AnoMark
-
Détection d'Anomalies dans des lignes de commande à l'aide de Chaînes de Markov

Alexandre Junius
alexandre.junius@ssi.gouv.fr

Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI)

Résumé. AnoMark est un algorithme de *Machine Learning* utilisant des méthodes de NLP (ou TAL : Traitement Automatique des Langues) pour analyser les lignes de commandes remontées à chaque création de processus dans les journaux d'événements d'un système d'information. En s'appuyant sur une décomposition en *n-grams* (sur les lettres composant la ligne de commandes), AnoMark entraîne un modèle statistique basé sur une chaîne de Markov. Ce dernier permet ensuite de calculer un score de vraisemblance de nouvelles lignes de commande, et d'en extraire les plus anormales vis-à-vis de l'activité passée.

1 Introduction

1.1 Contexte de création

L'étude des journaux d'événements système est une opportunité de détecter des intrusions sur des parcs informatiques, et peut se faire de plusieurs manières. Pour des comportements connus on peut détecter des indicateurs de compromissions dans les journaux, ou créer des règles plus complexes comme avec le format des règles SIGMA par exemple. Mais on peut aussi détecter des comportements inconnus grâce à des mécanismes de détection d'anomalies.

En apprentissage statistique (ou plus généralement en *Machine Learning*), la détection d'anomalies est un domaine à part entière. Ici, nous utiliserons un type de détection d'anomalies appartenant au sous-ensemble de l'apprentissage non supervisé, ce qui signifie que l'apprentissage se fait sans connaissance préalable sur les groupes que l'on cherche à isoler. Sans cette connaissance, on crée différents groupes de comportements à partir de mesures mathématiques d'éloignement de données. Ainsi, une anomalie peut être par exemple temporelle, dans le cadre de l'étude de séries temporelles, lorsqu'on observe un pic de données à une heure de la

journée inhabituelle. Mais on peut aussi définir d'autres types d'anomalies, dont de langage grâce à des modélisations d'habitude d'écriture. Or, les différentes informations des journaux d'événements système sont des données textuelles, car elles sont au départ destinées à être traitées par des humains. C'est le point de départ du présent travail, qui consiste à déterminer des anomalies dans les lignes de commandes remontées par certains événements en s'appuyant sur un modèle d'apprentissage du langage. Nous chercherons à produire des alertes de sécurité à partir de ces anomalies.

1.2 Journaux d'événements concernés

Les journaux d'événements qui nous intéressent ici sont les événements de création de processus. Cela correspond aux journaux `Windows Security 4688`, et aux journaux `Sysmon 1`. Tous deux contiennent un champ `Command Line` qui contient la plupart du temps le nom de l'exécutable et les arguments qui lui ont été passés. Pour être précis, le champ `Command Line` contient la ligne de commande une fois interprétée. Ce champ n'est cependant pas activé par défaut dans les 4688, il est nécessaire d'appliquer une politique de journalisation spécifique. De plus, il est à noter que si le présent travail s'est concentré sur des journaux Windows, la même information peut être retrouvée dans les journaux Auditd sous Linux.

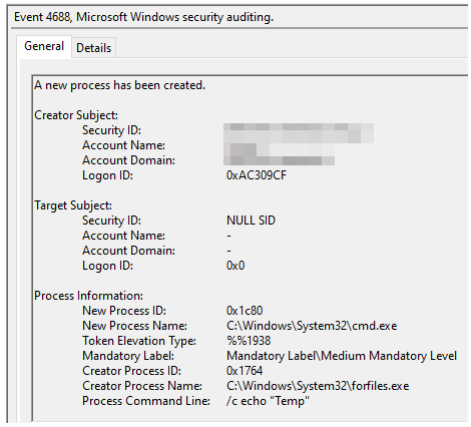

Fig. 1. Un journal d'événement Windows Security 4688

On retrouve aussi d'autres champs d'intérêt dans ces journaux, tels que : *timestamp*, *SubjectUserName*, *Computer*, *NewProcessName*, *Parent-*

ProcessName. Ils pourront nous servir à enrichir l'information que l'on remonte aux côtés d'une ligne de commande anormale.

1.3 État de l'art

En matière d'algorithmes de *Machine Learning* pour la production d'alertes sur des données d'événements de sécurité informatique la supervision réseau est aujourd'hui la plus en avance. Les données issues de cette supervision se prêtent par nature plus facilement à l'analyse statistique, car il est plus aisé de définir des variables numériques pour nourrir des algorithmes. Comme nous l'avons précisé plus haut, l'analyse sur des données système présente un enjeu différent, lié au type de nos données. Cependant, certaines informations de données réseau sont aussi textuelles. On peut donc penser que certains algorithmes choisis peuvent s'appliquer à la fois aux détections réseaux et systèmes.

En détection d'anomalies de langage sur des données réseau on retrouve notamment la détection automatisée de DGA (*Domain Generation Algorithm*), qui est la plus similaire à ce que l'on cherche à faire dans la présente étude. Il s'agit d'entraîner un modèle de langage sur des noms de domaines connus, puis de mettre un score de vraisemblance sur d'autres noms de domaines. Les chercheurs utilisant des algorithmes de *Machine Learning* pour la recherche de DGA ont obtenu des très bons résultats, qui sont le fait du caractère aléatoire des DGA. Ils utilisent notamment des modèles habituellement utilisés pour la génération automatique de texte (cf. [6]).

La génération de texte est un ensemble de méthodes appartenant au domaine du NLP (*Natural Language Processing*). Aujourd'hui l'application la plus connue est l'auto-complétion, comme sur nos claviers de téléphone portable. A partir d'un corpus de textes d'entraînement donné, ces modèles permettent de prédire lorsqu'on rédige une phrase le prochain mot ayant le plus de chances d'être écrit. Il existe plusieurs modélisations possibles pour arriver à ce but. Cela revient généralement pendant la phase d'apprentissage à *enregistrer* les habitudes d'écriture en déterminant les probabilités de transition d'un contexte de phrase au mot suivant.

Aujourd'hui les modèles les plus connus, comme *BERT* de Google, sont construits sur des architectures complexes telles que les *transformers*. Ces modèles de *Deep Learning*, bien que performants, peuvent être lourds à mettre en place. Ils reposent en effet sur des couches *encoder-decoder*, architectures de réseaux de neurones très utilisées pour le NLP. Si l'on schématise, ces modèles apprennent à compresser l'information de chaînes de caractères, puis à la décompresser. De cette manière on peut choisir

plusieurs tâches de traitement du langage une fois l'entraînement suffisant, comme la traduction ou la détection de sentiments dans des textes. Mais cela suppose de construire un *cluster* de calcul puissant, de préférence à partir de GPU (*Graphics Processing Unit*, processeurs graphiques). Leur avantage marginal face à des modèles plus simples devient intéressant dans des cas d'application précis, demandant par exemple une grande précision pour la prise de décision. Dans le présent travail nous choisissons de restreindre la complexité théorique de notre modèle statistique pour garder en intelligibilité, en efficacité algorithmique, et en portabilité. Ces restrictions permettent de convenir à des applications rapides pour des besoins opérationnels.

L'application de ces modèles d'apprentissage du langage aux lignes de commande issues des journaux d'événements `Windows Security 4688` ou des `Sysmon 1` n'est, à notre connaissance, pour le moment pas courante. Pourtant, au sein d'infrastructures Windows celles-ci sont très souvent stéréotypées, ce qui encourage à penser que la détection d'anomalies est une bonne piste pour détecter des intrusions. En effet, l'immense majorité des démarrages de processus sont le fait d'une exécution par un programme, ce qui produit des lignes de commande caractéristiques.

2 Modélisation

Nous fournissons dans cette partie des explications quant à la théorie mathématique et algorithmique qui sous-tend AnoMark. Nous allons dans un premier temps nous concentrer sur le principe du découpage en *n-grams*, puis dans un second temps sur la modélisation en chaînes de Markov, pour enfin donner la formule de calcul du score de vraisemblance.

2.1 N-grams

Le terme *n-grams* désigne un principe de découpage de chaînes de caractères selon un certain nombre de n lettres ou n mots. Dans le cas de l'étude des lignes de commande, nous choisissons de découper nos chaînes de caractères en *n-grams* de lettres. En pratique considérons la ligne de commande suivante :

```
cmd.exe arg
```

Un découpage en *6-grams* donne la liste suivante :

```
["cmd.ex", "md.exe", "d.exe ", ".exe a", "exe ar", "xe arg"]
```

Ce découpage nous permet de représenter une ligne de commande comme une suite *d'états* entre lesquels on transitionne. Ces transitions entre états ouvrent dès lors la possibilité d'une modélisation en chaîne de Markov.

2.2 Chaînes de Markov

L'expression "chaîne de Markov" fait référence à un concept mathématique permettant de modéliser les transitions entre états indépendamment du passé. C'est un processus stochastique [1] dont la prédiction du futur à partir du présent n'est pas rendu plus précise par le passé. Derrière ces aphorismes mathématiques se cache une théorie plutôt simple.

Par exemple, on peut considérer que les actions d'un chat pendant sa journée se limitent à l'ensemble suivant : `{manger, dormir, jouer}`. En observant ce dernier pendant plusieurs jours on peut déterminer les probabilités de transition entre chaque état, et donc définir la chaîne de Markov associé à un comportement vraisemblable de chat. Ensuite, on pourra déterminer la probabilité d'un futur état par rapport à l'état actuel du chat, ou bien déterminer à l'aide de la suite d'états d'un animal quelconque s'il est probable ou pas que ce soit un chat.

Dans le cas de notre étude, le formalisme des chaînes de Markov est utilisée pour porter la notion de probabilité de transition entre un morceau du découpage en *n-grams* et la lettre qui suit dans une ligne de commande. On considère alors les morceaux du découpage en *n-grams* comme des états, entre lesquels s'effectuent des transitions. Les probabilités calculées pendant la phase d'apprentissage permettront pendant la phase d'exploitation de distinguer les comportements vraisemblables des autres.

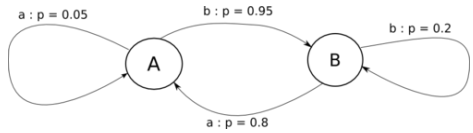

Fig. 2. Exemple de chaîne de Markov avec deux états A et B

2.3 Modèle mathématique

Prenons maintenant le temps de noter mathématiquement les calculs que l'on cherche à faire.

1. Processus stochastique (dans ce cas) : Évolution discrète d'une variable aléatoire.

Soit (CMD_i), $i \in [\![1, N]\!]$ l'ensemble des lignes de commande de notre jeu de données (de taille $N \in \mathbb{N}$). Pour un i donné, on peut représenter une ligne de commandes comme une suite de sous-chaînes de caractères après découpages suivant la méthode des *n-grams* (n est la taille du découpage) :

$$\text{CMD}_i = (c_{i,1}, c_{i,2}, ..., c_{i,m(i)}),\ m(i) \in \mathbb{N}$$

Les éléments $c_{i,j}$ $(j \in [\![1, m]\!])$ sont les morceaux après découpage et sont donc tous des chaînes de caractères de taille n.

Propriété. Il est à noter que l'on peut déterminer le nombre de découpages nécessaires pour une ligne de commande CMD_i, connaissant n la taille des *n-grams* avec la formule suivante :

$$m(i) = len(\text{CMD}_i) - n + 1$$

Ensuite, on définit les probabilités de transition observées entre une chaîne c et x la lettre qui suit, comme la probabilité conditionnelle suivante :

$$\mathbb{P}_c(x) = \frac{\text{Card}((c, x) \in \text{CMD}_i,\ i \in [\![1, N]\!])}{\text{Card}(c \in \text{CMD}_i,\ i \in [\![1, N]\!])}$$

Cela correspond à la phase d'entraînement de l'algorithme. Pour illustrer avec un exemple concret, si on découpe nos commandes en 4-grams et que l'on observe `exe` après `cmd.` dans 95 lignes de commandes, sur un total de 100 lignes de commandes où `cmd.` apparaît, alors la probabilité de transition entre `cmd.` et la lettre `e` est simplement de 95%.

Une fois l'algorithme entraîné on peut déterminer la vraisemblance d'une nouvelle ligne de commande $\text{CMD} = (c_1, c_2, ..., c_m)$ normalisée selon sa longueur (on note x_j la lettre qui suit le découpage c_j, et x_m correspond à un marquage de fin de ligne de commande) :

$$\mathcal{L}(\text{CMD}) = \left(\prod_{j=1}^{m} \mathbb{P}_{c_j}(x_j) \right)^{\frac{1}{m}}$$

Et pour plus de simplicité, on considérera en réalité la log-vraisemblance :

$$\begin{aligned} \ell(\text{CMD}) &= \log\left[\mathcal{L}(\text{CMD})\right] \\ &= \frac{1}{m} \sum_{j=1}^{m} \log\left[\mathbb{P}_{c_j}(x_j)\right] \end{aligned} \tag{1}$$

Pour résumer ce développement, et réconcilier les allergiques aux équations mathématiques, on retrouve en Figure 3 un exemple d'application du calcul de façon schématique.

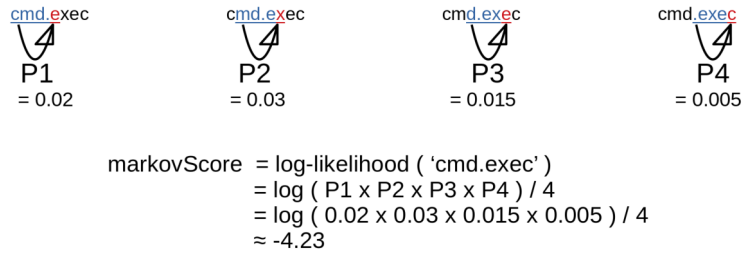

Fig. 3. Exécution schématique de l'algorithme (4-grams)

On notera qu'un modèle est alors spécifique à un parc informatique, notamment pour la valeur de vraisemblance minimum et parce qu'il sauvegarde une partie de l'information des lignes de commandes des machines considérées (ce qui peut poser des problèmes de confidentialité). Dès lors, on ne définit pas de modèle omniscient capable de détecter des anomalies dans n'importe quel système. Il sera nécessaire de faire des modèles spécifiques à chaque cas d'application.

2.4 Production d'alertes

Il est détaillé ici comment l'algorithme peut être utilisé pour produire des alertes dans un cas de détection de long terme. Une alerte de détection système fait suite à un comportement suspect sur un parc supervisé, et est traité par des analystes qui doivent pouvoir conclure sur la présence ou non d'un attaquant. Un des objectifs d'une alerte est donc d'être porteuse d'informations permettant la qualification. Les analystes sont habitués à traiter des alertes provenant de règles SIGMA, qui caractérisent de façon précise le comportement suspect. Elles permettent aussi de remonter aux journaux d'événements mis en cause (il peut en exister seulement un), ce qui facilite l'investigation. Il s'agit donc de coller le plus possible à ce schéma dans le cas d'une alerte provenant de la détection d'une ligne de commande anormale.

Étant donné que le score de vraisemblance dépend du jeu de données d'entraînement et du comportement du parc considéré, on ne peut pas fixer de seuil absolu de génération d'alertes. De plus on ne veut pas inonder les analystes bénéficiant de notre source d'alertes avec un trop grand nombre

de ces dernières. C'est à dire qu'on ne veut pas remonter K alertes par parc supervisé et par jour, car cela nuirait à la pertinence des alertes et ne passerait pas à l'échelle, ne permettant plus aux analystes de prendre le temps d'investiguer chaque alerte. Il a donc fallu imaginer un système adaptatif, qui prend en compte les alertes passées et permet de créer un seuil dynamique.

Pour ce faire, AnoMark lance chaque jour ses calculs sur les lignes de commandes de la veille, pour chaque parc supervisé. Il produit un top K de lignes de commandes anormales après les avoir triées en commençant par les plus invraisemblables (on peut choisir $K = 100$ par exemple, ou le faire dépendre du volume quotidien de journaux). Pour ne pas remonter K alertes, il compare au top K des 30 derniers jours (ce nombre de jours peut être modifié), et chaque ligne du top du jour qui pourrait rentrer dans ce top *historique* (*i.e.* dont le score de vraisemblance est inférieur à celui de la K^{eme} entrée du top historique) est une alerte.

Cette méthode permet de garder un seuil dynamique au cours du temps et de la vie du parc informatique. Dans l'exemple de la Figure 4, le seuil au jour J est à -5.4, c'est à dire qu'il faut un score inférieur pour qu'une ligne de commande soit considérée comme une alerte.

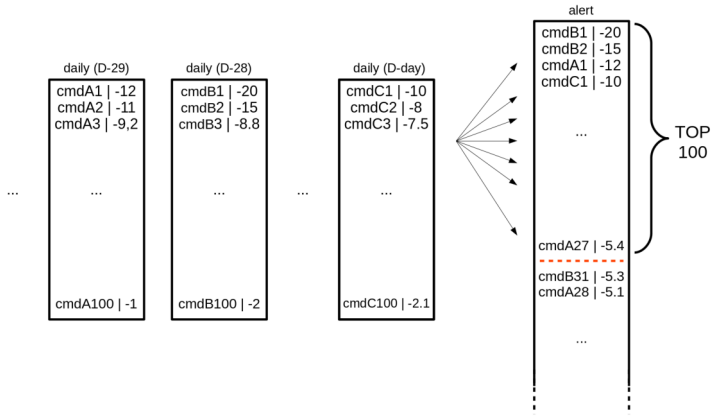

Fig. 4. Schéma de la production d'alertes pour K=100 et 30 jours d'historique

3 Développement de l'outil

3.1 Technologies utilisées

AnoMark a déjà été implémenté en Python et en PySpark, et testé en conditions opérationnelles. Ses performances permettent de garantir une installation et une première application en mode *Threat Hunting* sur un parc en une journée environ.

L'application de l'outil sur un parc nécessite de mettre en place une politique de journalisation adéquate, pour remonter les informations nécessaires. L'ANSSI décrit une telle politique dans son guide de journalisation. Pour la production d'alertes, nous avons testé plusieurs méthodes :

— dans le cas du *Threat Hunting* on peut fouiller les résultats d'analyses à l'aide d'un notebook Jupyter, ou en utilisant le projet en tant que *custom command* Splunk ;

— dans le cas d'une détection en continu on utilisera une architecture type ELK (*ElasticSearch-Logstash-Kibana*) avec un *dashboard* présentant l'analyse du jour, ou bien une architecture Splunk avec un *dashboard* dédié.

Ces idées d'architectures ne sont précisées qu'à titre d'exemples. Le projet a été pensé pour être le plus modulaire possible, afin d'être intégré rapidement dans n'importe quelles conditions. Il peut d'ailleurs même être appliqué à d'autres champs textuels que les lignes de commande...

3.2 Performances du modèle

Afin d'évaluer les performances de l'algorithme, nous avons entraîné puis appliqué AnoMark sur des données d'événements création de processus relatives à un incident. Nous nous sommes placés dans des conditions du quasi-direct, comme si le modèle avait été utilisé en détection continue sur le parc.

Le jeu complet comprenait 18 millions d'entrées, que nous avons découpé en deux pour avoir un jeu de données d'entraînement et un jeu de données de test. Ce découpage respecte l'ordre de déroulement temporel des événements, pour reproduire le fonctionnement d'un quasi-direct. Dans le top 50 (donc par rapport aux 9 millions de lignes du jeu de test) des résultats on retrouve dans le cas d'un entraînement avec des 4-grams :

— 22 actions de l'attaquant ;

— 23 actions d'administrateur ;

— 5 actions du début de remédiation.

Il est à noter que la définition de vrais et faux positifs n'est pas évidente. Pour une ligne de commande correspondant à un attaquant on peut évidemment classer dans les vrais positifs, mais pour une ligne correspondant à une action de remédiation ou d'administration les choses sont plus complexes. En effet, certaines commandes d'administrateurs sont proches d'un comportement potentiel d'attaquant (dans le sens où elles peuvent être totalement nouvelles et s'exécuter avec des pouvoirs élevés), et on veut donc les remonter. Suivant cette logique certaines de ces commandes sont donc aussi de vrais positifs et doivent intervenir dans le calcul des scores de précision de l'algorithme.

Nous avons aussi pu tester différentes tailles de *n-grams*, pour déterminer quelle valeur de n permettait d'avoir les meilleurs résultats. Nous avons observé que la valeur optimale est dépendante de la forme des lignes de commande observées sur le parc. Généralement des découpages en 4-grams ou 5-grams offrent les résultats les plus pertinents. Il est aussi tout à fait possible d'utiliser plusieurs modèles à la fois et de créer un arbre de décision *adhoc* s'appuyant sur ces derniers.

3.3 Détails des formats de lignes de commandes remontées

Il ressort de l'utilisation d'AnoMark quelques tendances dans les alertes remontées qui permettent de comprendre comment il détecte les anomalies. Nous listerons ici quelques exemples, sans toutefois être exhaustifs, car il existe une infinité de possibilités d'alertes.

On peut rassembler dans un premier groupe les lignes de commande qui sont présentes dans les alertes de l'algorithme mais qui pourraient être détectées de manière plus simple :

— les lignes contenant de l'information en base 64 du type :

```
» powershell -enc Q29uY2VudHJlLnRvaS5zdXIubWEucHJl
c2VudGF0aW9uIQ==
```

— les *ping* vers des domaines inhabituels :

```
» ping heeeeeeeey.com
```

— l'exécution de processus inconnus :

```
» iWillPawnYou.exe /user adminAccount
```

Il n'est pas étonnant, au vu de la modélisation construite, qu'AnoMark remonte ce type de commandes. Pour la base 64, l'algorithme voit un enchaînement quasi-aléatoire de caractères donc ayant des probabilités faibles. Pour les deux autres cas, il s'agit tout simplement de groupes de mots jamais vus. Mais dans les trois cas, on aurait pu écrire des règles ou des algorithmes plus simples.

Dans un second groupe on rassemble des lignes de commande détectées comme anormales par l'algorithme, et qui auraient été par ailleurs complexes à détecter :

— l'utilisation de *flags* inconnus :

```
» legit.exe -newflag newdata
```

— le changement de quelques lettres :

```
» CmD.eXe -someflag -someparam
```

— l'exécution de processus connus depuis des chemins inconnus :

```
» C:\newfolder\myproc.exe
```

Il apparaît un peu mieux dans ces exemples-ci l'intérêt de l'algorithme. Il serait plus complexe de remonter ce type d'activité avec des règles. On peut ainsi mieux appréhender l'intérêt d'une telle modélisation pour détecter des comportements nouveaux.

Enfin, l'algorithme a aussi pu remonter des mauvaises pratiques d'administration comme l'écriture de mots de passe en clair dans des lignes de commande dans lesquels ils n'auraient pas dû figurer. Ce sont des alertes qui permettent de prévenir des incidents suite à ces mauvaises pratiques, et elles ne sont donc pas considérées comme des faux positifs.

3.4 Limites du modèle

Il s'agit enfin d'estimer quels sont les limites de la modélisation, et les possibles biais statistiques qui peuvent apparaître lors des entraînements.

D'un point de vue détection, il faut déjà noter qu'on ne peut pas attendre de l'algorithme qu'il fonctionne sans être accompagné d'un ensemble de règles construites par des spécialistes. Pour prendre un exemple, il est par construction impossible d'assurer qu'il remontera toutes les lignes de commande utilisant de l'encodage en base 64 de manière illégitime, même si on constate qu'il a tendance à remonter cet encodage. En effet, il se peut que d'autres commandes soient considérées comme plus anormales au moment de l'analyse, et qu'ainsi une détection triviale ne soit pas faite. Dès lors, il faut garder une base de règles pour des comportements simples. AnoMark ne permet pas d'assurer une certitude de détection, mais propose plutôt une détection de comportements jusqu'ici inconnus. De plus, on considère qu'un attaquant visible dans les journaux produira plusieurs lignes de commande, et que c'est dans cet ensemble qu'on espère trouver une anomalie remontée par le modèle.

Ensuite, d'un point de vue plus statistique il peut arriver que des biais de sélection arrivent, à la suite d'une mauvaise construction de jeu d'entraînement. Il faudra toujours veiller à choisir une fenêtre temporelle

de données suffisamment grande pour que les journaux d'événements soient représentatifs du comportement sur le parc supervisé. Or, cette taille de fenêtre n'est pas aisée à définir. Il semble qu'une période d'environ 1 mois permette d'éviter de tomber sur l'absence d'un utilisateur (pour cause de congés par exemple), mais cela reste à définir en fonction du périmètre de supervision.

4 Conclusion

L'étude menée grâce à AnoMark nous a permis d'entrevoir les possibilités de détection apportées par l'apprentissage statistique et plus précisément la détection d'anomalies. Ce type d'algorithme ne remplace ceci dit pas les constructions de signatures pour la détection de comportements connus. C'est un complément pour les analystes, afin de pouvoir explorer d'autres pistes de recherche. D'ailleurs, l'algorithme ne cherche pas à qualifier l'aspect dangereux de la ligne de commande, mais plutôt à mettre en évidence des changements de comportement qui doivent être investigués en autonomie par l'analyste.

Nous avons vu que cette détection peut se faire aussi bien de manière exploratoire pour de la détection de circonstance en mode *Threat Hunting*, comme de manière continue pour une détection de long terme avec une production d'alertes. Ces deux aspects en font un outil profondément opérationnel, ce qui est un atout fort. La conception s'est voulue dès le départ la plus simple possible pour appuyer la portabilité de l'outil. C'est aujourd'hui un atout dans l'analyse de grands volumes de données, pour trouver rapidement des voies d'investigation.

Ce travail nous encourage à continuer de chercher des solutions d'algorithmes de détection d'anomalies pour détecter des intrusions de façon opérationnelle. Cela pourrait être toujours sur des données textuelles, avec d'autres techniques de NLP, ou sur des données de type différent mais en continuant d'utiliser des chaînes de Markov, peut-être cachées pour d'autres cas d'usage...

Références

1. Niki Parmar Jakob Uszkoreit Llion Jones Aidan N. Gomez Lukasz Kaiser Illia Polosukhin Ashish Vaswani, Noam Shazeer. Attention Is All You Need. *arXiv:1706.03762*, 2009.
2. Hugging Face. Get started with transformers. `https://huggingface.co/docs/transformers/index`, 2021.

3. Microsoft. Description for event 4688. https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4688.
4. Klaudia Nazarko. Practical text generation using gpt-2, lstm and markov chain. https://towardsdatascience.com/text-generation-gpt-2-lstm-markov-chain-9ea371820e1e, 2021.
5. Eugene Seneta. Markov and the the creation of markov chains. *In Amy N. Langville and William J. Stewart (Eds.) MAM2006 : Markov Anniversary Meeting. Boson Books, Raleigh, North Carolina, pp. 1-20*, 2006.
6. Austin Taylor. Applied data science and machine learning for cybersecurity - sans tactical detection summit 2018. https://youtu.be/m2AgYbbXz8k?t=1442, 2019.