Binacle : indexation « full-bin » de fichiers binaires pour la recherche et l'écriture de signatures Yara

Guillaume Jeanne
guillaume.jeanne@ssi.gouv.fr

ANSSI - Bureau Failles et Signatures

Résumé. Binacle (contraction de binaire et oracle) est une base de données probabiliste écrite en Rust et conçue pour indexer le contenu entier de fichiers binaires. Les bases de données classiques permettent d'indexer les chaînes de caractères pour des recherches *full-text* mais pas directement les données binaires. Un des domaines où cette fonctionnalité s'avère utile est l'analyse de codes malveillants. En effet, l'analyste est souvent amené à se demander s'il existe d'autres codes qui contiennent une séquence d'octets observée dans un code. Plusieurs expérimentations avec des bases de données classiques ont montré la nécessité de développer une base spécialisée pour ce type d'indexation. L'objectif de ce document est de présenter l'état de l'art dans le domaine, puis le fonctionnement interne de Binacle ainsi que des résultats obtenus expérimentalement pour conclure sur les limitations de cette approche et les améliorations possibles.

1 Introduction

Les CERTs maintiennent des collections de plusieurs milliers de codes malveillants. Certains outils, comme Polichombr [3], permettent de capitaliser et labelliser ces codes par famille afin de les retrouver plus facilement. Les analystes sont régulièrement confrontés au problème d'identifier s'il existe d'autres codes qui contiennent une séquence d'octets particulière (comme un domaine, une IP, un nom de fichier, une routine de déchiffrement) provenant d'un code malveillant. Ces ressemblances peuvent permettre de trouver de nouvelles variantes d'un code donné ou d'identifier deux familles de codes proches, peut-être issues du même groupe d'attaquants.

La plupart des bases de données classiques (MySQL, PostgreSQL, SQL Server, ElasticSearch...) offrent la possibilité de créer des index *full-text*, qui permettent de rechercher efficacement la présence de chaînes dans un document texte. C'est le principe utilisé par les moteurs de recherche. En général, ces bases fonctionnent grâce à un « index inversé » qui découpe les chaînes en mots clés et qui associe chaque mot clé avec la liste des

documents qui le contiennent. Aucune de ces bases ne fournit la possibilité d'indexer du code binaire en mode *full-bin*.

Contrairement au texte, où il est simple d'effectuer un découpage en un nombre restreint de mots clés du dictionnaire avec un caractère de séparation naturel (l'espace, par exemple la chaîne « Je soumets au SSTIC » pourra être découpée en `['Je', 'soumets', 'au', 'SSTIC']`), le code binaire ne permet pas de découpage universel et il est composé d'un nombre de mots potentiellement bien plus élevé. De plus, dans le cas du texte, certains mots clés comme 'je' ou 'le' peuvent être exclus, car ils n'interviennent pas dans la différenciation sémantique du document. Dans le cas du binaire, il faut a priori considérer l'ensemble des séquences possibles, ce qui conduit à des index beaucoup plus volumineux.

L'approche proposée dans ce document consiste à indexer le code binaire grâce à une base de données spécialisée qui repose sur l'utilisation d'un index inversé de n-grammes. Chaque document (code malveillant ou code sain) est associé à un identifiant sur 4 octets (ce qui permet d'indexer jusqu'à 4 294 967 296 documents par base). L'objectif est de pouvoir associer rapidement une séquence d'octets à la liste des documents qui la contienne. L'association entre l'identifiant d'un document et ses caractéristiques (sha256, nom...) est réalisée en amont de manière naturelle par les outils de capitalisation de code (par exemple dans Polichombr, chaque code est déjà associé à un identifiant unique en base). On supposera que la liste des identifiants des fichiers insérés est chronologiquement croissante (cas des bases de données classiques), ce qui permettra quelques optimisations afin de réduire l'espace mémoire demandé. Les contraintes techniques sont les suivantes :

- l'insertion d'un nouveau fichier dans la base doit s'effectuer rapidement, idéalement dans un temps inférieur à une seconde ;
- la recherche de tous les fichiers contenant une séquence d'octets donnée doit s'effectuer très rapidement ;
- l'espace mémoire occupé doit être aussi faible que possible. Ainsi, le facteur entre la taille induite par l'indexation et la taille des données « brutes » indexées doit être minimisé ;
- la scalabilité, c'est-à-dire qu'une machine standard doit être capable de faire tourner la base pour un « petit » nombre de documents (de l'ordre de 10 000) et que le nombre maximum de documents gérés doit être proportionnel aux caractéristiques de la machine ;
- enfin, il n'est pas nécessaire que la base permette de supprimer un fichier inséré. Cette propriété va permettre d'utiliser des structures de données plus simples et plus efficaces.

L'objectif de ce document est de présenter le fonctionnement interne de l'outil et les choix architecturaux, puis de donner des résultats obtenus expérimentalement. Enfin, l'utilité de cette base sera illustrée avec des cas concrets tels que l'accélération de scans Yara grâce à une technique de pré-filtrage ou la génération automatique de signatures en identifiant des séquences propres à certaines familles de code.

1.1 État de l'art

Initialement, plusieurs essais ont été faits avec des bases de données classiques pour tenter de répondre au problème. Le premier utilisait une base MySQL avec une table contenant deux champs « `ngram` » (`integer`) et « `file id` » (`integer`) avec un index sur le champ `ngram`. Cette solution fonctionne pour indexer quelques fichiers mais la taille de la base augmente de plus en plus et la vitesse de recherche est de plus en plus lente, à partir de quelques centaines de fichiers.

La seconde expérimentation concerne LMDB [4] (Lightning Memory-Mapped Database). LMDB est une base de donnée clé-valeur réputée pour ses performances. Ici les clés sont les n-grammes et les valeurs sont les identifiants des fichiers. Les performances s'avèvent bien meilleures qu'avec MySQL mais l'occupation mémoire est trop importante, la base pèse environ 3 fois la taille des données. Néanmoins plusieurs idées provenant de LMDB ont été reprises pour Binacle, notamment le fait que la base soit projetée directement en mémoire afin d'accélérer les I/O.

Durant la phase de développement de Binacle, le projet BigGrep [2] a été découvert. L'objectif de BigGrep est de fournir des fonctionnalités similaires à celles de Binacle pour des contraintes comparables en terme d'insertion, de recherche et d'occupation mémoire. BigGrep confirme les analyses concernant l'utilisation des bases de données classiques (leurs expérimentations concernent les bases PostgreSQL, Tokyo Cabinet, MongoDB et Redis). Néanmoins il possède plusieurs inconvénients pour lesquels des améliorations sont proposées :

- L'obligation d'insérer les fichiers par lot de plusieurs milliers de fichiers simultanément, un nouvel index étant créé à chaque mot. La recherche doit s'effectuer sur tous les index en parallèle. Dans le cadre de ses activités, le Bureau Failles et Signatures de l'ANSSI est amené à indexer un petit nombre de fichiers simultanément et à rechercher immédiatement dedans. La construction d'un nouvel index pour chaque insertion serait inefficace.

- L'impossibilité de mettre à un jour un index. Une fois qu'un index est créé, il ne peut plus être modifié. Notre base permet une modification incrémentale de l'index afin d'exploiter au mieux les capacités de la machine (notamment sa quantité de RAM).
- Une approche différente en fonction de la taille du fchier, les gros fichiers sont insérés dans un index via leurs 4-grammes et les autres dans un index différent avec leurs 3-grammes. L'approche que nous proposons repose sur un modèle hybride qui permet l'indexation avec des n-grammes de taille quelconque (en bits), en particulier, des 3,5-grammes (n-grammes de taille 28 bits) fournissent un bon compromis performance/stockage.

2 Fonctionnement

Binacle est implémenté dans le langage Rust. À l'instar de LMDB, il repose sur un fichier projeté en mémoire afin d'accélérer les lectures et écritures aléatoires dans le fichier et de profiter du système de cache fourni par le système d'exploitation.

2.1 Pourquoi Rust ?

Rust est un langage multi-paradigme développé par Mozilla Research, il supporte les styles de programmation fonctionnel, procédural et orienté objet. L'objectif de Rust est de proposer un langage sécurisé et performant. Sa philosophie est de proposer des abstractions syntaxiques qui sont vérifiées par le compilateur, telles qu'un typage statique strict, mais qui disparaissent dans le code compilé pour gagner en performances. Lorsque le compilateur ne peut prouver une assertion de manière statique, par exemple que l'accès à un tableau ne se situe pas au delà de son espace mémoire, il ajoute une assertion qui sera vérifiée à chaque fois que le code s'exécutera.

Rust garantit l'absence de vulnérabilités de type corruption mémoire, comme les *buffer overflows* ou les *use-after-free*, grâce à un modèle mémoire spécifique : chaque objet ne peut être détenu que par un unique élément. Ce modèle permet de s'affranchir de la présence d'un ramasse-miettes (*garbage collector*). Par exemple si une fonction détient une instance d'une classe, lorsque la fonction aura fini de s'exécuter, l'objet sera automatiquement libéré, à moins que cette fonction ne transmette l'appartenance, en retournant l'objet.

Le langage Rust a été choisi pour implémenter Binacle pour les raisons suivantes :

- l'implémentation facilitée grâce aux structures fournies en standard et aux nombreuses bibliothèques disponibles (les « crates ») qui sont facilement installables ;
- la portabilité entre Windows et Linux ;
- les performances proches du C [1] ;
- la gestion mémoire bas-niveau possible qui permet de maîtriser la gestion mémoire, par exemple en écrivant directement à une adresse virtuelle de la mémoire ;
- la robustesse et les garanties de sécurité intrinsèques au langage ;

2.2 Implémentation et détails techniques

L'implémentation de Binacle repose sur une structure de table de hachage. Chaque n-gramme est associé à une liste qui contient les identifiants de tous les documents qui contiennent ce n-gramme (c'est le principe des index inversés utilisée pour la recherche *full-text* mais avec des mots clés de taille fixe plutôt qu'en utilisant un séparateur naturel). Il y a un donc $2^{S_{ngram}}$ listes (pour des n-grammes sur 28 bits, $S_{ngram} = 28$, ce qui représente 268 435 456 listes). Pour des raisons d'occupation mémoire, les listes sont initialement petites (32 octets) et de nouvelles listes deux fois plus grandes sont allouées à chaque fois que les premières sont pleines. Ce compromis permet de limiter l'occupation mémoire des n-grammes peu présents. Lorsque qu'une nouvelle liste est alloué, celle-ci garde un pointeur sur l'ancienne liste, ce qui permettra ultérieurement de parcourir l'ensemble des listes pour un n-gramme.

L'en-tête de la base permet de faire la correspondance entre un n-gramme et la liste qui lui est associée. La taille de l'en-tête varie en fonction de la taille des n-grammes (S_{ngram}, en bits) et de la taille des offsets ($S_{offsets}$, en octets) selon la formule ci-dessous. Les offsets sont relatifs au début du fichier projeté, ils permettent de calculer les pointeurs qui font correspondre un n-gramme avec sa liste.

$$S_{en-tete} = S_{offset} * 2^{S_{ngram}}$$

S_{offset} et S_{ngram} sont des paramètres qu'on peut choisir librement (sous certaines contraintes) dans le code implémenté. $S_{offset} = 5$ convient dans la plupart des cas. Le tableau suivant donne la taille de l'en-tête en fonction de la taille des n-grammes pour $S_{offset} = 5$.

S_{ngram}	S_{offset}	$S_{en-tete}$
26 bits	5 octets	335 Mo
28 bits	5 octets	1,34 Go
30 bits	5 octets	5,36 Go
32 bits	5 octets	21,47 Go

Pour que les insertions de documents restent rapides, il est nécessaire que la base réside en RAM. On voit que des n-grammes de 28 bits représentent un bon compromis pour un en-tête de taille raisonnable sur une machine standard. Par contre on perdra en précision lorsqu'on fera une recherche dans la base. Des n-grammes de plus petite taille augmenteront le taux de faux positifs.

2.3 Insertion d'un document dans la base

Lorsqu'on souhaite insérer un fichier dans la base, on découpe le fichier en n-grammes de taille souhaitée avec une fenêtre glissante de 1 octet. Un fichier de n octets contient n-x n-grammes où x est la taille des n-grammes en octets, arrondie au supérieur. Pour chacun de ses n-grammes, on récupère via l'en-tête un pointeur sur la liste qui lui est associé puis on insère dans ce dernier l'identifiant du fichier. Si la liste est pleine, on alloue une nouvelle liste à la fin du fichier qui pointe sur l'ancienne liste et on met à jour l'en-tête pour indiquer que la liste courante est la nouvelle liste. La figure 1 récapitule ce principe.

Compression des identifiants Afin d'occuper moins de place en mémoire, les listes d'identifiants sont compressées avec une variante de l'algorithme `VariableByte`, spécialement adapté pour cette situation. L'algorithme `VariableByte` consiste à représenter un entier grâce à un nombre d'octets variable. Pour chaque octet lu, un bit indique s'il est nécessaire de lire le prochain octet pour décoder le nombre. On a donc sept bits utiles par octet. Par exemple, les nombres de 1 à 127 sont représentés sur un octet, ceux de 128 à 16383 sur deux octets, etc. Dans le cas présent, on n'encode pas directement l'identifiant mais sa différence avec l'identifiant précédent. Par exemple si deux fichiers avec deux identifiants successifs comportent le même n-gramme, ce qui arrive fréquemment en pratique puisqu'on a tendance à insérer simultanément des codes malveillants de la même famille, alors l'encodage de l'identifiant du second fichier n'occupe qu'un octet en mémoire au lieu de quatre. Dans cette implémentation, les listes sont optimisées pour permettre d'insérer un nouvel identifiant sans devoir recalculer toute la liste, ce qui garantit une insertion en temps

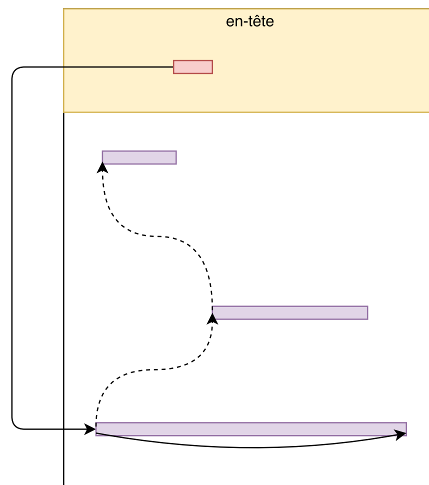

Fig. 1. Schéma de la base de données. Pour un n-gramme, on lit l'offset de la liste correspondant dans l'en-tête (en rouge), la liste (en violet) contient un pointeur sur son dernier élément pour insérer rapidement à la fin et un pointeur sur la liste précédente (en violet également).

constant. Cet algorithme permet d'économiser jusqu'à trois quarts de mémoire.

D'autres algorithmes de compression plus efficaces existent, par exemple `Patched Frame of Reference` (PFOR) qui utilise une variante de `VariableByte` mais il est moins adapté à la situation, notamment pour insérer sans devoir recalculer toute la liste.

Résultats expérimentaux d'insertion Le tableau ci-dessous représente les temps d'insertion de 119 182 fichiers dans la base en fonction de la taille des n-grammes. Ils proviennent de l'ensemble des fichiers présents dans `C:\Windows` pour Windows 10, ce qui représente 15,7 Go de données à insérer. La machine de test est un PC de bureau standard équipé d'un Core i7-4770 (3.40Ghz), 32 Go de RAM et d'un disque dur mécanique. Les tests sont effectués sur Windows 10.

S_{ngram}	Temps	T_{base}	Vitesse
24	27 min	9 Go	10.1 Mo/sec
26	35 min	15 Go	7.1 Mo/sec
28	39 min	17 Go	8.0 Mo/sec
30	1h27	75 Go	3.2 Mo/sec

Bien que le nombre de n-grammes total soit le même dans les différents cas, les performances diminuent lorsqu'on augmente la taille des n-grammes. Ceci est dû au nombre de listes qui augmente linéairement avec le nombre

de n-grammes possibles. Pour $S_{ngram} = 30$, la base dépasse la taille de la RAM, dans ce cas un nouvel index est créé et l'ancien est écrit sur le disque mécanique, expliquant la dégradation des performances. Celle-ci ne devrait pas avoir lieu si le système possède suffisament de mémoire.

2.4 Recherche dans la base

Pour rechercher une séquence dans la base, il faut décomposer la séquence en un ensemble de n-grammes. Par exemple la chaîne « Windows » devient l'ensemble `["Wind", "indo", "ndow", "dows"]`. La seconde étape consiste à extraire depuis la base le nombre d'identifiants de fichier pour chaque n-gramme. Par exemple : `[("Wind", 23842), ("indo", 1931), ("ndow", 293847), ("dows", 2918)]`. Cette étape s'effectue rapidement car ce nombre est présent dans l'en-tête des listes correspondant à chaque n-grammes. L'ensemble est ensuite classé par ordre croissant : `[("indo", 1931), ("dows", 2918), ("Wind", 23842), ("ndow", 23847)]`. On intersecte enfin les différentes listes d'identifiants une à une par ordre croissant. La liste résultante contient tous les fichiers qui contiennent l'ensemble des n-grammes.

Cette recherche est probabiliste car elle ne garantit pas que l'ensemble des fichiers retournés contient effectivement la séquence au complet (faux positifs). Par contre elle garantit l'absence de faux négatif, c'est à dire que si un fichier contient la séquence, alors il sera dans la liste. Pour confirmer et supprimer les faux positifs, il est possible d'aller vérifier si la séquence est bien présente dans le fichier, par exemple avec la fonction `memmem`.

Résultats expérimentaux de recherche On mesure expérimentalement les temps de recherche pour différentes séquences pour $S_{ngram} = 28$, sur les 119 182 fichiers insérés précedemment. Les temps affichés ne tiennent pas compte de l'étape de vérification nécessaire pour s'assurer que la séquence est effectivement présente.

Nb ngrammes	Séquence	Nb résultats	Temps	Nb vrai match
1	0x12345678	538	0.60 sec	23
4	windows	30164	0.94 sec	25187
11	GetProcAddress	10006	1.56 sec	9991
13	448D4201E8EBF...	5	1.57 sec	2
16	a long ascii string	5	1.39 sec	0

Conformément aux attentes, plus la taille de la séquence à rechercher augmente, plus le nombre de faux positifs est réduit. La recherche est

rapide dans tous les cas. La mesure exacte du temps est complexe à cause de la mise en cache par le système d'exploitation des pages du fichier de la base de données.

3 Cas d'utilisation

Binacle permet d'obtenir la liste des documents qui comportent une séquence quelconque d'octets. Deux cas d'usage concrets peuvent tirer parti de cette fonctionnalité : l'accélération du scan Yara grâce à une technique de pré-filtrage et la génération automatique de règles grâce à l'identification de séquences propres à certaines familles de code.

Yara [5] est un langage de signature créé par l'équipe de VirusTotal afin d'aider à l'identification et la classification des codes malveillants. Une règle Yara est une expression logique qui fait intervenir un ou plusieurs types de motifs. Les motifs peuvent être des chaînes de caractères, des chaînes hexadécimales ou des expressions rationnelles. L'outil Yara permet de parcourir un répertoire afin de trouver tous les fichiers qui valident une règle donnée. Dans cette partie, nous verrons comment Binacle permet d'accélérer les recherches de règles Yara, puis comment il peut être utile pour générer automatiquement des règles.

3.1 Accélération du scan Yara

Une règle Yara se présente de la manière suivante :

```
rule name_example_rule
{
    meta:
        description = "This is just an example"

    strings:
        $a = {6A 40 68 00 30 00 00 6A 14 8D 91}
        $b = {8D 4D B0 2B C1 83 C0 27 99 6A 4E 59 F7 F9}
        $c = "UVODFRYSIHLNWPEJXQZAKCBGMT"

    condition:
        ($a or $b) and not $c
}
```

Listing 1. Règle Yara

Cette règle possède 3 chaînes, `$a`, `$b` et `$c`, les deux premières sont des séquences d'octets alors que la troisième est une chaînes de caractères. La condition `($a or $b) and not $c` définit comment les chaînes interviennent pour savoir si un fichier correspond à cette règle. Ici, il suffit que

l'une des 2 chaînes `$a` ou `$b` soit présente dans le fichier et que `$c` soit absente. Il serait possible d'exprimer d'autres conditions telles que « au moins deux des chaînes doivent être présentes » ou « les chaînes `$a`, `$b` et `$c` doivent toutes être présentes en 1 seul exemplaire ».

Le principe d'accélération des scans consiste à extraire des n-grammes depuis les séquences et à rechercher les fichiers qui les contiennent. L'ensemble des fichiers qui est retourné est un sur-ensemble (qu'on espère proche) de l'ensemble des fichiers qui vérifient la règle Yara. Au lieu de faire passer cette règle sur l'ensemble des fichiers, on l'applique uniquement au sur-ensemble obtenu. Certaines règles Yara peuvent être longue à scanner. Le fait de disposer d'un pré-filtrage rapide peut permettre d'accélérer le scan de manière drastique dans certains cas.

Résultats expérimentaux de scan Yara Le cas le plus favorable est lorsque qu'aucun fichier n'est renvoyé, dans ce cas on est sûr que la règle Yara ne matche aucun fichier de la base. À l'inverse, si un grand nombre de fichiers est retourné, il faut faire passer la règle sur les fichiers pour vérifier que ce ne sont pas des faux positifs. Le tableau suivant donne quelques examples pour 3 règles Yara (disponibles en annexe) sur les 119 182 fichiers précédents. T_{yara} représente le temps mis par l'outil Yara pour scanner tous les fichiers, $T_{binacle}$ celui mis par Binacle seul puis celui en tenant compte de la phase de vérification avec Yara.

Règle Yara	Résultats	T_{yara}	Nb binacle	$T_{binacle}$	%
Règle 1	25 187	56,20 sec	30 164	1,07 sec -> 17,31 sec	324%
Règle 2	0	9,9 sec	0	0.40 sec -> 0.40 sec	2475%
Règle 3	2	10,3 sec	6	1.8 sec -> 2.0 sec	572%

3.2 Génération automatique de signatures Yara

On choisit un ensemble de fichiers similaires sur lesquels on veut générer une signature Yara. La première étape consiste à extraire les séquences de code binaire commun à ces fichiers. Cette opération est réalisée en interne grâce à un outil appelé `bincmp`, développé spécialement pour cette tâche. Les séquences obtenues sont des candidats pour la génération d'une règle Yara, reste à savoir si ces séquences sont uniquement représentatives de l'ensemble de fichiers qu'on a sélectionné. Binacle intervient à ce moment, la recherche de sous-chaînes parmi les candidats permet d'extraire des chaînes uniques de taille minimale qui caractérisent ces fichiers ou, au contraire de montrer que d'autres codes possèdent ces chaînes-là, donc

que le candidat n'est pas un bon représentant de la famille et que la règle Yara risque de générer des faux-positifs.

4 Limitations et travail restant

4.1 Propriétés ACID

Binacle ne supporte pas les propriétés ACID (atomicité, cohérence, isolation et durabilité) des bases de données traditionnelles. Ce choix délibéré permet d'améliorer les performances et de simplifier l'implémentation au détriment de quelques bonnes pratiques à respecter. La plus contraignante est l'impossibilité d'insérer de manière parallèle sur une même base, cette propriété est garantie grâce à la possibilité de verrouiller des fichiers sur le disque. Ainsi deux instances de Binacle ne peuvent pas accéder simultanément en écriture à la même base, la seconde attendra que la première ait terminé. Par contre, plusieurs instances peuvent avoir accès en lecture seule à la base à condition qu'aucune écriture ne soit en cours.

Néanmoins, il est possible de profiter des capacités multi-cœurs de nos processeurs en créant autant de bases (c'est à dire autant de fichiers physiques) que de cœurs sur la machine et en distribuant équitablement les tâches d'insertion dans ces bases. La recherche pourra être faite en parallèle, elle devra s'effectuer sur l'ensemble des bases.

Les changements d'architecture permettant de bénéficier de ces propriétés tout en gardant des performances honorables sont en cours de réflexion.

4.2 Interface

Binacle est un outil destiné à être intégré avec les autres outils de capitalisation du Bureau Failles et Signatures de l'ANSSI. En conséquence, sa vocation n'est pas de proposer une interface de type web, ni même de retenir quel fichier correspond à quel identifiant. Cette fonctionnalité est néanmoins fournie à titre de démonstration pour pouvoir utiliser l'outil sans interdépendances avec d'autres outils.

Remerciements Je remercie David Lesperon pour la relecture de cet article.

A Règles Yara

```
rule yara1
{
  strings:
    $a = "windows"
  condition:
    $a
}
```

Listing 2. Règle Yara 1

```
rule yara2
{
  strings:
    $a = { FE 36 [0-8] F4 23 15 82 A3 04 45 }
  condition:
    $a
}
```

Listing 3. Règle Yara 2

```
rule yara3
{
  strings:
    $b = "Microsoft Corporation"
    $c = { 00 48 8B CF E8 7B 0D FF FF 44 0F B7 00 49 8B 04 }
  condition:
    all of them
}
```

Listing 4. Règle Yara 3

Références

1. Gouy, Isaac. The computer language Benchmarks game. `https://benchmarksgame.alioth.debian.org/u64q/rust.html`, .
2. Jin, W. ; Hines, C. ; Cohen, C. ; Narasimhan, P., . BigGrep. `https://github.com/cmu-sei/BigGrep`, 2012.
3. Le Berre S., Chevalier A., Pourcelot T. Polichombr. `https://github.com/ANSSI-FR/polichombr`, 2016.
4. Symas. LMDB. `https://symas.com/products/lightning-memory-mapped-database/`, .
5. VirusTotal. Yara. `http://virustotal.github.io/yara/`, .