de trois mois entre le 18 novembre 2019 et le 21 janvier 2020.

13: https://any.run

14: https://github.com/ytisf/theZoo

Sur cette période, nous avons trouvé soixante-deux échantillons qui correspondent à au moins une de nos règles Yara sur les quelques milliers d'échantillons crawlés.

Afin d'éviter les faux positifs et être certains de ne retenir que les exécutables qui sont vraisemblablement les plus malveillants, nous gardons seulement ceux qui ne sont pas signés et marqués comme malveillants par VirusTotal [15]. Nous excluons les échantillons signés, car beaucoup de logiciels légitimes comme CCleaner ou MalwareBytes étaient présents dans nos premiers essais et une nouvelle étape de filtrage était nécessaire pour différencier les malwares signés des logiciels légitimes signés. Comme nous souhaitons obtenir des logiciels véritablement malveillants et évasifs quitte à en obtenir une quantité moindre, nous faisons le choix d'exclure les malware signés.

15: https://www.virustotal.com

La Table 3.6, montre le nombre d'échantillons récupérés par chaque règle Yara que nous avons utilisée. Les Message Digest 5 (MD5) des dix-huit malware retenus sont listés dans la Table 3.7.

Table 3.7 – Liste des échantillons sélectionnés.

Groupe	MD5
A	de3d1414d45e762ca766d88c1384e5f6
	2d57683027a49d020db648c633aa430a
	d3e89fa1273f61ef4dce4c43148d5488
	bd5934f9e13ac128b90e5f81916eebd8
	512d425d279c1d32b88fe49013812167
	2ccc21611d5afc0ab67ccea2cf3bb38b
	6b43ec6a58836fd41d40e0413eae9b4d
B	ee12bbee4c76237f8303c209cc92749d
	5253a537b2558719a4d7b9a1fd7a58cf
	8fdab468bc10dc98e5dc91fde12080e9
	e5b2189b8450f5c8fe8af72f9b8b0ad9
	ee88a6abb2548063f1b551a06105f425
	4d5ac38437f8eb4c017bc648bb547fac
C	f4d68add66607647bf9cf68cd17ea06a
D	862c2a3f48f981470dcb77f8295a4fcc
E	e51fc4cdd3a950444f9491e15edc5e22
F	812d2536c250cb1e8a972bdac3dbb123
G	5c8c9d0c144a35d2e56281d00bb738a4

Table 3.6 – Nombre de malware trouvés par chaque règle.

Règles	Tous les échantillons	Échantillons retenus
Debugger	20	16
Antivirus et sandbox	0	0
Manipulation de dossiers	5	0
Adresses MAC	37	2
Recherche de fenêtre graphique	0	0

Le fait que certaines règles ne retournent aucun résultat peut être dû au fait que nous avons fait une recherche à la main et donc n'avions pas suffisamment de malware à mettre en entrée du filtre Yara.

Ce filtre Yara est très restrictif et ne retient que quelques échantillons pendant la durée de notre récolte. Toutefois, notre but est d'obtenir des malware véritablement évasifs quitte à en posséder une petite quantité. L'amélioration de cette récolte de malware évasifs est l'objet de notre deuxième contribution que nous présentons dans le Chapitre 5.

3.4.3 Description des échantillons récupérés

Nous regroupons les échantillons en fonction de la ressemblance de leur code assembleur via l'outil diff de Ghidra. Tous les échantillons d'un même groupe contiennent le même code dans leur section *text*. Celle-ci semble contenir le code qui exécute l'évasion.

Les groupes A, B et C partagent une partie significative de leur section *text* avec quelques petits ajouts de codes ou modifications de la signature de certaines méthodes. Les autres groupes sont très différents et contiennent un code qui leur est propre.

Nous calculons l'entropie de chaque échantillon et trouvons une valeur minimale de 6.58 et une valeur maximale de 7.11. En se basant sur l'expérimentation faite par Lyda *et al.* [54], ces valeurs suggèrent que les échantillons sont empaquetés ou chiffrés. En effet, un exécutable classique possède une entropie autour de 5.099, un malware empaqueté se situe autour de 6.801, et s'il est chiffré autour de 7.175. Nous trouvons dans chaque échantillon une section qui contient les différentes ressources. Dans certains échantillons, cette section semble contenir un exécutable qui pourrait être dépaqueté ou déchiffré après les tests d'évasion.

[54]: Lyda et al. (2007), « Using Entropy Analysis to Find Encrypted and Packed Malware »

Toutes ces similarités entre les échantillons pourraient être dues au partage de code dans des projets publics tels que Al-Khaser [16] ou des projets de plus petite envergure [17]. Certains packers proposent également d'ajouter une partie évasion avant le dépaquetage tel que Trojka Crypter [18].

16: https://github.com/LordNotoeworthy/al-khaser

17: https://github.com/maikel233/X-HOOK-For-CSGO

18: MD5 : c74b6ad8ca7b1dd810e9704c34d3e217

3.5 Conclusion

Lors des tests de résistance des antivirus contre les techniques d'évasion de Nuky, nous utilisons un petit nombre de techniques basiques d'évasion bien connues des attaquants et facilement reproductibles. Ces techniques ne sont pas représentatives de toutes les manières possibles de s'évader. Pour l'instant, Nuky représente donc un malware que n'importe quel attaquant est capable d'écrire en suivant des tutoriels et non un malware avancé créé par de grands groupes ou états. Nous n'avons pas réussi à tester les techniques d'évasion qui ciblent les debuggers et VMs avec notre méthode, mais pensons possible leur évaluation à l'aide de tests plus adaptés.

Nuky est capable de détecter et identifier tous les antivirus de notre panel pour ensuite adapter son comportement. De plus, nous savons que ces techniques sont simples à implémenter et que plusieurs projets permettent de les intégrer dans un malware à moindre frais. Tout ceci nous laisse penser que créer une contre-mesure destinée à stopper ce genre de malware a un réel intérêt, et ce sujet fait l'objet du Chapitre 4.

L'ensemble de malware évasifs que nous récoltons contient dix-huit malware que nous utilisons dans la Section 4.4.2. Ce dataset est de taille modeste, mais tous les échantillons sont désassemblés et nous trouvons des éléments qui nous permettent de les étiqueter comme étant évasifs. Le Chapitre 5 est consacré à la création d'un dataset plus important et

utilise cet ensemble pour définir un seuil de détection au-delà duquel un malware est considéré comme évasif.

Contremesure Anti Malware Évasifs

4

Ce chapitre décrit la façon dont nous construisons notre contre-mesure anti malware évasifs ainsi que ses effets sur des malware évasifs et des logiciels légitimes. Nous commençons par décrire son fonctionnement général dans la Section 4.1 avant de présenter plus en détail son implémentation dans la Section 4.2. La plateforme utilisée pour exécuter l'expérimentation est détaillée dans la Section 4.3. Le bon fonctionnement de la contre-mesure est évalué à l'aide d'un malware évasif appelé Nuky que nous créons et qui est présenté dans la Section 3.3. Ce malware évasif permet également de vérifier l'effet des techniques d'évasion sur un panel d'antivirus. Nous créons un dataset décrit dans la Section 3.4 que nous utilisons dans notre expérimentation présentée dans la Section 4.4.

Ces travaux ont donné lieu à une première contribution publiée à la conférence SECRYPT 2020 puis à une seconde présentée à la conférence C&ESAR 2020.

4.1 But de la contre-mesure

Comme peu de malware évasifs sont disponibles, nous ne connaissons pas leurs cibles favorites et les comportements suivis les plus fréquemment lors de l'évasion. Cependant, nous supposons que ce genre de malware peut cibler les antivirus, car leur usage est répandu.

Installer plusieurs antivirus pourrait donc augmenter les chances de déclencher le comportement évasif de ces malware. Comme détaillé dans la Section 3.1, le fonctionnement d'un antivirus est assez complexe et représente souvent un coût important en termes de consommation de CPU et de mémoire. Installer plusieurs antivirus n'est pas envisageable à cause de ce coût en ressources trop important, mais également à cause de conflits qui peuvent apparaître entre eux [1].

1: https://www.kaspersky.com/resource-center/preemptive-safety/running-more-than-one-antivirus-program

Notre contre-mesure consiste à simuler la présence de plusieurs antivirus afin de faire s'évader un maximum de malware évasifs tout en limitant l'impact sur le système.

4.1.1 Principe de la contre-mesure anti malware évasifs

Nous proposons de mettre en place une contre-mesure qui induit en erreur les malware évasifs et les pousse à s'évader. Pour cela, nous instrumentons l'API Windows afin de reproduire les artéfacts générés par des antivirus de sorte à inciter le malware à stopper son exécution.

Pour effectuer ces instrumentations, nous utilisons Microsoft Detours [55] afin de produire une DLL que nous chargeons en utilisant la clef de registre `AppInit` [2]. Ce registre contient une liste de librairies destinées à être chargées dans tous les processus qui s'exécutent en mode utilisateur et plus précisément ceux qui utilisent la librairie `User32.dll`. Le chargement des librairies listées dans ce registre s'effectue automatiquement par le système d'exploitation au démarrage des programmes. Deux versions de

[55]: Brubacher (1999), « Detours : Binary Interception of Win32 Functions »

2: https://attack.mitre.org/techniques/T1103/

ce registre existent et permettent de fournir respectivement des librairies 32-bit et 64-bit.

Les instrumentations sont codées en C++ puis compilées pour générer les deux librairies à injecter dans tous les processus de la machine à protéger. Pour nous assurer d'affecter la plupart des malware évasifs, nous créons une version 32-bits et une 64-bit que nous ajoutons dans les registres `AppInit` correspondants.

Cette méthode d'injection est visible par le malware qui pourrait repérer notre contre-mesure et la désactiver, mais a pour avantage d'être facilement modifiable ce qui accélère nos expérimentations, car nous pouvons effectuer plusieurs essais rapidement. Pour contrer cela, nous pourrions, à terme, effectuer nos instrumentations depuis un driver qui serait ainsi plus privilégié que les programmes s'exécutant en espace utilisateur.

4.2 Simulation des artéfacts

Nous instrumentons sept fonctions *cibles* choisies, car elles sont souvent utilisées à des fins d'évasion [3]. Des pointeurs vers les fonctions à instrumenter sont créés comme présentés dans le Listing 4.1. Certaines fonctions cibles sont présentes en deux versions en fonction du format des chaînes des caractères supporté. Les fonctions qui se terminent par un *A* supportent les caractères au format `ANSI` et celles par *W* les caractères encodés au format `WIDE`.

3: https://malapi.io

Listing 4.1: Création de pointeur des fonctions cibles non instrumentées.

```
1  static BOOL(WINAPI * TrueDebuggerPresent)(void) = IsDebuggerPresent;
2  static BOOL(WINAPI * TrueCursorPos)(LPPOINT lpPoint) = GetCursorPos;
3  static LSTATUS(WINAPI * TrueOpenKeyA)(HKEY hKey, LPCSTR lpSubKey, DWORD
       ulOptions, REGSAM samDesired, PHKEY phkResult) = RegOpenKeyExA;
4  static LSTATUS(WINAPI * TrueOpenKeyW)(HKEY hKey, LPCWSTR lpSubKey, DWORD
       ulOptions, REGSAM samDesired, PHKEY phkResult) = RegOpenKeyExW;
5  static HANDLE(WINAPI * TrueCreateFileA)(LPCSTR lpFileName, DWORD
       dwDesiredAccess, DWORD dwShareMode, LPSECURITY_ATTRIBUTES
       lpSecurityAttributes, DWORD dwCreationDisposition, DWORD
       dwFlagsAndAttributes, HANDLE hTemplateFile) = CreateFileA;
6  static HANDLE(WINAPI * TrueCreateFileW)(LPCWSTR lpFileName, DWORD
       dwDesiredAccess, DWORD dwShareMode, LPSECURITY_ATTRIBUTES
       lpSecurityAttributes, DWORD dwCreationDisposition, DWORD
       dwFlagsAndAttributes, HANDLE hTemplateFile) = CreateFileW;
7  static HMODULE(WINAPI * TrueModuleHandleA)(LPCSTR lpModuleName) =
       GetModuleHandleA;
8  static HMODULE(WINAPI * TrueModuleHandleW)(LPCWSTR lpModuleName) =
       GetModuleHandleW;
9  static LSTATUS(WINAPI * TrueQueryValueA)(HKEY hKey, LPCSTR lpValueName,
       LPDWORD lpReserved, LPDWORD lpType, LPBYTE lpData, LPDWORD lpcbData)
       = RegQueryValueExA;
10 static LSTATUS(WINAPI * TrueQueryValueW)(HKEY hKey, LPCWSTR lpValueName,
       LPDWORD lpReserved, LPDWORD lpType, LPBYTE lpData, LPDWORD lpcbData)
       = RegQueryValueExW;
11 static DWORD(WINAPI * TrueFileAttributesA)(LPCSTR lpFileName) =
       GetFileAttributesA;
12 static DWORD(WINAPI * TrueFileAttributesW)(LPCWSTR lpFileName) =
       GetFileAttributesW;
13 static HANDLE(WINAPI * TrueCreateToolhelp32Snapshot)(DWORD dwFlags, DWORD
       th32ProcessID) = CreateToolhelp32Snapshot;
```

Les cinq premiers octets des fonctions *cible* sont ensuite remplacés par Microsoft Detours lors de l'exécution par un saut inconditionnel vers des fonctions *detours* que nous définissons. Ces cinq octets sont stockés dans

une fonction *trampoline* qui se termine par un saut inconditionnel vers le reste de la fonction *cible*.

Comme présenté dans le Listing 4.2, le remplacement de la fonction *cible* par la fonction *detour* associée s'effectue dans le point d'entrée de la librairie `DllMain`. Les auteurs de Microsoft Detours recommandent d'effectuer ces appels dans cette partie afin d'assurer qu'un seul thread effectue des opérations de l'espace mémoire de l'application nécessaires à la mise en place des interceptions.

Listing 4.2: Point d'entrée de la DLL dans laquelle sont remplacées les fonctions cibles par les fonctions detours.

```
BOOL WINAPI DllMain(HINSTANCE hinst, DWORD dwReason, LPVOID reserved)
{
    //LONG error;
    (void)hinst;
    (void)reserved;

    if (DetourIsHelperProcess()) {
        return TRUE;
    }

    if (dwReason == DLL_PROCESS_ATTACH) {
        DetourRestoreAfterWith();
        DetourTransactionBegin();
        DetourUpdateThread(GetCurrentThread());
        DetourAttach(&(PVOID&)TrueDebuggerPresent, FakeDebuggerPresent);
        DetourAttach(&(PVOID&)TrueCursorPos, DoublePoint);
        DetourAttach(&(PVOID&)TrueOpenKeyA, FalseOpenKeyA);
        DetourAttach(&(PVOID&)TrueOpenKeyW, FalseOpenKeyW);
        DetourAttach(&(PVOID&)TrueCreateFileA, FalseCreateFileA);
        DetourAttach(&(PVOID&)TrueCreateFileW, FalseCreateFileW);
        DetourAttach(&(PVOID&)TrueModuleHandleA, FalseModuleHandleA);
        DetourAttach(&(PVOID&)TrueModuleHandleW, FalseModuleHandleW);
        DetourAttach(&(PVOID&)TrueQueryValueA, FalseQueryValueA);
        DetourAttach(&(PVOID&)TrueQueryValueW, FalseQueryValueW);
        DetourAttach(&(PVOID&)TrueFileAttributesA, FalseFileAttributesA);
        DetourAttach(&(PVOID&)TrueFileAttributesW, FalseFileAttributesW);
        DetourAttach(&(PVOID&)TrueCreateToolhelp32Snapshot,
     FalseCreateToolhelp32Snapshot);
        DetourTransactionCommit();
    }
    else if (dwReason == DLL_PROCESS_DETACH) {
        DetourTransactionBegin();
        DetourUpdateThread(GetCurrentThread());
        DetourDetach(&(PVOID&)TrueDebuggerPresent, FakeDebuggerPresent);
        DetourAttach(&(PVOID&)TrueCursorPos, DoublePoint);
        DetourAttach(&(PVOID&)TrueOpenKeyA, FalseOpenKeyA);
        DetourAttach(&(PVOID&)TrueOpenKeyW, FalseOpenKeyW);
        DetourAttach(&(PVOID&)TrueCreateFileA, FalseCreateFileA);
        DetourAttach(&(PVOID&)TrueCreateFileW, FalseCreateFileW);
        DetourAttach(&(PVOID&)TrueModuleHandleA, FalseModuleHandleA);
        DetourAttach(&(PVOID&)TrueModuleHandleW, FalseModuleHandleW);
        DetourAttach(&(PVOID&)TrueQueryValueA, FalseQueryValueA);
        DetourAttach(&(PVOID&)TrueQueryValueW, FalseQueryValueW);
        DetourAttach(&(PVOID&)TrueFileAttributesA, FalseFileAttributesA);
        DetourAttach(&(PVOID&)TrueFileAttributesW, FalseFileAttributesW);
        DetourAttach(&(PVOID&)TrueCreateToolhelp32Snapshot,
     FalseCreateToolhelp32Snapshot);
        DetourTransactionCommit();
    }
    return TRUE;
}
```

Trois types différents de fonctions *detours* sont créés que nous présentons maintenant.

4.2.1 Retourner une valeur unique

Le premier type de modification change la fonction pour retourner une valeur constante. Dans ce cas, la fonction *cible* est totalement remplacée par notre fonction *detour*. Le code de la fonction detour qui remplace `isDebuggerPresent()` est présenté dans le Listing 4.3.

Listing 4.3: Code de la fonction *detour* d'isDebuggerPresent.

```
// Nouveau comportement d'isDebuggerPresent
BOOL WINAPI FakeDebuggerPresent() {
    return TRUE;
}
```

IsDebuggerPresent : Cette fonction retourne `True` si un debugger est attaché au processus qui l'appelle.

Un malware vérifiera la valeur retournée par `IsDebuggerPresent()`. Si `True` est retourné, alors un debugger et présent, et le malware stoppe son exécution.

Nous modifions cette fonction pour toujours retourner `True`.

4.2.2 Modification du comportement en fonction des arguments

Le second type de modification décide en fonction des valeurs passées en paramètre de retourner ou non une fausse valeur que nous construisons de sorte à provoquer l'évasion. Un exemple du code d'instrumentation pour la fonction `getModuleHandle()` est présenté dans le Listing 4.4.

Listing 4.4: Code de la fonction *detour* de getModuleHandle.

```
// Detour function for GetModuleHandle
HMODULE WINAPI FalseModuleHandleA(LPCSTR lpModuleName) {
    LPCSTR szDlls[] = {
        "avghookx.dll",
        "avghooka.dll",
        "snxhk.dll",
        "sbiedll.dll",
        "dbghelp.dll",
        "api_log.dll",
        "dir_watch.dll",
        "pstorec.dll",
        "vmcheck.dll",
        "wpespy.dll",
        "cmdvrt64.dll",
        "cmdvrt32.dll",
        "sbiedll.dll"
    };
    WORD dwlength = sizeof(szDlls) / sizeof(szDlls[0]);
    for (int i = 0; i < dwlength; i++) {
        if(lpModuleName){
            if (lstrcmp(szDlls[i], lpModuleName) == 0) {
                return TrueModuleHandleA("user32.dll");
            }
        }
    }
    return TrueModuleHandleA(lpModuleName);
}
```

GetModuleHandle : Cette fonction retourne un descripteur d'objet appelé handle et qui permet d'accéder au module dont le nom est passé en paramètre ou `NULL` si elle n'existe pas.

Les malware vérifient la présence de DLL spécifiques appartenant à un antivirus ou un outil d'analyse en demandant l'accès à un handle [4]. Si le malware obtient le handle, il déduit alors que la librairie est présente et chargée. Le malware évasif stoppe donc son exécution.

4: https://docs.microsoft.com/fr-fr/windows/win32/sysinfo/handles-and-objects

Nous créons une liste de noms de librairies susceptibles d'être recherchées par les malware évasifs. Nous modifions le comportement de la fonction pour faire en sorte que si un des noms de la liste est présent en argument nous retournions un handle à `User32.dll`. Si la librairie n'est pas dans notre liste, nous appelons la méthode `GetModuleHandle()` originale.

RegOpenKeyEx : La fonction `RegOpenKeyEx()` retourne un handle à une clef de registre ouverte.

Des clefs de registre peuvent être ajoutées ou modifiées par les VMs et les outils d'analyse. Un malware évasif peut chercher la présence de ces clefs de registre afin de vérifier la présence de ces outils.

Nous créons une liste de noms de clefs de registre susceptibles d'être recherchées par les malware évasifs. La fonction `RegOpenKeyEx()` est modifiée pour renvoyer `ERROR_SUCCESS` si la clef recherchée fait partie de cette liste. Sinon le comportement normal est utilisé et nous retournons le résultat de `RegOpenKeyEx()` original.

RegQueryValueEx : Cette fonction retourne le contenu du registre dont le nom est passé en paramètre.

En plus de vérifier la présence d'une clef de registre, un malware évasif peut décider de lire son contenu.

Nous dressons une liste de noms de clefs de registre associée à des valeurs potentiellement recherchées par des malware évasifs. Si un malware demande le contenu d'une clef présente dans notre liste, nous retournons la valeur conçue pour le faire s'évader.

CreateFile : La fonction `CreateFile()` est utilisée pour créer ou ouvrir un fichier ou un périphérique I/O.

Les antivirus ou outils d'analyses peuvent ajouter des fichiers ou des périphériques I/O. Les malware évasifs peuvent vérifier leur présence en essayant de les ouvrir.

Nous avons créé une liste de noms de fichier et périphérique I/O susceptibles d'être recherchés par les malware évasifs. Cette fonction modifiée retourne `ERROR_SUCCESS` si le fichier ou périphérique demandé fait partie de la liste.

GetFileAttributes : Cette fonction retourne les métadonnées associées à un fichier passé en paramètre.

Une autre façon de vérifier la présence de fichiers est de demander l'accès à ses métadonnées. Si le malware évasif arrive à accéder à ces métadonnées, le fichier est présent donc un antivirus est potentiellement installé.

Si le fichier est présent dans notre liste, `FILE_ATTRIBUTE_NORMAL` est retourné. De la même manière, `FILE_ATTRIBUTE_DIRECTORY` est renvoyé pour un répertoire.

4.2.3 Leurres persistants après l'exécution de la fonction

Le dernier type de modification crée des changements persistants qui contrairement aux deux autres types de modifications précédents seront toujours visibles après l'appel à la fonction. Un exemple du code utilisé pour instrumenter la fonction `createToolHelp32Snapshot()` est présenté dans le Listing 4.5.

Listing 4.5: Code de la fonction *detour* de createHelp32Snapshot.

```
HANDLE WINAPI FalseCreateToolhelp32Snapshot(DWORD dwFlags, DWORD
     th32ProcessID){
    const TCHAR *szProcesses[] = {
        _T("ollydbg.exe"),      // OllyDebug debugger
        _T("ProcessHacker.exe"),  // Process Hacker
        _T("tcpview.exe"),      // Part of Sysinternals Suite
        _T("autoruns.exe"),     // Part of Sysinternals Suite
        _T("autorunsc.exe"),    // Part of Sysinternals Suite
        _T("filemon.exe"),      // Part of Sysinternals Suite
        _T("procmon.exe"),      // Part of Sysinternals Suite
        _T("regmon.exe"),     // Part of Sysinternals Suite
        _T("procexp.exe"),      // Part of Sysinternals Suite
        _T("idaq.exe"),       // IDA Pro Interactive Disassembler
        _T("idaq64.exe"),     // IDA Pro Interactive Disassembler
        // ... the list continues
    };

    WORD dwlength = sizeof(szProcesses) / sizeof(szProcesses[0]);

    // Creates exe files if not launched
    for (int i = 0; i < dwlength; i++) {
        BOOLEAN bProcessLauchned = CheckIfLaunched();

        if (!bProcessLauchned){
            STARTUPINFO info = { sizeof(info) };
            PROCESS_INFORMATION processInfo;
            TCHAR newPath[256] = CreateEmptyExeNamed(szProcesses[i]);
            CreateProcess(newPath, NULL, NULL, NULL, FALSE,
                CREATE_NO_WINDOW, NULL, NULL, &info, &processInfo);
        }
    }

    // Returns a snapshot
    return TrueCreateToolhelp32Snapshot(dwFlags, th32ProcessID);
}
```

CreateToolhelp32Snapshot : La fonction `CreateToolhelp32Snapshot()` permet de créer un snapshot du tas ainsi que de tous les modules, et threads utilisés par tous les processus du système. D'autres fonctions

de l'API Windows permettent de lister des processus, mais celle-ci fonctionne pour les anciennes versions ainsi que les actuelles.

Un malware peut chercher en mémoire la présence de processus appartenant aux antivirus.

Au premier appel à `CreateToolhelp32Snapshot()`, nous créons plusieurs processus de boucles infinies nommés avec des noms d'antivirus connus. Aux prochains appels à `CreateToolhelp32Snapshot()`, la fonction instrumentée vérifiera que les processus ne soient pas déjà présents avant d'en recréer.

4.3 Matériel et méthode

Toutes les expérimentations de cette thèse sont effectuées au Laboratoire Haute Sécurité (LHS) [5] de l'Inria Rennes sur la plateforme Malware o Matic (MoM) [56]. Le LHS est un lieu à accès restreint depuis lequel nous avons la possibilité d'exécuter des malware sur des machines physiques tout en leur accordant un accès internet.

5: https://www.pole-excellence-cyber.org/recherche/laboratoire-haute-securite-lhs/

[56]: Palisse et al. (2016), « Malware'O'Matic a platform to analyze Malware »

4.3.1 Plateforme MoM-v1

La plateforme d'analyse MoM permet d'analyser le comportement d'un ensemble de malware sur des ordinateurs physiques. Elle est composée d'un Network Attached Storage (NAS) qui contient tous les malware que le LHS de Rennes possède, de quatre machines clientes qui exécutent les malware et d'une machine serveur qui orchestre les expérimentations. Plusieurs versions de MoM existent et utilisent un matériel différent et améliorent le système d'orchestration des expérimentations. Nous détaillons maintenant la première version de MoM que nous utilisons afin de tester notre contre-mesure. La deuxième version de MoM à laquelle nous avons contribué pour répondre aux besoins de notre seconde contribution est présentée dans la Section 5.3.2.

Serveur

Le serveur de MoM [6] offre une API utilisée par des machines clientes pour envoyer des résultats d'une exécution ou pour télécharger le prochain malware à analyser par exemple. La Figure 4.1 présente un exemple de communication entre le serveur et un client. Le serveur contient une base de données qui liste pour chaque expérimentation les malware à tester et le résultat des exécutions. Il possède également des informations concernant l'état des machines clientes «en cours d'exécution» ou «en cours de restauration» par exemple. Les résultats envoyés par les clients peuvent être des fichiers assez volumineux. Le serveur MoM agit également en tant que serveur File Transfer Protocol (FTP) afin de les récupérer.

6: Caractéristiques du serveur MoM-V1 :
- HP Z400 Workstation
- 4Go DDR3 1333MHz
- Intel Xeon W3550 3.07Ghz

Clients

Nous utilisons quatre ordinateurs pour paralléliser les exécutions de malware et donc accélérer les expérimentations. Pour pouvoir comparer les résultats qui proviennent de deux machines différentes, nous devons les rendre aussi ressemblantes l'une de l'autre que possible. Pour cela, chaque ordinateur utilise le même matériel [7] et les images du système d'exploitation qu'ils utilisent sont toutes dérivées d'une même image de base. Seules les configurations réseau sont changées pour attribuer à chaque machine une adresse IP différente.

7: Caractéristiques des clients MoM-V1 :
— HP Z400 Workstation
— 4Go DDR3 1333MHz
— Intel Xeon W3550 3.07Ghz

Les images utilisées dans MoM-v1 sont des *Windows 7 Pro 64-bit* sur lesquels nous avons installé des logiciels légitimes tels que VLC, Firefox et Notepad++. Un corpus de documents Govdocs1 est présent dans le dossier `Mes Documents` [8]. Celui-ci contient par exemple des photos dans plusieurs formats et des documents au format Portable Document Format (PDF), Excel, Word ou HyperText Markup Language (HTML) par exemple. Ces modifications ont pour but de faire ressembler le plus possible ces clients à des machines de particuliers sur lesquels une activité humaine est présente.

8: http://downloads.digitalcorpora.org/corpora/files/govdocs1/

Les clients sont tous paramétrés pour télécharger la configuration de démarrage sur le serveur MoM. Si le serveur ne sait pas quel OS est installé sur la machine cliente ou si une exécution a déjà eu lieu, une restauration du disque dur est effectuée à l'aide de Clonezilla. Si l'état est connu et que l'image présente sur le disque est l'image vierge, alors le client démarre normalement sur le disque. Réinstaller une image vierge entre chaque exécution permet de partir d'un état initial quasi identique pour chaque malware testé.

Chaque image est configurée pour lancer une routine au démarrage de l'OS via une tâche planifiée. Cette routine appelle l'API du serveur MoM afin de récupérer le prochain malware à tester ainsi qu'un script propre à chaque expérimentation, que nous détaillons dans la Section 4.3.3. Avant de s'éteindre, le client prévient le serveur qu'il a besoin d'une restauration de son image avant de passer au prochain malware. Il envoie également les résultats de chaque exécution au serveur.

4.3.2 Déroulement d'une expérimentation

La Figure 4.1 permet de visualiser les communications qui ont lieu entre tous les éléments de la plateforme MoM. Une seule machine cliente est utilisée dans cet exemple dans un but de simplification, mais le comportement est similaire lorsque plusieurs machines clientes sont utilisées.

La création d'une expérimentation débute par la récupération par le serveur d'un ensemble de malware que nous souhaitons exécuter (1). Puis chaque échantillon est marqué comme «non testé» et le fichier de suivi d'état des machines est créé (2). C'est ce fichier de suivi qui permet au serveur de définir si une restauration de la machine cliente doit être effectuée. La machine cliente est ensuite démarrée afin de lancer l'expérimentation. La machine utilise l'API du serveur pour demander une configuration de démarrage (3). Le serveur envoie une configuration en fonction de ce que contient son fichier de suivi (4). Si la machine est

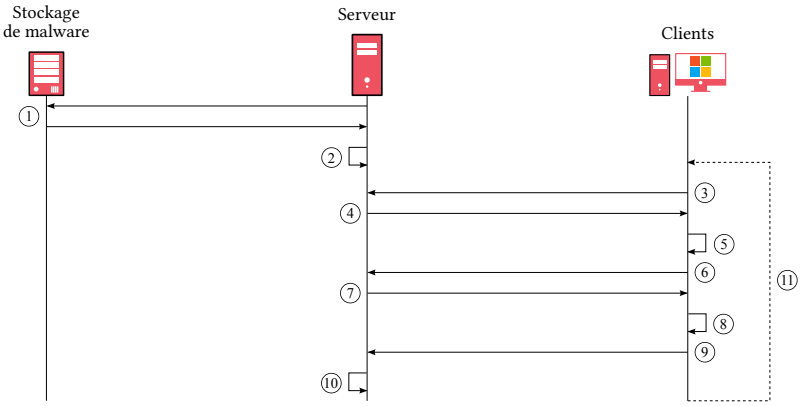


Figure 4.1 – Fonctionnement de la plateforme MoM :
1 - Récupération de malware
2 - Initialisation du fichier de suivi de l'état des machines
3 - Récupération de la configuration de démarrage
4 - Envoi de la configuration de démarrage
5 - Restauration puis redémarrage ou redémarrage seul
6 - Demande du prochain malware à exécuter
7 - Envoi du malware et du script d'expérimentation
8 - Exécution du script d'expérimentation
9 - Envoi des malware
10 - Mise à jour de l'état de la machine
11 - Redémarrage

dans un état inconnu ou exigeant une restauration d'après le fichier de suivi, le fichier de configuration envoyé contient une étape de restauration puis demandera un redémarrage classique sur le disque dur. Sinon, seul le redémarrage est demandé. Le client applique la configuration (5) et une fois le démarrage terminé, une tâche planifiée contacte l'API du serveur afin de demander l'envoi du prochain malware «non testé» de la liste ainsi qu'un script qui permet de l'exécuter et de le monitorer (6). Le serveur envoie les fichiers demandés (7) puis le client exécute le script pour démarrer l'analyse (8). Chaque script se termine par une partie conçue pour contacter l'API du serveur pour envoyer les résultats de l'exécution (9). À la réception des résultats, le serveur met à jour le fichier de suivi pour demander une restauration au prochain démarrage de la machine (10). Le client peut ensuite redémarrer pour analyser un nouveau malware (11).

4.3.3 Méthode

Le but de l'expérimentation mise en place consiste à mettre en évidence un effet de notre contre-mesure sur des malware évasifs, mais également une absence d'effet sur des malware non évasifs et des logiciels légitimes. Pour chaque malware nous effectuons deux exécutions, une première sans la présence de la contre-mesure et une seconde en sa présence.

La première exécution des échantillons s'effectue sur les machines clientes sur lesquelles est installée l'image de base décrite dans la Section 6. Le script d'expérimentation exécute un échantillon pendant cinq minutes et le surveille à l'aide de Process Monitor. Ce logiciel fait partie de la suite Sysinternals [9] et permet de surveiller l'activité du système de fichiers, des registres et des processus.

9: https://docs.microsoft.com/en-us/sysinternals/

La seconde exécution est ensuite effectuée à nouveau sur l'image de base, mais cette fois-ci avec notre contre-mesure installée. Nous utilisons le même script d'expérimentation qui démarre un échantillon et le surveille à l'aide de Process Monitor.

Pour chaque malware, nous obtenons deux fichiers Comma-Separated Values (CSV) générés par Process Monitor qui contiennent toutes les activités liées à l'échantillon étudié.

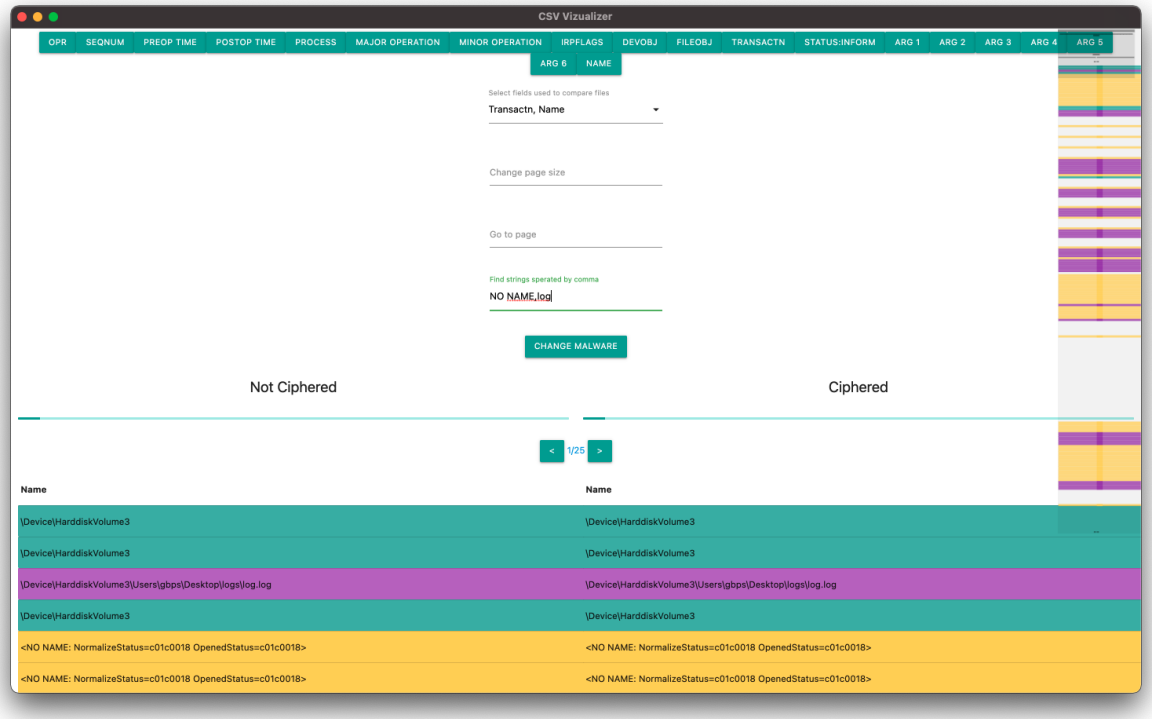


Figure 4.2 – Outil utilisé pour comparer deux CSV générés par Procmon.

Ces deux fichiers sont ensuite comparés à l'aide d'un outil que nous concevons dans le but de mettre en évidence leurs différences et qui est visible sur la Figure 4.2. L'outil permet de sélectionner des champs à comparer puis d'y rechercher des chaînes de caractères. Dans l'exemple présenté sur la Figure 4.2, nous comparons les champs *transaction* et *Name* à la recherche de deux chaînes de caractères *NO NAME* et *log*. Toutes les lignes dont un des deux champs sélectionnés contient la chaîne *NO NAME* sont colorées en violet et celles qui contiennent *log* en jaune. De plus, survoler une ligne dans une colonne colorera en vert la ligne de l'autre colonne dont les champs sélectionnés sont identiques. Une vue globale de la colorisation des fichiers est située à droite de la fenêtre et permet de repérer facilement une différence de colorisation. Dans notre exemple, nous observons que les couleurs sont parfaitement alignées, car les deux fichiers sont identiques. Une différence d'alignement indique qu'un des deux fichiers contient des étapes en plus ou en moins.

Dès qu'un point de divergence est trouvé, nous examinons la trace plus en détail et notons toutes les actions effectuées en plus ou en moins en présence de la contre-mesure. Si nous observons des différences telles qu'un temps d'exécution plus faible ou des sous-processus n'apparaissant plus dès lors que la contre-mesure est présente, alors nous considérons que la contre-mesure a eu un impact sur le malware étudié.

Dans cette expérimentation, nous testons Nuky, un malware évasif que nous avons créé et que nous présentons dans la Section 3.3 ainsi que le dataset de malware évasifs et de logiciels légitimes décrits dans la Section 3.4. Les résultats de l'effet de la contre-mesure sont présentés dans la Section 4.4.

4.4 Expérimentations

Dans le Chapitre 3, nous créons un malware évasif appelé Nuky ainsi qu'un petit ensemble de malware évasifs. Une première expérimentation permet de tester le bon fonctionnement de la contre-mesure à l'aide de notre malware de démonstration. Dans une seconde expérimentation, nous évaluons l'impact de notre contre-mesure sur l'ensemble de malware évasifs. Enfin, une dernière expérimentation mesure l'overhead induit par la contre-mesure sur quelques logiciels légitimes ainsi qu'Al-Khaser.

4.4.1 Validation du bon fonctionnement de notre contre-mesure

Nous vérifions le bon fonctionnement de la contre-mesure contre Nuky qui devrait stopper son exécution lorsqu'elle est présente. Pour cela, nous exécutons Nuky en suivant la méthode décrite dans la Section 4.3. Pour cette expérimentation, tous les blocs d'évasion et toutes les payloads de Nuky sont activés.

Lors de l'exécution dans l'environnement sans contre-mesure, Nuky chiffre tous les documents présents dans le dossier `Mes Documents`.

En présence de la contre-mesure, Nuky arrête son exécution immédiatement et ne lance aucune de ses payloads. Aucun fichier du dossier `Mes Documents` n'est donc chiffré.

Nous pouvons donc conclure que la contre-mesure fonctionne comme nous l'avons envisagé, car elle produit bien l'effet voulu sur Nuky qui stoppe son exécution.

4.4.2 Expérimentation sur des malware évasifs trouvés dans la nature

Nous testons la contre-mesure sur dix-huit malware récupérés d'après la méthode décrite dans la Section 3.4 et pour lesquels nous effectuons deux exécutions en suivant la méthode décrite dans la Section 4.3.3. La première exécution a lieu sur une image de base et permet d'avoir un comportement normal de l'application. La seconde utilise la même machine avec l'image de base qui est cette fois-ci équipée de la contre-mesure. Nous comparons ensuite les comportements du malware entre les deux exécutions.

Nous considérons que la contre-mesure a forcé avec succès un malware à s'évader si le comportement est différent entre les deux images. Les comportements des dix-huit malware sont comparés à la main en utilisant Process Monitor de la librairie *SysInternals* [10].

Pour quatorze malware (groupes A, B et C), nous observons un changement de comportement sur la machine équipée de la contre-mesure. Le processus parent s'arrête rapidement sans créer de sous-processus ce qui pourrait signifier qu'ils s'évadent.

10: https://docs.microsoft.com/en-us/sysinternals/downloads/procmon

Table 4.1 – Échantillons sélectionnés et résultats de la contre-mesure.
✓ : Le malware change de comportement
× : Le malware garde le même comportement

Groupe	MD5	Résultats
A	de3d1414d45e762ca766d88c1384e5f6	✓
	2d57683027a49d020db648c633aa430a	✓
	d3e89fa1273f61ef4dce4c43148d5488	✓
	bd5934f9e13ac128b90e5f81916eebd8	✓
	512d425d279c1d32b88fe49013812167	✓
	2ccc21611d5afc0ab67ccea2cf3bb38b	✓
	6b43ec6a58836fd41d40e0413eae9b4d	✓
B	ee12bbee4c76237f8303c209cc92749d	✓
	5253a537b2558719a4d7b9a1fd7a58cf	✓
	8fdab468bc10dc98e5dc91fde12080e9	✓
	e5b2189b8450f5c8fe8af72f9b8b0ad9	✓
	ee88a6abb2548063f1b551a06105f425	✓
	4d5ac38437f8eb4c017bc648bb547fac	✓
C	f4d68add66607647bf9cf68cd17ea06a	✓
D	862c2a3f48f981470dcb77f8295a4fcc	CRASH
E	e51fc4cdd3a950444f9491e15edc5e22	×
F	812d2536c250cb1e8a972bdac3dbb123	×
G	5c8c9d0c144a35d2e56281d00bb738a4	CRASH

Les résultats détaillés dans la Table 4.1 montrent que sur dix-huit malware, deux malware crashent dans les deux environnements et deux autres ne changent pas de comportement.

4.4.3 Overhead de la contre-mesure

Nous évaluons l'overhead produit par la contre-mesure sur quatre logiciels légitimes connus que sont VLC, Notepad, Firefox, et Microsoft Paint. Nous ajoutons également Al-Khaser, un programme exécutant plusieurs techniques d'évasion et générant un rapport sur leurs résultats. Nous avons retiré d'Al-Khaser toutes les techniques qui reposent sur l'observation du temps [11] ou qui attendent un certain temps qui permet d'atteindre le timeout de certaines sandbox qui arrêtent alors leur analyse. Ces techniques prennent beaucoup de temps à s'exécuter et notre contre-mesure n'a pas pour but de recréer ces artéfacts pour l'instant.

11: https://github.com/LordNotework/al-khaser/tree/master/al-khaser/TimingAttacks

Chaque logiciel est lancé cent fois, est manipulé pendant un temps en suivant un comportement scripté. Nous calculons ensuite les moyennes et écarts-types des temps d'exécution avec et sans la contre-mesure comme détaillées dans la Table 4.2.

Le seul overhead significatif que nous observons est avec Al-Khaser qui s'exécute pendant 13.29 secondes dans l'environnement standard et 15.24 secondes en présence de la contre-mesure, ce qui correspond à une

Table 4.2 – Temps d'exécution (en secondes) avec et sans la contre-mesure activée.

Logiciel	Moyenne sans contre-mesure	Moyenne avec contre-mesure	Ecart-type sans contre-mesure	Ecart-type avec contre-mesure
Al-Khaser	13.29	15.24 (+14.67%)	1.36	1.38
VLC	0.8593	0.8772 (+2.08%)	0.0677	0.0609
Notepad	0.0607	0.0615 (+1.32%)	0.0028	0.0033
Firefox	0.8026	0.8005 (-0.26%)	0.0259	0.0270
MS-Paint	0.0583	0.0581 (-0.34%)	0.0106	0.0096

augmentation de 14.67%. Cela peut être dû au fait qu'il appelle beaucoup les fonctions que nous avons instrumentées, ce que nous considérons comme le pire scénario. Les logiciels légitimes semblent quant à eux peu impactés, car aucune différence ne semble être significative.

4.5 Discussions et limitations

Tout comme une base de signatures, notre contre-mesure a besoin d'être mise à jour de façon continue pour être capable de stopper de nouvelles techniques. Naturellement, avec cette méthode nous pouvons seulement bloquer les malware évasifs qui utilisent l'API Windows et pas ceux qui passent par d'autres moyens. En effet, pour qu'un malware soit impacté par notre contre-mesure il doit absolument passer par les fonctions que nous avons instrumenté. Des preuves de concepts existent et prouvent qu'un malware peut éviter d'utiliser les fonctions instrumentées de plusieurs manières [12]. Effectuer les instrumentations de notre contre-mesure à un niveau plus bas et privilégié permettrait de rendre plus difficiles les techniques utilisées par ce genre de malware.

12: https://medium.com/@omribaso/this-is-how-i-bypassed-almost-every-edr-6e9792cf6c44

Notre contre-mesure semble fonctionner sur certains malware mais la taille du dataset ne nous permet pas de conclure réellement sur son efficacité pour le moment. Aucun répertoire public ne fournit de malware correctement détaillés avec les techniques d'évasion qu'ils utilisent et c'est pour cette raison que nous utilisons un petit dataset que nous avons créé et analysé à la main.

4.6 Conclusion

Nous avons élaboré une contre-mesure en instrumentant l'API Windows à l'aide de Microsoft Detours. Les faux artéfacts mis en place ont permis de stopper quatorze malware sur dix-huit, qui ont montré un changement de comportement en présence de la contre-mesure. L'overhead sur le temps d'exécution le plus important est mesuré sur Al-Khaser et est de 14.62% et nous ne notons pas de réelle différence sur des logiciels légitimes. Le code complet de la contre-mesure est accessible dans l'Annexe A et celui de la règle Yara dans l'Annexe B.

Une grande difficulté que nous identifions est l'obtention d'un dataset d'échantillons évasifs. L'évaluation de notre contre-mesure est faite sur un petit nombre d'échantillons, ce qui ne nous permet pas de conclure

sur son efficacité. Nous souhaitons donc créer un dataset de malware évasifs correctement étiquetés et qui permettrait à toute personne qui souhaite étudier le sujet de créer une expérimentation robuste.

Création d'un dataset de malware évasifs

5

Dans la section précédente, nous évaluons notre contre-mesure en utilisant quelques malware récupérés à l'aide de règles Yara. Nous concluons à la nécessité de répéter l'expérimentation sur un dataset plus grand et correctement étiqueté afin de réellement déterminer l'efficacité de cette contre-mesure.

Pour ce faire, nous avons effectué plusieurs essais de datasets disponibles publiquement ou sur invitation avant de prendre la décision d'en construire un nouveau. Les raisons qui nous ont poussés à ne pas recourir à l'existant, mais plutôt à utiliser un ensemble de malware non étiquetés sont détaillées dans la Section 5.1.

Notre seconde contribution est un dataset de malware évasifs étiquetés de deux façons.

Un premier étiquetage automatique se base sur la contre-mesure issue de notre première contribution. Sa mise en œuvre ainsi que ses résultats sont présentés en détail dans la Section 5.3.

En parallèle, nous créons The Evasive Malware PlatfOrm (TEMPO), une plateforme web présentée dans la Section 5.4 permettant à des experts de mettre une étiquette sur les mêmes malware que ceux traités par la détection automatique.

Ce second étiquetage manuel sert de vérité terrain afin de vérifier que les malware étiquetés à l'aide de notre contre-mesure soient bien évasifs.

Figure 5.1 – La plateforme Tempo se trouve au lien suivant : https://tempo.irisa.fr

5.1 Datasets existants

Deux critères qui nous semblent les plus importants dans le choix d'un dataset sont la justesse des informations fournies ainsi que la quantité d'échantillons disponibles. Un ensemble de malware qui réunit ces deux critères permet d'interpréter les résultats avec un niveau de confiance important.

Plusieurs datasets tels que Virus Total [1] ainsi qu'Androzoo [57] permettent d'acquérir une grande quantité d'échantillons. Certains échantillons peuvent être associés à des informations basiques telles que des hashs, l'architecture ciblée et la date de la première découverte par exemple. Ces informations basiques sont assez simples à obtenir et nous pouvons nous y fier sans avoir à grandement les remettre en question. En revanche, d'autres informations plus complexes à obtenir telles que la famille du malware, la présence d'un packer ou la présence d'un comportement évasif peuvent plus facilement être erronées.

Nous avons par exemple effectué des tests en récupérant des malware étiquetés «stealth» du dataset de VirusTotal accessible aux chercheurs sur demande. En analysant certains échantillons, nous n'arrivions pas à comprendre les raisons de la présence de ce label. Le problème est que les auteurs de ces grands datasets ne les donnent pas. Savoir à l'avance si un

1: https://www.virustotal.com

[57]: Allix et al. (2016), « AndroZoo : Collecting Millions of Android Apps for the Research Community »

échantillon est réellement un malware, s'il pourra s'exécuter et si les étiquettes sont correctes est complexe. Nous avons rencontré des problèmes similaires avec des échantillons qui proviennent de MalwareBazaar ou d'AnyRun par exemple.

Nos premières expérimentations qui utilisaient cinquante-trois malware labélisés «stealth» et qui provenaient de VirusTotal produisaient de mauvais résultats et nous n'arrivions pas à déterminer si cela venait de la méthode de construction ou d'un mauvais dataset. Sur les cinquante-trois malware récupérés, quarante-neuf font partie d'une même famille appelée «Virlock» et quarante-et-un ne s'exécutent pas. Enfin, ces plateformes contiennent également des échantillons qui ne sont pas des malware et nous ne possédons pas d'informations qui nous permettent de savoir qu'ils sont en réalité bénins.

Ces datasets sont plus pertinents lorsqu'ils alimentent des expérimentations qui nécessitent une grande quantité d'échantillons et qui attachent une moins grande importance à la qualité des labels. Dans notre cas en revanche, nous souhaitons pouvoir accorder une grande confiance aux labels quitte à posséder un dataset de taille plus modeste.

Ce sont pour ces raisons que nous avons décidé de créer un nouveau dataset en commençant par récupérer quelques milliers d'échantillons sur plusieurs sites internet comme nous le décrivons dans la Section 3.4.2. Ceux-ci sont ensuite filtrés grâce à notre règle Yara pour ne retenir que les malware les plus susceptibles de présenter un comportement évasif. Cependant, les faibles scores retournés par VirusTotal ainsi que les quelques exécutions que nous avons tentées nous laissent penser que ce dataset contient beaucoup d'échantillons qui ne sont pas des malware ou qui ne s'exécutent pas. Le filtrage nous a permis de ne récupérer que dix-huit malware évasifs en trois mois comme nous le présentons dans la Section 3.4.2.

Afin de récupérer un grand nombre de malware fortement susceptibles d'être exécutables, nous utilisons Malpedia [2], qui est un dataset qui fournit exclusivement des échantillons malveillants souvent associés à des rapports d'analystes. En six mois d'utilisation, nous n'avons pour l'instant pas trouvé d'échantillons qui sont des logiciels légitimes et non des malware. De plus, nous constatons que parmi les 3152 échantillons téléchargés 2263, soit 72%, sont bien exécutables ce qui nous semble être une part importante. Malpedia répond donc bien à notre besoin de posséder un grand nombre de malware exécutables. Bien que ces malware s'exécutent, cela ne signifie pas qu'ils sont encore actifs. Nous avons relevé quelques échantillons qui ne déclenchent pas leur payload, car les serveurs Command & Control (C&C) chargés de les transmettre ne sont plus disponibles. Malpedia reste malgré tout la source de malware exécutables la plus fiable que nous ayons trouvée à ce jour.

2: https://malpedia.caad.fkie.fraunhofer.de

Cependant, Malpedia ne fournit pas de labels «évasifs» et nous devons donc les étiqueter nous-mêmes. Dans le cas des malware évasifs, nous considérons qu'actuellement, un étiquetage effectué par un expert contiendra moins d'erreurs qu'un étiquetage automatique. En revanche, une analyse humaine ne permet pas de traiter beaucoup de malware, ni donc d'obtenir un dataset qui contient beaucoup d'échantillons. C'est pour cette raison que nous mettons en place un moyen pour les experts

d'étiqueter collaborativement le dataset de Malpedia et prévoyons ensuite d'effectuer cet étiquetage de manière automatique.

5.2 Objectifs du nouveau dataset de malware évasifs

Plusieurs articles présents dans notre bibliographie que nous présentons dans la Section 2.1 expliquent avoir effectué un étiquetage à la main afin de créer une vérité terrain. Nous souhaitons proposer une plateforme qui permet de mutualiser ces efforts en offrant à tout le monde la possibilité de participer à cet étiquetage. Un chercheur qui ne souhaite pas étiqueter lui-même les malware peut tout de même télécharger ce dataset et commencer directement ses expérimentations.

Cette collaboration permet de confronter les avis de chacun, mais également de comprendre les raisons qui ont poussé un expert à étiqueter un malware comme étant évasif. Le nombre de malware évasifs ainsi identifiés est moins important qu'en utilisant un étiquetage automatique, mais la confiance que l'on peut accorder aux labels est plus grande.

Nous listons quatre critères que le dataset doit respecter afin de faciliter son utilisation par les expérimentateurs, mais également pour renforcer la confiance que l'on peut avoir dans les expérimentations qui l'utilisent.

Le dataset doit contenir des informations pour chaque échantillon que nous devons pouvoir vérifier. Rendre publiques ces informations permet de savoir exactement de quoi il est composé et facilite ainsi l'identification d'éventuels biais. Un expérimentateur qui récupère notre dataset peut de ce fait comprendre comment nous étiquetons nos malware et peut vérifier que les labels que nous créons correspondent bien à sa définition d'un malware évasif. Mais cela permet également à n'importe quel expérimentateur de se rendre compte d'un mauvais étiquetage éventuel et peut alors les signaler pour les corriger.

De nouvelles informations concernant des malware peuvent être découvertes grâce à de nouvelles techniques d'analyse. Des malware considérés jusqu'alors comme non évasifs peuvent subir un nouvel étiquetage en réponse à l'apparition de ces nouveaux éléments. Si tel est le cas, il doit être nécessaire de pouvoir corriger le dataset en accord avec les nouvelles informations que nous possédons. Les expérimentations peuvent ensuite être réévaluées avec le dataset corrigé. L'interprétation des résultats est alors plus proche de la réalité et la confiance que nous pouvons leur accorder plus forte.

Les cas d'usage possibles des datasets peuvent changer énormément entre chaque expérimentateur. Une expérimentation peut souhaiter cibler les malware sur une période précise ou ceux qui sont inférieurs à une certaine taille par exemple. Posséder suffisamment d'échantillons est nécessaire afin de créer un dataset qui peut convenir aux divers besoins des expérimentateurs.

Un accès facilité au dataset permet aux utilisateurs de tester plus simplement et plus régulièrement leurs expérimentations. Cette facilité d'accès permet aux utilisateurs d'utiliser la dernière version du dataset à chaque

visite et ainsi leur assure de toujours utiliser la version la plus fiable disponible.

5.3 Étiquetage automatique

L'étiquetage automatique que nous mettons en place se base sur la Proposition 5.3.2 développée dans la Section 5.3.1 et qui considère qu'un malware évasif se comporte différemment dès lors qu'il se trouve en présence de notre contre-mesure. Nous testons la validité de cette proposition à l'aide des résultats issus de l'étiquetage collaboratif dans la Section 5.5.

Chaque malware est analysé grâce à plusieurs étapes présentées sur la Figure 5.2. Les échantillons sont exécutés au moyen de la plateforme MoM décrite dans la Section 5.3.2. Cette plateforme exécute plusieurs fois un malware dans deux environnements différents tout en effectuant un monitoring que nous présentons dans la Section 5.3.3. Les mesures relevées lors du monitoring sont ensuite prétraitées pour extraire des critères de comparaison que nous détaillons dans la Section 5.3.4. Enfin, nous calculons un score à partir de ces critères à l'aide d'un test statistique que nous décrivons dans la Section 5.3.6. Un seuil est par la suite fixé qui, s'il est dépassé, permet de considérer un malware comme évasif.

5.3.1 Définitions et propositions de départ

Dans cette section, nous utilisons les termes d'environnement standard et d'environnement d'analyse dont nous proposons les Définitions 5.3.1 et 5.3.2.

Définition 5.3.1 *L'**environnement standard** est conçu pour représenter ce qui correspond à ce qu'un auteur de malware souhaite cibler. Il contient des logiciels de bureautique classique tels que des éditeurs de texte, applications de prise de notes, messageries et calendriers, mais également des programmes très populaires comme des navigateurs web ou des lecteurs vidéo. Un environnement réaliste doit également comporter plusieurs fichiers tels que des documents, photos et vidéos.*

Définition 5.3.2 *L'**environnement d'analyse** est conçu pour représenter ce qui correspond à ce qu'un auteur de malware souhaite éviter. Il peut contenir les mêmes programmes légitimes qu'un environnement standard, mais comporte en plus des logiciels d'analyse utiles à la rétro-ingénierie tels que des debuggers, désassembleurs et des machines virtuelles par exemple. Tout comme un environnement standard, il peut comporter plusieurs fichiers tels que des documents, photos et vidéos.*

Les résultats présentés dans le chapitre précédent nous laissent penser que la contre-mesure impacte les malware évasifs et affecte peu les logiciels légitimes. De plus, plusieurs échantillons évasifs montrent des comportements différents dès qu'ils se trouvent en sa présence. Des malware évasifs ont par exemple réagi à notre contre-mesure en ne créant pas de sous-processus.

De ces deux constatations, nous émettons les Propositions 5.3.1 et 5.3.2.

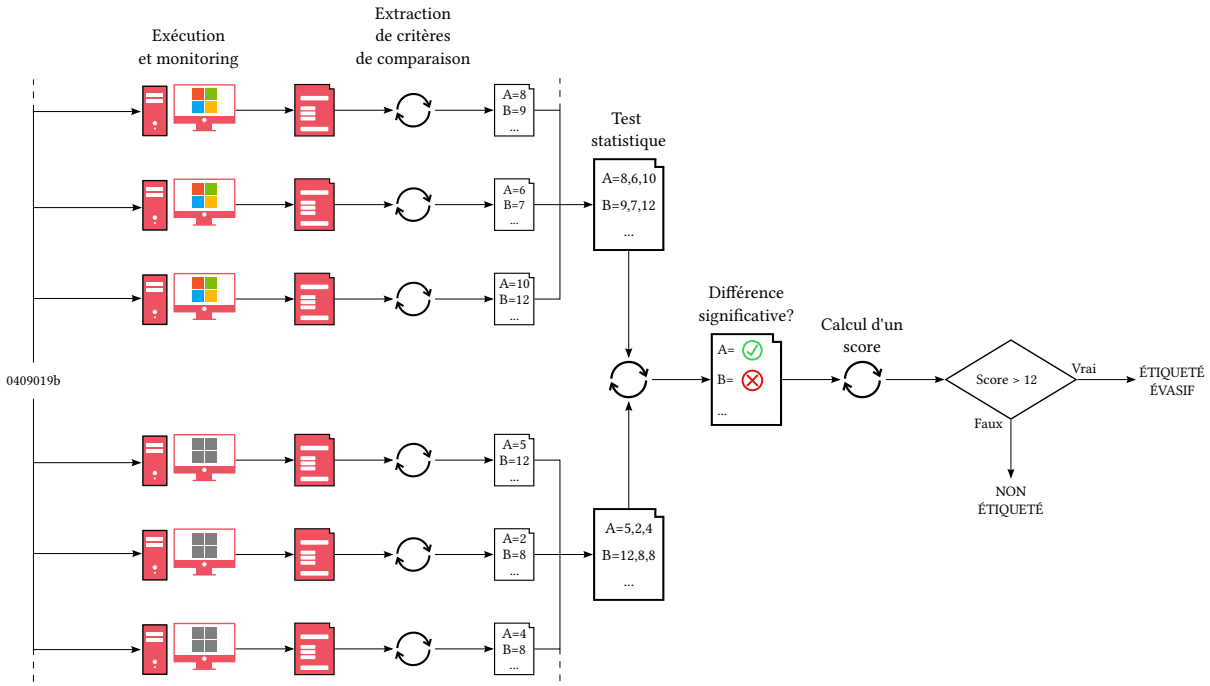


Figure 5.2 – Fonctionnement de l'étiquetage automatique d'un malware.

Proposition 5.3.1 *La présence de notre contre-mesure dans un environnement standard permet de le faire passer pour un environnement d'analyse.*

Proposition 5.3.2 *Si le comportement d'un malware dans un environnement standard est différent de son comportement dans un environnement d'analyse, alors nous considérons que ce malware est évasif.*

La robustesse de notre détection automatique repose sur la confiance que nous portons sur l'efficacité de notre contre-mesure et sur la validité de notre Proposition 5.3.2, que nous vérifions grâce à l'étiquetage collaboratif, présenté dans la Section 5.5.

5.3.2 Matériel utilisé pour l'expérimentation

La nouvelle version de MoM utilisée pour cette nouvelle expérimentation est toujours présente au LHS mais a été légèrement améliorée par l'ajout d'une fonctionnalité que nous présentons dans la Section 4.

Le matériel des machines clientes [3] est changé ainsi que celui du serveur. L'orchestration des expérimentations effectuée par le serveur [4] est également modifiée pour offrir de nouvelles fonctionnalités que nous présentons maintenant.

3: Caractéristiques des clients MoM-V2 :
- OptiPlex 7060 micro MFF
- 8Go DDR4 2666MHz
- i5-8500T 2.1GHz

4: Caractéristiques du serveur MoM-V2 :
- OptiPlex 7060
- 8Go DDR4 2666MHz
- i5-8400 2.8GHz

Serveur

Le stockage de la liste de malware à tester ne s'effectue plus dans une base de données, mais dans des fichiers JavaScript Objet Notation (JSON) qui permettent également de configurer l'expérimentation. Une expérimentation créée à l'aide de MoM-v1 ne permet d'exécuter une

série de malware qu'une fois et pour une image. Le nouveau fichier de configuration de MoM-v2 ajoute la possibilité de définir un nombre quelconque d'exécutions pour chaque malware ainsi que plusieurs images sur lesquelles elles doivent avoir lieu.

Cette simplicité de configuration nous permet de mettre en place une seule expérimentation et de recevoir les résultats après chaque exécution. Ceux-ci sont traités au fur et à mesure ce qui rend possible l'affichage des résultats en temps réel sur une interface web.

Nous utilisons cette fonctionnalité pour mettre en place l'expérimentation de détection automatique de malware évasifs présentée dans la Section 5.3. Chaque malware est exécuté quinze fois dans deux environnements pour un total de trente exécutions. La raison de ce nombre d'exécutions est présentée dans la Section 5.3.6.

Clients

Les quatre machines clientes utilisent à nouveau toutes le même hardware qui a été changé pour correspondre aux machines actuelles auxquelles toute personne peut avoir accès.

Contrairement à la première version où une image était créée par machine avec une configuration réseau différente, cette nouvelle version utilise une unique image dont les adresses Internet Protocol (IP) sont attribuées par Dynamic Host Configuration Protocol (DHCP) pour toutes les machines. Cela évite les erreurs humaines où il était possible de se tromper dans la configuration d'une des images. En utilisant une unique image, nous sommes dorénavant certains que toutes les machines sont configurées exactement de la même manière et utilisent les mêmes versions des scripts ou drivers par exemple.

Le fait d'utiliser un hardware et une image identiques pour tous les ordinateurs permet d'obtenir un état initial des machines physiques le plus proche possible.

Images utilisées

Nous utilisons deux images créées à partir de la même image *Windows 10 version 10.0.19042* sur lesquelles nous avons installé les mêmes logiciels légitimes que pour MoM-v1. Le corpus de documents Govdocs1 a encore une fois été déposé dans le dossier `Mes Documents` [5]. L'interpréteur Python est cette fois-ci renommé en pychon afin d'empêcher un malware évasif de repérer l'exécution d'un script en listant les processus à la recherche d'un programme contenant python dans le nom. Nous pensons que la présence d'un processus python en mémoire peut être une condition d'évasion.

5: http://downloads.digitalcorpora.org/corpora/files/govdocs1/

Les images sont paramétrées pour se connecter automatiquement à une session possédant les droits administrateur. Une tâche planifiée est exécutée au bout de deux minutes afin de laisser le temps à tous les programmes de démarrer. Cette tâche planifiée utilise l'interpréteur Python que nous avons renommé pour exécuter un même script d'initialisation présent dans les deux images. Ce script d'initialisation contacte le serveur MoM à la recherche du prochain malware à exécuter ainsi que le

script d'expérimentation. C'est ce script d'expérimentation qui démarre le malware et effectue des mesures tout au long de son exécution. La procédure est identique à celle détaillée dans la Section 4.3.2.

Les deux images sont identiques à l'exception de notre contre-mesure installée dans une seule des images. Ici, la contre-mesure est utilisée afin de simuler un environnement qu'un malware évasif doit éviter. Nous obtenons ainsi un environnement où le malware évasif devrait exécuter sa payload et un second où il devrait se désactiver.

5.3.3 Monitoring effectué lors de l'exécution d'un malware

Dans le cas des malware évasifs, nous souhaitons que notre outil d'analyse soit invisible du malware et évitons par conséquent les programmes tels que les debuggers qui sont facilement détectables.

Les malware évasifs que nous avons en notre possession effectuent des actions en moins lorsqu'ils s'évadent. Dans ce cas, il paraît probable que le malware consomme moins de ressources mémoires, de CPU et mette un peu moins de temps à s'exécuter. De plus, les logiciels légitimes semblent peu affectés par la présence de notre contre-mesure comme nous le décrivons dans la Section 4.4.3. Un monitoring est effectué afin de repérer ces différences de consommation de ressources. Pour cela, nous utilisons trois fonctions de l'API Windows.

Chaque malware est créé dans un état suspendu à l'aide du creation flag `CREATE_SUSPENDED` [6] afin que l'on ait le temps de mettre en place le monitoring avant de le démarrer et ainsi capturer toute son exécution.

6: https://docs.microsoft.com/en-us/windows/win32/procthread/process-creation-flags

Un sous-processus est ensuite créé afin d'enregistrer la consommation en ressources du malware. Les trois fonctions de l'API Windows utilisées pour le monitoring nécessitent un handle permettant d'accéder au processus du malware et de l'observer. Ce handle est récupéré à l'aide de la fonction OpenProcess et en demandant un accès `PROCESS_QUERY_INFORMATION` [7].

7: https://docs.microsoft.com/en-us/windows/win32/procthread/process-security-and-access-rights

Le processus du malware est ensuite résumé à l'aide de la fonction NtResumeProcess afin de démarrer son exécution. Ainsi, le démarrage du malware est enregistré par le script de monitoring.

Le malware est exécuté et observé pendant cinq minutes avant d'être stoppé s'il n'a pas terminé son exécution avant.

Nous détaillons maintenant les données enregistrées à chaque exécution et listées dans la Table 5.1.

Durées

Nous enregistrons tout d'abord les dates de création et d'extinction du processus afin de définir sa durée d'exécution (Execution Time). Les temps d'exécution en mode noyau (Kernel Time) et en mode utilisateur (User Time) sont également relevés. Pour cela nous utilisons la fonction de l'API Windows `GetProcessTimes()` définie dans le Listing 5.1.

Table 5.1 – Tableau récapitulatif des mesures effectuées pour une exécution.

Type	Noms des mesures	Format
Durées	Execution Time	Float
	Kernel Time	Float[]
	User Time	Float[]
Mémoire	Page Fault Count	SIZE_T[]
	Working Set Size	SIZE_T[]
	Quota Paged Pool Usage	SIZE_T[]
	Quota Non Paged Pool Usage	SIZE_T[]
	Page File Usage	SIZE_T[]
CPU	CPU Cycle	PULONG64[]

Listing 5.1: Fonction GetProcessTimes (processthreadsapi.h)

```
BOOL GetProcessTimes(
  HANDLE     hProcess,
  LPFILETIME lpCreationTime,
  LPFILETIME lpExitTime,
  LPFILETIME lpKernelTime,
  LPFILETIME lpUserTime
);
```

Consommation mémoire

L'utilisation de la fonction `GetProcessMemoryInfo()` définie dans le Listing 5.2 permet d'enregistrer le nombre de fautes de pagination (Page Fault Count) ainsi que la quantité de mémoire requise par le processus (Working Set Size). La consommation des pages mémoire utilisées par le noyau et les pilotes (Quota Non Paged Pool Usage) et les pages mémoire utilisées par exemple par les registres (Quota Paged Pool Usage) sont également enregistrées . La dernière mesure effectuée concerne l'utilisation des fichiers de page (Page File Usage) qui peuvent être utilisés afin d'étendre la mémoire physique de la Random Access Memory (RAM) par exemple.

Listing 5.2: Fonction GetProcessMemoryInfo (psapi.h)

```
BOOL GetProcessMemoryInfo(
  HANDLE Process,
  PPROCESS_MEMORY_COUNTERS
    ppsmemCounters,
  DWORD  cb
);
```

Consommation CPU

La consommation des ressources CPU est représentée par le nombre de cycles d'horloge CPU (CPUCycle) utilisé par le processus. La fonction de l'API Windows utilisée est `QueryProcessCycleTime()` définie dans le Listing 5.3

Listing 5.3: Fonction QueryProcessCycleTime (realtimeapiset.h)

```
BOOL QueryProcessCycleTime(
  HANDLE   ProcessHandle,
  PULONG64 CycleTime
);
```

Toutes les mesures sont écrites au fur et à mesure de l'exécution du malware dans un fichier qui se trouve à la racine du disque `C:\` de la machine cliente. Ce fichier au format CSV d'environ 95Mo est ensuite transmis au serveur de MoM-v2 qui le traite afin d'extraire des critères de comparaison puis qui les transforme dans un même format qui facilite leur manipulation.

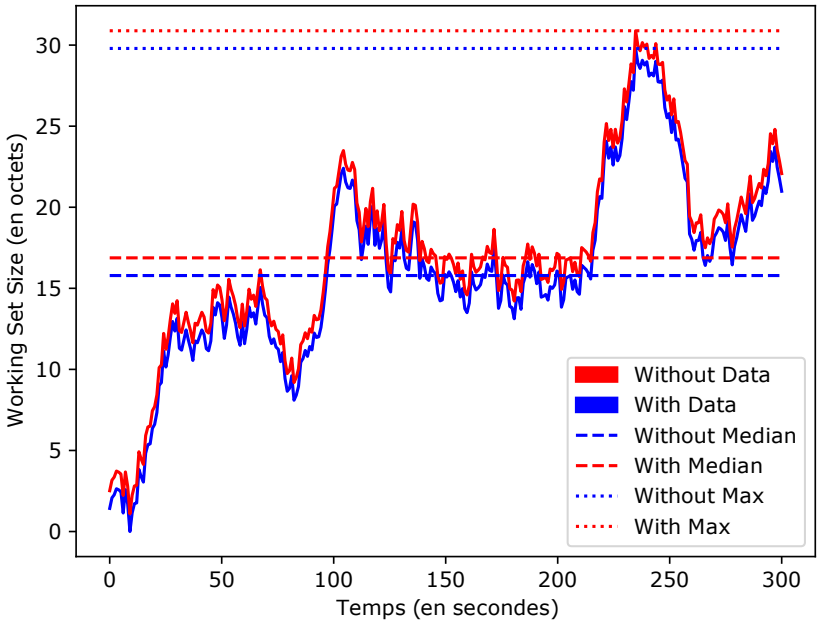
5.3.4 Extraction de critères de comparaison

Les neuf données brutes mesurées par le monitoring de chaque exécution permettent de définir si le malware s'est comporté différemment dans les deux environnements.

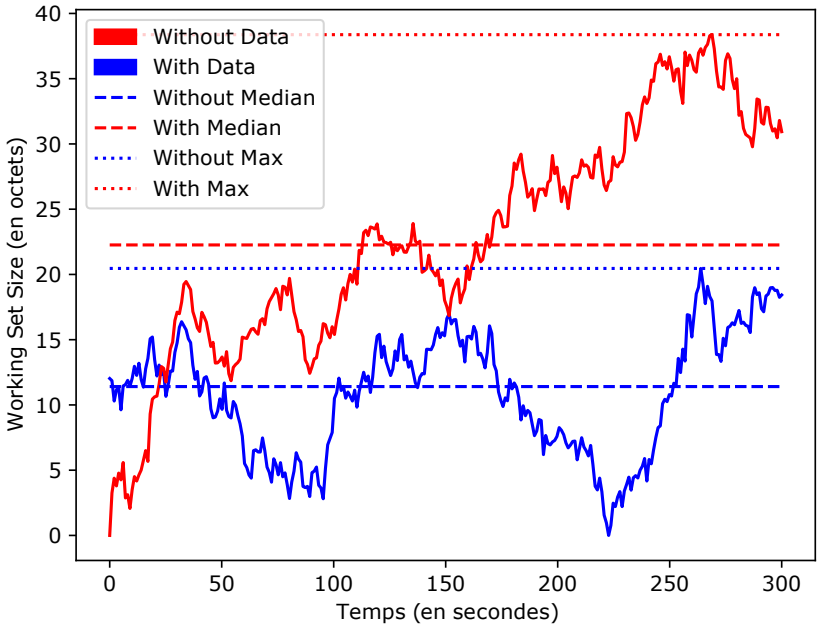
	Sans contre-mesure	Avec contre-mesure
Présence d'un impact	116.4	0.555 (-99.5%)
Absence d'un impact	103.4	103.36 (-0.039%)

Table 5.2 – Observation des temps d'exécution en secondes dans chaque environnement pour déduire un éventuel impact de la contre-mesure. Ces valeurs sont simulées.

Parmi celles-ci, le temps d'exécution est une donnée simple à comparer, car une seule valeur est retournée par exécution. Une valeur significativement plus faible ou plus élevée entre l'environnement standard et celui d'analyse indiquerait une différence de comportement comme présenté dans la Table 5.2.



(a) Cas 1 : L'évolution du *Working Set Size* est similaire en présence ou non de la contre-mesure.



(b) Cas 2 : L'évolution du *Working Set Size* est différente en présence de la contre-mesure

Figure 5.3 – Comparaison des mesures du *Working Set Size* d'abord sans puis avec la présence de la contre-mesure. Les maximums et médianes de chaque courbe sont également représentés. Ces données ont été générées aléatoirement.

Les huit autres données enregistrées lors du monitoring ne sont pas des valeurs uniques, mais des séries de mesures effectuées tout au long d'une exécution et dont deux exemples sont présentés sur la Figure 5.3.

Afin de comparer les deux courbes, nous extrayons deux critères que sont la moyenne et le maximum, qu'il est possible de comparer simplement. Par exemple, dans le cas présenté sur la Figure 5.3a, nous observons que la moyenne des mesures du *Working Set Size* effectuées dans l'environnement standard est proche de la moyenne des mesures effectuées dans l'environnement d'analyse. La même constatation s'applique pour les maximums des deux courbes.

En revanche, dans le cas présenté sur la Figure 5.3b, les moyennes et les maximums des mesures du *Working Set Size* effectuées dans l'environnement d'analyse semblent significativement plus élevés de ceux relevés dans l'environnement standard.

Nous obtenons ainsi un critère de comparaison qui correspond au temps d'exécution du malware que nous pouvons comparer immédiatement. Seize autres critères proviennent de l'extraction des moyennes et maximums des huit mesures sous forme de série de données. Au total, nous utilisons dix-sept critères listés dans la Table 5.3 et qui nous permettent de comparer deux exécutions d'un malware.

Or en exécutant deux fois à la suite un même malware dans un environnement identique et sur un ordinateur inchangé, nous observions ponctuellement des variations sur ces critères. Les machines subissent des aléas lors du démarrage bien qu'utilisant toutes un hardware ainsi qu'une image Windows identique. L'état initial d'un ordinateur peut alors différer à chaque itération ce qui entraîne des exécutions et donc des mesures dissimilaires.

Une solution qui permet d'atténuer l'impact de ces aléas consiste à ne pas comparer deux uniques exécutions prises dans deux environnements, mais deux ensembles d'exécutions effectuées dans chacun des deux environnements. Nous effectuons quinze exécutions dans chacun des deux environnements et expliquons les raisons de ce nombre dans la Section 5.3.6.

5.3.5 Adaptation pour comparer de multiples exécutions

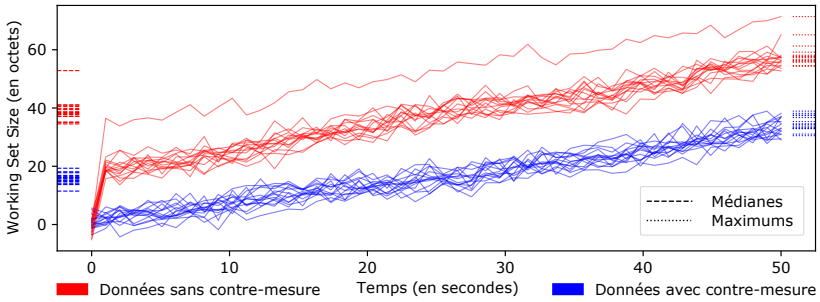
Effectuer n exécutions dans deux environnements implique une comparaison de deux ensembles de n mesures et non plus seulement de deux mesures uniques.

Une visualisation qui nous permet de nous donner une idée sur une éventuelle présence d'un écart important entre deux ensembles de mesures est le diagramme en boîte. La Figure 5.5 permet par exemple de visualiser les mesures des temps d'exécution effectuées dans deux environnements. Grâce à cela, nous pouvons identifier visuellement qu'une différence significative existe, mais également de repérer si des valeurs aberrantes apparaissent.

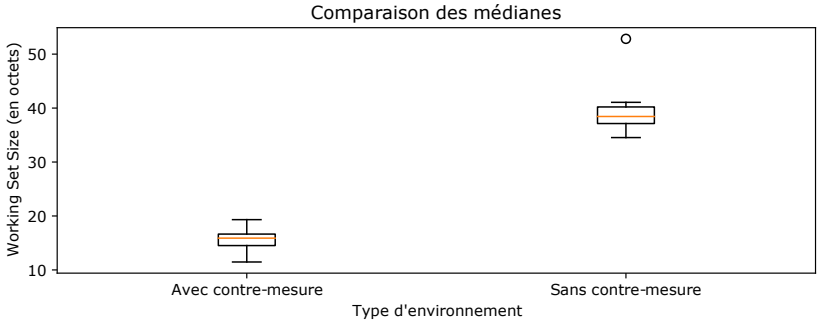
Un exemple de mesures de type série de données est représenté sur la Figure 5.4a où l'on peut observer une superposition de toutes les mesures du *Working Set Size* prises pour un malware dans deux environnements.

Tout comme pour la comparaison de deux courbes présentée dans la Section 5.3.4, nous commençons par extraire les maximums et les moyennes de chaque courbe. Pour n exécutions dans deux environnements, nous obtenons ainsi deux ensembles de n moyennes et deux ensembles de n maximums.

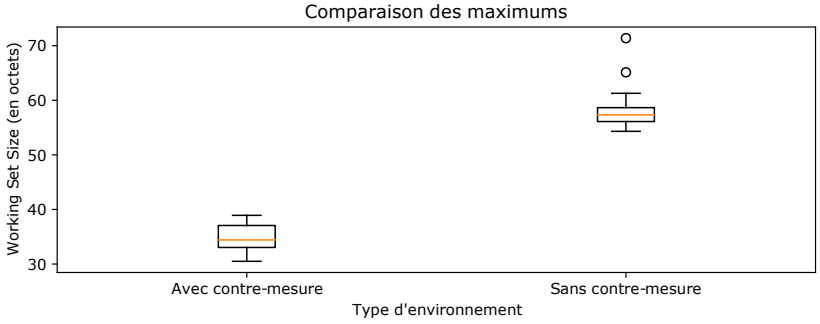
Les n moyennes des mesures du *Working Set Size* effectuées dans un environnement d'analyse peuvent ensuite être comparées aux n moyennes des mesures effectuées dans un second environnement standard.



(a) Superposition de toutes les mesures du *Working Set Size* et affichage de leurs moyennes et maximums. Deux environnements sont comparés, le premier contient notre contre-mesure et le second ne le contient pas.



(b) Comparaisons des moyennes du *Working Set Size* en présence ou non de notre contre-mesure.



(c) Comparaisons des maximums du *Working Set Size* en présence ou non de notre contre-mesure.

Figure 5.4 – Représentation d'une mesure de type série de données telle que le *Working Set Size* pour toutes les exécutions d'un malware dans deux environnements. Pour chaque courbe les moyennes ainsi que les maximums sont extraits puis visualisés sous forme de diagramme en boîte. Ces données ont été générées aléatoirement.

Il est ainsi possible de visualiser chacun des dix-sept critères sous forme de diagramme à boîte pour observer la présence ou non de différence significative.

Le fait d'avoir multiplié le nombre d'exécutions permet d'atténuer les différences dues aux aléas. Sur la Figure 5.4a, nous pouvons voir que dans l'environnement sans contre-mesure, une exécution s'est comportée différemment en ayant des mesures supérieures à toutes les autres. Grâce

au diagramme en boîte présenté sur la Figure 5.4c, nous pouvons voir que cette exécution est représentée par un cercle signifiant qu'elle est aberrante par rapport à celles effectuées dans le même environnement.

Cependant, ces diagrammes permettent seulement d'avoir une indication visuelle de la présence d'une différence significative et nous souhaitons pouvoir la détecter automatiquement. Pour cela, nous utilisons un test statistique que nous appliquons sur chacun des dix-sept critères.

5.3.6 Application d'un test statistique sur les critères de comparaison

Afin de savoir si une différence que nous observons entre deux environnements est due à des aléas ou a un effet véritable de notre contre-mesure, nous utilisons un test statistique. Nous souhaitons effectuer plusieurs exécutions afin de limiter l'impact des aléas, mais devons limiter leur nombre, car chaque exécution dure cinq minutes suivie de quinze minutes de restauration de la machine. En considérant quinze exécutions dans chacun des deux environnements, un malware prend donc dix heures à être analysé. Comme nous utilisons quatre machines, nous pouvons analyser un peu plus de huit malware par jour.

Or une grande partie des tests statistiques tels que le test de Student [58] implique que les ensembles qui doivent être comparés suivent une loi normale. Nos ensembles contenant seulement quinze valeurs, conclure sur la loi que suit leur distribution ne semble pas possible. Ces tests sont également assez sensibles à la présence de valeurs aberrantes qui peuvent survenir d'après nos tests préliminaires.

[58]: Boudec (2011), *Performance Evaluation of Computer and Communication Systems*

En revanche, d'après ce que nous observons à l'aide des diagrammes en boîte, les distributions sont décalées. Nous essayons de repérer ce changement d'emplacement des distributions à l'aide d'un test non paramétrique appelé test de Wilcoxon-Mann-Whitney [8]. Un test non paramétrique ne se base pas sur le type de loi de la distribution de données contrairement à un test paramétrique tel que le test de Student. Les tests non paramétriques sont très utilisés sur des ensembles de petite taille ou lorsqu'il est difficile de définir la loi suivie par la distribution des données ou encore lorsque des valeurs aberrantes sont présentes [59].

8: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ranksums.html

[59]: Chin et al. (2008), « Chapter 15 - Analysis of Data »

Un test d'hypothèse consiste à rejeter ou non une première hypothèse appelée hypothèse nulle et notée H_0. En cas de rejet d'H_0, nous adoptons une hypothèse complémentaire et opposée appelée hypothèse alternative et notée H_1 [9]. Soit A l'ensemble des n_A observations d'un critère dans un environnement d'analyse, et B l'ensemble des n_B observations de ce même critère dans un environnement standard, les hypothèses que nous utilisons sont les suivantes :

9: https://fr.wikipedia.org/wiki/Test_statistique

— H_0 : Il est crédible de penser que A est homogène à B
— H_1 : A est différente de B

Afin de décider du rejet ou non d'H_0, nous utilisons la valeur-p retournée par le test statistique. Cette valeur indique dans quelle mesure les données sont conformes à l'hypothèse H_0 [60]. Cette valeur permet de répondre à la question : quelle est la probabilité d'obtenir une différence entre les

[60]: Greenland et al. (2016), « Statistical tests, P values, confidence intervals, and power : a guide to misinterpretations »

populations en admettant que seule la chance produise les écarts entre observations ?

Une valeur-p supérieure à 0.05 indique que l'écart par rapport à l'hypothèse testée serait au moins aussi grand que celui observé dans plus 5% du temps si la chance seule était responsable des écarts. Au contraire, une valeur-p inférieure à 0.05 indique que l'écart par rapport à l'hypothèse nulle serait au moins aussi grand que celui observé dans moins de 5% du temps si la chance seule était responsable des écarts.

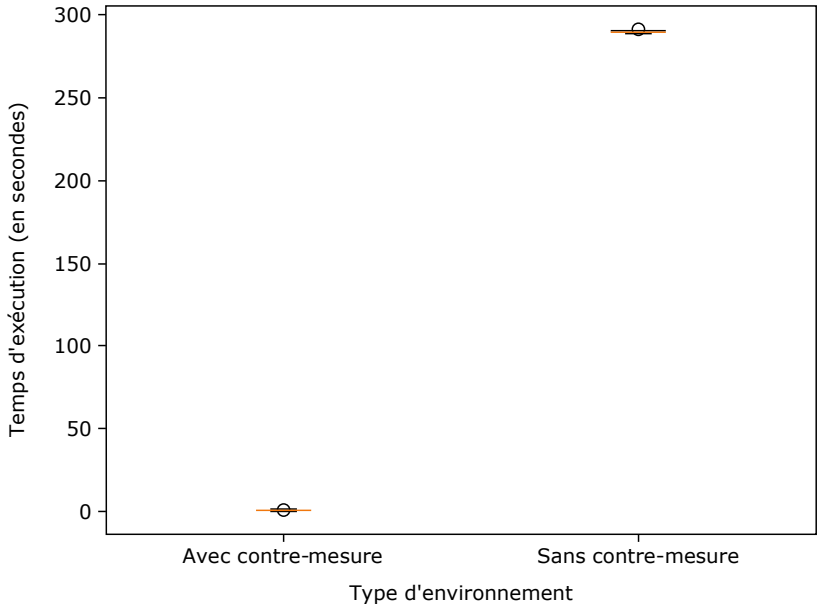
Nous définissons avant l'expérimentation un taux d'erreur α représentant la probabilité de rejeter à tort une hypothèse nulle. La valeur-p est ensuite comparée à α afin de décider de l'hypothèse à rejeter.

— $p < \alpha$: H_0 rejetée au profit d'H_1
— $p \geq \alpha$: H_0 non rejetée

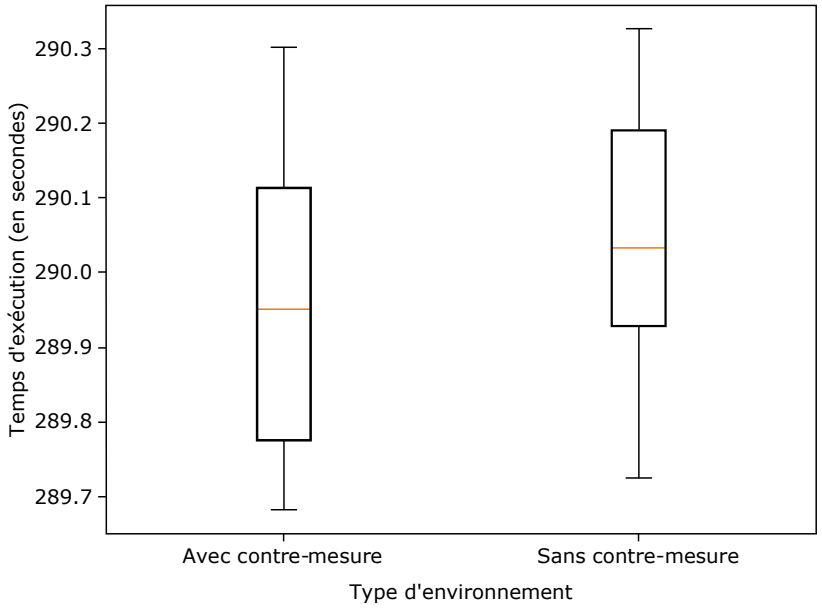
Nous fixons pour nos expérimentations $\alpha = 0.01$ qui est une valeur classique du taux d'erreur et qui permet d'obtenir moins de faux positifs qu'avec $\alpha = 0.05$ également fréquemment utilisée. Pour rejeter l'hypothèse nulle, la valeur-p doit être inférieure à 0.01 ce qui signifie que la chance seule peut créer de tels écarts entre les observations dans pas plus de 1% du temps.

Le test statistique peut réagir à de très petites différences et être rapidement significatif. Si toutes les mesures prises dans un environnement sont supérieures de 0.5% par rapport à l'autre environnement, le test statistique trouvera une différence significative. En effectuant des tests sur le dataset utilisé dans la partie qui présente la contre-mesure, nous avons remarqué que de telles différences pouvaient survenir même sur des logiciels légitimes. En revanche, les malware évasifs qui réagissent à notre contre-mesure avaient des mesures plus hautes ou plus basses d'au moins 20% en présence de la contre-mesure.

Nous considérons un critère comme réagissant à notre contre-mesure si le test statistique retourne une valeur-p inférieure à α et si la médiane du critère mesuré dans l'environnement d'analyse augmente ou baisse d'au moins 20% par rapport celle de l'environnement standard.



(a) Cas 1 : Les temps d'exécution sont significativement différents entre les deux environnements. https://tempo.irisa.fr/malware/438



(b) Cas 2 : Les temps d'exécution sont similaires entre les deux environnements. https://tempo.irisa.fr/malware/1474

Figure 5.5 – Représentation sous forme de diagramme en boîte des temps d'exécution en présence ou non de la contre-mesure.

Exemple d'application sur des mesures de temps d'exécution

Nous observons sur la Figure 5.5 les distributions des mesures des temps d'exécution de deux malware. Pour chacun des deux malware sont représentées la distribution de quinze mesures effectuées dans un environnement d'analyse (1) ainsi que la distribution de quinze autres mesures effectuées dans un environnement standard (2).

Dans le premier cas représenté sur la Figure 5.5a, les quinze mesures en présence de la contre-mesure sont situées autour de 0.15 seconde. Les quinze mesures sans la présence de la contre-mesure sont situées autour de 290 secondes ce qui représente une hausse de plus de 20%. Le test de Wilcoxon-Mann-Whitney retourne une valeur-p de 3e−6

qui est inférieure à α. Grâce à cela, nous pouvons rejeter l'hypothèse nulle H_0 avec une confiance de 99%. Nous considérons alors que les mesures des temps d'exécution effectuées dans l'environnement standard sont significativement différentes par rapport aux mesures des temps d'exécution effectuées dans l'environnement d'analyse. La contre-mesure a donc un impact sur le temps d'exécution de ce malware.

Dans le deuxième cas représenté sur la Figure 5.5b, les quinze mesures en présence de la contre-mesure sont situées autour de 289.95 secondes. Les quinze mesures sans la présence de la contre-mesure sont situées autour de 290.03 secondes et augmentent donc de moins de 20% par rapport à l'autre groupe. Le test de Wilcoxon-Mann-Whitney retourne une valeur-p de 0.44 qui est supérieure à α. Nous ne pouvons pas rejeter l'hypothèse nulle H_0 avec une confiance de 99% avec ce résultat. Nous considérons alors que les mesures des temps d'exécution effectuées dans l'environnement standard ne sont pas significativement différentes par rapport aux mesures des temps d'exécution effectuées dans l'environnement d'analyse. Nous ne pouvons donc pas conclure que la contre-mesure a un impact sur le temps d'exécution de ce malware.

Pour chaque critère de comparaison d'un malware, nous relevons la valeur-p retournée par le test statistique ainsi que la différence en pourcentage de leurs médianes dans chaque environnement. La Table 5.3 liste pour deux malware les valeur-p calculées ainsi que la différence relevée pour les dix-sept critères de comparaison. Ces valeurs entrent ensuite dans le calcul d'un score qui s'il dépasse un seuil, définit si notre contre-mesure a un impact ou non et permet le cas échéant d'attribuer un label évasif.

Table 5.3 – Exemples du calcul du score à partir des valeurs-p et de l'overhead induit par la contre-mesure pour un malware évasif et un second non évasif.

Critères \ Malware ID	n°1339	n°1771
Execution Time	3e−6 : 41.3%	8.4e−3 : -0.05%
Moyennes :		
Kernel Time	3e−6 : -95.12%	0.76 : 2.47%
User Time	3e−6 : -38.37%	0.49 : -0.51%
Page Fault Count	2.1e−5 : -8.26%	0.29 : 0.06%
Working Set Size	3e−6 : -63.06%	6e−6 : 3.50%
Quota Paged Pool Usage	3e−6 : -43.19%	3.1e−5 : 2.92%
Quota Non Paged Pool Usage	3e−6 : -39.64%	3e−6 : 2.33%
Page File Usage	3e−6 : -30.68%	1.5e−5 : 3.63%
CPU Cycle	3e−6 : -72.78%	0.95 : -0.54%
Maximum :		
Kernel Time	3e−6 : -95.24%	0.59 : 0.00%
User Time	3e−6 : -41.46%	0.42 : -4.44%
Page Fault Count	2.1e−5 : -16.70%	0.19 : 0.08%
Working Set Size	3e−6 : -65.74%	3e−6 : 0.60%
Quota Paged Pool Usage	3e−6 : -44.43%	1 : 0.01%
Quota Non Paged Pool Usage	3e−6 : -43.89%	3e−6 : 2.35%
Page File Usage	3e−6 : -56.44%	3.4e−5 : 0.43%
CPU Cycle	3e−6 : -73.52%	0.69 : 0.00%
Score	15/17	0/17

5.3.7 Calcul du score et étiquetage

Pour chaque malware, nous calculons un score sur dix-sept points qui représente le nombre de critères de comparaison impactés par notre contre-mesure et qui ont subi un overhead de plus ou moins 20%. La

Table 5.3 détaille pour deux malware les valeur-p et overhead de chaque critère ainsi que le score final obtenu.

Table 5.4 – Résultat du calcul des scores sur un dataset de test étiqueté à la main.

MD5/Nom	Score	Évasif
4d5ac38437f8eb4c017bc648bb547fac	17	✓
6b43ec6a58836fd41d40e0413eae9b4d	17	✓
5c8c9d0c144a35d2e56281d00bb738a4	0	×
ee88a6abb2548063f1b551a06105f425	17	✓
2ccc21611d5afc0ab67ccea2cf3bb38b	17	✓
512d425d279c1d32b88fe49013812167	17	✓
de3d1414d45e762ca766d88c1384e5f6	17	✓
e51fc4cdd3a950444f9491e15edc5e22	0	×
2d57683027a49d020db648c633aa430a	17	✓
8fdab468bc10dc98e5dc91fde12080e9	17	✓
ee12bbee4c76237f8303c209cc92749d	17	✓
f4d68add66607647bf9cf68cd17ea06a	17	✓
bd5934f9e13ac128b90e5f81916eebd8	17	✓
d3e89fa1273f61ef4dce4c43148d5488	17	✓
e5b2189b8450f5c8fe8af72f9b8b0ad9	17	✓
862c2a3f48f981470dcb77f8295a4fcc	0	×
5253a537b2558719a4d7b9a1fd7a58cf	17	✓
Notepad	0	×
Inkscape	3	×
Nuky	12	✓

Concernant le malware numéro 1339, tous les critères observés sont significativement différents entre les deux environnements. Cependant, les moyennes des mesures du *Page Fault Count* entre les deux environnements subissent un overhead de 8.26% ce qui est inférieur à notre seuil de 20%. Les mêmes observations s'appliquent pour la comparaison des maximums des courbes mesurées dans chaque environnement. Ces deux critères n'entrent pas en compte dans le score qui atteint donc 15/17.

Le malware numéro 1771 obtient des valeurs-p inférieures à α pour huit critères parmi les dix-sept observés. Cependant, aucun de ces critères n'a d'overhead plus grand que 20% ce qui n'ajoute par conséquent aucun point au score qui reste à 0/17.

Nous avons effectué ce calcul de score pour tous les logiciels présentés dans la Table 5.4.

Un seuil est défini en observant les scores obtenus par l'ensemble de malware présenté dans la Section 3.4.2 et qui est également utilisé lors

de notre première contribution. Nous remarquons que tous les malware de ce petit dataset que nous avions étiquetés comme non évasifs ont un score faible. Au contraire, tous les malware étiquetés comme étant évasifs ont un score plus élevé. Nuky, notre ransomware évasif, obtient un score de 12/20 ce qui est le score le plus faible parmi tous les malware évasifs de notre dataset test.

D'après ce dataset test, notre contre-mesure a bien un impact uniquement sur les malware évasifs et faible sur les malware non évasifs et les logiciels légitimes.

Afin d'évaluer l'efficacité de notre contre-mesure, nous calculons un score pour tous les malware de Malpedia et procédons à un étiquetage. Un label évasif est apposé sur ceux qui ont un score supérieur ou égal à douze.

5.3.8 Résultats de l'étiquetage automatique sur les malware provenant de Malpedia

Parmi les 2580 malware provenant de Malpedia, 120 sont étiquetés comme étant évasifs. Cela correspond à environ 4.7% des malware analysés comme représentés sur la Figure 5.6. Ces malware ont été traités par la plateforme pendant dix mois.

Figure 5.6 – Proportion de malware détéctés comme évasifs automatiquement.
Évasifs : 4.7%
Non Évasifs : 95.3%

Sur la Figure 5.7, nous pouvons voir que 1771 malware soit 68.6% du dataset ne semblent pas fortement impactés par la contre-mesure et ont des scores de 0. Environ 26.7% des malware subissent des changements modérés et ont des scores qui s'échelonnent de 1 à 11. Enfin, les malware fortement impactés représentent environ 4.7% avec 102 malware qui possèdent des scores entre 12 et 17.

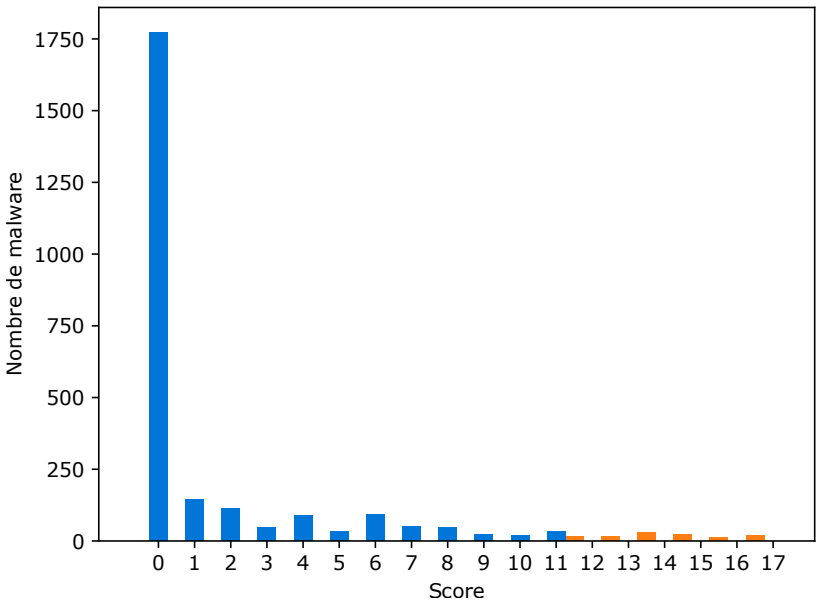


Figure 5.7 – Répartition des malware en fonction des scores obtenus.
Évasifs
Non Évasifs

Certains critères semblent plus fortement impactés par notre contre-mesure que d'autres. Par exemple, 87.5% des malware labélisés évasifs ont le critère *maximums des Page File Usage* qui réagit au changement d'environnement. Le critère *maximums des Quota Non Paged Pool Usage* en revanche ne réagit que pour 72.5% des malware du même groupe.

La Figure 5.8 permet de visualiser pour chaque critère la proportion de malware pour lesquels la présence de la contre-mesure a eu un impact.

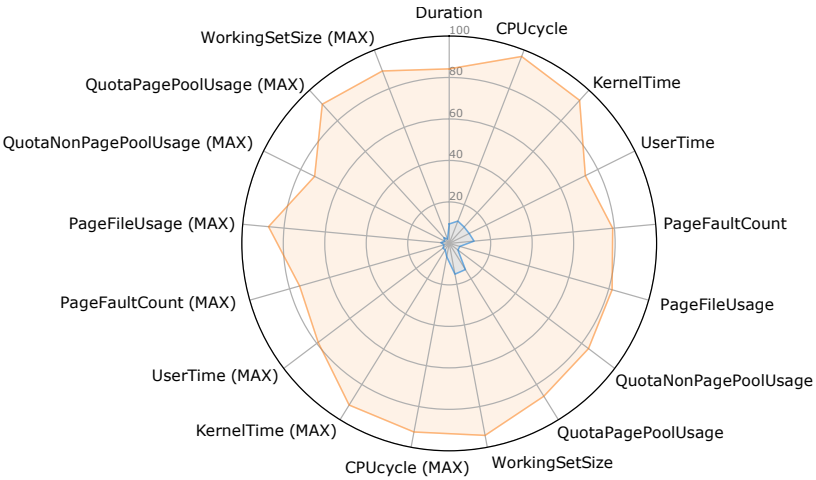


Figure 5.8 – Pourcentage de fois ou chaque critère a subi un impact de la contre-mesure. En orange sont représenté les malware ayant un score supérieur ou égal à 12 et en bleu les malware ayant un score inférieur à 12.
Évasifs
Non Évasifs

Avant de tirer des conclusions de ces résultats, nous les comparons à ceux obtenus par l'étiquetage collaboratif. Chaque malware traité par notre détection automatique est envoyé sur la plateforme TEMPO qui permettra à des experts de les étiqueter afin de créer une vérité terrain qui sert à évaluer les performances de l'étiquetage automatique.

5.4 Étiquetage collaboratif

Dans la section précédente, nous évaluons si la présence de notre contre-mesure censée reproduire un environnement d'analyse a bien un impact sur des malware qui proviennent de Malpedia. Afin d'évaluer si l'étiquetage automatique effectué a bien lieu sur des malware évasifs nous utilisons une vérité terrain construite collaborativement à l'aide d'une plateforme web.

Comme présenté sur la Figure 5.9, une première étape présentée dans la Section 5.4.2 consiste à faire étiqueter les malware de Malpedia par des experts. Une seconde étape présentée dans la Section 5.5 permet ensuite de comparer l'étiquetage automatique avec celui qui est collaboratif et de visualiser son efficacité.

5.4.1 Matériel utilisé pour la plateforme TEMPO

TEMPO est une plateforme web permettant premièrement de visualiser les résultats issus de la détection automatique. Elle permet également aux reviewers d'étiqueter les malware et de les commenter. Enfin, cette plateforme offre la possibilité d'évaluer la détection automatique par rapport aux évaluations humaines.

Les données sont stockées dans une base Structured Query Language (SQL) divisée en deux parties. La première stocke les résultats de la détection automatique et la seconde, les reviews faites par les experts. Une application PHP : Hypertext Preprocessor (PHP) permet ensuite d'accéder à ces données en lecture pour la partie détection et lecture/écriture pour la partie qui stocke les évaluations humaines.

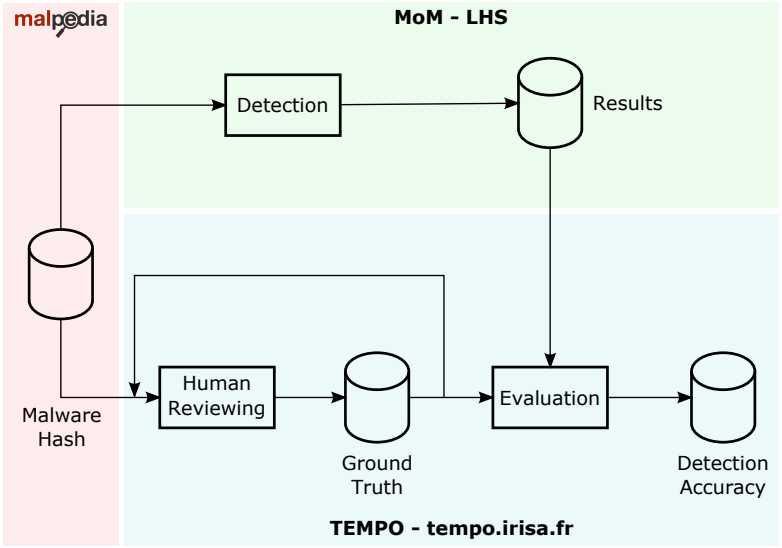


Figure 5.9 – Schema de présentation de la plateforme TEMPO.

5.4.2 Méthode de reviewing

Beaucoup d'articles présents dans notre bibliographie mentionnent avoir étiqueté des malware manuellement afin de créer une vérité terrain. Le but de la partie reviewing de la plateforme TEMPO est de mettre en commun ce travail qui pour le moment est effectué par un grand nombre de personnes, mais n'est pas partagé. Cela est fait également dans le but d'avoir plus de transparence sur la façon dont sont étiquetés les malware.

Chaque utilisateur peut étiqueter n'importe quel malware qui provient de Malpedia. Il est possible de sélectionner les malware en fonction de plusieurs critères tels que le score calculé lors de la détection automatique et qui sont listés sur la Figure 5.10.

En plus du score provenant de notre détection automatique, un second score créé à partir de l'étiquetage humain est calculé. Comme présenté sur la Figure 5.11, chaque utilisateur peut étiqueter les malware comme étant évasifs ou non et a la possibilité d'argumenter son choix anonymement. Il est également demandé de renseigner un niveau de confiance que l'utilisateur a en sa review. Ce niveau de confiance est noté de 0 (peu confiance en sa review) à 10 (grande confiance en sa review). Ce score est ensuite multiplié par 10 pour être affiché sous forme de pourcentage.

La décision de considérer un malware comme évasif se fait en fonction d'un score calculé à partir des reviews.

Pour calculer le score, nous additionnons les indices de confiance des review apposant une étiquette "évasif" et soustrayons les indices de confiance des review apposant une étiquette "non évasif". Cette somme est ensuite divisée par le nombre de reviews.

Soit α l'ensemble des indices de confiance des reviews de taille a étiquetant un malware m comme étant évasif et β l'ensemble des indices de confiance des reviews de taille b étiquetant ce même malware m comme étant non évasif, le score calculé à partir des reviews pour le malware m est défini d'après l'Équation 5.1.

SHA-256	Score	Number Reviews	Experiment score	Date Analyzed	Actions
ed01ebfbc9	30.00%	1	16/17	27/09/2021	show / edit
7bb5bca7c0	10.00%	1	12/17	26/09/2021	show / edit
aba452ab65	10.00%	1	14/17	22/09/2021	show / edit
cea10e8afb	70.00%	1	15/17	19/09/2021	show / edit
eb5bff95e0	10.00%	1	15/17	18/09/2021	show / edit
59cbc88027	60.00%	1	15/17	13/09/2021	show / edit
378aaaeaf2	50.00%	1	14/17	12/09/2021	show / edit
d26161bc38	20.00%	1	14/17	11/09/2021	show / edit
98eb5465c6	-10.00%	1	12/17	08/09/2021	show / edit
de38455675	-20.00%	1	13/17	08/09/2021	show / edit
2d317bccce	50.00%	1	16/17	26/08/2021	show / edit
edb1ff2521	-50.00%	1	17/17	26/08/2021	show / edit
dc2d2f5e3a	40.00%	1	15/17	04/06/2021	show / edit
538c2aef88	-90.00%	1	12/17	10/04/2021	show / edit
33afa2f1d5	70.00%	1	17/17	19/03/2021	show / edit
8ab344a190	80.00%	1	11/17	19/03/2021	show / edit
051463a147	80.00%	1	14/17	10/03/2021	show / edit
02f2cac217	70.00%	1	15/17	27/02/2021	show / edit
94c1b963cc	70.00%	1	13/17	22/02/2021	show / edit
008c4bd6ee	-70.00%	1	0/17	20/02/2021	show / edit
0068a72054	-50.00%	1	0/17	20/02/2021	show / edit
0047843092	-50.00%	1	0/17	19/02/2021	show / edit
00410373a2	-40.00%	1	0/17	19/02/2021	show / edit
003a7905e5	20.00%	1	0/17	19/02/2021	show / edit
003673cf04	-10.00%	1	0/17	19/02/2021	show / edit
0034c4fa21	-30.00%	1	0/17	18/02/2021	show / edit
001268d7fa	-40.00%	1	2/17	18/02/2021	show / edit

Figure 5.10 – Liste des malware provenant de Malpedia.

Id	438
Hash	33afa2f1d53d5279b6fc87ce6834193fdd7e16e4b44e895aae4b9da00be0c502.csv
Virustotal	33afa2f1d5
Date	19/03/2021
Score	70.00%(Evasive)
Number of Human Reviews	1
Reviews	User 1 ✓ Validated ✓ Evasive Confidence: 7 Maze Family https://malpedia.caad.fkie.fraunhofer.de/details/win.maze 2019-08-01 https://github.com/albertzsigovits/malware-notes/blob/master/Ransomware/Maze.md Checks AV software: Select * From AntiVirusProduct via root\SecurityCenter2 Not the same sample. https://www.Nreeye.com/blog/fr-threat-research/2020/05/tactics-techniques-procedures-associated-with-maze-ransomware-incidents.html A104-490 - Host CLI - Defense Evasion, Discovery: Terminate Processes, Malware Analysis Tools A104-491 - Host CLI - Defense Evasion, Persistence: MAZE, Create Target.lnk A104-500 - Host CLI - Discovery, Defense Evasion: Debugger Detection Sample in IOC Edit Delete

Figure 5.11 – Exemple d'une review écrite pour un malware. Ici, l'utilisateur étiquette le malware comme étant évasif et commente sa décision.

$$score_m = \frac{\sum_{i=1}^{a} \alpha_i - \sum_{j=1}^{b} \beta_j}{a+b} \tag{5.1}$$

Un score positif indique que le malware est jugé comme évasif et un score négatif indique que le malware est jugé comme non évasif. Un score égal à 100% qu'il y a un fort consensus entre les reviewers qui considèrent tous avec une grande certitude que le malware est évasif. Un score de −100% indique qu'il y a un fort consensus, mais pour considérer le malware comme non évasif. Enfin, un score de 0% signifie que les reviewers n'ont pas pu trancher et qu'aucun consensus ne s'est dégagé.

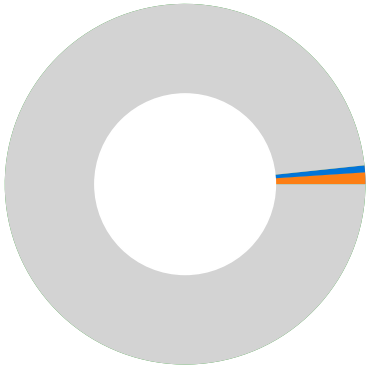


Figure 5.12 – Proportion de malware qui possèdent une review.
Évasifs : 1.2%
Non Évasifs : 0.8%
Pas encore reviewé : 98%

Figure 5.13 – Proportion de malware étiquetés collaborativement comme évasifs.
Évasifs : 62.8%
Non Évasifs : 37.2%

5.4.3 Résultats de l'étiquetage collaboratif

La plateforme n'étant pas encore accessible publiquement, nous avons étiqueté quelques malware nous-mêmes. Pour cela nous lisons les rapports d'analystes présents dans Malpedia et étiquetons le malware comme évasif si des preuves d'évasion sont décrites dans ceux-ci. Pour le moment 43 malware ont été étiquetés ce qui représente 1.6% des malware comme nous pouvons l'observer sur la Figure 5.12.

La Figure 5.13 montre que parmi ces 42 malware, nous en étiquetons collaborativement 27 comme étant évasifs et 16 comme non évasifs.

Nous pouvons également représenter les malware en fonction des scores obtenus avec notre expérimentation mesurant l'impact de notre contre-mesure. Nous pouvons observer sur la Figure 5.14 que les malware que nous avons étiquetés manuellement comme évasifs ont des scores d'au minimum 11 sauf un ayant un score de 0.

Cela nous conforte dans l'idée que notre contre-mesure a bien un fort impact sur les malware évasifs. Nous notons toutefois que celle-ci semble également avoir un impact sur les malware non évasifs qui bien que moins important n'est pas négligeable.

Nous étudions ces cas plus en détail dans la Section 5.5 qui détaille les performances de notre contre-mesure.

Nous pouvons également visualiser pour chaque critère, le pourcentage de malware du groupe évasifs ayant subi un impact. Nous observons sur la Figure 5.15 que les maximums des *Page Fault Count* est impacté pour 63% des malware du groupe évasif et seulement 9% du groupe non évasif.

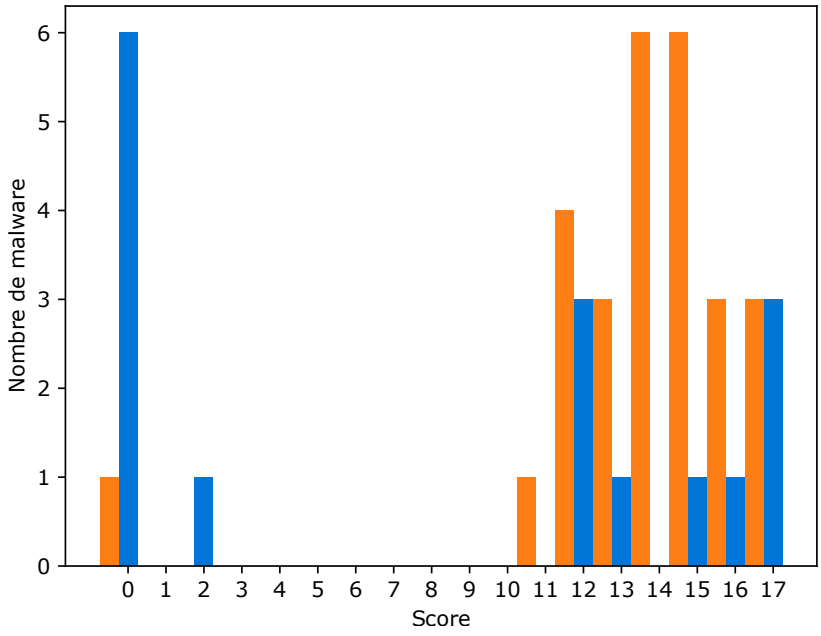


Figure 5.14 – Répartition des malware en fonction des scores obtenus.
Évasifs
Non Évasifs

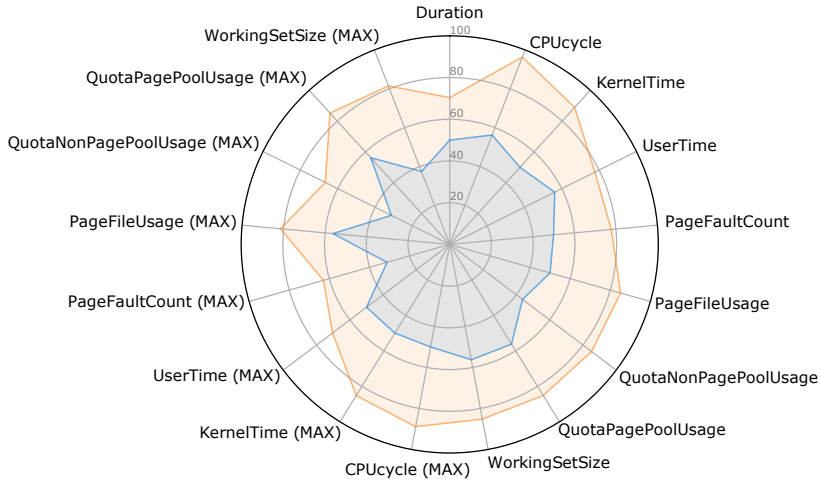


Figure 5.15 – Pourcentage de fois ou chaque critère a subi un impact de la contre-mesure. En orange sont représentés les malware ayant un score supérieur ou égal à 12 et en bleu les malware ayant un score inférieur à 12.
Évasifs
Non Évasifs

5.5 Évaluation de l'efficacité de notre contre-mesure

En comparant l'étiquetage de notre expérimentation avec celui effectué collaborativement, nous souhaitons vérifier si les malware fortement impactés par notre contre-mesure sont bien des malware évasifs ce qui permettrait de valider les Propositions 5.3.1 et 5.3.2.

Nous définissons les termes de faux positifs, faux négatifs, vrais positifs et vrais négatifs d'après la Table 5.5.

Afin d'interpréter les résultats, nous utilisons les notions de **précision** et de **sensibilité** définies respectivement par les Équations 5.2 et 5.3. La précision représente la fraction de malware évasifs détectés automatiquement qui sont véritablement évasifs. La sensibilité représente la fraction de malware étiquetés collaborativement comme évasifs que la méthode de détection automatique est capable de labéliser correctement.

Une détection parfaite a une précision et une sensibilité de 100%. Dans

Table 5.5 – Définition de faux positif, faux négatif, vrai positif et vrai négatif.

	Étiquetage automatique	Étiquetage manuel
Faux Positif	Évasif	Non Évasif
Faux Négatif	Non Évasif	Évasif
Vrai Positif	Évasif	Évasif
Vrai Négatif	Non Évasif	Non Évasif

la pratique de tels scores ne sont pas atteignables et un choix doit être fait pour décider de quel critère privilégier. Dans notre cas, nous souhaitons que tous les malware détectés par notre solution automatique soient véritablement évasifs ce qui correspond à une précision élevée. Augmenter la précision est souvent possible au prix d'une baisse de la sensibilité.

$$precision = \frac{|\{evasifs\_collaboratif\} \cap \{evasifs\_automatique\}|}{|\{evasifs\_automatique\}|} \tag{5.2}$$

$$sensibilite = \frac{|\{evasifs\_collaboratif\} \cap \{evasifs\_automatique\}|}{|\{evasifs\_collaboratif\}|} \tag{5.3}$$

Nous calculons les valeurs de ces deux critères en utilisant les Équations 5.4 et 5.5.

$$precision = \frac{Vrais\_Positifs}{Vrais\_Positifs + Faux\_Positifs} \tag{5.4}$$

$$sensibilite = \frac{Vrais\_Positifs}{Vrais\_Positifs + Faux\_Negatifs} \tag{5.5}$$

Figure 5.16 – Affichage des Faux Positifs, Faux Négatifs, Vrais Positifs et Vrais Négatifs issus de la comparaison de l'étiquetage automatique avec l'étiquetage collaboratif.
Faux Positifs : 20.9%
Faux Négatifs : 4.7%
Vrais Positifs : 58.1%
Vrais Négatifs : 16.3%

Une représentation générale des performances de la détection automatique par rapport à l'étiquetage collaboratif est présentée sur la Figure 5.16. Vingt-cinq malware sont des vrais positifs, neuf des faux positifs, sept des vrais négatifs et deux des faux négatifs.

Nous pouvons donc calculer la précision et la sensibilité et obtenons des valeurs de 73.5% et 92.6% respectivement. La sensibilité nous semble satisfaisante, car notre détection automatique est capable de détecter vingt-cinq parmi les vingt-sept malware qu'il y avait à détecter.

En revanche, la précision est moins bonne, car parmi les trente-quatre malware détectés automatiquement, neuf sont des faux positifs. Comme nous souhaitons une précision la plus élevée possible, nous cherchons des pistes d'amélioration qui permettraient de réduire le nombre de faux positifs.

Une première explication semble apparaître lorsque nous présentons la distribution des performances en fonction du champ *version* qui provient de Malpedia. Ce champ peut contenir une date que nous extrayons si elle existe et qui permet de tracer la Figure 5.17. Nous nous apercevons

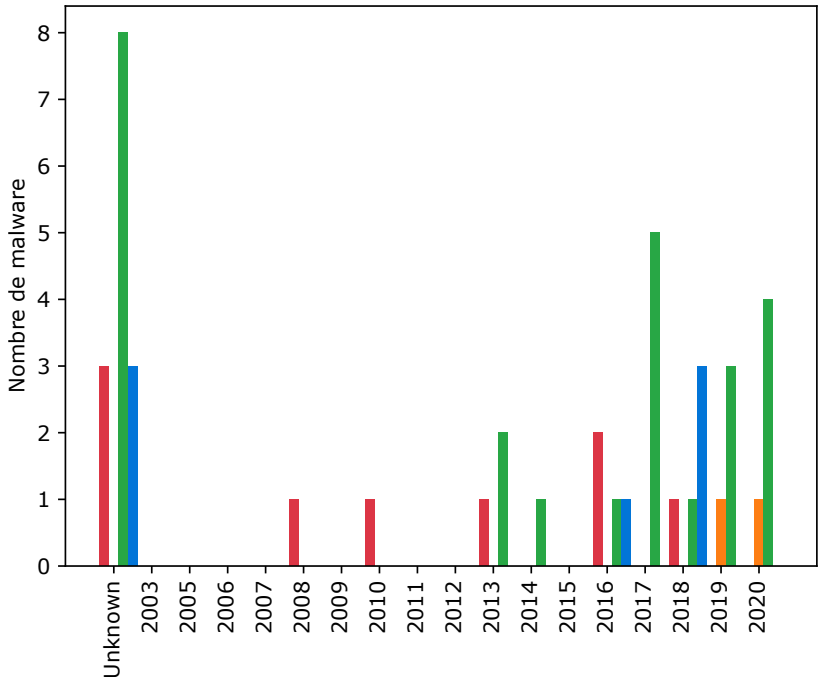


Figure 5.17 – Performances de l'étiquetage automatique par rapport à l'étiquetage collaboratif en fonction de l'année retrouvée dans le champ *version* de chaque malware.
Faux Positifs
Faux Négatifs
Vrais Positifs
Vrais Négatifs

alors que deux faux positifs sont des malware qui datent de 2008 et 2010. Les résultats du monitoring effectué pour ces deux malware montrent que ceux-ci s'exécutent pendant moins d'une seconde avant de s'arrêter, et ce, même dans l'environnement standard. Il se pourrait que ces échantillons soient trop anciens pour continuer leur exécution plus de quelques millisecondes sur le système d'exploitation Windows 10 que nous utilisons. Nous avons trouvé trois malware qui s'exécutent dans un temps aussi court parmi les neuf faux positifs. Lors de la prochaine itération de notre expérimentation, nous pourrions ajouter une condition qui vérifie que le temps d'exécution dure au moins quelques secondes avant de labéliser automatiquement les malware.

Trois autres malware semblent crasher systématiquement après le démarrage dès que la contre-mesure est présente. Pour l'instant, nos investigations ne permettent pas de mettre en évidence une cause commune de ces arrêts. Il semble qu'aucun de nos vrais positifs ne présente un temps d'exécution inférieur à une seconde lorsque la contre-mesure est présente. Ce problème serait également corrigé en vérifiant que la durée des exécutions est plus longue que quelques secondes avant de labéliser les malware.

Enfin, les trois derniers malware ne semblent pas évasifs, mais sont soit empaquetés, soit présentent un comportement qui cherche à identifier l'environnement d'exécution sans pour autant s'évader ensuite. Pour chacun de ces malware, nous manquons de documentations qui décrivent en détail leur comportement et la rétro-ingénierie de ces échantillons obfusqués est très chronophage. Les recherches que nous avons effectuées ne permettent pour l'instant pas d'identifier de comportement véritablement évasif de ces malware ni d'expliquer pourquoi la présence de notre contre-mesure provoque un tel changement de comportement.

La Figure 5.18 permet d'évaluer l'efficacité des critères en visualisant par exemple la proportion de faux positifs et de vrais positifs pour lesquels ils sont significativement différents. Un critère optimal est significativement

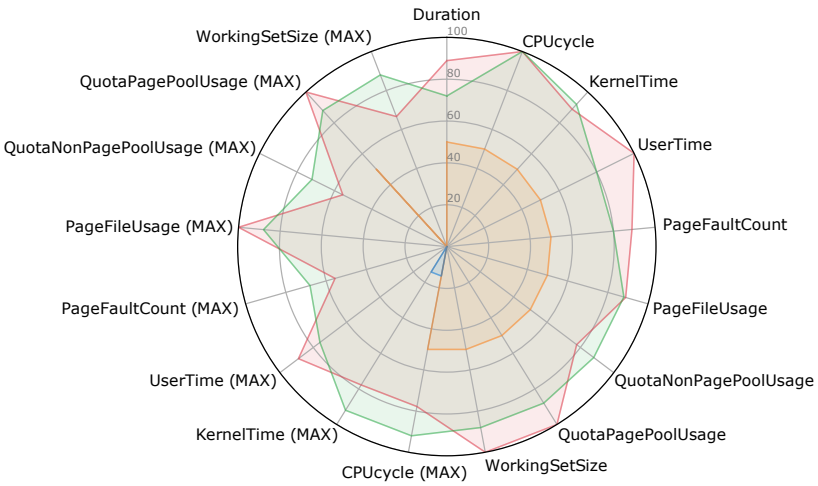


Figure 5.18 – Visualisation pour chaque critère du pourcentage de malware pour lesquels ce critère est significativement différent entre l'environnement standard et celui d'analyse.
Faux Positifs
Faux Négatifs
Vrais Positifs
Vrais Négatifs

différent pour 100% des vrais positifs et pour 0% des faux positifs. Nous pouvons voir par exemple que les maximums des mesures du *WorkingSetSize* sont significativement différents pour 88% des vrais positifs et 66.7% des faux positifs. Nous attendons cependant d'étiqueter plus de malware pour pouvoir conclure de l'efficacité de chacun des critères.

5.6 Discussion

Nous utilisons les résultats d'un monitoring afin de détecter une différence de comportement entre deux exécutions d'un programme. Il est possible d'utiliser un autre moyen de comparaison tel que l'extraction des Master File Table (MFT). Nous avons effectué des tests créant une image du disque à l'aide de clonezilla après chaque exécution et en extrayant une timeline de la MFT à l'aide de sleuthkit [10]. Les deux timeline sont ensuite comparées à la recherche de divergences et nous arrivons à trouver des différences de comportements sur notre dataset test. Toutefois, cette méthode demande beaucoup de temps pour une seule analyse, car il faut effectuer une image du disque qui prend environ dix minutes puis extraire les MFT pendant cinq minutes.

10: https://www.sleuthkit.org

Une autre solution est de manipuler la table SSDT qui contient les adresses des appels système de Windows. Cela permet par exemple de modifier les fonctions présentes par les nôtres et de leur faire écrire une ligne dans un fichier de log à chaque appel. Nous obtenons ainsi une trace d'appels système pour chaque exécution. La manipulation de la SSDT n'est plus possible depuis la mise en place du KPP [11]. Cependant des projets tels que gbhv [12] permettent de créer un hyperviseur qui prend alors le contrôle de l'hôte sur lequel il s'exécute et permet de créer des hooks de fonctions du noyau. Ceci est possible, car l'hyperviseur s'exécute alors à un niveau plus privilégié que le système d'exploitation hôte. Cela a pour avantage d'être invisible des malwares s'exécutant au niveau utilisateur au contraire de notre contre-mesure par exemple. Nous avions envisagé d'implémenter notre contre-mesure à l'aide de cet outil ou d'enregistrer une trace d'appels système pour chaque exécution. Cependant, l'outil est encore très récent et ne permet pas encore de créer plusieurs instrumentations de manière stable.

11: https://en.wikipedia.org/wiki/Kernel_Patch_Protection

12: https://github.com/Gbps/gbhv

La plateforme permet ici de créer un dataset de malware évasifs, mais pourrait être utilisée pour créer un dataset possédant d'autres caractéristiques. Une solution de détection de ransomware peut être confrontée à un étiquetage qui au lieu d'évaluer si un malware est évasif ou non, décrit si le malware est un ransomware ou non. Actuellement, les malware sont exécutés puis ne sont pas manipulés. Nous envisageons d'utiliser une solution de fuzzing d'interface graphique pour pouvoir déclencher plus de malware. Il est possible d'évaluer l'impact de cette solution de fuzzing d'interface graphique à l'aide de notre plateforme. Dans ce cas nous pouvons créer un environnement équipé de la solution de fuzzing et un second sans. Le même script de monitoring que celui décrit dans la Section 5.3.3 peut être utilisé. Les résultats peuvent ensuite être visualisés pour voir si la solution de fuzzing a eu un impact ou non sur les logiciels testés.

5.7 Conclusion

Notre méthode d'analyse automatique a traité 2181 malware parmi lesquels 102 sont étiquetés comme évasifs. La comparaison de 34 de ces malware avec l'étiquetage collaboratif qui sert de vérité terrain permet d'évaluer les performances de notre méthode de détection. 73.5% des malware étiquetés automatiquement comme évasifs le sont véritablement, et 92.6% des malware étiquetés collaborativement l'ont également été par la détection automatique.

Notre dataset a été créé dans le but de pouvoir vérifier et réduire la présence de biais qui peuvent être retrouvés dans les datasets existants.

L'effet expérimentateur est réduit, car l'étiquetage est effectué collaborativement et non plus par la personne qui effectue également l'expérimentation. En effet, les scores de l'étiquetage automatique sont cachés aux reviewers qui ne peuvent ainsi pas être influencés. De plus, il est possible de vérifier les choix qui ont poussé un humain à étiqueter un malware d'une certaine manière. Il est donc plus difficile pour une personne d'influencer les résultats de son expérimentation en étiquetant le dataset d'une manière biaisée.

Grâce aux différentes visualisations, nous pouvons vérifier qu'un dataset n'a pas été construit en sélectionnant des malware de sorte à améliorer les résultats des expérimentations. Un dataset composé uniquement d'anciens malware serait immédiatement repéré, car les informations sont publiques et observables par tous. Bien que toujours présents, ces biais de sélection sont plus visibles et un expérimentateur peut en prendre connaissance et effectuer ses expérimentations en connaissance de cause.

Le fait d'avoir un étiquetage humain régulier permet d'évaluer fréquemment les performances de la méthode de détection automatique. Le dataset est également alimenté régulièrement par de nouveaux malware afin de garantir un dataset d'actualité. Cela permet d'éviter d'utiliser un dataset vieillissant passé d'expérimentation en expérimentation et ne gardant que les échantillons ayant eu des résultats positifs, ce qui limite les biais du survivant.

Chaque personne utilisant le dataset de malware évasif peut communiquer la période utilisée et les paramètres qui ont été renseignés pour le filtrer. Par exemple, si une expérimentation est basée sur un dataset contenant des malware analysés en 2020 et étiquetés à la main, il suffit pour l'auteur de l'indiquer pour que tout le monde puisse visualiser ce dataset et vérifier sa composition. Il est alors possible pour une autre personne de récupérer le même dataset puis de créer sa propre expérimentation et de comparer les résultats.

Aucun dataset dédié aux malware évasifs n'existait jusqu'à présent. Il est maintenant simple pour un utilisateur souhaitant mettre en place une expérimentation sur des malware évasifs de le faire. Il suffit de se rendre sur la plateforme TEMPO et de filtrer la liste pour n'afficher que les échantillons évasifs. Une liste des hash peut ensuite être exportée afin de télécharger les exécutables depuis Malpedia.

Ce premier dataset de malware évasifs répond donc aux critères que nous nous sommes fixés et pourra être amélioré en modifiant par exemple la solution de détection automatique, la méthode de comparaison des exécutions ou la plateforme TEMPO pour assurer des expérimentations en double aveugle.

Epilogue

6 Conclusion

Dans cette thèse nous décrivons trois contributions qui visent à répondre au problème grandissant des malware évasifs. Notre malware évasif ainsi que la contre-mesure ont été présentés à la conférence SECRYPT 2020 [2] et C&ESAR 2020 [3]. Notre méthode de création d'un dataset et la plateforme TEMPO ont également pour but d'être publiées.

[2]: Herzog et al. (2020), « Evasive Windows Malware : Impact on Antiviruses and Possible Countermeasures »

[3]: Herzog et al. (2020), « Malware Windows Evasifs : Impact sur les Antivirus et Possible Contre-mesure »

6.1 Nuky

Premièrement, nous créons Nuky, un malware évasif de démonstration que nous utilisons pour vérifier la facilité d'implémentation des techniques d'évasion ainsi que leur efficacité sur des antivirus. Les techniques utilisées sont basiques et leurs codes sont présents dans des répertoires publics ce qui rend aisée leur implémentation dans des malware.

L'efficacité de ces techniques d'évasion est également évaluée contre un panel d'antivirus. Nuky est capable de repérer la présence des sept antivirus mais également d'identifier leur nom. Cela prouve que ces techniques d'évasion basiques permettent à un malware évasif de repérer précisément un antivirus puis d'adapter son comportement évasif. Les malware qui utilisent ce genre de techniques représentent donc bel et bien un danger que nos deux contributions suivantes visent à contrer.

6.2 Contre-mesure

Nous répondons tout d'abord à l'absence de moyens de protection contre les malware évasifs. Peu d'informations concernant le comportement adopté par les malware après avoir détecté un outil d'analyse sont disponibles. Nous considérons cependant que ceux-ci optent soit pour un comportement inoffensif, soit arrêtent leurs exécutions. Dans les deux cas, ces comportements ne représentent plus de danger pour la machine qui les exécute.

Notre contre-mesure consiste à forcer les malware évasifs à adopter ces comportements inoffensifs en simulant la présence d'outils d'analyse.

Plusieurs articles présents dans notre bibliographie détaillent le fonctionnement de techniques d'évasion. Celles-ci consistent à chercher des artéfacts laissés par les outils d'analyse lors de leur installation ou exécution et qui trahissent leur présence. Notre contre-mesure consiste à instrumenter l'API Windows de sorte à reproduire artificiellement ces artéfacts et ainsi faire s'évader les malware qui utilisent ce genre de technique d'évasion.

Une expérimentation permet d'évaluer les performances de cette contre-mesure sur dix-huit malware évasifs récoltés dans la nature. Parmi ceux-ci, quatorze changent de comportement en présence de la contre-mesure. La présence de notre contre-mesure ajoute un overhead négligeable sur des logiciels légitimes et une augmentation du temps d'exécution de

14.62% pour Al-Khaser, un logiciel évasif expérimental, mais ne semble pas avoir d'impact sur les logiciels légitimes.

Cette contribution s'est heurtée à l'absence de dataset de malware évasifs ce qui nous a obligés à tester notre contre-mesure sur un faible nombre de malware. Ce manque de dataset de malware évasif est mentionné dans plusieurs articles de notre bibliographie et semble être un problème récurrent.

6.3 Dataset de malware évasifs

[1]: Viet Triem Tong et al. (2019), « Isolating malicious code in Android malware in the wild »

Les six premiers mois de cette thèse ont permis d'achever des travaux concernant l'extraction de code malveillant de malware Android [1]. Nous avons également eu des difficultés à évaluer les performances de notre extraction, car aucun dataset ne fournit d'information aussi précise que la liste des méthodes malveillantes contenues dans les applications. L'évaluation de notre solution s'est faite à l'aide de malware que nous avions analysés manuellement. Cette étape d'analyse manuelle est commune à beaucoup d'articles présents dans notre bibliographie. Ces efforts sont produits séparément par les équipes de recherche et les datasets qui en résultent sont rarement partagés.

Notre seconde contribution répond à ce besoin en fournissant un dataset de malware évasifs. Ce dataset est étiqueté premièrement collaborativement afin d'obtenir une vérité terrain. En effet, nous avons noté que plusieurs auteurs d'articles mentionnent avoir étiqueté plusieurs malware eux-mêmes. Pour mettre ces efforts en commun, nous avons créé TEMPO, une plateforme web accessible aux chercheurs qui souhaitent créer ou étiqueter des malware.

Un second étiquetage automatique est utilisé et évalué en fonction de l'étiquetage collaboratif. Cela permet d'obtenir une quantité plus importante de malware évasifs tout en ayant conscience des performances de cette détection en temps réel.

Nous attachons une grande importance à ce que les données soient publiquement accessibles. Ainsi, il est possible pour n'importe quelle personne de vérifier les raisons de l'étiquetage de certains malware. Bien que le dataset ne soit pas exempt de biais, ceux-ci peuvent être plus simplement repérés grâce à ce partage d'informations.

Ce premier dataset de malware évasifs permet donc à quiconque le souhaite, de mettre en place une expérimentation simplement ou de reproduire une expérimentation.

6.4 Travaux futurs

La méthode de détection de malware évasifs que nous utilisons actuellement peut-être améliorée comme nous l'indiquons dans la Section 5.5. Pour cela, il suffit de réanalyser les résultats des expérimentations et d'étiqueter à nouveau les malware. Le temps pour analyser les résultats du monitoring est très court comparé au temps nécessaire pour effectuer les mesures. Le nouvel étiquetage de 2000 malware peut s'effectuer

en quelques heures et nous pouvons donc envisager de tester d'autres méthodes d'étiquetage à partir des données de monitoring.

La plateforme TEMPO ne permet pour l'instant d'évaluer qu'une seule méthode de détection par rapport à l'étiquetage collaboratif. Une piste d'amélioration possible serait de pouvoir comparer les performances de deux méthodes de détection différentes, toujours par rapport à l'étiquetage collaboratif.

Pour l'instant seuls les malware qui proviennent de Malpedia sont analysés et nous avons l'intention de l'alimenter avec d'autres sources de malware. Nous vérifierons en revanche que ces datasets sont accessibles aux chercheurs publiquement ou sur invitation et qu'ils contiennent bien exclusivement de réels malware.

Enfin, lorsqu'un utilisateur écrit une review, celui-ci ne peut choisir d'étiqueter le malware que comme étant évasif ou non. Nous souhaitons offrir la possibilité d'étiqueter les malware plus finement en fonction des techniques d'évasion utilisées par exemple.

Annexes

Code des instrumentations

A

```
#include <stdio.h>
#include <windows.h>
#include "detours.h"
#include <TlHelp32.h>
#include <tchar.h>
#include <shlwapi.h>


static BOOL(WINAPI * TrueDebuggerPresent)(void) = IsDebuggerPresent;
static BOOL(WINAPI * TrueCursorPos)(LPPOINT lpPoint) = GetCursorPos;
static LSTATUS(WINAPI * TrueOpenKeyA)(HKEY hKey, LPCSTR lpSubKey, DWORD
     ulOptions, REGSAM samDesired, PHKEY phkResult) = RegOpenKeyExA;
static LSTATUS(WINAPI * TrueOpenKeyW)(HKEY hKey, LPCWSTR lpSubKey, DWORD
     ulOptions, REGSAM samDesired, PHKEY phkResult) = RegOpenKeyExW;
static HANDLE(WINAPI * TrueCreateFileA)(LPCSTR lpFileName, DWORD
     dwDesiredAccess, DWORD dwShareMode, LPSECURITY_ATTRIBUTES
     lpSecurityAttributes, DWORD dwCreationDisposition, DWORD
     dwFlagsAndAttributes, HANDLE hTemplateFile) = CreateFileA;
static HANDLE(WINAPI * TrueCreateFileW)(LPCWSTR lpFileName, DWORD
     dwDesiredAccess, DWORD dwShareMode, LPSECURITY_ATTRIBUTES
     lpSecurityAttributes, DWORD dwCreationDisposition, DWORD
     dwFlagsAndAttributes, HANDLE hTemplateFile) = CreateFileW;
static HMODULE(WINAPI * TrueModuleHandleA)(LPCSTR lpModuleName) =
     GetModuleHandleA;
static HMODULE(WINAPI * TrueModuleHandleW)(LPCWSTR lpModuleName) =
     GetModuleHandleW;
static LSTATUS(WINAPI * TrueQueryValueA)(HKEY hKey, LPCSTR lpValueName,
     LPDWORD lpReserved, LPDWORD lpType, LPBYTE lpData, LPDWORD lpcbData)
     = RegQueryValueExA;
static LSTATUS(WINAPI * TrueQueryValueW)(HKEY hKey, LPCWSTR lpValueName,
     LPDWORD lpReserved, LPDWORD lpType, LPBYTE lpData, LPDWORD lpcbData)
     = RegQueryValueExW;
static DWORD(WINAPI * TrueFileAttributesA)(LPCSTR lpFileName) =
     GetFileAttributesA;
static DWORD(WINAPI * TrueFileAttributesW)(LPCWSTR lpFileName) =
     GetFileAttributesW;
static HANDLE(WINAPI * TrueCreateToolhelp32Snapshot)(DWORD dwFlags, DWORD
     th32ProcessID) = CreateToolhelp32Snapshot;


// Other local variables (GetPosCursor)
POINT point;
LPPOINT firstPoint = &point;

// Nouveau comportement d'isDebuggerPresent
BOOL WINAPI FakeDebuggerPresent() {
    return TRUE;
}

// Detour function for the GetPosCursor
BOOL WINAPI DoublePoint(LPPOINT lpPoint) {
    BOOL t;
    if (firstPoint->x == 0 && firstPoint->y == 0) {
        t = TrueCursorPos(firstPoint);
        if (t) {
            lpPoint->x = firstPoint->x;
            lpPoint->y = firstPoint->y;
        }
    }
    else {
        lpPoint->x = firstPoint->x;
```

Listing A.1: Code complet de la contre-mesure issue de notre première contribution.

```
        lpPoint->y = firstPoint->y;
        t = TrueCursorPos(firstPoint);
    }
    return t;
}

// Detour function for RegOpenKeyExA
LSTATUS WINAPI FalseOpenKeyA(HKEY hKey, LPCSTR lpSubKey, DWORD ulOptions,
     REGSAM samDesired, PHKEY phkResult) {
    LPCSTR szKeys[] = {
            "HARDWARE\\DEVICEMAP\\Scsi\\Scsi Port 0\\Scsi Bus 0\\Target Id
      0\\Logical Unit Id 0",
            "HARDWARE\\DEVICEMAP\\Scsi\\Scsi Port 1\\Scsi Bus 0\\Target Id
      0\\Logical Unit Id 0",
            "HARDWARE\\DEVICEMAP\\Scsi\\Scsi Port 2\\Scsi Bus 0\\Target Id
      0\\Logical Unit Id 0",
            "HARDWARE\\Description\\System",
            "HARDWARE\\ACPI\\DSDT\\VBOX__",
            "HARDWARE\\ACPI\\FADT\\VBOX__",
            "HARDWARE\\ACPI\\RSDT\\VBOX__",
            "SOFTWARE\\Oracle\\VirtualBox Guest Additions",
            "SYSTEM\\ControlSet001\\Services\\VBoxGuest",
            "SYSTEM\\ControlSet001\\Services\\VBoxMouse",
            "SYSTEM\\ControlSet001\\Services\\VBoxService",
            "SYSTEM\\ControlSet001\\Services\\VBoxSF",
            "SYSTEM\\ControlSet001\\Services\\VBoxVideo",
            "SYSTEM\\ControlSet001\\Control\\SystemInformation",
            "SOFTWARE\\Wine",
            "SOFTWARE\\VMware, Inc.\\VMware Tools",
            "SOFTWARE\\Microsoft\\Virtual Machine\\Guest\\Parameters"
    };
    WORD dwlength = sizeof(szKeys) / sizeof(szKeys[0]);

    for (int i = 0; i < dwlength; i++) {
        if(lpSubKey){
            if (lstrcmp(szKeys[i], lpSubKey) == 0) {
                TrueOpenKeyA(hKey, lpSubKey, ulOptions, samDesired,
     phkResult);
                return ERROR_SUCCESS;
            }
        }
    }
    return TrueOpenKeyA(hKey, lpSubKey, ulOptions, samDesired, phkResult);
}

// Detour function for RegOpenKeyExW
LSTATUS WINAPI FalseOpenKeyW(HKEY hKey, LPCWSTR lpSubKey, DWORD ulOptions,
      REGSAM samDesired, PHKEY phkResult) {
    LPCWSTR szKeys[] = {
        L"HARDWARE\\DEVICEMAP\\Scsi\\Scsi Port 0\\Scsi Bus 0\\Target Id
     0\\Logical Unit Id 0",
        L"HARDWARE\\DEVICEMAP\\Scsi\\Scsi Port 1\\Scsi Bus 0\\Target Id
     0\\Logical Unit Id 0",
        L"HARDWARE\\DEVICEMAP\\Scsi\\Scsi Port 2\\Scsi Bus 0\\Target Id
     0\\Logical Unit Id 0",
        L"HARDWARE\\Description\\System",
        L"HARDWARE\\ACPI\\DSDT\\VBOX__",
        L"HARDWARE\\ACPI\\FADT\\VBOX__",
        L"HARDWARE\\ACPI\\RSDT\\VBOX__",
        L"SOFTWARE\\Oracle\\VirtualBox Guest Additions",
        L"SYSTEM\\ControlSet001\\Services\\VBoxGuest",
        L"SYSTEM\\ControlSet001\\Services\\VBoxMouse",
        L"SYSTEM\\ControlSet001\\Services\\VBoxService",
        L"SYSTEM\\ControlSet001\\Services\\VBoxSF",
        L"SYSTEM\\ControlSet001\\Services\\VBoxVideo",
        L"SYSTEM\\ControlSet001\\Control\\SystemInformation",
        L"SOFTWARE\\Wine",
```

```
        L"SOFTWARE\\VMware, Inc.\\VMware Tools",
        L"SOFTWARE\\Microsoft\\Virtual Machine\\Guest\\Parameters"
    };
    WORD dwlength = sizeof(szKeys) / sizeof(szKeys[0]);

    for (int i = 0; i < dwlength; i++) {
        if(lpSubKey){
            if (wcscmp(szKeys[i], lpSubKey) == 0) {
                TrueOpenKeyW(hKey, lpSubKey, ulOptions, samDesired,
     phkResult);
                return ERROR_SUCCESS;
            }
        }
    }
    return TrueOpenKeyW(hKey, lpSubKey, ulOptions, samDesired, phkResult);
}

// Detour function for CreateFile
HANDLE WINAPI FalseCreateFileA(LPCSTR lpFileName, DWORD dwDesiredAccess,
     DWORD dwShareMode, LPSECURITY_ATTRIBUTES lpSecurityAttributes, DWORD
     dwCreationDisposition, DWORD dwFlagsAndAttributes, HANDLE
     hTemplateFile) {
    LPCSTR devices[] = {
        "\\\\.\\VBoxMiniRdrDN",
        "\\\\.\\VBoxGuest",
        "\\\\.\\pipe\\VBoxMiniRdDN",
        "\\\\.\\VBoxTrayIPC",
        "\\\\.\\pipe\\VBoxTrayIPC",
        "\\\\.\\HGFS",
        "\\\\.\\vmci"
    };
    WORD iLength = sizeof(devices) / sizeof(devices[0]);
    for (int i = 0; i < iLength; i++) {
        if(lpFileName){
            if (lstrcmp(lpFileName, devices[i]) == 0) {
                return ERROR_SUCCESS;
            }
        }
    }
    return TrueCreateFileA(lpFileName, dwDesiredAccess, dwShareMode,
     lpSecurityAttributes, dwCreationDisposition, dwFlagsAndAttributes,
     hTemplateFile);
}

// Detour function for CreateFile
HANDLE WINAPI FalseCreateFileW(LPCWSTR lpFileName, DWORD dwDesiredAccess,
     DWORD dwShareMode, LPSECURITY_ATTRIBUTES lpSecurityAttributes, DWORD
     dwCreationDisposition, DWORD dwFlagsAndAttributes, HANDLE
     hTemplateFile) {
    LPCWSTR devices[] = {
        L"\\\\.\\VBoxMiniRdrDN",
        L"\\\\.\\VBoxGuest",
        L"\\\\.\\pipe\\VBoxMiniRdDN",
        L"\\\\.\\VBoxTrayIPC",
        L"\\\\.\\pipe\\VBoxTrayIPC",
        L"\\\\.\\HGFS",
        L"\\\\.\\vmci"
    };
    WORD iLength = sizeof(devices) / sizeof(devices[0]);
    for (int i = 0; i < iLength; i++) {
        if(lpFileName){
            if (wcscmp(lpFileName, devices[i]) == 0) {
                return ERROR_SUCCESS;
            }
        }
    }
    return TrueCreateFileW(lpFileName, dwDesiredAccess, dwShareMode,
```

```
    lpSecurityAttributes, dwCreationDisposition, dwFlagsAndAttributes,
    hTemplateFile);
}

// Detour function for GetModuleHandle
HMODULE WINAPI FalseModuleHandleA(LPCSTR lpModuleName) {
    LPCSTR szDlls[] = {
        "avghookx.dll",
        "avghooka.dll",
        "snxhk.dll",
        "sbiedll.dll",
        "dbghelp.dll",
        "api_log.dll",
        "dir_watch.dll",
        "pstorec.dll",
        "vmcheck.dll",
        "wpespy.dll",
        "cmdvrt64.dll",
        "cmdvrt32.dll",
        "sbiedll.dll"
    };
    WORD dwlength = sizeof(szDlls) / sizeof(szDlls[0]);
    for (int i = 0; i < dwlength; i++) {
        if(lpModuleName){
            if (lstrcmp(szDlls[i], lpModuleName) == 0) {
                return TrueModuleHandleA("user32.dll");
            }
        }
    }
    return TrueModuleHandleA(lpModuleName);
}

// Detour function for GetModuleHandle
HMODULE WINAPI FalseModuleHandleW(LPCWSTR lpModuleName) {
    LPCWSTR szDlls[] = {
        L"avghookx.dll",
        L"avghooka.dll",
        L"snxhk.dll",
        L"sbiedll.dll",
        L"dbghelp.dll",
        L"api_log.dll",
        L"dir_watch.dll",
        L"pstorec.dll",
        L"vmcheck.dll",
        L"wpespy.dll",
        L"cmdvrt64.dll",
        L"cmdvrt32.dll",
        L"sbiedll.dll"
    };
    WORD dwlength = sizeof(szDlls) / sizeof(szDlls[0]);
    for (int i = 0; i < dwlength; i++) {
        if(lpModuleName){
            if (wcscmp(szDlls[i], lpModuleName) == 0) {
                return TrueModuleHandleW(L"user32.dll");
            }
        }
    }
    return TrueModuleHandleW(lpModuleName);
}

// Detour function for RegQueryValueEx
LSTATUS WINAPI FalseQueryValueA(HKEY hKey, LPCSTR lpValueName, LPDWORD
    lpReserved, LPDWORD lpType, LPBYTE lpData, LPDWORD lpcbData) {
    LPCSTR szEntries[][2] = {
        { "Identifier", "VBOX" },
    { "SystemBiosVersion", "VBOX" },
    { "VideoBiosVersion", "VIRTUALBOX" },
```

```
    { "SystemBiosDate", "06/23/99" },
    { "SystemManufacturer", "VMWARE" },
    { "SystemProductName", "VMWARE" }
    };
    WORD dwLength = sizeof(szEntries) / sizeof(szEntries[0]);
    //HKEY hkResult = FALSE;
    for (int i = 0; i < dwLength; i++) {
        if(lpValueName){
            if (lstrcmp(szEntries[i][0], lpValueName) == 0) {
                memcpy(lpData, szEntries[i][1], *lpcbData);
                return ERROR_SUCCESS;
            }
        }
    }
    return TrueQueryValueA(hKey, lpValueName, lpReserved, lpType, lpData,
     lpcbData);
}

// Detour function for RegQueryValueEx
LSTATUS WINAPI FalseQueryValueW(HKEY hKey, LPCWSTR lpValueName, LPDWORD
     lpReserved, LPDWORD lpType, LPBYTE lpData, LPDWORD lpcbData) {
    LPCWSTR szEntries[][2] = {
        { L"Identifier", L"VBOX" },
    { L"SystemBiosVersion", L"VBOX" },
    { L"VideoBiosVersion", L"VIRTUALBOX" },
    { L"SystemBiosDate", L"06/23/99" },
    { L"SystemManufacturer", L"VMWARE" },
    { L"SystemProductName", L"VMWARE" }
    };

    WORD dwLength = sizeof(szEntries) / sizeof(szEntries[0]);
    dwLength;
    for (int i = 0; i < dwLength; i++) {
        //Certains programmes appellent QueryValue avec lpValueName ==
     NULL
        if(lpValueName){
            if (wcscmp(szEntries[i][0], lpValueName) == 0) {
                memcpy(lpData, szEntries[i][1], *lpcbData);
                return ERROR_SUCCESS;
            }
        }
    }

    return TrueQueryValueW(hKey,(LPCWSTR) lpValueName, lpReserved, lpType,
      lpData, lpcbData);
}

// Detour function for detecting files and directories
DWORD WINAPI FalseFileAttributesA(LPCSTR lpFileName) {
    LPCSTR szPaths[] = {
        "system32\\drivers\\VBoxMouse.sys",
        "system32\\drivers\\VBoxGuest.sys",
        "system32\\drivers\\VBoxSF.sys",
        "system32\\drivers\\VBoxVideo.sys",
        "system32\\vboxdisp.dll",
        "system32\\vboxhook.dll",
        "system32\\vboxmrxnp.dll",
        "system32\\vboxogl.dll",
        "system32\\vboxoglarrayspu.dll",
        "system32\\vboxoglcrutil.dll",
        "system32\\vboxoglerrorspu.dll",
        "system32\\vboxoglfeedbackspu.dll",
        "system32\\vboxoglpackspu.dll",
        "system32\\vboxoglpassthroughspu.dll",
        "system32\\vboxservice.exe",
        "system32\\vboxtray.exe",
        "system32\\VBoxControl.exe",
```

```
        "C:\\Windows\\system32\\drivers\\vmmouse.sys",
        "system32\\drivers\\vmhgfs.sys",
        "system32\\drivers\\vm3dmp.sys",
        "system32\\drivers\\vmci.sys",
        "system32\\drivers\\vmhgfs.sys",
        "system32\\drivers\\vmmemctl.sys",
        "system32\\drivers\\vmmouse.sys",
        "system32\\drivers\\vmrawdsk.sys",
        "system32\\drivers\\vmusbmouse.sys"
    };

    LPCSTR szPathsDir[] = {
        "oracle\\virtualbox guest additions\\",
        "VMWare\\"
    };

    WORD dwlength = sizeof(szPaths) / sizeof(szPaths[0]);
    WORD dwlengthDir = sizeof(szPathsDir) / sizeof(szPathsDir[0]);

    for (int i = 0; i < dwlength; i++) {
        if(lpFileName){
            if (lstrcmp(lpFileName, szPaths[i]) != 0) {
                return FILE_ATTRIBUTE_NORMAL;
            }
        }
    }

    for (int i = 0; i < dwlengthDir; i++) {
        if(lpFileName){
            if (lstrcmp(lpFileName, szPathsDir[i]) != 0) {
                return FILE_ATTRIBUTE_DIRECTORY;
            }
        }
    }

    return TrueFileAttributesA(lpFileName);
}

// Detour function for detecting files and directories
DWORD WINAPI FalseFileAttributesW(LPCWSTR lpFileName) {
    LPCWSTR szPaths[] = {
        L"system32\\drivers\\VBoxMouse.sys",
        L"system32\\drivers\\VBoxGuest.sys",
        L"system32\\drivers\\VBoxSF.sys",
        L"system32\\drivers\\VBoxVideo.sys",
        L"system32\\vboxdisp.dll",
        L"system32\\vboxhook.dll",
        L"system32\\vboxmrxnp.dll",
        L"system32\\vboxogl.dll",
        L"system32\\vboxoglarrayspu.dll",
        L"system32\\vboxoglcrutil.dll",
        L"system32\\vboxoglerrorspu.dll",
        L"system32\\vboxoglfeedbackspu.dll",
        L"system32\\vboxoglpackspu.dll",
        L"system32\\vboxoglpassthroughspu.dll",
        L"system32\\vboxservice.exe",
        L"system32\\vboxtray.exe",
        L"system32\\VBoxControl.exe",
        L"C:\\Windows\\system32\\drivers\\vmmouse.sys",
        L"system32\\drivers\\vmhgfs.sys",
        L"system32\\drivers\\vm3dmp.sys",
        L"system32\\drivers\\vmci.sys",
        L"system32\\drivers\\vmhgfs.sys",
        L"system32\\drivers\\vmmemctl.sys",
        L"system32\\drivers\\vmmouse.sys",
        L"system32\\drivers\\vmrawdsk.sys",
        L"system32\\drivers\\vmusbmouse.sys"
```

```
    };

    LPCWSTR szPathsDir[] = {
        L"oracle\\virtualbox guest additions\\",
        L"VMWare\\"
    };

    WORD dwlength = sizeof(szPaths) / sizeof(szPaths[0]);
    WORD dwlengthDir = sizeof(szPathsDir) / sizeof(szPathsDir[0]);

    for (int i = 0; i < dwlength; i++) {
        if(lpFileName){
            if (wcsstr(lpFileName, szPaths[i]) != 0) {
                return FILE_ATTRIBUTE_NORMAL;
            }
        }
    }

    for (int i = 0; i < dwlengthDir; i++) {
        if(lpFileName){
            if (wcsstr(lpFileName, szPathsDir[i]) != 0) {
                return FILE_ATTRIBUTE_DIRECTORY;
            }
        }
    }

    return TrueFileAttributesW(lpFileName);
}

HANDLE WINAPI FalseCreateToolhelp32Snapshot(DWORD dwFlags, DWORD
     th32ProcessID){
    const TCHAR *szProcesses[] = {
        _T("ollydbg.exe"),      // OllyDebug debugger
        _T("ProcessHacker.exe"),  // Process Hacker
        _T("tcpview.exe"),      // Part of Sysinternals Suite
        _T("autoruns.exe"),     // Part of Sysinternals Suite
        _T("autorunsc.exe"),    // Part of Sysinternals Suite
        _T("filemon.exe"),      // Part of Sysinternals Suite
        _T("procmon.exe"),      // Part of Sysinternals Suite
        _T("regmon.exe"),     // Part of Sysinternals Suite
        _T("procexp.exe"),      // Part of Sysinternals Suite
        _T("idaq.exe"),       // IDA Pro Interactive Disassembler
        _T("idaq64.exe"),     // IDA Pro Interactive Disassembler
        _T("ImmunityDebugger.exe"), // ImmunityDebugger
        _T("Wireshark.exe"),    // Wireshark packet sniffer
        _T("dumpcap.exe"),      // Network traffic dump tool
        _T("HookExplorer.exe"),   // Find various types of runtime hooks
        _T("ImportREC.exe"),    // Import Reconstructor
        _T("PETools.exe"),      // PE Tool
        _T("LordPE.exe"),     // LordPE
        _T("SysInspector.exe"),   // ESET SysInspector
        _T("proc_analyzer.exe"),  // Part of SysAnalyzer iDefense
        _T("sysAnalyzer.exe"),    // Part of SysAnalyzer iDefense
        _T("sniff_hit.exe"),    // Part of SysAnalyzer iDefense
        _T("windbg.exe"),     // Microsoft WinDbg
        _T("joeboxcontrol.exe"),  // Part of Joe Sandbox
        _T("joeboxserver.exe"),   // Part of Joe Sandbox
        _T("prl_cc.exe"),
        _T("prl_tools.exe"),
        _T("xenservice.exe"),
        _T("qemu-ga.exe"),
        _T("VMSrvc.exe"),
        _T("VMUSrvc.exe"),
        _T("vboxtray.exe"),
        _T("vboxservice.exe")
    };

```

```
421     WORD dwlength = sizeof(szProcesses) / sizeof(szProcesses[0]);
422     PROCESSENTRY32 pe32;
423     SecureZeroMemory(&pe32, sizeof(PROCESSENTRY32));
424     HANDLE hSnapshot = TrueCreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0)
         ;
425     pe32.dwSize = sizeof(PROCESSENTRY32);
426 
427     //Creates exe files if not launched
428     for (int i = 0; i < dwlength; i++) {
429         BOOLEAN bProcessLauchned = FALSE;
430         if (Process32First(hSnapshot, &pe32) == TRUE)
431         {
432             if (StrCmpI(pe32.szExeFile, szProcesses[i]) == 0)
433             {
434                 bProcessLauchned = TRUE;
435             }
436             else {
437                 while (Process32Next(hSnapshot, &pe32))
438                 {
439                     if (StrCmpI(pe32.szExeFile, szProcesses[i]) == 0)
440                     {
441                         bProcessLauchned = TRUE;
442                         break;
443                     }
444                 }
445             }
446         }
447         if (!bProcessLauchned){
448             STARTUPINFO info = { sizeof(info) };
449             PROCESS_INFORMATION processInfo;
450             TCHAR path[] = _T("C:\\falseExe\\empty.exe");
451             //TODO ameliorer la gestion de la taille
452             TCHAR newPath[256] = _T("C:\\falseExe\\");
453             _tcscat_s(newPath, szProcesses[i]);
454             CopyFile(path, newPath, TRUE);
455             CreateProcess(newPath, NULL, NULL, NULL, FALSE,
     CREATE_NO_WINDOW, NULL, NULL, &info, &processInfo);
456         }
457     }
458 
459     return TrueCreateToolhelp32Snapshot(dwFlags, th32ProcessID);
460 }
461 
462 HANDLE WINAPI FalseCreateToolhelp32Snapshot(DWORD dwFlags, DWORD
     th32ProcessID){
463     const TCHAR *szProcesses[] = {
464         _T("ollydbg.exe"),      // OllyDebug debugger
465         _T("ProcessHacker.exe"),  // Process Hacker
466         _T("tcpview.exe"),      // Part of Sysinternals Suite
467         _T("autoruns.exe"),     // Part of Sysinternals Suite
468         _T("autorunsc.exe"),    // Part of Sysinternals Suite
469         _T("filemon.exe"),      // Part of Sysinternals Suite
470         _T("procmon.exe"),      // Part of Sysinternals Suite
471         _T("regmon.exe"),     // Part of Sysinternals Suite
472         _T("procexp.exe"),      // Part of Sysinternals Suite
473         _T("idaq.exe"),       // IDA Pro Interactive Disassembler
474         _T("idaq64.exe"),     // IDA Pro Interactive Disassembler
475         // ... the list continues
476     };
477 
478     WORD dwlength = sizeof(szProcesses) / sizeof(szProcesses[0]);
479 
480     // Creates exe files if not launched
481     for (int i = 0; i < dwlength; i++) {
482         BOOLEAN bProcessLauchned = CheckIfLaunched();
483 
484         if (!bProcessLauchned){
```

```
485             STARTUPINFO info = { sizeof(info) };
486             PROCESS_INFORMATION processInfo;
487             TCHAR newPath[256] = CreateEmptyExeNamed(szProcesses[i]);
488             CreateProcess(newPath, NULL, NULL, NULL, FALSE,
489                 CREATE_NO_WINDOW, NULL, NULL, &info, &processInfo);
490         }
491     }
492
493     // Returns a snapshot
494     return TrueCreateToolhelp32Snapshot(dwFlags, th32ProcessID);
495 }
496
497 BOOL WINAPI DllMain(HINSTANCE hinst, DWORD dwReason, LPVOID reserved)
498 {
499     //LONG error;
500     (void)hinst;
501     (void)reserved;
502
503     if (DetourIsHelperProcess()) {
504         return TRUE;
505     }
506
507     if (dwReason == DLL_PROCESS_ATTACH) {
508         DetourRestoreAfterWith();
509         DetourTransactionBegin();
510         DetourUpdateThread(GetCurrentThread());
511         DetourAttach(&(PVOID&)TrueDebuggerPresent, FakeDebuggerPresent);
512         DetourAttach(&(PVOID&)TrueCursorPos, DoublePoint);
513         DetourAttach(&(PVOID&)TrueOpenKeyA, FalseOpenKeyA);
514         DetourAttach(&(PVOID&)TrueOpenKeyW, FalseOpenKeyW);
515         DetourAttach(&(PVOID&)TrueCreateFileA, FalseCreateFileA);
516         DetourAttach(&(PVOID&)TrueCreateFileW, FalseCreateFileW);
517         DetourAttach(&(PVOID&)TrueModuleHandleA, FalseModuleHandleA);
518         DetourAttach(&(PVOID&)TrueModuleHandleW, FalseModuleHandleW);
519         DetourAttach(&(PVOID&)TrueQueryValueA, FalseQueryValueA);
520         DetourAttach(&(PVOID&)TrueQueryValueW, FalseQueryValueW);
521         DetourAttach(&(PVOID&)TrueFileAttributesA, FalseFileAttributesA);
522         DetourAttach(&(PVOID&)TrueFileAttributesW, FalseFileAttributesW);
523         DetourAttach(&(PVOID&)TrueCreateToolhelp32Snapshot,
     FalseCreateToolhelp32Snapshot);
524         DetourTransactionCommit();
525     }
526     else if (dwReason == DLL_PROCESS_DETACH) {
527         DetourTransactionBegin();
528         DetourUpdateThread(GetCurrentThread());
529         DetourDetach(&(PVOID&)TrueDebuggerPresent, FakeDebuggerPresent);
530         DetourAttach(&(PVOID&)TrueCursorPos, DoublePoint);
531         DetourAttach(&(PVOID&)TrueOpenKeyA, FalseOpenKeyA);
532         DetourAttach(&(PVOID&)TrueOpenKeyW, FalseOpenKeyW);
533         DetourAttach(&(PVOID&)TrueCreateFileA, FalseCreateFileA);
534         DetourAttach(&(PVOID&)TrueCreateFileW, FalseCreateFileW);
535         DetourAttach(&(PVOID&)TrueModuleHandleA, FalseModuleHandleA);
536         DetourAttach(&(PVOID&)TrueModuleHandleW, FalseModuleHandleW);
537         DetourAttach(&(PVOID&)TrueQueryValueA, FalseQueryValueA);
538         DetourAttach(&(PVOID&)TrueQueryValueW, FalseQueryValueW);
539         DetourAttach(&(PVOID&)TrueFileAttributesA, FalseFileAttributesA);
540         DetourAttach(&(PVOID&)TrueFileAttributesW, FalseFileAttributesW);
541         DetourAttach(&(PVOID&)TrueCreateToolhelp32Snapshot,
     FalseCreateToolhelp32Snapshot);
542         DetourTransactionCommit();
543     }
544     return TRUE;
545 }
```

Code de la règle Yara | B

Listing B.1: Règle Yara complète utilisée pour créer un dataset de malware évasifs.

```
import "pe"

/*
* DEBUGGER RELATED RULES
*/
private rule Check_Debug_String
{
    meta:
        Author = "Cedric HERZOG"
    Description = "Checks for anti debugger attack"
    strings:
        $a = "%s%s%s%s" wide nocase ascii fullword
    condition:
        pe.imports("kernel32.dll", "SetLastError") and
        pe.imports("kernel32.dll", "OutputDebugStringA") and
        pe.imports("kernel32.dll", "GetLastError") and $a
}

private rule Check_Running_Debugger
{
    meta:
        Author = "Cedric HERZOG"
    Description = "Checks for debuggers and analysis tools process names"
    strings:
        $name1 = "ollydbg" wide nocase ascii fullword
        $name2 = "processhacker" wide nocase ascii fullword
        $name3 = "tcpview" wide nocase ascii fullword
        $name4 = "autoruns" wide nocase ascii fullword
        $name5 = "autorunsc" wide nocase ascii fullword
        $name6 = "filemon" wide nocase ascii fullword
        $name7 = "procmon" wide nocase ascii fullword
        $name8 = "idaq" wide nocase ascii fullword
        $name9 = "immunitydebugger" wide nocase ascii fullword
        $name10 = "wireshark" wide nocase ascii fullword
        $name11 = "dumpcap" wide nocase ascii fullword
        $name12 = "hookexplorer" wide nocase ascii fullword
        $name13 = "importrec" wide nocase ascii fullword
        $name14 = "petools" wide nocase ascii fullword
        $name15 = "lordpe" wide nocase ascii fullword
        $name16 = "sysinspector" wide nocase ascii fullword
        $name17 = "proc_analyzer" wide nocase ascii fullword
        $name18 = "sysanalyzer" wide nocase ascii fullword
        $name19 = "sniff_hit" wide nocase ascii fullword
        $name20 = "windbg" wide nocase ascii fullword
        $name21 = "joeboxcontrol" wide nocase ascii fullword
        $name22 = "joeboxserver" wide nocase ascii fullword
        $name23 = "netmon" wide nocase ascii fullword
        $name24 = "x64dbg" wide nocase ascii fullword
        $name25 = "x32dbg" wide nocase ascii fullword
    condition:
        10 of ($name*) and
        pe.imports("kernel32.dll", "CreateToolhelp32Snapshot") and
        pe.imports("kernel32.dll", "Process32First") and
        pe.imports("kernel32.dll", "Process32Next")
}

rule Debugger
{
    meta:
        Author = "Cedric HERZOG"
    Description = "Checks for debugger checking"
```

```
    condition:
        Check_Debug_String and
        Check_Running_Debugger
}


/*
* AV AND SANDBOX RELATED RULES
*/
private rule Check_Running_AV
{
    meta:
        Author = "Cedric HERZOG"
    Description = "Checks for antivirus process names"
    strings:
        $name1 = "aswidagent" wide nocase ascii fullword
        $name2 = "Avast" wide nocase ascii fullword
        $name3 = "avgnt" wide nocase ascii fullword
        $name4 = "avshadow" wide nocase ascii fullword
        $name5 = "Avira" wide nocase ascii fullword
        $name6 = "klnagent" wide nocase ascii fullword
        $name7 = "avp" wide nocase ascii fullword
        $name8 = "ksde" wide nocase ascii fullword
        $name9 = "mbam" wide nocase ascii fullword
        $name10 = "MBAMService" wide nocase ascii fullword
        $name11 = "mbamtray" wide nocase ascii fullword
    condition:
        4 of ($name*) and
        pe.imports("kernel32.dll", "CreateToolhelp32Snapshot") and
        pe.imports("kernel32.dll", "Process32First") and
        pe.imports("kernel32.dll", "Process32Next")
}

private rule Check_Username
{
    meta:
        Author = "Cedric HERZOG"
    Description = "Checks for common sandbox usernames"
    strings:
        $name1 = "sandbox" wide nocase ascii fullword
        $name2 = "maltest" wide nocase ascii fullword
        $name3 = "malware" wide nocase ascii fullword
        $name4 = "virus" wide nocase ascii fullword
        $name5 = "sample" wide nocase ascii fullword
        $a = "getenv_s" wide nocase ascii fullword
        $b = "USERNAME" wide ascii fullword
    condition:
        3 of ($name*) and $a and $b
}

private rule Check_Running_VM
{
    meta:
        Author = "Cedric HERZOG"
    Description = "Checks for common sandbox process names"
    strings:
        $name1 = "vmware" wide nocase ascii fullword
        $name2 = "vbox" wide nocase ascii fullword
    condition:
        any of ($name*) and
        pe.imports("kernel32.dll", "CreateToolhelp32Snapshot") and
        pe.imports("kernel32.dll", "Process32First") and
        pe.imports("kernel32.dll", "Process32Next")
}

private rule Check_Dlls
{
```

```
  meta:
    Author = "Nick Hoffman"
    Description = "Checks for common sandbox dlls"
    Sample = "de1af0e97e94859d372be7fcf3a5daa5"
  strings:
    $dll1 = "sbiedll.dll" wide nocase ascii fullword
    $dll2 = "dbghelp.dll" wide nocase ascii fullword
    $dll3 = "api_log.dll" wide nocase ascii fullword
    $dll4 = "dir_watch.dll" wide nocase ascii fullword
    $dll5 = "pstorec.dll" wide nocase ascii fullword
    $dll6 = "vmcheck.dll" wide nocase ascii fullword
    $dll7 = "wpespy.dll" wide nocase ascii fullword
  condition:
    3 of them
}

rule AV
{
    meta:
        Author = "Cedric HERZOG"
    Description = "Checks for AV and sandbox checking"
    condition:
        Check_Running_AV and
        Check_Username and
        Check_Running_VM and
        Check_Dlls
}

/*
* FOLDER MANIPULATION RULES
*/
private rule Check_Installation_Folder
{
    meta:
        Author = "Cedric HERZOG"
    Description = "Checks for antivirus directories"
    strings:
        $name1 = "avast" wide nocase ascii fullword
        $name2 = "avira" wide nocase ascii fullword
        $name3 = "kaspersky" wide nocase ascii fullword
        $name4 = "malwarebytes" wide nocase ascii fullword
        $name5 = "avg" wide nocase ascii fullword
        $a = "directory_iterator" wide nocase ascii fullword
    condition:
        3 of ($name*) and $a
}

private rule Check_Folder
{
    meta:
        Author = "Cedric HERZOG"
    Description = "Checks for common sandbox directories"
    strings:
        $name1 = "sandbox" wide nocase ascii fullword
        $name2 = "maltest" wide nocase ascii fullword
        $name3 = "malware" wide nocase ascii fullword
        $name4 = "virus" wide nocase ascii fullword
        $name5 = "sample" wide nocase ascii fullword
        $a = "directory_iterator" wide nocase ascii fullword
    condition:
        3 of ($name*) and $a
}

rule Folder
{
    meta:
        Author = "Cedric HERZOG"
```

```
    Description = "Checks for folder name comparison"
    condition:
        Check_Installation_Folder and Check_Folder
}

/*
* STANDALONE RULES
*/
rule Check_MAC_Address
{
    meta:
        Author = "Cedric HERZOG"
    Description = "Checks for common sandbox MAC addresses"
    strings:
        $name1 = "00-05-69" wide nocase ascii fullword
        $name2 = "00-0C-29" wide nocase ascii fullword
        $name3 = "00-1C-14" wide nocase ascii fullword
        $name4 = "00-50-56" wide nocase ascii fullword
        $name5 = "08-00-27" wide nocase ascii fullword
        $name6 = "00-1C-42" wide nocase ascii fullword
        $name7 = "00-0F-4B" wide nocase ascii fullword
        $name8 = "00-16-3E" wide nocase ascii fullword
        $name9 = "00-03-FF" wide nocase ascii fullword
        $name10 = "00-15-5D" wide nocase ascii fullword
        $name11 = "00-0D-3A" wide nocase ascii fullword
        $name12 = "00-12-5A" wide nocase ascii fullword
        $name13 = "00-17-FA" wide nocase ascii fullword
        $name14 = "00-50-F2" wide nocase ascii fullword
        $name15 = "00-1D-D8" wide nocase ascii fullword
    condition:
        6 of ($name*) and
        pe.imports("Iphlpapi.dll", "GetAdaptersInfo")
}

rule Check_Window
{
    meta:
        Author = "Cedric HERZOG"
    Description = "Checks for FindWindow evasion"
    strings:
        $name1 = "OLLYDBG" wide nocase ascii fullword
        $name2 = "WinDbgFrameClass" wide nocase ascii fullword
        $name3 = "Immunity Debugger" wide nocase ascii fullword
        $name4 = "Zeta Debugger" wide nocase ascii fullword
        $name5 = "Rock Debugger" wide nocase ascii fullword
        $name6 = "ObsidianGUI" wide nocase ascii fullword
        $name7 = "ID" wide nocase ascii fullword
        $name8 = "x32dbg" wide nocase ascii fullword
        $name9 = "x64dbg" wide nocase ascii fullword
    condition:
        4 of ($name*) and
        pe.imports("User32.dll", "FindWindow")
}
```

Code du monitoring

C

```python
import time
import ctypes
import subprocess
from ctypes.wintypes import HANDLE, LONG, ULONG
from monitor import measure

## DEBUT Manipulation pour lancer un process en suspendu ##
ntdll = ctypes.WinDLL("ntdll.dll")
RtlNtStatusToDosError = ntdll.RtlNtStatusToDosError
NtResumeProcess = ntdll.NtResumeProcess

def errcheck_ntstatus(status, *etc):
   if status < 0: raise ctypes.WinError(RtlNtStatusToDosError(status))
   return status

RtlNtStatusToDosError.argtypes = (LONG,)
RtlNtStatusToDosError.restype  = ULONG

NtResumeProcess.argtypes = (HANDLE,)
NtResumeProcess.restype  = LONG
NtResumeProcess.errcheck = errcheck_ntstatus

def resume_subprocess(proc):
    NtResumeProcess(int(proc._handle))
## FIN Manipulation pour lancer un process en suspendu ##

# On créé un process dans un état suspendu : https://docs.microsoft.com/en-us/windows/win32/procthread/process-creation-flags
p = subprocess.Popen('notepad', creationflags=4)

# Un sous processus est démarré qui lance le monitoring avec en argument le PID du processus à surveiller
subprocess.Popen(['python', 'monitor.py', str(p.pid)])

# On attend 2 secondes pour voir s'il reste bien dans cet état
time.sleep(2)

# On redémarre le process en appellant directement l'API Windows
resume_subprocess(p)

# On attend 5min pour voir si le process démarre bien
time.sleep(300)

# On kill si ça dépasse 5min
p.kill()

# On attend 2 secondes  pour que le kill soit fait
time.sleep(2)
```

Listing C.1: Code complet du script qui démarre le malware et le monitoring. Le programme démarré dans l'exemple est notepad.

```python
from os import cpu_count
import sys
import time
import ctypes
from datetime import datetime,timedelta
from ctypes.wintypes import DWORD, WORD, HANDLE, LONG, BOOL, LPFILETIME, ULONG, FILETIME

def measure(pid):
    """Effectue un monitoring d'un processus.
    Un handle est ouvert avec OpenProcess et un timestamp est enregistrer pour chaque mesure avec GetSystemTime. Trois fonctions sont lancées à
```

Listing C.2: Code complet du script de monitoring.

```
    la suite pour effectuer les mesures : GetProcessTimes,
    GetProcessMemoryInfo, QueryProcessCycleTime

    Args:
        pid (int): PID du processus à monitorer
    """
    kernel32 = ctypes.WinDLL("kernel32.dll")

    # GetSystemTime Definition
    class SYSTEMTIME (ctypes.Structure):
        _fields_ = [
            ('wYear', WORD),
            ('wMonth', WORD),
            ('wDayOfWeek', WORD),
            ('wDay', WORD),
            ('wHour', WORD),
            ('wMinute', WORD),
            ('wSecond', WORD),
            ('wMilliseconds', WORD),
        ]

    GetSystemTime = kernel32.GetSystemTime
    GetSystemTime.argtypes = (ctypes.POINTER(SYSTEMTIME) ,)
    GetSystemTime.restype = None

    lpSystemTime = SYSTEMTIME()

    # GetProcessTimes Definition
    GetProcessTimes = kernel32.GetProcessTimes
    GetProcessTimes.argtypes = (HANDLE, LPFILETIME, LPFILETIME, LPFILETIME
     , LPFILETIME, )
    GetProcessTimes.restype = LONG

    lpCreationTime = FILETIME()
    lpExitTime = FILETIME()
    lpKernelTime = FILETIME()
    lpUserTime = FILETIME()

    # OpenProcess Definition
    OpenProcess = kernel32.OpenProcess
    OpenProcess.argtypes = (DWORD, BOOL, DWORD,)
    OpenProcess.restype = HANDLE

    dwDesiredAccess = DWORD()
    dwDesiredAccess.value = 0x0400
    bInheritHandle = BOOL()
    bInheritHandle.value = False
    dwProcessId = DWORD()
    dwProcessId.value = pid

    # QueryProcessCycleTime Definition
    QueryProcessCycleTime = kernel32.QueryProcessCycleTime
    QueryProcessCycleTime.argtypes = (HANDLE, ctypes.POINTER(ULONG),)
    QueryProcessCycleTime.restype = BOOL

    CycleTime = ULONG()

    # GetProcessMemoryInfo Definition
    SIZE_T = ctypes.c_size_t
    class PROCESS_MEMORY_COUNTERS(ctypes.Structure):
        _fields_ = [
            ('cb', DWORD),
            ('PageFaultCount', DWORD),
            ('PeakWorkingSetSize', SIZE_T),
            ('WorkingSetSize', SIZE_T),
            ('QuotaPeakPagedPoolUsage', SIZE_T),
            ('QuotaPagedPoolUsage', SIZE_T),
```

```
            ('QuotaPeakNonPagedPoolUsage', SIZE_T),
            ('QuotaNonPagedPoolUsage', SIZE_T),
            ('PagefileUsage', SIZE_T),
            ('PeakPagefileUsage', SIZE_T),
        ]

    psapi = ctypes.WinDLL("psapi.dll")
    GetProcessMemoryInfo = psapi.GetProcessMemoryInfo
    GetProcessMemoryInfo.argtypes = (HANDLE, ctypes.POINTER(
     PROCESS_MEMORY_COUNTERS), DWORD,)
    GetProcessMemoryInfo.restype = BOOL

    ppsmemCounters = PROCESS_MEMORY_COUNTERS()



    with open('mesure.csv', 'w') as file:
        file.write('\"TimeStamp\";'
        +'\"CreationTime\";'
        +'\"ExitTime\";'
        +'\"KernelTime\";'
        +'\"UserTime\";'
        + '\"PageFaultCount\";'
        + '\"WorkingSetSize\";'
        + '\"QuotaPagedPoolUsage\";'
        + '\"QuotaNonPagedPoolUsage\";'
        + '\"PagefileUsage\";'
        + '\"CPUCycle\";'
        +'\n')
        while True:
            # On prend d'abord toutes les mesures dans le plus court temps
     possible avant de les traiter
            handle = OpenProcess(dwDesiredAccess, bInheritHandle,
     dwProcessId)

            GetSystemTime(lpSystemTime)

            GetProcessTimes(handle, lpCreationTime, lpExitTime,
     lpKernelTime, lpUserTime)

            GetProcessMemoryInfo(handle, ctypes.byref(ppsmemCounters),
                                 ctypes.sizeof(ppsmemCounters))

            QueryProcessCycleTime(handle, ctypes.byref(CycleTime))

            # On traite les mesures pour les afficher dans un format
     lisible
            # e.g.: 2021-01-21 10:56:31.749374
            sysDate = dict((name, getattr(lpSystemTime, name))
                for name, _ in lpSystemTime._fields_)
            timestamp = str(sysDate['wYear']) + '-' +str(sysDate['wMonth'
    ]).zfill(2) + '-' + str(sysDate['wDay']).zfill(2) + ' ' + str(sysDate
    ['wHour']).zfill(2) + ':' + str(sysDate['wMinute']).zfill(2) + ':' +
    str(sysDate['wSecond']).zfill(2) + '.' + str(sysDate['wMilliseconds'
    ]).zfill(3)

            # On récupère les informations de GetProcessMemoryInfo
            info = dict((name, getattr(ppsmemCounters, name))
                for name, _ in ppsmemCounters._fields_)
            PageFaultCount = info['PageFaultCount']
            WorkingSetSize = info['WorkingSetSize']
            QuotaPagedPoolUsage = info['QuotaPagedPoolUsage']
            QuotaNonPagedPoolUsage = info['QuotaNonPagedPoolUsage']
            PagefileUsage = info['PagefileUsage']

            # On récupère les informations de GetProcessTimes
            creationDate = (lpCreationTime.dwHighDateTime << 32) +
```

```
    lpCreationTime.dwLowDateTime
133         creationDate = datetime(1601,1,1) + timedelta(microseconds=
    creationDate/10)
134
135         exitTime = (lpExitTime.dwHighDateTime << 32) + lpExitTime.
    dwLowDateTime
136         exitTime = datetime(1601,1,1) + timedelta(microseconds=
    exitTime/10)
137
138         kernelTime = (lpKernelTime.dwHighDateTime << 32) +
    lpKernelTime.dwLowDateTime
139         kernelTime = datetime(1601,1,1) + timedelta(microseconds=
    kernelTime/10)
140
141         userTime = (lpUserTime.dwHighDateTime << 32) + lpUserTime.
    dwLowDateTime
142         userTime = datetime(1601,1,1) + timedelta(microseconds=
    userTime/10)
143
144         # On écrit les résultats de cette mesure dans le fichier
145         file.write(
146             str(timestamp) +';' +
147             str(creationDate) +';' +
148             str(exitTime) + ';' +
149             str(kernelTime) + ';' +
150             str(userTime) + ';' +
151             str(PageFaultCount) + ';' +
152             str(WorkingSetSize) + ';' +
153             str(QuotaPagedPoolUsage) + ';' +
154             str(QuotaNonPagedPoolUsage) + ';' +
155             str(PagefileUsage) + ';' +
156             str(CycleTime.value) +
157             '\n')
158
159         # On attend 0.001 seconde avant de prendre la mesure suivante
160         time.sleep(0.001)
161
162
163 if __name__ == "__main__":
164     measure(int(sys.argv[1]))
```

Références de la bibliographie interactive

[6] Joanna Rutkowska. *Red Pill. . . or how to detect VMM using (almost) one CPU instruction*. 2004. URL : http://web.archive.org/web/20110726182809/http://invisiblethings.org/papers/redpill.html (visité le 25/05/2021) (cf. p. 8, 13, 16).

[7] Tobias Klein. *ScoopyNG : The VMware detection tool.* 2006. URL : https://www.trapkit.de/tools/scoopyng/ (visité le 18/06/2021) (cf. p. 8).

[8] Ilfak Guilfanov. *2005 - Simple trick to hide IDA debugger*. 2005. URL : https://hex-rays.com/blog/simple-trick-to-hide-ida-debugger/ (visité le 22/06/2021) (cf. p. 8).

[9] Kostya kortchinsky. *Counter measures to VMware fingerprinting*. 2004. URL : https://seclists.org/honeypots/2004/q1/15 (visité le 14/06/2021) (cf. p. 8, 17).

[10] Matthew Carpenter, Tom Liston et Ed Skoudis. « Hiding Virtualization from Attackers and Malware ». In : *IEEE Secur. Priv.* 5.3 (2007), p. 62-65. DOI : 10.1109/MSP.2007.63 (cf. p. 8).

[11] Li Sun, Tim Ebringer et S. Boztag. « An automatic anti-anti-VMware technique applicable for multi-stage packed malware ». In : *3rd International Conference on Malicious and Unwanted Software, MALWARE 2008, Alexandria, Virginia, USA, October 7-8, 2008*. IEEE Computer Society, 2008, p. 17-23. DOI : 10.1109/MALWARE.2008.4690853 (cf. p. 8).

[12] Amit Vasudevan et Ramesh Yerraballi. « Cobra : Fine-grained Malware Analysis using Stealth Localized-executions ». In : *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*. IEEE Computer Society, 2006, p. 264-279. DOI : 10.1109/SP.2006.9 (cf. p. 8).

[13] Anh M. Nguyen et al. « MAVMM : Lightweight and Purpose Built VMM for Malware Analysis ». In : *Twenty-Fifth Annual Computer Security Applications Conference, ACSAC 2009, Honolulu, Hawaii, USA, 7-11 December 2009*. IEEE Computer Society, 2009, p. 441-450. DOI : 10.1109/ACSAC.2009.48 (cf. p. 8).

[14] Clemens Kolbitsch, Engin Kirda et Christopher Kruegel. « The Power of Procrastination : Detection and Mitigation of Execution-Stalling Malicious Code ». In : *Proceedings of the 18th ACM Conference on Computer and Communications Security*. CCS '11. Chicago, Illinois, USA : Association for Computing Machinery, 2011, p. 285-296. DOI : 10.1145/2046707.2046740 (cf. p. 9).

[15] Zhui Deng, Xiangyu Zhang et Dongyan Xu. « SPIDER : Stealthy Binary Program Instrumentation and Debugging via Hardware Virtualization ». In : ACSAC '13. New Orleans, Louisiana, USA : Association for Computing Machinery, 2013, p. 289-298. DOI : 10.1145/2523649.2523675 (cf. p. 9).

[16] Tamas K. Lengyel et al. « Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System ». In : *Proceedings of the 30th Annual Computer Security Applications Conference*. ACSAC '14. New Orleans, Louisiana, USA : Association for Computing Machinery, 2014, p. 386-395. DOI : 10.1145/2664243.2664252 (cf. p. 9, 26).

[17] Fengwei Zhang et al. « Using Hardware Features for Increased Debugging Transparency ». In : *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, p. 55-69. DOI : 10.1109/SP.2015.11 (cf. p. 9, 26).

[18] Kevin Leach et al. « Towards Transparent Introspection ». In : *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. IEEE Computer Society, 2016, p. 248-259. DOI : 10.1109/SANER.2016.25 (cf. p. 9).

[19] Min Gyung Kang et al. « Emulating Emulation-Resistant Malware ». In : *Proceedings of the 1st ACM Workshop on Virtual Machine Security*. VMSec '09. Chicago, Illinois, USA : Association for Computing Machinery, 2009, p. 11-22. DOI : 10.1145/1655148.1655151 (cf. p. 9).

[20] Lok-Kwong Yan et al. « V2E : Combining Hardware Virtualization and Softwareemulation for Transparent and Extensible Malware Analysis ». In : *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*. VEE '12. London, England, UK : Association for Computing Machinery, 2012, p. 227-238. DOI : 10.1145/2151024.2151053 (cf. p. 9).

[21] Manuel Egele et al. « A Survey on Automated Dynamic Malware-Analysis Techniques and Tools ». In : *ACM Comput. Surv.* 44.2 (2008). doi : 10.1145/2089125.2089126 (cf. p. 9).

[22] Tal Garfinkel et al. « Compatibility Is Not Transparency : VMM Detection Myths and Realities ». In : *Proceedings of HotOS'07 : 11th Workshop on Hot Topics in Operating Systems, May 7-9, 2005, San Diego, California, USA*. Sous la dir. de Galen C. Hunt. USENIX Association, 2007 (cf. p. 9).

[23] Davide Balzarotti et al. « Efficient Detection of Split Personalities in Malware ». In : *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*. The Internet Society, 2010 (cf. p. 10, 26).

[24] Yen Ju Liu et al. « Fast Discovery of VM-Sensitive Divergence Points with Basic Block Comparison ». In : *Eighth International Conference on Software Security and Reliability, SERE 2014, San Francisco, California, USA, June 30 - July 2, 2014*. IEEE, 2014, p. 196-205. doi : 10.1109/SERE.2014.33 (cf. p. 10).

[25] Chia-Wei Hsu et al. « Divergence Detector : A Fine-Grained Approach to Detecting VM-Awareness Malware ». In : *IEEE 7th International Conference on Software Security and Reliability, SERE 2013, Gaithersburg, MD, USA, June 18-20, 2013*. IEEE, 2013, p. 80-89. doi : 10.1109/SERE.2013.23 (cf. p. 10).

[26] Smita Naval et al. « Environment-Reactive Malware Behavior : Detection and Categorization ». In : *Data Privacy Management, Autonomous Spontaneous Security, and Security Assurance - 9th International Workshop, DPM 2014, 7th International Workshop, SETOP 2014, and 3rd International Workshop, QASA 2014, Wroclaw, Poland, September 10-11, 2014. Revised Selected Papers*. T. 8872. Lecture Notes in Computer Science. Springer, 2014, p. 167-182. doi : 10.1007/978-3-319-17016-9_11 (cf. p. 10, 20, 26).

[27] Ming-Kung Sun et al. « Malware Virtualization-Resistant Behavior Detection ». In : *17th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2011, Tainan, Taiwan, December 7-9, 2011*. IEEE Computer Society, 2011, p. 912-917. doi : 10.1109/ICPADS.2011.78 (cf. p. 10).

[28] Dhilung Kirat, Giovanni Vigna et Christopher Kruegel. « BareCloud : Bare-metal Analysis-based Evasive Malware Detection ». In : *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. Sous la dir. de Kevin Fu et Jaeyeon Jung. USENIX Association, 2014, p. 287-301 (cf. p. 10, 21, 26).

[29] Kalpa Vishnani, Alwyn R. Pais et Radhesh Mohandas. « Detecting and Defeating Split Personality Malware ». In : *SECURWARE 2011 : The Fifth International Conference on Emerging Security Information, Systems and Technologies*. Nice/Saint Laurent du Var, France : International Academy, Research, et Industry Association (IARIA), août 2011 (cf. p. 10, 26).

[30] Joash W. J. Tan et Roland H. C. Yap. « Detecting Malware Through Anti-analysis Signals - A Preliminary Study ». In : *Cryptology and Network Security - 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings*. Sous la dir. de Sara Foresti et Giuseppe Persiano. T. 10052. Lecture Notes in Computer Science. 2016, p. 542-551. doi : 10.1007/978-3-319-48965-0_33 (cf. p. 10, 26).

[31] Abdurrahman Pektas et Tankut Acarman. « A dynamic malware analyzer against virtual machine aware malicious software ». In : *Secur. Commun. Networks* 7.12 (2014), p. 2245-2257. doi : 10.1002/sec.931 (cf. p. 10).

[32] Martina Lindorfer, Clemens Kolbitsch et Paolo Milani Comparetti. « Detecting Environment-Sensitive Malware ». In : *Recent Advances in Intrusion Detection - 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings*. Sous la dir. de Robin Sommer, Davide Balzarotti et Gregor Maier. T. 6961. Lecture Notes in Computer Science. Springer, 2011, p. 338-357. doi : 10.1007/978-3-642-23644-0_18 (cf. p. 10, 26).

[33] Dhilung Kirat et Giovanni Vigna. « MalGene : Automatic Extraction of Malware Analysis Evasion Signature ». In : *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA : Association for Computing Machinery, 2015, p. 769-780. doi : 10.1145/2810103.2813642 (cf. p. 11, 21, 26).

[34] Lenny Zeltser. *Virtual Machine Detection in Malware via Commercial Tools*. 2006. url : https://isc.sans.edu/diary/Virtual+Machine+Detection+in+Malware+via+Commercial+Tools/1871 (visité le 30/06/2021) (cf. p. 11).

[35] Boris Lau et Vanja Svajcer. « Measuring virtual machine detection in malware using DSD tracer ». In : *J. Comput. Virol.* 6.3 (2010), p. 181-195. DOI : 10.1007/s11416-008-0096-y (cf. p. 11, 13).

[36] Mariano Graziano et al. « Needles in a Haystack : Mining Information from Public Dynamic Analysis Sandboxes for Malware Intelligence ». In : *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. Sous la dir. de Jaeyeon Jung et Thorsten Holz. USENIX Association, 2015, p. 1057-1072 (cf. p. 11).

[37] Ashish Jadhav, Deepti Vidyarthi et Hemavathy M. « Evolution of evasive malwares : A survey ». In : *2016 International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT)*. New Delhi, India : IEEE, mars 2016, p. 641-646. DOI : 10.1109/ICCTICT.2016.7514657 (cf. p. 12).

[38] Xu Chen et al. « Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware ». In : *The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings*. IEEE Computer Society, 2008, p. 177-186. DOI : 10.1109/DSN.2008.4630086 (cf. p. 12).

[39] Cédric Herzog et al. « Evasive Windows Malware : Impact on Antiviruses and Possible Countermeasures ». In : *SECRYPT 2020 - 17th International Conference on Security and Cryptography*. Proceedings of the 17th International Joint Conference on e-Business and Telecommunications - (Volume 3). Lieusaint - Paris, France : SCITEPRESS - Science and Technology Publications, juill. 2020, p. 302-309. DOI : 10.5220/0009816703020309 (cf. p. 12, 26).

[40] Jialong Zhang et al. « Scarecrow : Deactivating Evasive Malware via Its Own Evasive Logic ». In : *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Valencia, Spain : IEEE, juin 2020, p. 76-87. DOI : 10.1109/DSN48063.2020.00027 (cf. p. 12, 20, 21, 26).

[41] John Scott Robin et Cynthia E. Irvine. « Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor ». In : *9th USENIX Security Symposium (USENIX Security 00)*. Denver, CO : USENIX Association, août 2000 (cf. p. 13).

[42] Stéphane Charette. *How not to detect virtualization*. 2009. URL : http://charette.no-ip.com:81/programming/2009-12-30_Virtualization/ (visité le 30/06/2021) (cf. p. 14, 15).

[43] Anoirel Issa. « Anti-virtual machines and emulations ». In : *J. Comput. Virol.* 8.4 (2012), p. 141-149. DOI : 10.1007/s11416-012-0165-0 (cf. p. 16).

[44] Gaël Delalleau. « Mesure locale des temps d'exécution : application au contrôle d'intégrité et au fingerprinting ». In : Rennes, FRANCE : SSTIC, juin 2004 (cf. p. 17).

[45] Thorsten Holz et Frédéric Raynal. « Detecting honeypots and other suspicious environments ». In : *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop* (juill. 2005), p. 29-36 (cf. p. 17).

[46] Chew Keong Tan. « Defeating Kernel Native API Hookers by Direct Service Dispatch Table Restoration ». In : juill. 2004 (cf. p. 18).

[49] Alexei Bulazel et Bülent Yener. « A Survey On Automated Dynamic Malware Analysis Evasion and Counter-Evasion : PC, Mobile, and Web ». In : *Proceedings of the 1st Reversing and Offensive-Oriented Trends Symposium*. ROOTS. Vienna, Austria : Association for Computing Machinery, 2017. DOI : 10.1145/3150376.3150378 (cf. p. 30).

[50] Catalin-Valeriu Lita, Doina Cosovan et Dragos Gavrilut. « Anti-emulation trends in modern packers : a survey on the evolution of anti-emulation techniques in UPA packers ». In : *J. Comput. Virol. Hacking Tech.* 14.2 (2018), p. 107-126. DOI : 10.1007/s11416-017-0291-9 (cf. p. 30).

[51] Amir Afianian et al. « Malware Dynamic Analysis Evasion Techniques : A Survey ». In : *ACM Comput. Surv.* 52.6 (2019). DOI : 10.1145/3365001 (cf. p. 30).

[61] Elias Bachaalany. *Detect if your program is running inside a Virtual Machine*. 2005. URL : https://www.codeproject.com/Articles/9823/Detect-if-your-program-is-running-inside-a-Virtual (visité le 25/06/2021).

[62] Benjamin Armstrong. *Detecting Microsoft virtual machines*. 2005. URL : https://docs.microsoft.com/en-us/archive/blogs/virtual_pc_guy/detecting-microsoft-virtual-machines (visité le 25/06/2021).

[63] Peter Ferrie. « Attacks on Virtual Machine Emulators ». In : *9th AVAR Association of anti-Virus Asia Researchers*. Auckland, NEW ZEALAND, déc. 2006.

[64] Danny Quist et Val Smith. *Detecting the Presence of Virtual Machines Using the Local Data Table*. Rapp. tech. 2006.

[65] Danny Quist et Val Smith. *Further Down the VM Spiral Detection of full and partial emulation for IA-32 virtual machines*. Rapp. tech. 2006.

[66] Alfredo Andrés Omella. *Methods for Virtual Machine Detection*. Rapp. tech. 2006.

[67] Tom Liston et Ed Skoudis. *On the Cutting Edge : Thwarting Virtual Machine Detection*. Rapp. tech. 2006.

[68] Peter Ferrie. *Attacks on More Virtual Machine Emulators*. Rapp. tech. 2007.

[69] Thomas Raffetseder, Christopher Krügel et Engin Kirda. « Detecting System Emulators ». In : *Information Security, 10th International Conference, ISC 2007, Valparaíso, Chile, October 9-12, 2007, Proceedings*. Sous la dir. de Juan A. Garay et al. T. 4779. Lecture Notes in Computer Science. Springer, 2007, p. 1-18. DOI : 10.1007/978-3-540-75496-1_1.

[70] John Bethencourt, Dawn Song et Brent Waters. « Analysis-Resistant Malware ». In : *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society, 2008.

[71] Jason Franklin et al. « Remote Detection of Virtual Machine Monitors with Fuzzy Benchmarking ». In : *SIGOPS Oper. Syst. Rev.* 42.3 (2008), p. 83-92. DOI : 10.1145/1368506.1368518.

[72] Jason Franklin et al. « Towards Sound Detection of Virtual Machines ». In : *Botnet Detection : Countering the Largest Security Threat*. Sous la dir. de Wenke Lee, Cliff Wang et David Dagon. T. 36. Advances in Information Security. Springer, 2008, p. 89-116. DOI : 10.1007/978-0-387-68768-1_5.

[73] Roberto Paleari et al. « A Fistful of Red-Pills : How to Automatically Generate Procedures to Detect CPU Emulators ». In : *Proceedings of the 3rd USENIX Conference on Offensive Technologies*. WOOT'09. Montreal, Canada : USENIX Association, 2009, p. 2.

[74] Ulrich Bayer et al. « A View on Current Malware Behaviors ». In : *2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats, LEET '09, Boston, MA, USA, April 21, 2009*. Sous la dir. de Wenke Lee. USENIX Association, 2009.

[75] Clemens Kolbitsch et al. « Effective and Efficient Malware Detection at the End Host ». In : *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*. Sous la dir. de Fabian Monrose. USENIX Association, 2009, p. 351-366.

[76] Lorenzo Martignoni et al. « Testing CPU Emulators ». In : *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. ISSTA '09. Chicago, IL, USA : Association for Computing Machinery, 2009, p. 261-272. DOI : 10.1145/1572272.1572303.

[77] Christopher Thompson, Maria Huntley et Chad Link. « Virtualization detection : New strategies and their effectiveness ». In : *Univ. of Minn., Minneapolis, MN, USA* (2010).

[78] Alexandros Kapravelos et al. « Escape from Monkey Island : Evading High-Interaction Honeyclients ». In : *Detection of Intrusions and Malware, and Vulnerability Assessment - 8th International Conference ; DIMVA 2011, Amsterdam, The Netherlands, July 7-8, 2011. Proceedings*. Sous la dir. de Thorsten Holz et Herbert Bos. T. 6739. Lecture Notes in Computer Science. Springer, 2011, p. 124-143. DOI : 10.1007/978-3-642-22424-9_8.

[79] Gábor Pék, Boldizsár Bencsáth et Levente Buttyán. « NEther : In-Guest Detection of out-of-the-Guest Malware Analyzers ». In : *Proceedings of the Fourth European Workshop on System Security*. EUROSEC '11. Salzburg, Austria : Association for Computing Machinery, 2011. DOI : 10.1145/1972551.1972554.

[80] Katsunari Yoshioka et al. « Your Sandbox is Blinded : Impact of Decoy Injection to Public Malware Analysis Systems ». In : *J. Inf. Process.* 19 (2011), p. 153-168. DOI : 10.2197/ipsjjip.19.153.

[81] Rodrigo Rubira Branco, Gabriel Negreira Barbosa et Pedro Drimel Neto. *Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies*. Rapp. tech. 2012.

[82] Gábor Pék, Levente Buttyán et Boldizsár Bencsáth. « A Survey of Security Issues in Hardware Virtualization ». In : 45.3 (2013). doi : 10.1145/2480741.2480757.

[83] Abhishek Singh et Zheng Bu. *HOT KNIVES THROUGH BUTTER : Evading File-based Sandboxes*. Rapp. tech. 2013.

[84] Hao Shi, Abdulla Alwabel et Jelena Mirkovic. « Cardinal Pill Testing of System Virtual Machines ». In : *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. Sous la dir. de Kevin Fu et Jaeyeon Jung. USENIX Association, 2014, p. 271-285.

[85] Christopher Kruegel. *Full System Emulation : Achieving Successful Automated Dynamic Analysis of Evasive Malware*. Rapp. tech. 2014.

[86] Olivier Ferrand. « How to detect the Cuckoo Sandbox and to Strengthen it ? » In : *J. Comput. Virol. Hacking Tech.* 11.1 (2015), p. 51-58. doi : 10.1007/s11416-014-0224-9.

[87] Yuxin Gao, Zexin Lu et Yuqing Luo. « Survey on malware anti-analysis ». In : *Fifth International Conference on Intelligent Control and Information Processing*. Dalian, China : IEEE, 2014, p. 270-275. doi : 10.1109/ICICIP.2014.7010353.

[88] Yosuke Chubachi et Kenji Aiko. *TENTACLE : Environment-Sensitive Malware Palpation*. 2014. url : https://www.ffri.jp/assets/files/research/research_papers/pacsec2014_chubachi_aiko_EN.pdf (visité le 15/01/2022).

[89] Candid Wueest. *Threats to virtual environments*. Rapp. tech. 2014.

[90] Hassan Mourad. *Sleeping Your Way out of the Sandbox*. Rapp. tech. 2015.

[91] Ping Chen et al. « Advanced or Not ? A Comparative Study of the Use of Anti-debugging and Anti-VM Techniques in Generic and Targeted Malware ». In : *ICT Systems Security and Privacy Protection - 31st IFIP TC 11 International Conference, SEC 2016, Ghent, Belgium, May 30 - June 1, 2016, Proceedings*. Sous la dir. de Jaap-Henk Hoepman et Stefan Katzenbeisser. T. 471. IFIP Advances in Information and Communication Technology. Springer, 2016, p. 323-336. doi : 10.1007/978-3-319-33630-5_22.

[92] Yaniv Assor. *Anti-VM and Anti-Sandbox Explained*. 2016. url : https://www.cyberbit.com/blog/endpoint-security/anti-vm-and-anti-sandbox-explained/ (visité le 30/06/2021).

[93] Jeremy Blackthorne et al. « AVLeak : Fingerprinting Antivirus Emulators through Black-Box Testing ». In : *10th USENIX Workshop on Offensive Technologies, WOOT 16, Austin, TX, USA, August 8-9, 2016*. Sous la dir. de Natalie Silvanovich et Patrick Traynor. USENIX Association, 2016.

[94] Najmeh Miramirkhani et al. « Spotless Sandboxes : Evading Malware Analysis Systems Using Wear-and-Tear Artifacts ». In : *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 2017, p. 1009-1024. doi : 10.1109/SP.2017.42.

[95] Yoshihiro Oyama. « Trends of anti-analysis operations of malwares observed in API call logs ». In : *J. Comput. Virol. Hacking Tech.* 14.1 (2018), p. 69-85. doi : 10.1007/s11416-017-0290-x.

[96] Akira Yokoyama et al. « SandPrint : Fingerprinting Malware Sandboxes to Provide Intelligence for Sandbox Evasion ». In : *Research in Attacks, Intrusions, and Defenses - 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings*. Sous la dir. de Fabian Monrose et al. T. 9854. Lecture Notes in Computer Science. Springer, 2016, p. 165-187. doi : 10.1007/978-3-319-45719-2_8.

Autres références

Les références sont classées par ordre de citation.

[4] William Sutherland et Robert Thomas. « Natural Communication with Computers. Volume III. Distributed Computation Research at BBN ». In : *BOLT, BERANEK AND NEWMAN INC*. Cambridge, MA, déc. 1974 (cf. p. 3).

[5] Ilsun You et Kangbin Yim. « Malware Obfuscation Techniques : A Brief Survey ». In : *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*. 2010, p. 297-300. doi : 10.1109/BWCCA.2010.85 (cf. p. 4).

[47] Michael Hale Ligh et al. *The Art of Memory Forensics : Detecting Malware and Threats in Windows, Linux, and Mac Memory*. 1st. Wiley Publishing, 2014 (cf. p. 18).

[48] Joxean Koret et Elias Bachaalany. *The Antivirus Hacker's Handbook*. 1. Wiley Publishing, 2015 (cf. p. 29).

[52] National Institute of Standards et Technology. « Advanced Encryption Standard ». In : *NIST FIPS PUB 197* (2001) (cf. p. 32).

[53] David A. Huffman. « A Method for the Construction of Minimum-Redundancy Codes ». In : *Proceedings of the IRE* 40.9 (1952), p. 1098-1101. doi : 10.1109/JRPROC.1952.273898 (cf. p. 32).

[54] Robert Lyda et James Hamrock. « Using Entropy Analysis to Find Encrypted and Packed Malware ». In : *IEEE Security and Privacy* 5.2 (2007), p. 40-45. doi : 10.1109/MSP.2007.48 (cf. p. 43).

[55] Doug Brubacher. « Detours : Binary Interception of Win32 Functions ». In : *Windows NT 3rd Symposium (Windows NT 3rd Symposium)*. Seattle, WA : USENIX Association, juill. 1999 (cf. p. 45).

[56] Aurélien Palisse, Antoine Durand et Jean-Louis Lanet. « Malware'O'Matic a platform to analyze Malware ». In : *French Japanese workshop on CyberSecurity*. Inria. Rennes, France, sept. 2016 (cf. p. 51).

[58] Jean-Yves Le Boudec. *Performance Evaluation of Computer and Communication Systems*. EFPL Press, 2011 (cf. p. 70).

[59] Richard Chin et Bruce Y. Lee. « Chapter 15 - Analysis of Data ». In : *Principles and Practice of Clinical Trial Medicine*. Sous la dir. de Richard Chin et Bruce Y. Lee. New York : Academic Press, 2008, p. 325-359. doi : https://doi.org/10.1016/B978-0-12-373695-6.00015-6 (cf. p. 70).

[60] Sander Greenland et al. « Statistical tests, P values, confidence intervals, and power : a guide to misinterpretations ». In : *European Journal of Epidemiology* 31 (2016), p. 337-350 (cf. p. 70).

Index

Glossaire

AES Advanced Encryption Standard. 32
API Application Programming Interface. i, 10, 12, 18, 29, 30, 32, 33, 45, 51–53, 57, 65, 66, 89
ASLR Address Space Layout Randomization. 16

C&C Command & Control. 60
CARO Computer Research Organization Antivirus. 33
CBC Cipher Block Chaining. 32
CPU Central Processing Unit. 9, 13, 15–17, 31, 45, 65, 66
CSV Comma-Separated Values. 53, 66

DHCP Dynamic Host Configuration Protocol. 64
DLL Dynamic Link Library. 29, 39, 45, 49
DOS Disk Operating System. 3

ECB Electronic Code Book. 32
EICAR European Institute for Computer Antivirus Research. 33–35

FTP File Transfer Protocol. 51

GDT Global Descriptor Table. 11

HTML HyperText Markup Language. 52

IDE Integrated Drive Electronics. 17
IDT Interrupt Descriptor Table. 8, 11
IDTR Interrupt Descriptor Table Register. 16
IP Internet Protocol. 64

JSON JavaScript Objet Notation. 63

KPP Kernel Patch Protection. 18, 84

LHS Laboratoire Haute Sécurité. 51, 63

MAC Media Access Control. 32, 36, 40
MD5 Message Digest 5. 41
MFT Master File Table. 84
MoM Malware o Matic. 51, 52, 62–64, 66

NAS Network Attached Storage. 51

OS Operating System. 13, 15, 16, 52

PDF Portable Document Format. 52
PHP PHP : Hypertext Preprocessor. 77

RAM Random Access Memory. 66
RDTSC ReaD Time Stamp Counter. 8

SDT Service Description Table. 11
SIDT Store Interrupt Descriptor Table Register. 13, 16
SLAT Second Level Address Translation. 15
SMM System Management Mode. 9

SQL Structured Query Language. 77
SSDT System Service Dispatch Table. 18, 84

TEMPO The Evasive Malware PlatfOrm. 59, 77, 78, 86, 89–91
TLB Translation Lookaside Buffer. 15

VM Virtual Machine. 13, 14, 16, 17, 30, 36, 43

x86 Famille de jeu d'instruction. 14, 15, 30

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

CentraleSupélec

Titre : Étude des malware évasifs : conception, protection et détection

Mot clés : Malware, Évasif, Windows, Contre-mesure, Dataset

Résumé : Il y a une confrontation permanente entre les malware et les antivirus conduisant les deux partie à évoluer continuellement. D'un côté, les antivirus mettent en place des solutions de plus en avancées et proposent des techniques de détection complexes venant s'ajouter à la classique détection par signature. Lorsqu'un nouveau malware est détecté, les antivirus conduisent des recherches plus approfondies afin de produire une signature et garder leur base de données à jour rapidement. Cette complexification conduit les antivirus à laisser des traces de leur présence sur les machines qu'ils protègent.

D'un autre côté, des auteurs de malware souhaitant créer des malware durables à bas coûts peuvent quant à eux utiliser de simples méthodes permettant d'éviter une détection et une analyse approfondie par ces antivirus. Il est possible pour un malware de rechercher la présence de traces ou d'artéfacts laissés par les antivirus pour ensuite décider d'exécuter ou non leur charge malveillante. Nous désignons de tels programmes comme étant des *malware évasifs*.

Cette thèse se concentre sur l'étude de malware évasifs ciblant le système d'exploitation Windows. Une première contribution vise à évaluer l'efficacité de techniques d'évasion contre un panel d'antivirus. Nous proposons par la suite une contre-mesure visant à stopper l'exécution de malware utilisant ce genre de techniques d'évasion en simulant les modifications faites par un antivirus dans le système d'exploitation. Ces leurres sont créés via l'instrumentation de l'API Windows à l'aide de Microsoft Detours. Nous évaluons cette contre-mesure sur quelques exemples de malwares évasifs récupérés dans la nature.

Une seconde contribution a pour objectif de répondre à l'absence de dataset de malware évasifs. Pour cela, nous étiquetons un dataset de malware Windows existant de deux manières. Premièrement, à l'aide d'une détection automatique utilisant la contre-mesure issue de notre première contribution et deuxièmement, à l'aide d'un étiquetage collaboratif accessible publiquement.

Title : Evasive Malware Study : conception, protection and detection

Keywords : Malware, Evasive, Windows, Countermeasure, Dataset

Abstract : There is a permanent confrontation between malware and antiviruses, leading both parties to evolve continuously. On the one hand, the antiviruses put in place solutions that are more and more advanced and propose complex detection techniques in addition to the classic signature detection. Once a new malware is detected, the antiviruses conduct deeper analysis to extract a signature and keep their database quickly updated. This complexification leads the antiviruses to leave traces of their presence on the machine they protect.

On the other hand, malware authors willing to create long-lasting malware at a low cost can use simple techniques to avoid being deeply analyzed by these antiviruses. It is then possible for malware to search for the presence of traces or artifacts left by the antiviruses and then decide to execute or not their malicious payload. We define such software as being *evasive malware*.

This thesis focuses on the study of evasive malware targeting the Windows operating system. A first contribution aims at evaluating the efficiency of evasion techniques against a panel of antiviruses. We then propose a countermeasure designed to stop the execution of the malware using this kind of technique by simulating the presence of the modifications made by the antiviruses within the operating system. These decoys are created by instrumenting the Windows API using Microsoft Detours. Finally, we evaluate this countermeasure on a few samples of malware collected in the wild.

A second contribution aims at answering to the lack of a dataset of evasive malware. To do so, we label an existing dataset of Windows malware using two ways. First, with the help of an automatic detection that exploits the countermeasure we created, and second, thanks to collaborative labeling available on a public platform.