Semantic Malware Detection by Deploying Graph Mining [5]

Dylan Marinho

30 mars 2018

Rapport de synthèse de l'article *Semantic Malware Detection by Deploying Graph Mining [5]*, sous la direction de Jean Quilbeuf, équipe TAMIS, INRIA.

Résumé

Cet article porte sur la détection automatique de *malware*. Nous présenterons un système de détection reposant sur des graphes de comportement avec un taux de détection important.

Dans cette méthode, un logiciel est analysé dynamiquement dans une machine virtuelle pour observer son comportement et on étudie sur les relations qu'il entretient avec son environnement.

Mots clés : Détection de malware, Sous-graphes fréquents, Analyse sémantique , Appel système, Isomorphisme de graphes

1 Introduction

Donner une définition exhaustive de ce qu'est un *malware* n'est pas une tâche facile. Ce terme provient de la contraction de « *malicious software* » et est généralement utilisé pour décrire des logiciels endommageant le système ou ses données. Nous pouvons citer comme exemples les vers, les virus, les chevaux de Troie ou encore les logiciels espions (*spyware*). Leur mode de fonctionnement est aussi très varié : ils peuvent se propager à travers de *spam* ou en envoyer, être utilisés pour des attaques par déni de service (*DoS*) ou pour du vol de données, *etc*.. Cependant, ils revêtent également une dimension économique, comme avec du minage de *bitcoin* sur les machines infectées ou encore avec de l'espionnage industriel. Étudier de nouvelles méthodes de détection des *malware* est donc un des enjeux de la sécurité informatique.

Aujourd'hui, la plupart des détections de *malware* se fait en analysant la signature d'un logiciel. Celle-ci est une suite de bytes commune à un échantillon de *malware*. Elle sera donc présente dans les *malware* ou fichiers infectés et non dans les fichiers sains. Ainsi, la détection est une méthode statique, c'est à dire sans exécuter le logiciel, et peu coûteuse en temps de calcul. Cependant, les concepteurs de logiciels malveillants peuvent facilement la contourner : des techniques d'offuscation comme des réarrangement du code source leur permettent de passer pour des *goodware*.

Plusieurs méthodes s'offrent alors aux analystes. On peut tenter d'analyser le comportement d'un code binaire de façon statique. Une telle analyse est complexe et requiert d'importantes ressources de calcul car il faut explorer tous les états atteignables. C'est pourquoi on a favorisé un approche dynamique, c'est-à-dire qui exécute le logiciel.

L'algorithme présenté ici étudie les appels système effectués par le logiciel. En effet, en dehors de ces appels, il peut être difficile de comprendre le code exécuté, notamment s'il utilise des méthodes d'offuscation. Cependant, tout logiciel devra se conformer aux bibliothèques standard pour un certain nombre d'opérations, comme l'ouverture ou la fermeture d'un fichier. Cet algorithme générera un graphe représentant ces différents appels, ce qui permettra de différencier un *malware* d'un *goodware*.

Cet article présente tout d'abord une vision globale du système de détection, puis en détaillera le fonctionnement avant de procéder à la validation de la méthode mise en œuvre.

2 Vision globale du système

L'objectif de ce système est d'étudier un logiciel encore inconnu et de déterminer s'il s'agit d'un *malware*. Pour cela, il est exécuté au sein d'une machine virtuelle et la détection se fonde sur l'observation et l'enregistrement des interactions entre le logiciel et son environnement. Le principe général de ce système est présenté en figure 1.

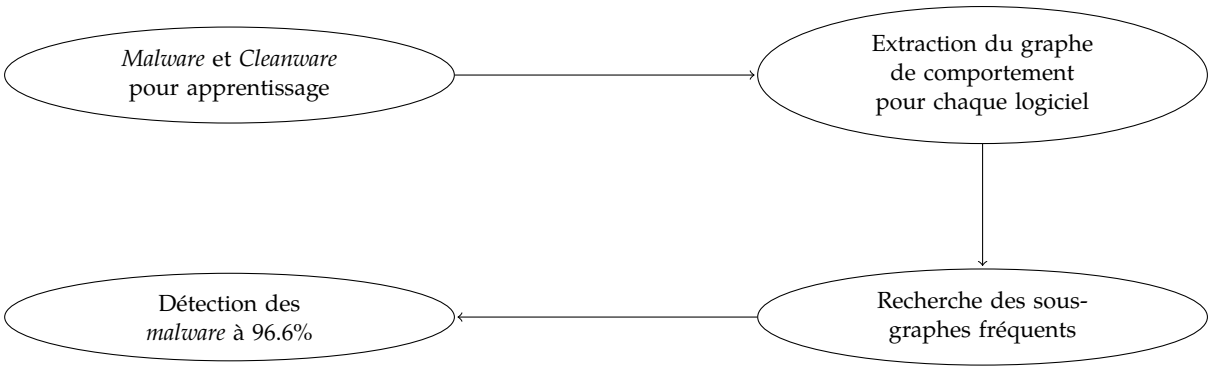

FIGURE 1 – Fonctionnement global du système

2.1 Modélisation de la sémantique

De nombreuses méthodes [1, 7] utilisent des séquences d'appels système afin de représenter le comportement d'un logiciel. Cependant, cette modélisation permet aux concepteurs de *malware* de contourner ces détections.

Le système présenté ici s'intéresse aux dépendances des données qui existent entre les appels système effectués par le logiciel. Le graphe d'étude de ce logiciel sera construit en définissant pour nœuds les appels système et en créant une flèche entre deux nœuds si la valeur de sortie de l'un est la valeur d'entrée de l'autre. Il n'est donc pas affecté par un réarrangement de leur ordre dans le code source.

2.2 Rendre cette modélisation plus efficace

Afin d'améliorer l'efficacité de ce système, nous ne nous intéresserons qu'aux appels *intéressants* (ce terme sera défini plus loin) pour cette analyse. De plus, nous ne nous considérerons pas les flèches représentant les valeurs 0 et 1. En effet, ces valeurs ne signifieraient que le succès ou l'échec d'un appel, et n'apporterait pas d'informations sur l'état du logiciel.

3 Détails de fonctionnement

3.1 Appels intéressants

Considérer toutes les bibliothèques et appels système utilisés par le logiciel serait trop coûteux et rendrait notre graphe de comportement bien trop imposant. De plus, nombre de ces éléments ne seraient pas utiles à l'analyse. Ainsi, seules six bibliothèques sont utilisées lors de la détection de *malware* : *kernel32.dll*, *user32.dll*, *ws_s32.dll*, *advapi32.dll*, *wininet.dll* et *CreateProcess.dll* [3].

Une étude a alors été menée afin de déterminer quels sont les types d'appels systèmes les plus utiles à la classification. Celle-ci a été faite sur une base de 400 *malware* et 397 *goodware* en utilisant une méthode de validation croisée [1] avec une classification créée par des forêts d'arbres décisionnels (*random forests*).

Cette étude a permis de définir les appels les plus significatifs pour notre système. Les opérations les plus courantes des *malware* sont donc la gestion de fichiers, l'accès aux informations du système et aux registres ainsi que le réseau. L'étude de ces différentes classes d'appels est de plus nécessaire, car elles sont utilisées de façon évidente par les *malware*. Par exemple, nombreux sont ceux qui s'assureront de fonctionner dans l'environnement à attaquer – ils liront donc les informations du système – et les registres contiennent de nombreuses données confidentielles en clair.

3.2 Quelques définitions

Revoir les defs. Nous utiliserons ici ces graphes non orientés avec des étiquettes.

Définition 1 (Graphe étiqueté)**.** Un graphe étiqueté est un $4-$uplet (S, A, E, e) avec :
- S l'ensemble des sommets ;
- $A \subseteq S \times S$ l'ensemble des arêtes ;
- E l'ensemble des étiquettes ;
- $e : S \cup A \to E$ la fonction d'étiquetage des sommets et des arêtes.

Définition 2 (Isomorphisme de graphes – isomorphisme de sous-graphes)**.** Un isomorphisme entre G et H est une fonction bijective $f : S_G \to S_H$ telle que :
- $\forall u \in S_G, e_G(u) = e_H(f(u))$;
- $\forall (u, v) \in A_G, (f(u), f(v)) \in A_H$;
- $e_G(u, v) = e_H(f(u), f(v))$.

Un isomorphisme de sous-graphes de G dans H est un isomorphisme de G dans un sous-graphe de H.

Définition 3 (Recherche de sous-graphes courants)**.** Soit $GS = \{G_i | i \in \{1, ..., n\}\}$ un ensemble de graphes, un graphe g et un support $s \in \mathbb{N}$. Considérons alors

$$\zeta(g, G) = \begin{cases} 1 & g \text{ est isomorphe à un sous-graphe de } G \\ 0 & g \text{ n'est pas isomorphe à un sous-graphe de } G \end{cases}$$

et ensuite $\sigma(g, GS) = \sum_{G \in GS} \zeta(g, G)$.

1. Il s'agit ici d'une $10-$*fold-cross-validation*. L'échantillon de départ est divisé en 10 sous-ensembles, on en choisit un qui sera l'échantillon de test et les neuf autres constitueront l'échantillon d'apprentissage. On répétera cette expérience 10 fois en changeant l'ensemble de test.

$\sigma(g, GS)$ décrit le nombre d'occurrences de g (à isomorphisme près) comme sous-graphes d'éléments de GS.

Rechercher les sous-graphes fréquents d'une collection consiste à trouver tous les graphes g tels que $\sigma(g, GS) \geq s$

3.3 Construction des graphes

Dans notre système, chaque programme sera associé à un graphe. Celui-ci est construit en considérant les appels systèmes intéressants qu'il effectue. Ceux-ci seront inscrits dans les nœuds du graphe. Ensuite, on reliera deux appels x et y si la valeur de sortie de x est une valeur d'entrée de y. On utilisera comme étiquette pour l'arête reliant x à y le nombre d'adresses mémoires uniques étant en entrée de l'un et en sortie de l'autre.

La figure 2 illustre ce principe. Dans cet exemple, l'adresse de sortie de CreateFileW et en valeur d'entrée de CloseHandle.

TABLE 1 – Exemple d'appels systèmes liés

Appel système	Paramètres	Valeur de retour
CreateFileW	$IpFileName : 0 \times 00415F2C$	$\mathbf{0 \times 000025A8}$
CloseHandle	$hObject : \mathbf{0 \times 000025A8}$	0×00000001

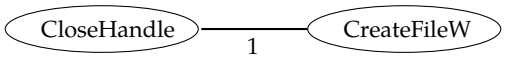

FIGURE 2 – Création d'un graphe basé sur la table 1

3.4 Recherche des sous-graphes fréquents

La recherche des sous-graphes fréquents est exécutée par l'algorithme gSpan (*graph-based substructure pattern mining*) [8]. Cet algorithme construit un ordre lexicographique sur les graphes en les associant à un parcours en profondeur (*DFS*). Il permet de trouver tous les sous-graphes fréquents d'une collection de graphes avec une certaine fréquence minimale.

gSpan sera exécuté avec un support (que l'on représente désormais par une fréquence) variant de 0,04 à 0,09. On cherchera donc les sous-graphes présents dans 4% à 9% des graphes de la collection.

4 Validation

Ce travail a été effectué sur 404 *malware* et 349 *goodware*, récoltés à partir de [6].

Afin de montrer la capacité de détection du système présenté, trois familles de *malware* ont été générées et sont représentés par la table 2. Ceux-ci sont utilisés car ils sont populaires selon les rapports de logiciels antiviraux [4].

2. Accès secret au logiciel inconnu de l'utilisateur.
3. Élément d'un programme permettant à un logiciel d'exploiter une faille de sécurité.

TABLE 2 – Répartition des *malware* de validation

Nom	Nombre
Constructor	188
Porte dérobée [2]	162
Exploit [3]	54

TABLE 3 – Nombre de retours de gSpan

Support	Nombre de sous-graphes fréquents
0,04	4187
0,05	1188
0,06	784
0,07	579
0,08	501
0,09	471

Il est important de noter que certains de ces logiciels utilisent un code polymorphique (c'est-à-dire un code fournissant une interface unique à des objets de types différents), ce qui rend la détection difficile pour un analyseur basé sur la signature.

De plus, il suffit d'exécuter un logiciel pendant 120 secondes pour récupérer une trace d'analyse suffisante. En effet, ce temps est suffisant pour que le *malware* exécute son attaque [2]. Ceux pour lesquels le temps d'attente serait plus long attendraient généralement une action extérieure pour démarrer sa partie offensive (comme une connexion réseau) [2].

Les échantillons d'apprentissage sont alors utilisés pour extraire les graphes de comportement. Ceux-ci sont utilisés comme entrées de gSpan afin d'obtenir les graphes courants. L'algorithme est lancé pour un support variant entre 0,04 et 0,09. Les résultats sont présentés par le tableau 3. Nous noterons que 0,09 est la valeur maximale de support retenue car au delà l'algorithme ne rend que des graphes ayant un nœud, ce qui n'est pas souhaitable ici.

Les sous-graphes fréquents obtenus sont alors utilisés pour l'apprentissage. Chaque sous-graphe trouvé par gSpan constituera une *feature* de la collection d'apprentissage : on lui associera 1 s'il est présent dans un *malware* ou un *goodware*, 0 sinon.

Enfin, une validation croisée (*10-fold cross validation*) avec une classification par forêts d'arbres de décision est utilisée afin d'évaluer la détection du système.

Comme on peut le voir sur le tableau 4, le système présenté obtient en moyenne un rappel de 96,6% pour un support de 0,09.

On constate alors que le nombre de faux négatifs a tendance à diminuer avec l'augmentation du support. De même, le rappel a tendance à augmenter et se situe entre 90% et 97%, même pour des faibles valeurs de support. La précision augmente également, jusqu'à atteindre 98,7%.

Le détail de la détection pour un support de 0,09 est détaillé dans le tableau 5. Ces données

4. Rapport entre les vrais positifs et les éléments pertinents (c'est-à-dire vrais positifs et faux négatifs).
5. Rapport entre les vrais positifs et les éléments renvoyés (c'est-à-dire vrais positifs et faux positifs).
6. Pourcentage par rapport aux nombre de *malware* à détecter (404).
7. $F - mesure = \frac{prcision \times rappel}{prcision + rappel}$

TABLE 4 – Détection du système (en pourcentages)

Support	Rappel [4]	Précision [5]	Faux négatifs [6]	F-mesure [7]
0,04	89,9	88,2	10,1	89,1
0,05	88,7	87,4	11,3	88
0,06	89,9	87,8	10,1	88,8
0,07	94,6	96,3	5,4	95,4
0,08	96,1	98,7	3,9	97,4
0,09	96,6	98,7	3,4	97,6

TABLE 5 – Détection du système, en valeur absolue, pour un support de 0,09.

Réel / Détection	Positifs	Négatifs	Total
	404	349	753
Positifs	Vrais positifs 390	Faux positifs 5	395
Négatifs	Faux négatifs 14	Vrais négatifs 344	358

ont été calculées à partir des données de l'article [5] : précision, rappel et nombre de *malware* et *goodware*. Cette étude a montré qu'il y a eu une confusion dans l'article original entre faux-positifs et faux-négatifs, corrigée ici.

5 Conclusion

La détection de *malware* est une tâche difficile car les concepteurs de logiciels malveillants ont de nombreuses techniques pour contourner les premiers systèmes de détection. Le système présenté ici permet de distinguer un code malveillant d'un code bénin par une analyse des graphes de comportement du logiciel. Celui-ci obtient un score de détection (rappel) de 96,6%.

Les graphes sont ici utilisés pour représenter le comportement des logiciels car ils permettent de représenter de façon simplifier des relations complexes entre des données.

Une prochaine étape pour ces travaux serait de déterminer quel sont les sous-graphes les plus intéressants afin de distinguer un *malware* d'un *goodware*. On pourrait aussi s'intéresser à la diminution des *features* pour l'apprentissage.

Références

[1] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for unix processes. *IEEE Symposium on Security and Privacy*, 1996.

[2] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specification from suspicious behaviors. *IEE Symposium on Security and Privacy*, 2010.

[3] http ://msdn.microsoft.com/.

[4] A. Inokuchi, T. Washio, and H. Motoda. Frequent substructure from graph data. *PKDD2000*, 2000.

[5] F. Karbalaie, A. Sami, and M. Ahmadi. Semantic malware detection by deploying graph mining. *IJCSI International Journal of Computer Science Issues*, 9(3), Janvier 2012.

[6] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamze. Malware detection based on mining api calls. *SAC'10 March 22-26*, 2010.

[7] D. Wagner and D. Dean. Intrusion detection via static analysis. *IEEE Symposium on Security and Privacy*, 2001.

[8] X. Yan and J. Han. gspan : Graph-based substructure pattern mining. *IEE Symposium on Security and Privacy*.