Université de Montréal

Vers la Sécurité des Conteneurs - Les Comprendre et les Sécuriser

Par

Hugo B. Lapointe

Département d'Informatique et de Recherche Opérationnelle, Faculté des Arts et des Sciences

Mémoire présenté en vue de l'obtention du grade de Maître ès sciences (M.Sc.) en informatique

Juin 2022

Université de Montréal

Département d'Informatique et de Recherche Opérationnelle, Faculté des Arts et des Sciences

---

Ce mémoire intitulé

Vers la Sécurité des Conteneurs - Les Comprendre et les Sécuriser

Présenté par

Hugo B. Lapointe

A été évalué par un jury composé des personnes suivantes

Nadia El-Mabrouk
Président-rapporteur

Esma Aïmeur
Directrice de recherche

Louis Salvail
Membre du jury

Résumé

Afin de faciliter les cycles de développement moderne plus courts, ainsi que la nature éphémère de l'infonuagique, de nombreuses organisations exécutent désormais leurs applications dans des conteneurs, une forme de virtualisation du système d'exploitation. Ces nouveaux environnements sont souvent appelés environnements conteneurisés. Cependant, ces environnements ne sont pas sans risque. Des études récentes ont montré que les applications conteneurisées sont, comme tous les types d'applications, sujettes à diverses attaques. Un autre problème pour ceux qui travaillent dans le domaine de la sécurité informatique est que les applications conteneurisées sont souvent très dynamiques et de courte durée, ce qui aggrave le problème, car il est plus difficile d'auditer leurs activités ou encore de faire une enquête en cas d'intrusion.

Dans ce mémoire, nous proposons un système de détection d'intrusion basé sur l'apprentissage machine pour les environnements conteneurisés. Les conteneurs assurent l'isolation entre le système hôte et l'environnement conteneurisé en regroupant efficacement, les applications ainsi que leurs dépendances. De cette façon, les conteneurs deviennent un environnement logiciel portable. Cependant, contrairement aux machines virtuelles, les conteneurs partagent le même noyau que le système d'exploitation hôte. Afin de pouvoir faire la détection d'anomalies, notre système utilise cette caractéristique pour surveiller les appels système envoyés d'un conteneur vers un système hôte. Ainsi, le conteneur surveillé n'a pas à être modifié et notre système n'est pas tenu de connaitre la nature du conteneur pour le surveiller.

Les résultats de nos expériences montrent qu'il est en effet possible d'utiliser les appels système afin de détecter des comportements anormaux faits par une application conteneurisée et ce sans à avoir à modifier le conteneur.

Mots-clés : Conteneur, Sécurité, Apprentissage Machine, Détection d'Intrusion

Abstract

To facilitate shorter modern development cycles, as well as the ephemeral nature of cloud computing, many organizations are now running their applications in containers, a form of operating system virtualization. These new environments are often referred to as containerized environments. However, these environments are not without risk. Recent studies have shown that containerized applications are, like all other types of applications, prone to various attacks. Another problem for those working in IT security is that containerized applications are often very dynamic and short-lived, which compounds the problem because it is more difficult to audit their activities or even make an investigation in case of intrusion.

In this thesis, we propose an intrusion detection system based on machine learning for containerized environments. Containers provide isolation between the host system and the containerized environment by efficiently grouping applications and their dependencies. Therefore, containers become a portable software environment. However, unlike virtual machines, containers share the same kernel as the host operating system. To be able to do anomaly detection, our system uses this feature to monitor system calls sent from a container to a host system. Thus, the monitored container does not have to be modified.

The results of our experiments show that it is indeed possible to use system calls to detect abnormal behaviour made by a containerized application without having to modify the container.

Keywords: Container, Security, Machine Learning, Intrusion Detection

Table des matières

Liste des tableaux

Liste des figures

Liste des acronymes et abréviations

ADFA-LD	Australian Defence Force Academy Linux Dataset
ADFA-WD	Australian Defence Force Academy Windows Dataset
API	Application Public Interface
AUC	Area Under the Curve
BSD	Berkeley Software Distribution
CGROUP	Control Group
CHROOT	Change Root
CIS	Center for Internet Security
CNCF	Cloud Native Computing Foundation
Cosmos	Container Smart Monitoring System
CPU	Central Processing Unit
CRI-O	Container Runtime Interface Open Container Initiative
CVE	Common Vulnerabilities and Exposures
DB	Database
DTs	Decision Trees
FP	Faux Positif
FN	Faux Négatif
FTP	File Transfer Protocol
GPU	Graphical Processing Unit
HTTP	Hypertext Transfer Protocol
ID3	Iterative Dichotomiser 3
IDS	Intrusion Detection System
IP	Internet Protocol
IPC	Inter-Process Communication
JSON	JavaScript Object Notation
KMC	K-Means Clustetring
LSTM	Long Short-Term Memory

LTS	Long Term Support
LXC	Linux Container
MS/DOS	Microsoft Disk Operating System
NB	Naive Bayes
OCI	Open Container Initiative
OS	Operating System
PID	Process Identifier
PPID	Parent Process Identifier
RNN	Recurrent Neural Network
ROC	Receiver Operating Characteristic
SQL	Structured Query Language
SSH	Secure Shell
SVM	Support Vector Machine
SysCalls	System Calls
TFP	Taux de Faux Positifs
TPU	Tensor Prosessing Unit
TVP	Taux de Vrai Positifs
VM	Virtual Machine
VMM	Virtual Machine Monitor
VN	Vrai Négatif
VP	Vrai Positif

Remerciements

J’aimerais remercier les personnes suivantes.

Professeure Esma Aïmeur, ma directrice de recherche. Je suis reconnaissant et je profite de l’occasion pour la remercier sincèrement. Elle a trouvé le temps de me soutenir, de m’offrir de précieux conseils et de m’encourager, malgré la période difficile que nous visons tous en ce moment. Merci.

Je tiens également à remercier les différents groupes de travail que j’ai eu durant ce diplôme. Vous m’avez fait apprendre énormément durant cette courte période et surtout, j'ai partagé des moments fantastiques avec vous tous. Merci.

J’aimerais remercier mes parents et mes amis pour leur soutien. Du fond de mon cœur, j’aimerais remercier ma petite famille, ma femme Claire et mes deux beaux enfants, Francis et Simon. Sans leur support, leur amour et leurs compréhensions, ce projet de recherche n’aurait pas été possible. Papa vous aime.

Chapitre 1 - Introduction

Dans ce premier chapitre, nous présentons le contexte de notre travail de recherche et définissons les problèmes qui sont abordés dans ce mémoire. Nous présentons également les questions de recherche correspondantes, nos objectifs et finalement nos contributions.

1.1 Énoncé du Problème

La technologie des *conteneurs* est largement adoptée dans les environnements informatiques à cause de son efficacité et de sa faible surcharge d'isolation. Ceux-ci peuvent être définis comme étant une forme de virtualisation au niveau du système d'exploitation qui permet de facilement exécuter une application dans divers environnements informatiques. Cependant, des études récentes (Wenhao & Zheng, 2020) ont montré que les conteneurs sont sujets à divers types d'attaques qui remettent en question leur sécurité. Ceci est devenu l'une des principales préoccupations des utilisateurs et freine l'adoption de cette nouvelle technologie. En effet, une étude récente (Shu, Gu, & Enck, 2017) révèle un nombre de vulnérabilités alarmant dans *Docker Hub*, le registre officiel de *Docker*. D'autre part, à cause de leur nature immuable, il est difficile de les modifier afin de les rendre plus sécuritaire.

1.1.1 Immutabilité

Les conteneurs sont fondamentalement conçus pour être immuables et sans état. Le principe d'immuabilité du conteneur concerne ce qu'on appelle son image. Une fois que celle-ci est construite, il est nécessaire de créer une nouvelle image si des modifications doivent être apportées[1]. Cette approche permet de déployer la même image de conteneur dans différents environnements, en les rendant aussi identiques que possible. Du point de vue de la sécurité, ceci peut être un problème, car il n'est pas possible de modifier un conteneur. Par exemple, il n'est pas possible de le modifier pour installer un antivirus puisque ceci briserait son immutabilité. De plus, si un attaquant infecte un conteneur et que ce dernier est redéployé par l'orchestrateur, les traces de l'attaque disparaissent, ce qui rend l'investigation difficile. Afin de garder une trace, il est possible d'enregistrer les fichiers journaux dans un endroit non éphémère,

[1] https://www.infosecurity-magazine.com/opinions/mmutable-infrastructure-modern

comme une base de données externe, mais ceux-ci permettent uniquement de détecter une anomalie longtemps après l'attaque.

1.1.2 Méthodes de détection d'anomalies inefficaces

Bien que plusieurs méthodes existent pour détecter les intrusions, aucune n'est efficace dans le contexte dynamique des environnements conteneurisés. Les systèmes de détection d'intrusion basés sur les signatures sont efficaces pour détecter les attaques connues, mais ces systèmes ne peuvent pas détecter de nouvelles attaques[2]. Ceux basés sur les anomalies sont principalement utilisés pour détecter les attaques inconnues. Malheureusement, même si ces systèmes sont efficaces pour détecter les intrusions, ils sont sujets à un taux élevé de faux positifs (Khraisat & Alazab, 2021). Ceux-ci peuvent rapidement dégrader la confiance dans le système, entrainant même l'ignorance totale des alertes. Un autre problème est que les applications conteneurisées sont souvent éphémères et s'exécutent généralement pendant une courte période avant l'arrêt du conteneur pour économiser des ressources, car le redémarrage d'un conteneur est généralement rapide et peu couteux[3]. Par conséquent, il est difficile pour le système de détection d'anomalies de collecter suffisamment de données d'entrainement pour créer un comportement normal fiable. De plus, ces systèmes ont tendance à consommer beaucoup de ressources.

1.1.3 Impact sur la Performance

Dans le contexte de la sécurité de l'hôte, la détection d'anomalies fait référence à l'identification d'intrus ou de violations inattendues. En moyenne, il faut des dizaines de jours pour qu'une violation du système soit détectée[4]. Cependant, une fois qu'un attaquant est entré, les dégâts sont généralement causés en quelques jours ou moins, il est donc important de détecter les anomalies le plus rapidement possible. De plus, les systèmes de détection d'anomalies peuvent avoir un impact négatif sur la performance du système (Dreger, Feldmann, Paxson, & Sommer, 2008), ce qui ralentit leur adoption. La pénalité de performance dans ce type d'application peut être quelque peu élevée, par exemple l'un des clients de l'entreprise Capsule8 a constaté une

[2] https://www.n-able.com/blog/intrusion-detection-system
[3] https://www.blackhat.com/docs/eu-15-Bettini-Vulnerability-Exploitation-In-Docker-Container-Environments.pdf
[4] https://www.verizon.com/business/resources/reports/dbir/

augmentation de 80 % de la surcharge des appels système dans l'espace utilisateur[5]. Cette surcharge est inacceptable, surtout dans le contexte des environnements conteneurisés qui sont souvent hébergés dans l'infonuagique où les prix sont basés sur l'utilisation des ressources, en d'autres mots la mémoire vive et la puissance de calcul du processeur.

Ces problèmes ont conduit aux questions suivantes que nous avons tenté d'aborder tout au long de notre mémoire :

- Comment **auditer les activités d'un conteneur sans le modifier**, ou encore sans connaitre d'avance l'application qu'il contient ?
- Comment pouvons-nous **utiliser l'apprentissage machine** pour créer un système de détection d'anomalies dans le contexte d'un environnement conteneurisé.
- Comment faire pour que le système de détection d'intrusion soit **efficace, mais ne consomme pas trop de ressources** sur l'hôte ?

1.2 Objectifs

L'objectif de ce mémoire est de produire un système de détection d'intrusion basé sur l'apprentissage machine pour les environnements conteneurisés et ainsi ajouter un outil à l'arsenal des ingénieurs en sécurité pour protéger nos données dans cet environnement dynamique et éphémère. Plus spécifiquement, les objectifs de ce mémoire sont :

- **Proposer** un système qui permet d'auditer les activités des conteneurs sans modifier le contenu de ceux-ci ou encore de connaitre leur nature, c'est-à-dire l'application qu'ils contiennent.
- **Concevoir** un système adaptatif et faiblement couplé qui peut être facilement connecté à d'autres applications, par exemple une base de données, afin que d'autres développeurs ou chercheurs puissent l'utiliser ;
- **Utiliser l'apprentissage machine** afin que le système performe mieux dans la détection d'anomalies que les systèmes traditionnels sans apprentissage machine ;

[5] https://capsule8.com/blog/auditd-what-is-the-linux-auditing-system/

- **Développer** un système portable et qui n'affecte pas négativement les performances du système hôte.

1.3 Principales Contributions et Originalité

L'originalité de notre mémoire réside dans la conception d'un système de détection d'anomalies basé sur l'apprentissage machine pour les environnements conteneurisés. Les principaux apports sont les suivants :

- Le système proposé permet d'**auditer le comportement des conteneurs** sans briser l'immutabilité de ceux-ci ;
- Le système utilise des techniques de détection d'intrusion **basées sur l'apprentissage machine** afin de minimiser le taux de faux positifs ;
- Puisque que la plupart des environnements conteneurisés sont dans l'infonuagique, le système **minimise son utilisation des ressources**.

1.4 Structure de la thèse

Le reste de la thèse est organisé comme suit. Dans le chapitre 2, nous discutons des concepts et des travaux reliés. Dans le chapitre 3, nous présentons l'architecture de notre système de détection d'anomalie basé sur l'apprentissage machine pour les environnements conteneurisés. Le chapitre 4 explore la mise en œuvre du projet, plus précisément comment nous avons configuré l'environnement utilisé pour faire nos expériences, les technologies utilisées pour faire notre implémentation et les ensembles de données utilisés. Le chapitre 5 explique comment nous avons évalué notre modèle, les expériences que nous avons faites, ainsi que les résultats de ceux-ci. Finalement, le chapitre 6 présente la conclusion et les travaux futurs.

Chapitre 2 - Concepts et Travaux Reliés

Dans ce chapitre, nous explorons les origines des conteneurs, les concepts fondamentaux qui leur permettent d'exister, ainsi que les différences qu'ils ont avec les machines virtuelles. Par la suite, nous abordons les concepts et travaux reliés à l'apprentissage machine et aux systèmes de détection d'intrusions.

2.1 Les Origines des Conteneurs

Même si les conteneurs, qui sont en ce moment en train de transformer l'informatique, semblent être une nouvelle technologie sortie de nulle part, ils sont plus anciens qu'on pourrait le penser. En effet, l'idée originale a vu le jour vers la fin des années 1970, lorsque le concept a été utilisé pour la première fois sur les systèmes d'exploitation *UNIX* afin de mieux isoler les applications. Lors du développement d'UNIX V7, en 1979, l'appel système *chroot* a été introduit[6]. Celui-ci permet de modifier le répertoire racine d'un processus et de ses enfants vers un nouvel emplacement dans le système de fichiers (Eder, 2016). Ce nouveau concept a été le début de l'isolement des processus.

Près de 20 ans plus tard, dans les années 2000, un fournisseur d'environnement partagé qui utilisait *FreeBSD*, un système d'exploitation de type UNIX basé sur BSD, propose de réaliser une séparation nette entre ses services et ceux de ses clients. Le but de cette séparation est de faciliter l'administration de ses serveurs et d'augmenter la sécurité. Pour y parvenir, ce fournisseur a introduit le mécanisme appelé *Jails*. Les Jails *FreeBSD* permettent de partitionner le système d'exploitation en plusieurs petits systèmes indépendants. Chaque système possède, par exemple sa propre adresse IP, ainsi que sa propre configuration. En 2001, Linux introduit *Linux V-Server*, un mécanisme similaire qui permet de partitionner les ressources du système.

Quelques années plus tard, en 2006, Google introduit un nouveau mécanisme appelé *Process Containers*. Celui-ci permet de limiter, comptabiliser et isoler l'utilisation des ressources sur un ensemble de processus[7]. Par la suite, il fut renommé *Groupes de Contrôle* avant d'être ajouté au

[6] https://en-academic.com/dic.nsf/enwiki/175578
[7] https://www.dataversity.net/a-brief-history-of-data-containers/

noyau Linux à partir de la version 2.6.24. Basé sur les Groupes de Contrôle et les espaces de noms, en 2008, *LXC* voit le jour. Il s'agit de la première implémentation complète d'un gestionnaire de conteneurs Linux.

Finalement, en 2013, Docker est créé et les conteneurs explosent en popularité[8]. Ce n'est pas un hasard si la croissance de l'utilisation de Docker et des conteneurs va de pair. À ses débuts, Docker utilisait LXC. Il a ensuite remplacé ce gestionnaire de conteneurs par sa propre implémentation appelé *libcontainer*. Celle-ci permet de créer des conteneurs et permet aussi de gérer leur cycle de vie en effectuant des opérations supplémentaires après sa création.

La figure 1 donne les noms et les dates importantes concernant l'origine des conteneurs :

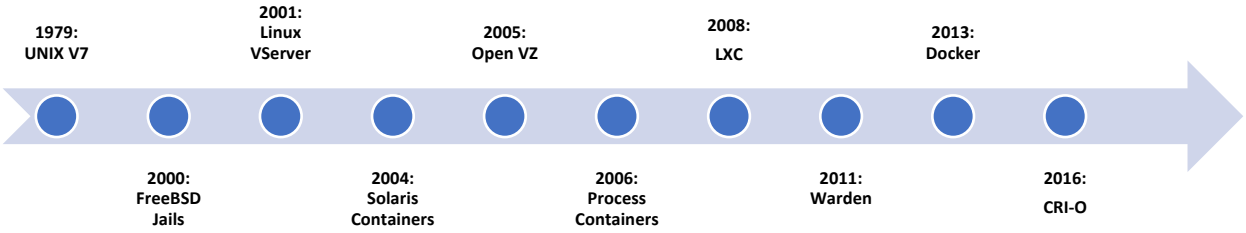

Figure 1- Histoire des conteneurs

2.2 Conteneur

On peut définir un conteneur comme étant une unité logiciel exécutable dans laquelle le code de l'application est regroupé avec les librairies et les dépendances nécessaires à son fonctionnement. De façon générale, les conteneurs sont petits, rapides et portables, car contrairement aux machines virtuelles ils n'ont pas besoin d'inclure un système d'exploitation complet. En effet, ils tirent parti des fonctionnalités et des ressources du système d'exploitation hôte qui les exécutent (Bhatia & Choudhary, 2017). Un exemple de cas d'utilisation des conteneurs serait pour standardiser les environnements de développement. En effet, étant donné qu'un conteneur documente les instructions nécessaires à sa création dans son fichier de

[8] https://containerjournal.com/features/docker-4-milestones-docker-history/

configuration, il permet de minimiser l'incohérence entre les différents environnements de développement, de test et de production. Le moteur du conteneur s'assurerait que tous les environnements créés soient cohérents.

Un point intéressant est que bien qu'ils soient appelés conteneurs, le terme processus conteneurisé serait plus précis. Un conteneur est simplement un processus exécuté sur la machine hôte avec une vue limitée de cette machine, des ressources limitées et un accès à une sous-arborescence du système de fichier[9].

Dans la majorité des cas, les conteneurs sont exécutés sur une machine hôte qui roule un système d'exploitation utilisant le noyau Linux[10]. Afin de comprendre le fonctionnement de cette technologie, une compréhension des mécanismes suivants est requise : les appels système, les groupes de contrôle, les espaces de noms et comment changer le répertoire racine.

La figure 2 montre les différentes couches nécessaires à la virtualisation aux conteneurs :

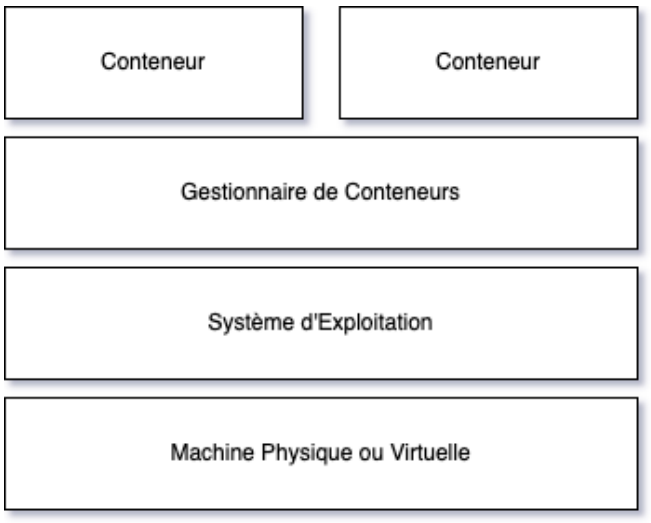

Figure 2- Virtualisation des conteneurs

Il est important de garder en tête que même si les conteneurs ont un accès limité à ce qu'ils peuvent voir et aux ressources qu'ils peuvent consommer, l'hôte lui peut voir les processus qui sont exécutés par ces derniers (Combe, Martin, & Di Pietro, 2016).

[9] https://www.youtube.com/watch?v=8fi7uSYlOdc
[10] https://www.docker.com/resources/what-container

2.2.1 Appels Système

Un appel système est une méthode utilisée par les applications pour communiquer avec le noyau du système d'exploitation. Dans les systèmes d'exploitation modernes, cette méthode est utilisée si une application ou un processus doit transmettre des informations au matériel informatique, à un autre processus, ou encore au noyau lui-même. La plupart des applications avec lesquelles un utilisateur interagit, par exemple Firefox, sont exécutées dans ce qu'on appelle l'espace utilisateur, cet espace a un niveau de privilège inférieur à l'espace noyau du système d'exploitation[11]. Si l'application veut effectuer une tâche, par exemple accéder à un fichier, utiliser le réseau ou encore avoir l'heure, elle doit faire la demande au noyau qui le fera pour elle (Merkel, 2014). À l'origine, les appels système ont été créés pour régler les deux problèmes majeurs suivants qu'avaient les vieux systèmes d'exploitation :

- **L'absence de séparation entre le code des applications** utilisateur et le code du noyau fonctionne bien pour les systèmes mono-utilisateur qui ne sont pas multitâches. Cependant, dès que plusieurs applications peuvent s'exécuter simultanément dans un même système d'exploitation, il faut synchroniser l'accès aux matériels informatiques et contrôler l'utilisation de la mémoire ;
- **L'absence de protection du système d'exploitation** contre une application mal programmée. Dans un système d'exploitation moderne, un programme erroné peut planter, sans pour autant faire planter le système d'exploitation lui-même. Par exemple, sous MS/DOS, un plantage d'un programme se terminait généralement par un redémarrage complet de la machine.

Peu importe qu'il s'agisse d'une application conteneurisée ou non, les appels système s'utilisent exactement de la même façon. La figure 3 montre comment un processus conteneurisé, dans l'espace utilisateur, utilise les appels système pour demander l'accès aux matériels informatiques, dans l'espace noyau :

[11] https://man7.org/linux/man-pages/man2/syscalls.2.html

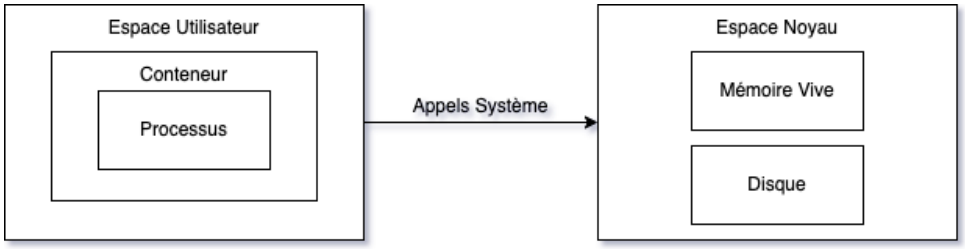

Figure 3 – Processus conteneurisé qui fait un appel système

Du point de vue de la sécurité, le fait que tous les conteneurs sur un même hôte effectuent leurs appels système au même noyau peut sembler dangereux. Heureusement, le noyau Linux offre trois mécanismes essentiels, c'est-à-dire les groupes de contrôle, les espaces de noms et la capacité de changer le répertoire racine afin de limiter les permissions et les accès aux ressources de ces derniers.

2.2.2 Groupe de Contrôle

Le premier mécanisme essentiel aux conteneurs est le groupe de contrôle. Il s'agit d'une fonctionnalité du noyau Linux qui permet la gestion hiérarchique et l'allocation des ressources système telles que la mémoire vive, l'accès au processeur et les entrées et sorties réseau qu'un groupe de processus peut utiliser[12]. En créant une limite sur la quantité de mémoire vive et aux quantités des autres ressources auxquelles un processus peut accéder, certains types d'attaques peuvent être évités[13]. Par exemple, si un processus est autorisé à consommer une quantité de mémoire vive illimitée, il peut affamer d'autres processus sur le même hôte. Avec ce mécanisme, nous sommes en mesure de limiter la quantité de ressources qu'un conteneur peut utiliser. Cependant, ce dernier peut encore accéder à toutes les ressources, comme la mémoire vive, le processeur et le disque dur du système d'exploitation hôte.

2.2.3 Espace de Noms

Le deuxième mécanisme essentiel est l'espace de nom. En plaçant un processus dans un espace de noms, il est possible de restreindre les ressources visibles par ce processus[14]. Le premier espace de noms a été introduit dans la version 2.4.19 du noyau Linux en 2002 (Kerrisk, 2021). Il

[12] https://man7.org/linux/man-pages/man7/cgroups.7.html
[13] https://redhat.com/documentation/html/resource_management_guide/sec-default_cgroup_hierarchies
[14] https://man7.org/linux/man-pages/man7/namespaces.7.html

s'agissait de l'espace de noms de montage. À l'époque, la plupart des systèmes d'exploitation avaient un seul espace de noms. Le système d'exploitation *Plan 9* est à l'origine de ce concept (Pike, Presotto, Thompson, Trickey, & Winterbottom, 1992). De nous jours, il existe plusieurs types d'espaces de noms pris en charge par le noyau Linux. Dans un ordinateur mono-utilisateur, un seul environnement système peut convenir. Cependant, sur un serveur où les services sont exécutés, il est essentiel pour la sécurité et la stabilité que les services soient isolés les uns des autres. Prenons l'exemple d'un serveur exécutant plusieurs services, où l'un est compromis par un acteur malveillant. Dans un tel cas, cet acteur malveillant est en mesure d'exploiter ce service et de se frayer un chemin vers les autres services. Il est même en mesure de compromettre l'ensemble du serveur. L'isolation de l'espace de noms fournit un environnement sécurisé et élimine ce type d'attaque. En revanche, l'arborescence du système du fichier est encore complètement accessible et donc un acteur malveillant serait facilement en mesure de modifier des fichiers qui appartiennent à d'autres applications sans aucune restriction.

2.2.4 Changer le répertoire racine (chroot)

Finalement, le troisième mécanisme essentiel est la capacité de pouvoir changer le répertoire racine d'un processus[15]. Sous Linux, il est possible de changer le répertoire racine d'un processus en cours et de ses enfants pour que celui-ci pointe vers un autre emplacement dans le système de fichier (Kovács, 2017). Après avoir modifié la racine, le processus perd l'accès à tout ce qui est plus haut dans la hiérarchie des fichiers, car il n'existe aucun moyen d'aller plus haut que ce répertoire dans le système de fichier. En bref, lorsqu'une application est exécutée dans un tel environnement modifié, il lui est impossible d'accéder aux fichiers ou aux utilitaires qui sont plus haut dans l'arborescence. Cet environnement modifié est communément appelé *chroot jail*. Changer le répertoire racine n'est pas un mécanisme de sécurité à toute épreuve. En effet, il ne conteneurise pas complètement l'application et ne doit pas être considéré comme un pare-feu qui sauvera un système des attaquants. Cependant, à moins qu'un processus essaie spécifiquement de sortir d'une prison chroot, il accomplit son travail de sectionnement du

[15] https://man7.org/linux/man-pages/man2/chroot.2.html

système de fichiers pour la plupart des processus, et peut être configuré avec des mesures de sécurité supplémentaires pour bloquer les principales méthodes d'échappement.

Les trois mécanismes fondamentaux précédents sont utilisés par tous les gestionnaires de conteneur moderne, cependant, comme nous pouvons le voir dans le tableau 1, certains détails d'implémentation sont différents (Rajdeep, A Reddy, & Dharmesh, 2014) :

Tableau 1 - Comparaison de l'implémentation des conteneurs

	OpenVZ	**Warden**	**LXC**	**Docker**
Isolation des processus	Espaces de noms PID	Espaces de noms PID	Espaces de noms PID	Espaces de noms PID
Isolation des ressources	Groupe de Contrôle	Groupe de Contrôle	Groupe de Contrôle	Groupe de Contrôle
Isolation du réseau	Espaces de noms réseau	Espaces de noms réseau	Espaces de noms réseau	Espaces de noms réseau
Isolation du système de fichier	Chroot	Superposition système de fichiers avec « overlayfs »	Chroot	Chroot
Cycle de vie du conteneur	« vzctl » pour gérer le cycle de vie des conteneurs	Les conteneurs sont gérés, mais exécutent des commandes sur un client et un serveur « Warden »	« lcx create », « lcx stop », « lcx start », pour créer des conteneurs de démarrage et d'arrêt	Daemon Docker et un client à pour gérer les conteneurs

2.3 Sécurité des Conteneurs

Les conteneurs permettent un déploiement plus rapide que les machines virtuelles avec des performances quasi natives. Cette rapidité est l'une des raisons pour laquelle, au cours des

dernières années, l'utilisation des conteneurs a explosé (Lin, et al., 2018). Même si les concepts nécessaires à leur implémentation existent depuis plusieurs années, c'est Docker qui a popularisé l'utilisation des conteneurs (Bernstein, 2014). Plus qu'une solution de conteneur, Docker est un outil complet d'emballage et de livraison de logiciels (Martin, Raponi, Combe, & Pietro Di, 2018). Cette popularité a aussi attiré l'attention des acteurs malveillants, dans un rapport fait par Tripwire[16] qui incluait 311 professionnels de la sécurité informatique qui travaillent pour une entreprise de plus de 100 employés, 60% ont rapporté avoir déjà eu un incident de sécurité lié à aux conteneurs. De plus, avec l'adoption généralisée d'applications basées sur des conteneurs, les systèmes sont devenus plus complexes et les risques ont augmenté. Des vulnérabilités récentes n'ont fait qu'approfondir cette réflexion.

La figure 4 montre les différentes couches qui peuvent être utilisé comme vecteurs d'attaques contre un conteneurs durant son cycle de vie :

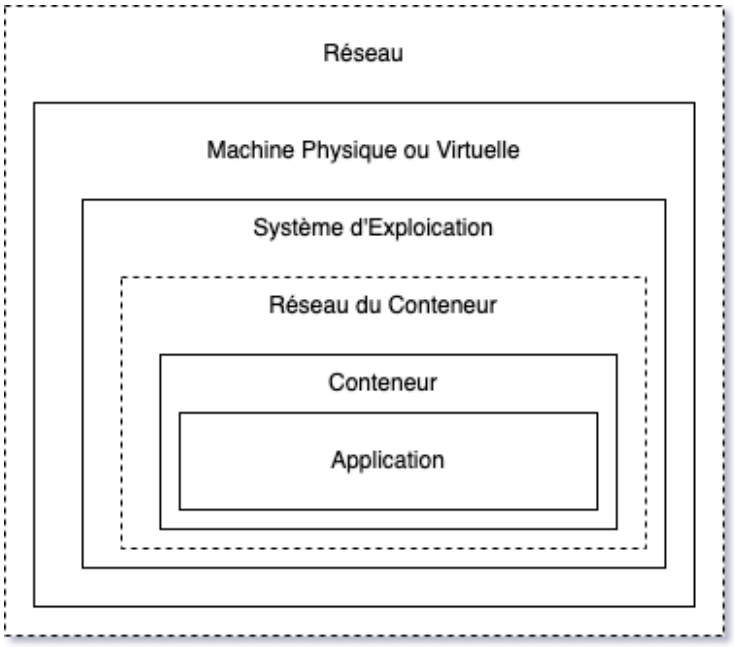

Figure 4- Vecteurs d'attaques des conteneurs

Le reste de cette section sera dédié à ces vecteurs d'attaque.

[16] https://www.tripwire.com/solutions/devops/tripwire-dimensional-research-state-of-container-security-report-register

2.3.1 Code vulnérable

Le cycle de vie d'une application commence avec le code écrit d'un développeur. Ce code, qui est développé à l'interne, et ces dépendances tierces, qui sont écrites par des développeurs externes, peuvent contenir des vulnérabilités. Il existe des milliers de vulnérabilités connues, publiées sur des plateformes comme https://cve.mitre.org/. Chacune d'elle peut potentiellement être utilisée par un acteur malveillant si elles ne sont pas mitigées dans l'application. À ce jour, le meilleur moyen d'éviter l'utilisation d'un conteneur avec des vulnérabilités connues consiste à l'analyser. Il ne s'agit pas d'une activité ponctuelle, car de nouvelles vulnérabilités sont découvertes dans le code existant tout le temps. Il existe deux grandes familles d'outils de détection de vulnérabilités :

- **L'analyse statique de l'image d'un conteneur** : Ce type d'analyse se concentre principalement sur la détection de vulnérabilités dans l'image (He, Dai, & Gu, 2019). Ce type d'analyse peut détecter des vulnérabilités connues en faisant correspondre les dépendances et leur version avec une base de données de CVE (Common Vulnerabilities and Exposures). Le problème avec ce type d'analyse c'est qu'elle n'inclue pas les vulnérabilités non divulguées publiquement et les vulnérabilités du jour zéro, c'est-à-dire une vulnérabilité de logiciel informatique jusqu'alors inconnue de ceux qui devraient s'intéresser à son atténuation, comme le fournisseur du logiciel ;
- **L'analyse dynamique du comportement du conteneur** : Ce type d'analyse se concentre sur le comportement du conteneur durant son exécution afin de détecter des activités anormales (Kharraz, Arshad, & Mulliner, 2016). Cependant, la plupart de ces outils sont basés sur des politiques qui ne peuvent pas s'adapter à des changements dans le comportement d'un conteneur. Tunde-Onadele et coll. ont comparé plusieurs outils de ce type et suggèrent que la détection dynamique surpasse l'analyse statique dans le cas des conteneurs (Tunde-Onadele, He, Dai, & Gu, 2019).
-

Un des outils disponibles sur le marché qui permet de faire ce type d'analyse est Aqua Security. La figure 5 est un exemple de résultat produit par Aqua sur un conteneur basé sur Alpine Linux :

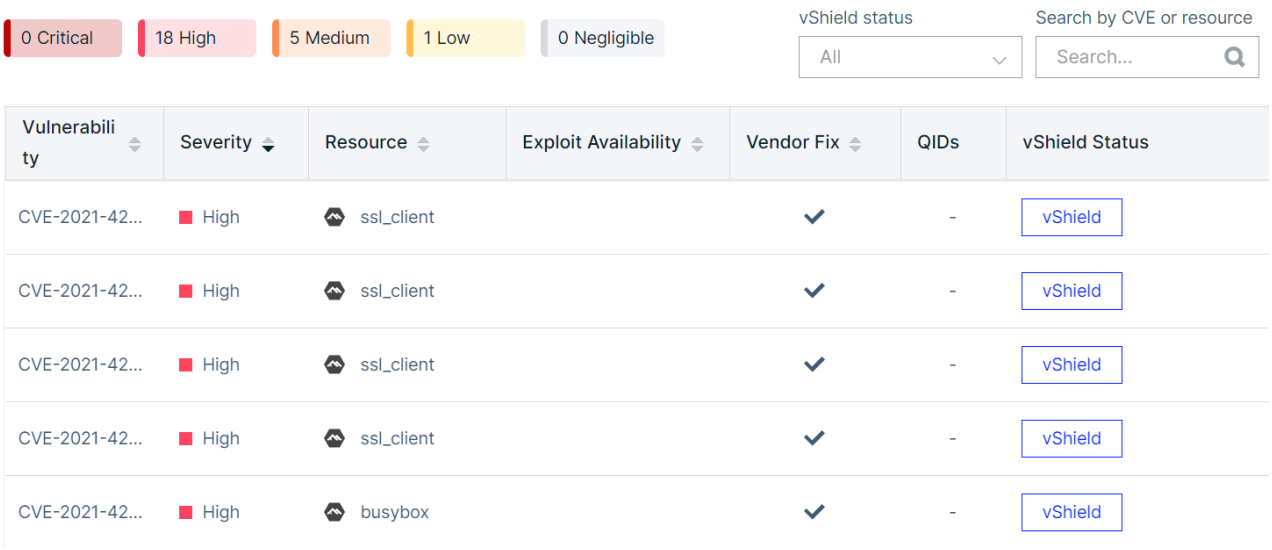

Figure 5 - Résultats d'Aqua Security

2.3.2 Hôte mal configuré

Les conteneurs s'exécutent sur des machines hôtes. Celle-ci peut être réelle ou virtuelle. De plus, les conteneurs et leur hôte partagent un noyau, si un hôte est compromis, tous les conteneurs que cet hôte exécute sont des victimes potentielles pour l'attaquant, particulièrement si ce dernier obtient des privilèges élevés (Hua, et al., 2021). Comment sécuriser un hôte Linux est un domaine important avec déjà beaucoup de littérature. Si nous nous concentrons uniquement sur le contexte des conteneurs, il est fortement recommandé d'exécuter des applications de conteneur sur des machines hôtes dédiées.

La configuration du gestionnaire de conteneurs sécurisée (accès restreint et authentification, communication cryptée, etc.) aide également à réduire le nombre de vecteurs d'attaques. Dans le cas de Docker, des outils comme *Docker Bench Security*[17] permettent de vérifier automatiquement si la configuration respecte les bonnes pratiques listées par le *Center for Internet Security*[18], une organisation à but non lucratif qui aide les entreprises à améliorer leur posture de sécurité.

[17] https://github.com/docker/docker-bench-security
[18] https://www.cisecurity.org/benchmark/docker/

L'utilisation de systèmes hôtes minimaux et centrés sur les conteneurs tels que *CoreOS*, *Red Hat Atomic* et *RancherOS* qui sont des systèmes d'exploitation pour exécuter des charges de travail conteneurisées de façon sécuritaire et à grande échelle peuvent également réduire la surface d'attaque. Ils peuvent également apporter de nouvelles fonctionnalités utiles, telles que l'exécution de services système dans des conteneurs.

Un orchestrateur de conteneurs, comme *Kubernetes*, est un programme qui automatise le déploiement, la gestion et la mise en réseau des conteneurs. L'utilisation d'un orchestrateur signifie que les interactions humaines avec l'hôte peuvent être minimisées, ceci permet de détecter plus facilement des tentatives de connexions non autorisées. Cependant, un orchestrateur est une plateforme complexe et nécessite d'être correctement configuré pour ne pas devenir un autre vecteur d'attaque[19].

2.3.3 Attaque Contre l'Orchestrateur

Un orchestrateur de conteneurs est utilisé pour automatiser et gérer des tâches telles que l'approvisionnement, le déploiement, la configuration, la planification, l'allocation des ressources et la sécurité des interactions entre les conteneurs (Islam Shamim, 2021). Plusieurs orchestrateurs sont disponibles sur le marché tel que :

- **Kubernetes** : Originellement développé par Google, il s'agit d'un orchestrateur pour le déploiement d'applications conteneurisées. Celui-ci a été inspiré par une décennie d'expérience dans le déploiement de systèmes évolutifs et fiables. Google a fait don du projet à la nouvelle CNCF ;
- **Nomad** : Développé par HashiCorp, il s'agit d'un orchestrateur de charge de travail flexible pour déployer et gérer tout type d'applications à l'aide d'un flux de travail unique et unifié. La différence majeure entre Kubernetes et Nomad est que ce dernier peut gérer des applications qui ne sont pas conteneurisées, ainsi que des tâches par lots, ce qui sauve énormément de temps et d'argent aux organisations qui nous pas encore conteneurisé leurs applications existantes.

[19] https://www.aquasec.com/cloud-native-academy/kubernetes-in-production/kubernetes-security-best-practices-10-steps-to-securing-k8s/

Le tableau 2 montre les principales différences entre les deux orchestrateurs :

Tableau 2 - Comparaison entre Kubernetes et Nomad

	Kubernetes	**Nomad**
Complexité	Plus complexe mais offre un niveau de contrôle plus élevé	Plus facile à démarrer, mais moins mature
Communauté	Communauté plus importante, fournissant des outils, des ressources et un soutien	Communauté très petite et donc moins d'outils, ressources et soutien
Coût	Coûts potentiellement plus élevés en raison de grandes équipes et d'une architecture plus exigeante	Nécessite des équipes plus petites, moins de serveurs et prend moins de temps
Type d'application	Principalement les applications conteneurisées sous Linux	Les applications natives, virtualisés (Java) sous Linux, Windows et MacOs.
Écosystème	Vaste et soutenu par la communauté	Étroitement lié aux produits d'HashiCorp

Même s'il existe différents orchestrateurs, les vecteurs d'attaques restent généralement les mêmes. Tout comme dans la section précédente, sécuriser un orchestrateur est un domaine important avec beaucoup de littérature. La matrice d'attaque suivante, développée par Microsoft (Threat matrix for Kubernetes, 2021) est un excellent résumé de celle-ci :

Tableau 3 - Matrice d'attaques de Kubernetes

Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery	Lateral Movement	Impact
Using Cloud credentials	Exec into container	Backdoor container	Privileged container	Clear container logs	List K8S secrets	Access the K8S API server	Access cloud resources	Data Destruction
Compromised images in registry	bash/cmd inside container	Writable hostPath mount	Cluster-admin binding	Delete K8S events	Mount service principal	Access Kubelet API	Container service account	Resource Hijacking
Kubeconfig file	New container	Kubernetes CronJob	hostPath mount	Pod / container name similarity	Access container service account	Network mapping	Cluster internal networking	Denial of service
Application vulnerability	Application exploit (RCE)		Access cloud resources	Connect from Proxy server	Applications credentials in configuration files	Access Kubernetes dashboard	Applications credentials in configuration files	
Exposed Dashboard	SSH server running inside container					Instance Metadata API	Writable volume mounts on the host	
							Access Kubernetes dashboard	
							Access tiller endpoint	

Il est important de garder en tête que les conteneurs sont souvent exécutés par Kubernetes ou un autre orchestrateur et cet aspect de la sécurité des conteneurs ne doit pas être négligé.

2.3.4 Images de Conteneur

Une image de conteneur est un modèle en lecture seule qui contient un ensemble d'instructions permettant la création de ce dernier et son exécution sur un engin de conteneur. En juin 2015, Docker et d'autres joueurs majeurs de l'industrie ont standardisé les spécifications des images de conteneurs, ainsi que leurs spécifications d'exécution. Les conteneurs construits grâce à Docker sont basés sur les images Docker (Merkel, 2014).

La figure 6 est un exemple de fichier de configuration Docker qui permet de déployer un server *Git*, un système de contrôle de version distribué :

```
FROM alpine:latest

ENV HOME /root

RUN apk --no-cache add \
    bash \
    git \
    openssh \
    && sed -i "s/#PasswordAuthentication yes/PasswordAuthentication no/" /etc/ssh/sshd_config \
    && sed -i "s/#PubkeyAuthentication yes/PubkeyAuthentication yes/" /etc/ssh/sshd_config \
    && echo -e "AllowUsers git\n" >> /etc/ssh/sshd_config \
    && echo -e "Port 22\n" >> /etc/ssh/sshd_config \
    && addgroup git \
    && adduser -D -S -s /usr/bin/git-shell -h /home/git -g git git \
    && mkdir -p /home/git/.ssh \
    && chown -R git:git /home/git \
    && passwd -u git

ENV HOME /home/git
EXPOSE 22
WORKDIR $HOME

COPY ./start.sh /
COPY create_repo /usr/bin/create_repo

ENTRYPOINT ["/start.sh"]
CMD ["/usr/sbin/sshd", "-D", "-e", "-f", "/etc/ssh/sshd_config"]
```

Figure 6- Définition d'une image pour un conteneur Docker

Si un attaquant a accès à la machine qui construit l'image d'un conteneur, il peut modifier ou influencer la façon dont l'image est construite par exemple en y injectant du code malicieux. Une fois l'image construite, elle peut être distribuée en l'entreposant dans un répertoire externe, celui-ci peut être public ou privé (Shu, Gu, & Enck, 2017).

Un registre de conteneurs est un système où les images de ceux-ci peuvent être entreposé et distribué. Docker Hub est le registre par défaut de Docker, mais il ne s'agit pas de la seule option. Amazon, Google et Microsoft offrent également des registres. L'entreposage d'une image dans un registre est généralement appelé un *push*, et sa récupération est appelé un *pull*. Les répertoires publics sont parfois utilisés pour distribuer des conteneurs malveillants. Par exemple, il y a eu des cas de programmes de minage de cryptomonnaie caché (Karn, Kudva, Huang, Suneja, & Elfadel, 2020). Ce genre de problème est dû au fait que les utilisateurs se questionnent rarement sur la sécurité d'une image avant de commencer à l'utiliser (Zerouali, Mens, Robles, & Gonzalez-Barahona, 2018). Pour y remédier, K. Soonhong et L. Jong-Hyouk (Kwon & Lee, 2020)

ont proposé un système de *Diagnostic de Vulnérabilité d'Image Docker*. Ce système détecte les vulnérabilités connues et évalue les images Docker en se basant sur les scores de vulnérabilités. Ce score est basé sur des formules qui incluent les métriques exploitables, les moyens par lesquels la vulnérabilité peut être exploitée et les impacts si elle l'est. En d'autres mots, elles reflètent les conséquences d'une attaque réussie.

2.3.5 Vulnérabilités d'échappement

Pour s'attaquer à l'hôte d'un conteneur, un adversaire peut exploiter une vulnérabilité qui lui permet de briser la virtualisation et donc de s'échapper d'un conteneur. Ce type d'attaque permet à l'attaquant d'exécuter du code directement sur l'hôte ou à un autre conteneur (Sultan, Ahmad, & Dimitriou, 2019). Par exemple, un conteneur exécutant un service avec distribution d'images peut partager le même hôte qu'un conteneur qui gère des informations sensibles telles que les numéros de d'assurance sociale ou les mots de passe. Sans isolation, si le conteneur de distribution d'images n'est pas sécuritaire, il peut permettre à un attaquant d'accéder à l'hôte et d'exploiter le conteneur de gestion des numéros d'assurance sociale et ce même si une sécurité renforcée est appliquée sur ce dernier.

Bien que les mécanismes essentiels utilisés pour créer des conteneurs, c'est-à-dire les groupes de contrôle, les espaces de noms et la capacité de pouvoir changer le répertoire racine, soient assez simples à comprendre, leurs interactions avec les autres fonctionnalités du noyau Linux peuvent être complexes. Du point de vue de la sécurité, la plus grande faiblesse des conteneurs est qu'ils partagent le même noyau que leur hôte. Les environnements d'exécution de conteneurs les plus utilisés, incluant *containerd* et *CRI-O*, sont maintenant matures et sécuritaires[20]. Cependant, il est toujours possible qu'il ait des failles permettant à du code malveillant de s'exécuter à l'intérieur d'un conteneur et de se propager dans l'hôte. Ce type de problème est connu sous le nom de *runecespace*. L'un des cas les plus récents a eu lieu en 2019[21]. Lin et coll. (Xin, et al., 2018) ont étudié 11 vulnérabilités qui permettent de briser l'isolement d'un

[20] https://thenewstack.io/a-security-comparison-of-docker-cri-o-and-containerd/
[21] https://redhat.com/security/vulnerabilities/runcescape

conteneur et d'obtenir une élévation des privilèges sur l'hôte et proposent un mécanisme de défense pour contrer ce genre d'attaque.

2.4 Machines Virtuelles

Tout comme un ordinateur physique, une machine virtuelle possède un processeur, de la mémoire vive et un disque dur. La différence entre une machine réelle et une machine virtuelle est simple. Dans le premier cas, les composants sont tangibles, c'est-à-dire qu'ils existent physiquement. Dans le deuxième cas, les composants sont virtualisés, en d'autres mots ils existent sous forme de code. Naturellement, une machine virtuelle doit rouler sur une machine physique qui s'occupera de faire la virtualisation.

La figure 7 montre les différentes couches nécessaires à la virtualisation des machines :

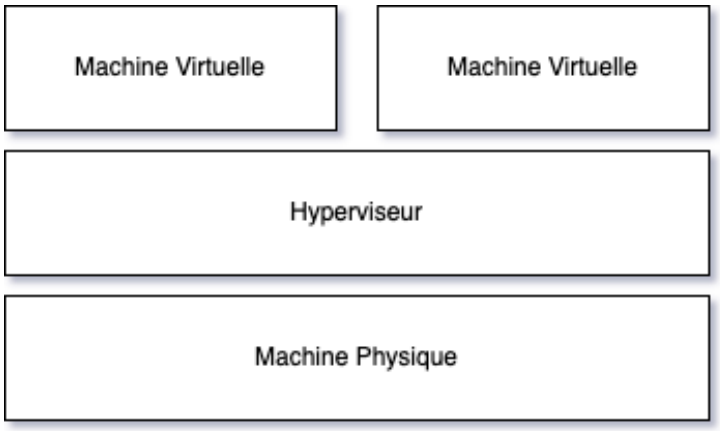

Figure 7- Virtualisation des machines virtuelles

Afin de pouvoir apprécier le fonctionnement des machines virtuelles, il est nécessaire de comprendre leurs mécanismes fondamentaux. En d'autres mots, les anneaux de protection du processeur et comment le moniteur de machines virtuelles créent s'occupe du cycle de vie de celles-ci (Aalam, Kumar, & Gour, 2021).

2.4.1 Anneaux de Protection

Un système d'exploitation moderne sépare généralement sa mémoire virtuelle en deux espaces, l'*espace utilisateur* et l'*espace noyau*. Principalement, cette séparation sert à fournir une protection de la mémoire et une protection matérielle contre les comportements logiciels

malveillants. Une application opère dans l'espace utilisateur. Elle doit donc utiliser le mécanisme des appels système afin que le noyau fasse certaines tâches pour elle, par exemple avoir accès à l'heure du système. Quant à lui, le noyau du système d'exploitation opère dans l'espace noyau. Ce niveau de privilège lui permet d'interagir avec la mémoire, le matériel, etc. Un rôle important du noyau est de gérer la mémoire vive du système d'exploitation. Effectivement, le noyau affecte des blocs de mémoire à chaque processus et s'assure que les processus ne peuvent pas accéder aux blocs de mémoire des autres (Harini & Fancy, 2020). Comme n'importe quel processus, le noyau est exécuté sur le processeur sous la forme d'instructions de code machine, naturellement ces instructions peuvent inclure des instructions privilégiées. Ce processeur utilise probablement l'architecture *x86* de Intel. Celle-ci organise les différents niveaux de privilèges en anneaux de protection. L'anneau 0 est celui avec le plus de privilèges. C'est là que le noyau opère. L'anneau 3 est celui avec le moins de privilèges. C'est dans celui-ci qu'une application est exécutée (Adams & Agesen, 2007). L'utilisation efficace de l'architecture en anneau nécessite une coopération étroite entre le matériel et le système d'exploitation.

La figure 8 montre les anneaux de privilèges pour l'architecture x86, principalement les processeurs d'Intel, disponibles en mode protégé :

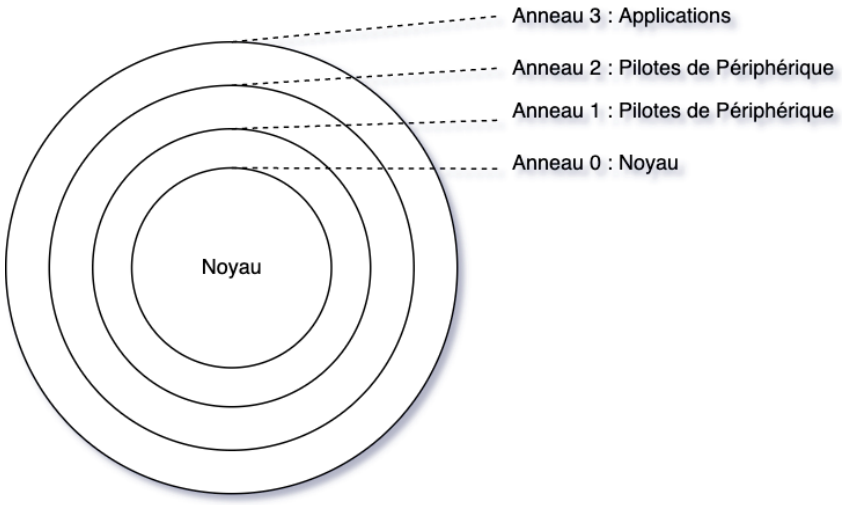

Figure 8- Anneaux de protection pour les processeurs x86

Dans le cadre de la virtualisation, un programme appelé *hyperviseur* est exécuté directement sur le matériel du système hôte dans l'*anneau 0*. La tâche de cet hyperviseur est de gérer l'allocation des ressources et de la mémoire pour les machines virtuelles.

2.4.2 Moniteur de Machine Virtuelle

Un moniteur de machine virtuelle est un logiciel qui permet la création, la gestion et la gouvernance de machines virtuelles. Il gère également le fonctionnement d'un environnement virtualisé au-dessus d'une machine hôte physique (Sahoo, Mohapatra, & Lath, 2010). Pour chaque machine virtuelle qu'il gère, il attribue de la mémoire vive, un pourcentage du temps de calcul du processeur, configure les divers périphériques virtuels et démarre un noyau invité avec accès à ces ressources. De plus, le moniteur de machine virtuelle est chargé de s'assurer que le système d'exploitation invité, le système d'exploitation installé sur une machine virtuelle, et ses applications ne puissent pas dépasser la limite des ressources qui lui ont été allouées. Il existe principalement deux types de moniteurs de machine virtuelle :

- **Type 1 :** Sur une machine physique, le bootloader exécute le noyau du système d'exploitation. Dans un environnement virtuel de Type 1, le *bootloader*, un programme qui est responsable du démarrage d'un ordinateur, est remplacé par un hyperviseur. De plus, le noyau du système d'exploitation invité est exécuté dans l'anneau de protection 1, ce qui signifie qu'il a moins de privilèges que l'hyperviseur ;
- **Type 2 :** Celui-ci s'exécute à l'intérieur d'un autre système d'exploitation. Une application dédiée qui gère les machines virtuelles comme VMWare est nécessaire afin que le système d'exploitation invité puisse coexister avec l'hôte. Ces systèmes d'exploitation invités n'ont pas besoin d'être les mêmes que celui de l'hôte.

La figure 9 montre les différentes couches de virtualisations entre un moniteur de machines virtuelles de type 1 et un moniteur de machines virtuelles de type 2 :

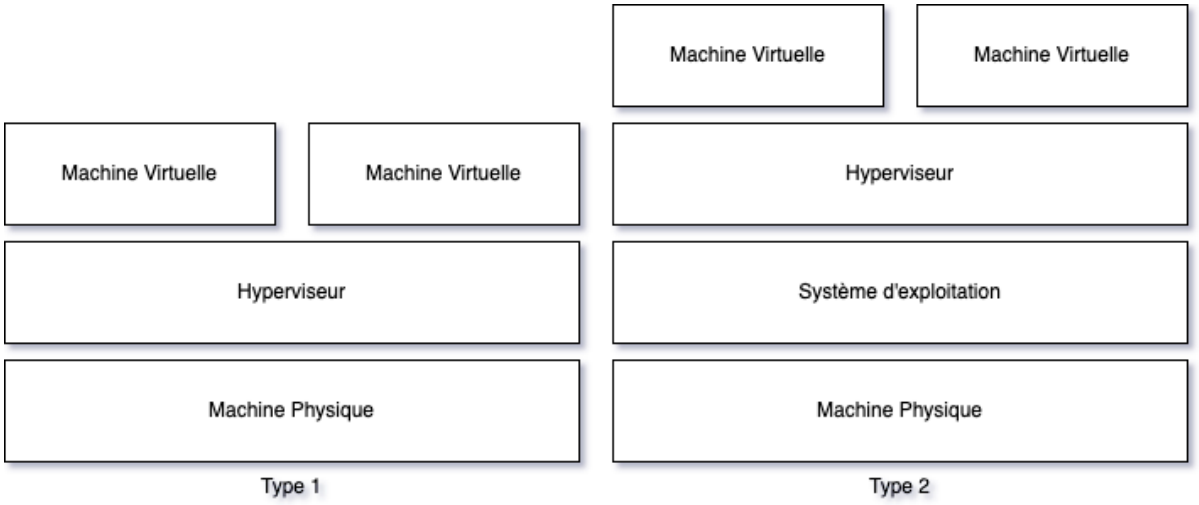

Figure 9- Types de moniteurs pour les machines virtuelles

Les machines virtuelles sont souvent comparées aux conteneurs, notamment parce que les deux technologies offrent une manière d'isoler une application du système hôte.

2.4.3 Différences entre une Machine Virtuelle et un Conteneur

Fondamentalement, la différence entre les deux technologies concerne la gestion de l'isolation entre une application et le système hôte. Comme nous l'avons vu précédemment, dans le cas des conteneurs, l'application a accès à un sous-ensemble du système d'hôte et partage le même noyau. Cependant, dans le cas des machines virtuelles, une copie complète d'un système d'exploitation, y compris le noyau, est virtualisée. (Laureano, Maziero, & Jamhour, 2004).

Le tableau 4 est une comparaison de différents aspects entre les machines virtuelles et les conteneurs :

Tableau 4 - Comparaison entre les machines virtuelles et les conteneurs

	Machines Virtuelles	Conteneurs
Système d'exploitation	Chaque machine virtuelle fonctionne sur du matériel virtualisé et a son propre noyau chargé en mémoire	Toutes les applications partagent le même système d'exploitation et le même noyau
Communication	Se produit via un périphérique Ethernets	Mécanismes IPC standard comme les signaux, les tuyaux et les soquets
Performance	Les machines virtuelles souffrent d'une baisse de performance, car les instructions doivent être traduite pour le système d'exploitation hôte	Les conteneurs offrent des performances presque natives, car les instructions ne doivent pas être traduite pour le système d'exploitation hôte
Temps de démarrage	Les machines virtuelles prennent plusieurs minutes pour démarrer	Les conteneurs peuvent être démarrés en quelques secondes, car le système d'exploitation hôte est déjà en cours d'exécution
Isolation	Le partage de bibliothèques et de fichiers entre les invités et les hôtes invités n'est pas vraiment possible	Les sous-répertoires peuvent être montés de manière transparente et peuvent être partagés
Sécurité	Dépends de la mise en œuvre de l'hyperviseur	Le contrôle d'accès obligatoire peut être utilisé

Nous avons exploré ce qu'est un conteneur, ce qu'est une machine virtuelle et la différence entre ces deux technologies, il est temps de voir les autres concepts nécessaires à la création de notre système en commençant par l'apprentissage machine.

2.5 Apprentissage Machine

L'apprentissage machine est la science et l'art de donner à un ordinateur la capacité d'apprendre grâce à des données. Certains nomment cette nouvelle manière de programmer *Software 2.0*, un terme inventé en 2017 par Andrey (Karpathy, 2017). Dans la manière classique de programmer, Software 1.0, le programmeur doit formellement définir le problème et écrire ligne par ligne un algorithme pour le résoudre. En faisant cela, il identifie un point spécifique dans

l’espace des programmes. Cependant, Software 2.0, qui a grandement gagné en popularité ces dernières années, consiste à amasser des données d’apprentissage, les nettoyer et les donner en entrées à un système d’apprentissage automatique qui produira une fonction approximative. (Dilhara, Ketkar, & Dig, 2021). À la base, un algorithme d'apprentissage automatique prend un ensemble de données d'entrainement et génère un modèle. Le modèle est un algorithme qui prend de nouveaux points de données sous la même forme que les données d'entrainement et génère une prédiction. Tous les algorithmes d'apprentissage automatique sont définis par les trois composants interdépendants suivants :

- **Une famille de modèles** : Elle décrit l'univers des modèles à partir desquels on peut choisir ;
- **Une fonction de perte** : Elle permet de comparer quantitativement différents modèles ;
- **Une procédure d'optimisation** : Elle permet de choisir le meilleur modèle dans la famille.

Il est possible de classer les systèmes d’apprentissage automatique en trois grandes catégories en fonction de la quantité de supervision humaine qu’ils ont durant la phase d’apprentissage. Dans la première famille, l’apprentissage supervisé, un humain supervise le système tandis que dans le second, l’apprentissage non supervisé, il n’y a pas d’intervention humaine. La troisième est un mélange des deux précédentes, dans le sens que l’apprentissage sera en parti supervisé par un humain.

La figure suivante est une représentation hiérarchique de ces trois grandes catégories, ainsi que les types de problèmes qu’elles tentent de résoudre :

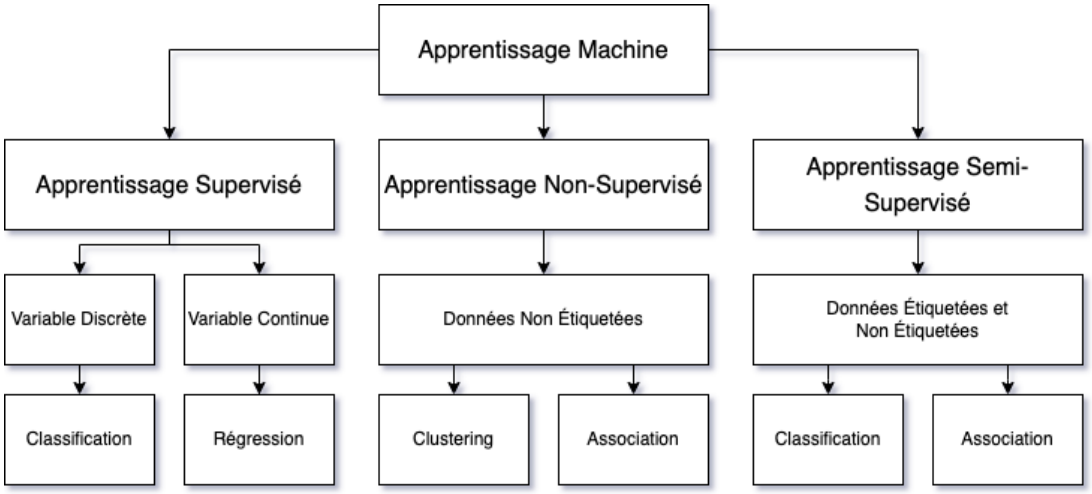

Figure 10 - Grandes Catégories d'Apprentissage Machine

2.5.1 Apprentissage Supervisé

En apprentissage supervisé, les données d'apprentissage sont fournies en entrée au système afin d'être mappées à la solution recherchée appelée la *cible*. Comme tous les algorithmes d'apprentissage automatique, l'apprentissage supervisé est basé sur l'entraînement. Au cours de sa phase d'apprentissage, le système est alimenté avec des ensembles de données étiquetées, qui indiquent au système quelle sortie est liée à chaque valeur d'entrée spécifique. Le modèle formé est ensuite présenté avec des données de test : il s'agit de données qui ont été étiquetées, mais les étiquettes n'ont pas été révélées à l'algorithme. L'objectif des données de test est de mesurer la précision avec laquelle l'algorithme fonctionnera sur des données non étiquetées. Une fois la phase d'apprentissage terminé, le système est en mesure de fournir la cible pour de nouvelles entrées qui n'ont jamais été vues (A Nichols, Chan, & Baker, 2019). Si la cible est exprimée en classes, alors il s'agit d'un problème de classification.

Le filtre antipourriel, comme on peut le voir dans la figure suivante, en est un bon exemple de cas d'utilisation de classification :

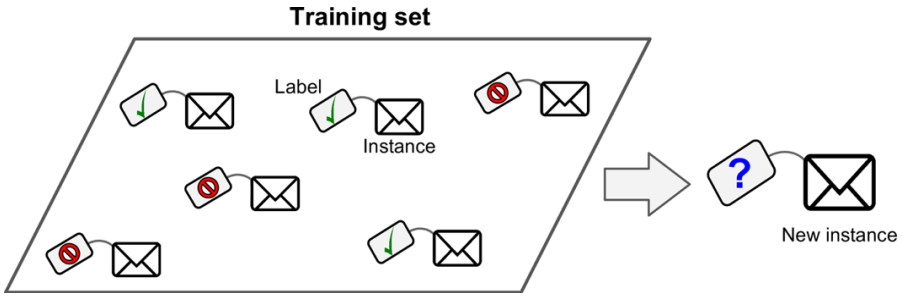

Figure 11 – Filtre antipourriel

Il peut également être utilisé pour résoudre des problèmes de régression, dans lesquels nous essayons de prédire la valeur d'une variable de nombre réel. Une version modifiée du cas d'utilisation précédent serait de prédire le nombre de pourriels qu'un employé recevra au cours d'un mois donné en fonction des données sur son poste, ses privilèges d'accès, son ancienneté dans l'entreprise, son score d'hygiène de sécurité, etc.

Malheureusement, les données d'entraînement étiquetées peuvent être un goulot d'étranglement important. Selon certaines estimations, cette étape peut représenter jusqu'à 80% du temps d'un projet[22]. Heureusement, il est possible de laisser le système apprendre par lui-même sur des ensembles de données non étiquetées. Dans ce cas, on parle d'apprentissage non supervisé.

2.5.2 Apprentissage Non-Supervisé

L'apprentissage non supervisé fait référence à l'utilisation d'algorithmes d'intelligence artificielle pour identifier des modèles dans les données. Dans ce mode d'apprentissage, le système essaie de différencier les données qui lui sont fournies en entrées en se basant sur les caractéristiques de celles-ci. En d'autres mots, il génère une relation entre les données (Sarker, 2021). L'un des algorithmes les plus importants ce de type d'apprentissage est le *clustering*. Étant donné un tas de points de données, lesquels sont similaires les uns aux autres ? L'algorithme consiste à analyser les données d'entrées et de les séparer dans différents groupes. Quand une nouvelle entrée est fournie, il mappe l'entrée au groupe approprié (Rodriguez, et al., 2019). Un exemple d'utilisation serait d'analyser un grand ensemble de données de trafic Internet vers un site web

[22] https://www.cognilytica.com/document/data-preparation-labeling-for-ai-2020

pour déterminer les différents groupes d'utilisateurs. Dans ce cas, les groupes possibles pourraient être, un utilisateur légitime, un robot d'exploration de Google ou encore un botnets.

La figure 12 montre comment l'algorithme pourrait faire les groupes :

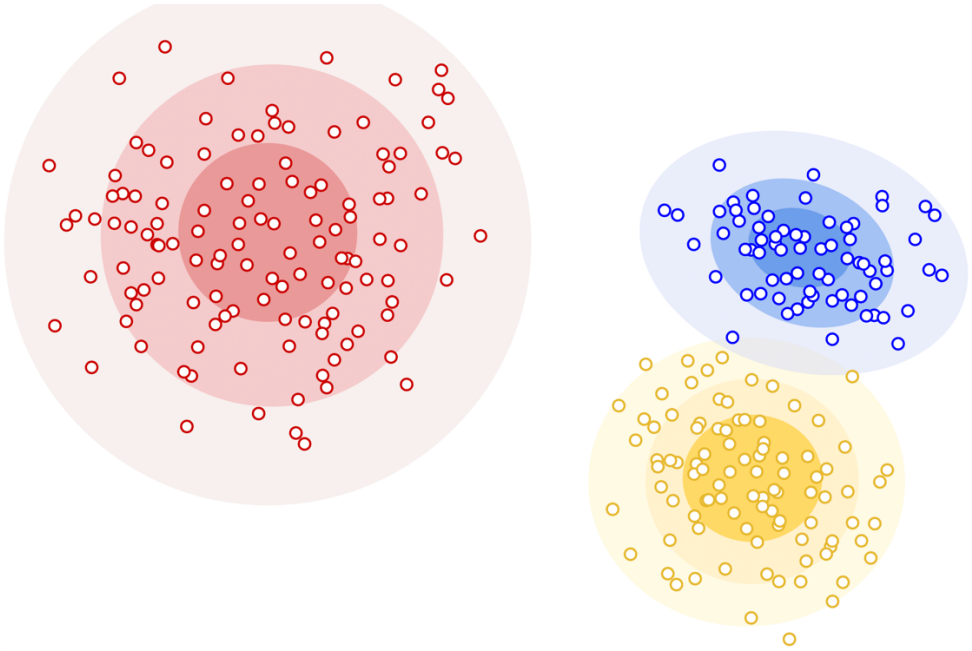

Figure 12 - Algorithme de regroupement

2.5.3 Apprentissage Semi-Supervisé

L'apprentissage semi-supervisé se situe quelque part entre les deux autres catégories (P. Bennett & Demiriz, 1999). Il résout principalement des problèmes de classification, ce qui signifie qu'un algorithme d'apprentissage supervisé doit être utilisé pour la tâche. Cependant, afin d'éviter d'avoir à étiqueter chaque exemple utilisé lors de la formation, un algorithme d'apprentissage non-supervisé peut être utilisé afin de créer les différents clusters.

Par exemple, supposons que nous voulons entraîner un modèle à classer les chiffres écrits à la main, mais que ceux-ci n'ont pas encore été étiquetés avec la réponse. Annoter chaque exemple serait une tâche laborieuse. Il est possible d'utiliser l'apprentissage semi-supervisé pour créer ce modèle[23]. En effet, il est possible d'utiliser l'algorithme *k-means* pour regrouper nos échantillons.

[23] https://github.com/ageron/handson-ml2/blob/master/09_unsupervised_learning.ipynb

K-means est un algorithme d'apprentissage non supervisé, rapide et efficace, ce qui signifie qu'il ne nécessite aucune étiquette (Li & Wu, 03). Celui-ci va calculer la similarité entre nos échantillons en mesurant la distance entre leurs caractéristiques.

Après la formation du modèle, nos données seront divisées en clusters dans lesquels les nombres que se ressemblent le plus seront regroupé. Par exemple, dans l'image suivante, on peut remarquer que le modèle a créé six clusters pour le chiffre « 6 », car ceux si sont écrit de manière trop différente.

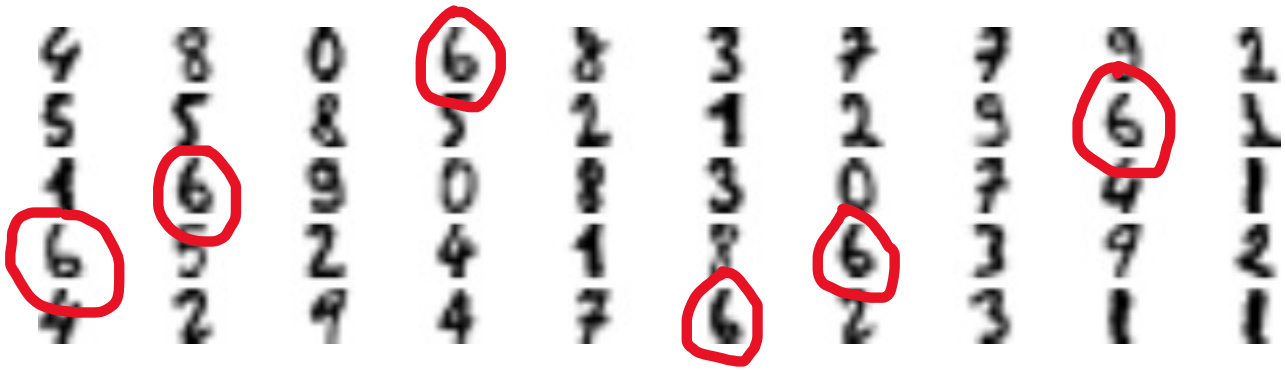

Figure 13 - Cluster de chiffres

Maintenant, il faut étiqueter ces images et les utiliser pour former notre deuxième modèle d'apprentissage automatique, le classifieur. Après avoir étiqueté les échantillons représentatifs de chaque cluster, nous pouvons propager la même étiquette à d'autres échantillons du même cluster. En utilisant cette méthode, il est possible d'annoter des milliers d'exemples en peu de temps.

2.5.4 Apprentissage Machine en Sécurité de l'Information

L'apprentissage automatique a le potentiel de rendre la sécurité des technologies de l'information plus simple, plus proactive, moins couteuse et bien plus efficace[24]. En effet, dans son livre « Introduction to Machine Learning with Applications in Information Security », Mark Stamp, fournit plusieurs applications réelles en sécurité des TI où les algorithmes d'apprentissage automatique peuvent être utilisés afin d'augmenter la sécurité des systèmes informatiques. Ces applications incluent la détection de virus et la classification des courriels. Ces algorithmes

[24] https://www.theguardian.com/media-network/2016/jan/28/ai-developers-think-smart-to-boost-cybersecurity

peuvent aussi aider dans la prévention des menaces et dans la réponse aux attaques actives en temps réel[25]. D'autres cas d'utilisation incluent :

- **Détection et classification des menaces** : Des algorithmes d'apprentissage automatique peuvent être implémentés dans des applications pour identifier et répondre aux cyberattaques avant qu'elles ne prennent effet (Dolev & Lodha, 2017)
- **Notation des risques du réseau** : L'apprentissage automatique peut être utilisé pour automatiser ce processus en analysant les ensembles de données historiques de cyberattaques et en déterminant quelles zones des réseaux étaient principalement impliquées dans certains types d'attaques ;
- **Automatisez les tâches de sécurité routinière** : En analysant les rapports d'actions passées prises par les analystes de sécurité pour identifier et répondre avec succès à certaines attaques il est possible de créer un modèle qui peut identifier des attaques similaires et répondre en conséquence sans intervention humaine.

Cependant, l'apprentissage automatique peut aussi bien être utilisé dans des activités d'attaque que de défense (Rege & Blanch K. Mbah, 2018). Par exemple, des études ont montré que les cybercriminels tirent parti de l'apprentissage automatique pour créer des logiciels malveillants intelligents capables de déjouer les systèmes de défense intelligents actuels[26]. D'autres cas d'utilisation malveillantes incluent :

- **L'accès non autorisé** : L'apprentissage machine peut être utilisé pour obtenir un accès non autorisé aux systèmes, tels que ceux impliquant des *captchas*, un test pour différencier les ordinateurs des humains (Wang, Chau, & Chen, 2017). Par exemple il est possible de l'utiliser pour créer un programme qui identifie et sélectionne les bonnes images de taxis de la figure 14 ;
- **Maliciel évasif** : Il y a eu des cas où l'apprentissage machine a été utilisé pour générer du code malveillant que d'autres programmes de sécurité n'ont pas pu détecter, y compris des systèmes basés sur l'apprentissage automatique (Weiwei & Tan, 2017)

[25] https://cset.georgetown.edu/publication/machine-learning-and-cybersecurity/
[26] https://www.businessinsider.com/darktrace-dave-palmer-artificial-intelligence-powered-malware-hacks

- **Hameçonnage** : L'apprentissage machine peut être utilisé pour mener des attaques d'hameçonnage avancées. Par exemple, il est possible de collecter des données de courriel envoyées par des organisations comme Netflix afin de générer automatiquement des courriels d'hameçonnage qui semblent légitimes.

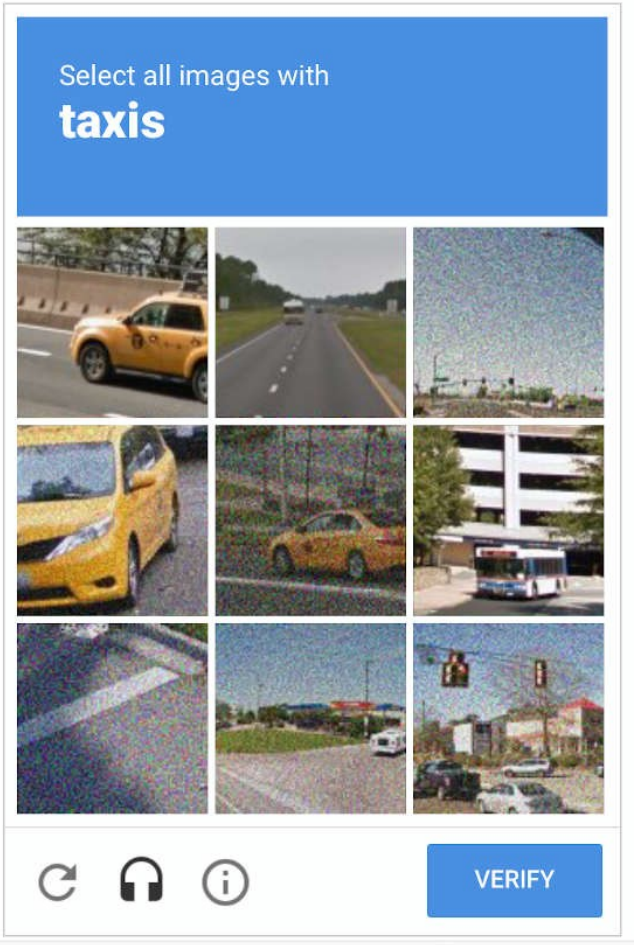

Figure 14 - Exemple de CAPTCHA

Un autre cas d'utilisation important, et qui nous intéresse le plus pour réaliser notre système, est la *détection d'anomalies*. Par exemple, la détection de transactions inhabituelles par carte de crédit pour prévenir la fraude. Ce genre de système est entraîné à l'aide d'instances normales. Lorsqu'il voit une nouvelle instance, il peut dire si elle ressemble à une instance normale ou s'il s'agit d'une anomalie. Cependant, avant d'y aller plus en détail, nous allons explorer les systèmes d'intrusion classique, c'est-à-dire sans apprentissage machine.

2.6 Système de détection d'intrusion

Les systèmes de détection d'intrusion jouent un rôle clé dans la sécurité du système en identifiant les attaques potentielles et en donnant des réponses appropriées. Le problème de l'intrusion dans des systèmes informatiques a été documenté pour la première fois par James P. Anderson dans les années 80s (Anderson, 1980). Le but de son étude était d'améliorer la capacité d'auditer des systèmes afin d'augmenter leur sécurité. Dans celle-ci, il introduit également une stratégie de détection d'intrusion basée sur l'analyse des fichiers journaux d'un système. Quelques années plus tard, au Stanford Research Institute International, un institut de recherche scientifique américain à but non lucratif, Dorothy Denning, propose le premier système de détection d'intrusion (Denning, 1987). Depuis, de nombreux systèmes de détection d'intrusion ont été proposés à la fois dans le monde de la recherche et dans le monde commercial.

2.6.1 L'architecture des systèmes de détection d'intrusion

Bien que ces systèmes soient extrêmement divers dans les techniques qu'ils emploient pour collecter et analyser les données, la plupart d'entre eux reposent sur un cadre architectural relativement général et sont composés de trois composants fonctionnels (Lazarevic, Kumar, & Srivastava, 2005). Le premier composant d'un système de détection d'intrusion est une source de données, celui-ci peut également être appelé *générateur d'événements*. Les données collectées pour l'analyse de l'intrusion peuvent être collectées à partir de sources variées. Il est possible de classer le premier composant en deux classes en fonction de la provenance des données :

- **Basé sur l'hôte** : est installé sur le système ou l'hôte local ;
- **Basé sur le réseau** : prend en compte l'ensemble de l'environnement réseau pour surveiller les activités à l'intérieur et à l'extérieur du réseau.

Le deuxième composant d'un système de détection d'intrusion est appelé le *moteur d'analyse*. Ce composant récupère les informations de la source de données et examine les données à la recherche de symptômes d'attaques ou d'autres violations de stratégie. Le moteur d'analyse peut utiliser une ou les deux approches d'analyse suivantes :

- **Anomalie** : Les modèles significatifs sont examinés pour refléter tout écart par rapport aux modèles normaux ;
- **Mauvaise utilisation** : Ici, le système est construit sur la base des vulnérabilités du système et des signatures d'attaque qui sont déjà connues. Il est possible de définir une signature d'attaque comme étant un arrangement unique d'informations qui peut être utilisé pour identifier la tentative d'un attaquant d'exploiter une vulnérabilité connue du système d'exploitation ou de l'application.

Le troisième composant d'un système de détection d'intrusion est le gestionnaire de réponses. Celui-ci n'agira que lorsque des inexactitudes, c'est-à-dire des intrusions possibles, sont trouvées sur le système, en émettant des réponses. Le mécanisme de réponses est la façon dont le système répond, les actions faites par celui-ci, lorsqu'une intrusion s'est produite. Il peut s'agir d'une réponse :

- **Active** : Le mécanisme de réponse peut être défini comme le système conçu pour bloquer les intrusions et les attaques instantanément au moment où elles sont détectées sans même concerner l'expert en sécurité ;
- **Passive** : Le mécanisme de réponse peut être défini comme le système conçu pour surveiller le trafic réseau en tyrannisant les opérations réseau ayant un schéma ou une activité réseau inhabituelle.

La figure 15 est une représentation hiérarchique des composants fonctionnels d'un système de détection d'intrusion, ainsi que les types de chacun :

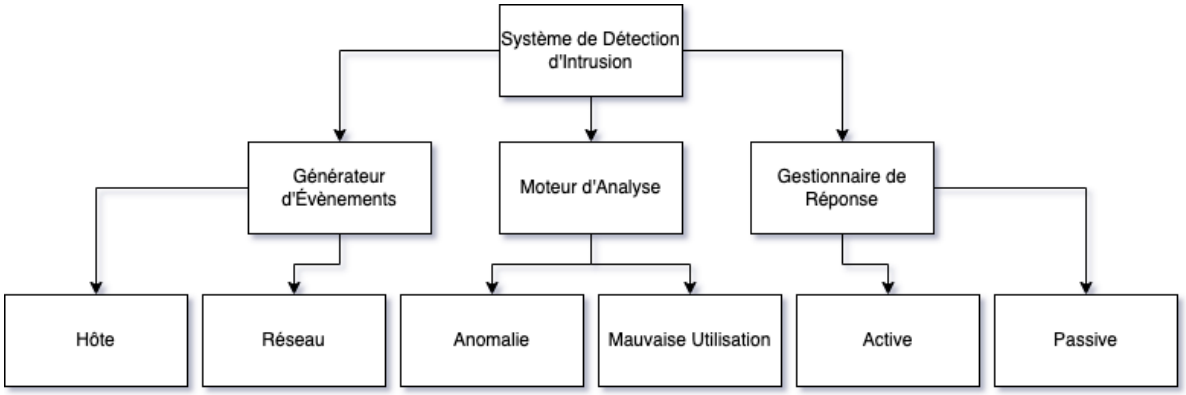

Figure 15 - Les composants d'un système de détection d'intrusion

2.6.2 Problèmes avec les systèmes existants

Naturellement, comme n'importe quel type de système, les systèmes de détection d'intrusion ne sont pas parfaits. Ceux-ci peuvent souffrir des problèmes suivants :

- Un problème important avec un système de détection d'intrusion est qu'il émet régulièrement **des faux positifs**. Dans de nombreux cas, les faux positifs sont plus fréquents que les menaces réelles ce qui met en doute l'efficacité du système [27] ;
- Un système de détection d'intrusion est l'une des pratiques de sécurité qui, par leur nature, **consomme beaucoup de ressource sur un hôte** (Benkhelifa, Welsh, & Hamouda, 2018). Celui-ci utilise en permanence des ressources supplémentaires dans le système qu'il surveille même lorsqu'aucune intrusion ne se produit pas, car les composants du système de détection d'intrusion doivent fonctionner en permanence ;
- Étant donné que les composants du système de détection d'intrusion sont implémentés en tant que programmes distincts, ils sont susceptibles d'être falsifiés. Un intrus peut potentiellement **désactiver ou modifier** les programmes exécutés sur un système, rendant le système de détection d'intrusion inutile ou peu fiable ;
- La base de données des signatures d'attaques **doit être mise à jour régulièrement** pour incorporer de nouvelles signatures d'attaques.

[27] https://www.rapid7.com/blog/post/2017/01/11/the-pros-cons-of-intrusion-detection-systems/

Maintenant que nous avons exploré les composants fonctionnels des systèmes de détection d'intrusion classique, ainsi que certains de leurs problèmes, nous allons voir comment ceux-ci peuvent être améliorés grâce à l'apprentissage machine.

2.6.3 Apprentissage Machine et Systèmes de Détection d'Intrusion

Dans les dernières années, les méthodes de détection basées sur l'apprentissage machine sont devenues un point chaud dans la recherche de systèmes de détection d'intrusion (Yang & Hui, 2015). Cependant, plusieurs défis dans la conception et la mise en œuvre de ces systèmes doivent être résolus avant qu'ils ne puissent être viables. En effet, les *données déséquilibrées* sont un problème pour la construction d'un système de détection d'intrusion. Les données déséquilibrées font généralement référence à un problème de classification où les classes ne sont pas représentées de manière égale.

Par exemple les opérations bancaires, comme nous pouvons le voir dans la figure 16, produisent ce genre d'ensemble de données :

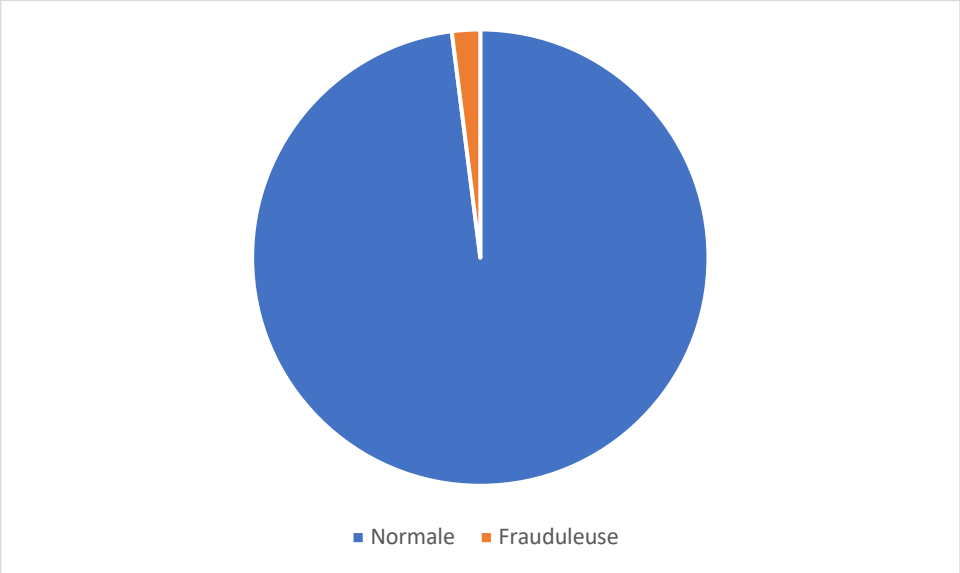

Figure 16 - Exemple d'ensemble de données déséquilibrées

Ces données affectent négativement les performances du modèle (Jiang, Wang, Wang, & Wu, 2020). En effet, elles se traduisent par des modèles qui ont de mauvaises performances prédictives, en particulier pour la classe minoritaire. Cependant, en 2018, l'entreprise

Greenhouse (Jun Lee, Gottschlich, Tatbul, Metcalf, & Zdonik, 2018) a réussi a créer un système de détection d'anomalies basé sur l'apprentissage machine de type *zéro positif*, c'est-à-dire qui ne nécessite pas la présence d'échantillons anormaux dans les données d'entrainement. Si nous revenons sur l'exemple des opérations bancaires, il serait possible de créer un système de détection d'anomalies qui serait entraîné uniquement avec des transactions normales. Pour y parvenir, l'équipe a dû utiliser un réseau de *mémoire à long et court terme.*

Récemment, des modèles d'apprentissage profond ont été explorés pour faire la détection d'anomalies (Yin, Zhu, Fei, & He, 2017). Les réseaux de *neurones récurrents*, et plus particulièrement leur variant les réseaux de *mémoire à long et cours terme*, sont excellent dans la détection de dépendances dans des données séquentielles (Olah, 2015). Dans le contexte de l'informatique, un exemple de données séquentielles serait des données d'évènements horodatées produites par un système, en d'autres mots des *logs*. Les réseaux de mémoire à long et court terme sont explicitement conçus pour éviter le problème de dépendance à long terme, c'est-à-dire de se souvenir d'informations pendant de longtemps. Cette propriété les rend utiles dans la détection d'intrusion, car l'exécution d'une attaque peut se produire sur une grande période.

Dans ce chapitre, nous avons vu les origines des conteneurs, comment ceux-ci fonctionnent et les différents vecteurs qui peuvent être utilisés pour les attaquer. Nous avons aussi vu les différences entre les conteneurs et les machines virtuelles. Par la suite, nous avons exploré les grandes catégories d'apprentissage machine pour enfin terminer avec l'architecture des systèmes de détections d'intrusion. Ces concepts sont nécessaires afin de comprendre *Cosmos*, le système que nous avons développé.

Chapitre 3 – L'architecture du Système

Dans ce chapitre, nous explorons l'architecture de base que nous proposons pour *Cosmos*, notre système. Celle-ci inclut un module de collecte de surveillance des appels système, un module de prédiction et finalement un module de seuil. De plus, une base de données externe peut optionnellement être ajoutée afin que les données collectées soient entreposées à l'extérieur du système hôte.

3.1 Diagramme d'Architecture

Le diagramme ci-dessous montre les différents modules de Cosmos et leur localisation par rapport au système d'exploitation hôte et aux applications conteneurisées. Comme on peut le voir, celui-ci n'a pas besoin d'être installé dans l'application conteneurisée. En d'autres mots, Cosmos n'a pas d'impact sur le déploiement ou le comportement du conteneur.

La figure 17 est un diagramme à haut niveau de la localisation de Cosmos et des différents modules qui le compose :

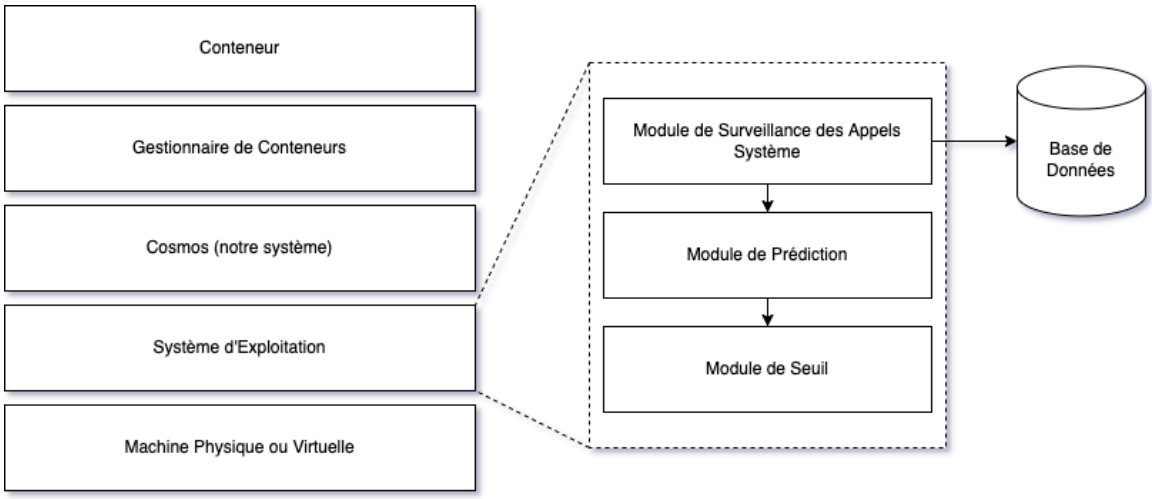

Figure 17 - Architecture du système

La base de données externe est optionnelle, les données sont entreposées par défaut dans un fichier texte sur l'hôte. Nous avons décidé d'utiliser *Elasticsearch*. Celle-ci est utilisée pour entreposer les données collectées sur l'hôte à l'extérieur de celui-ci dans l'optique de les garder

pour une investigation future ou encore les utiliser pour d'autres expériences. Les données collectées incluent :

- La date de l'appel système ;
- Le numéro du processus, ainsi que son nom ;
- L'appel système fait.

Afin d'améliorer la lisibilité et l'intégration avec d'autres types d'outils, les données sont converties dans le format *JSON*. Nous avons choisi ce format, car il s'agit du format d'échange de données les plus utilisées aujourd'hui[28].

La figure 18 est un exemple de données capturées par notre module de surveillance d'appels système convertie en JSON :


```
[
    {
        "serial"    : 369,
        "time"      : "2022-03-27 22:51:67.301",
        "name"      : "mysql",
        "pid"       : "1355",
        "data"      : {
            "syscall"   : {
                "syscall"   : ["times", "100"],
                "success"   : ["yes"],
                "exit"      : ["0"]
            }
        }
    }
]
```


Figure 18 – Données Capturées en JSON

Une fois la séquence capturée, nous pouvons utiliser les appels système dans celles-ci afin de détecter si un processus agit de façon normale ou non. Puisque Cosmos détecte les anomalies

[28] https://www.toptal.com/web/json-vs-xml.

d’une application conteneurisée grâce aux appels système qu’elle fait avec l’hôte, les deux assertions suivantes doivent être vraies pour que notre système puisse fonctionner :

1. **Les activités anormales apparaitront dans les arguments des appels système** : Si une attaque peut être effectuée sans effectuer d'invocations d'appels système ou sans affecter la valeur des arguments de ces invocations, alors Cosmos ne sera pas en mesure de la détecter ;
2. **Les arguments d'appel système utilisés dans l'exécution d'activités anormales diffèrent des valeurs utilisées lors de l'exécution normale d'une application** : Si une attaque peut être effectuée en utilisant des valeurs d'argument d'appel système qui ne peuvent être distinguées des valeurs utilisées pendant l'exécution normale, alors l'attaque ne sera pas détectée par Cosmos. La capacité à identifier des valeurs anormales dépend de l'efficacité et de la sophistication du modèle à créer des profils pour les fonctions d'appel système. Un bon modèle devrait rendre extrêmement difficile l'exécution d'une attaque sans être détectée.

Avec ses deux assertions en têtes, nous pouvons maintenant décrire les différents modules qui seront implémentés dans Cosmos.

3.2 Module de Surveillance des Appels Système

Comme nous l’avons vu dans le chapitre 2, les appels système sont une manière par laquelle un programme dans l’espace utilisateur peut demander un service au noyau du système d’exploitation sur lequel il est exécuté, par exemple la lecture de fichiers. Puisque les appels système sont la façon dont tous les logiciels communiquent avec le noyau et que les conteneurs partagent le même noyau que l’hôte, nous avons besoin de les collecter en **temps réel** afin de pouvoir faire la détection d’anomalie.

3.2.1 Trace

Il existe déjà certaines solutions pour y parvenir, nous aurions pu collecter les appels système en utilisant un des utilitaires suivants :

- **Dtrace** : Créé à l'origine par Sun Microsystems[29], une entreprise technologique américaine connue pour avoir créé Java, afin de résoudre les problèmes de noyau et les problèmes d'applications sur les systèmes de production en temps réel ;
- **Ptrace** : Utilisé par les débogueurs, comme *gdb*[30], un débogueur portable qui s'exécute sur de nombreux systèmes de type UNIX, et d'autres outils d'analyse de code, il est principalement utilisé pour aider au développement de logiciels ;
- **Strace** : Créé à l'origine pour le système d'exploitation SunOS[31] dans le but de surveiller et altérer les interactions entre les processus et le noyau Linux.

Cependant, ceux-ci ont été développés pour auditer les appels système dans le but de faire du débogage et non pour faire de la **surveillance en continu**. Dans le contexte de Cosmos, c'est ce deuxième but qui nous intéresse, sans cela ce dernier ne sera pas en mesure de détecter des activités anormales en temps réel.

Par exemple si une application, sous Linux, veut interagir avec un fichier, celle-ci devrait faire les appels suivants pour y parvenir :

1. **open :** Ouvre simplement le fichier, pour effectuer des opérations telles que la lecture et l'écriture ;
2. **read :** Cet appel système ouvre le fichier en mode lecture. Il n'est pas possible de modifier un fichier avec cet appel système. Plusieurs processus peuvent exécuter simultanément cet appel sur le même fichier ;
3. **write :** Ouvre le fichier en mode écriture. Il est possible d'éditer un fichier avec cet appel système. Un seul processus peut exécuter cet appel sur le même fichier à la fois ;
4. **close :** Cet appel système ferme le fichier ouvert.

Dans le cas où une application compromise par un attaquant commencerait à écrire dans des fichiers à une fréquence hors de l'ordinaire, il est primordial que ces appels système soient capturés en temps réel afin que Cosmos puisse réagir rapidement.

[29] http://dtrace.org/blogs/about/
[30] https://man7.org/linux/man-pages/man2/ptrace.2.html
[31] https://man7.org/linux/man-pages/man1/strace.1.html

3.2.2 Linux Auditing System

Une alternative, pour collecter en temps réel les appels système, est d'utiliser le *Linux Auditing System*. Il s'agit d'une fonctionnalité du noyau Linux qui peut enregistrer les appels système en fonction de règles qui peuvent être définies par un administrateur. Le Linux Auditing System a l'avantage d'exister depuis très longtemps, depuis Linux 2.6, et de vivre dans le noyau, donc nous n'avons pas besoin d'installer d'autres logiciels sur l'hôte pour le faire fonctionner (Ma, et al., 2018). Du point de vue de la sécurité, un de ses nombreux avantages est qu'avec ce système, il est possible de verrouiller la configuration d'audit afin que celle-ci ne puisse pas être modifiée après sa configuration. Toute tentative de modification de la configuration sera auditée et refusée. La configuration ne peut être modifiée qu'en redémarrant la machine, ce qui peut déclencher d'autres alertes.

Le système se compose de deux éléments principaux :

- **Du code dans le noyau** pour accrocher les appels système ;
- **Un daemon**, un programme qui s'exécute en arrière-plan, plutôt que d'être sous le contrôle direct d'un utilisateur, dans l'espace utilisateur qui enregistre les évènements d'appel système[32].

Le système fonctionne sur à l'aide d'un ensemble de règles qui définissent ce qui doit être capturé dans les fichiers journaux. Il est capable de surveiller les trois choses suivantes (Zeng, Xiao, & Chen, 2015) :

- **Appels système** : Qui ont été appelés, ainsi que des informations contextuelles telles que les arguments qui lui sont transmises, les informations utilisateur, etc. ;
- **Accès aux fichiers** : Il s'agit d'un autre moyen de surveiller l'activité d'accès aux fichiers, plutôt que de surveiller directement l'appel système ouvert et les appels associés ;
- **Règles de Contrôle** : Utilisé pour modifier la configuration et les paramètres du système d'audit lui-même.

[32] https://linux.die.net/man/8/auditd

L’image suivante montre la syntaxe utilisée pour définir une règle d'appel système, le cas qui nous intéresse, car c'est le moyen universel pour comprendre le comportement des applications. À partir des évènements d'appel système, nous pouvons savoir, par exemple, quel fichier a été créé, lu ou écrit :

```
1
2   auditctl -a action,filter -S system_call -F field=value -k key_name
3
```

Figure 19 - Syntaxe pour définir une règle d'appel système

Supposons que nous voulons enregistrer un événement chaque fois que quelqu'un lit le fichier « /var/log/secure ». Ce fichier contient tous les messages liés à la sécurité, y compris les échecs d'authentification. Afin d’effacer ses traces, un acteur malveillant s’assurait tenterais de modifier ce fichier.

```
1
2   auditctl -w var/log/secure -p rwxa
3
```

Figure 20 - Exemple d'utilisation d'auditctl

Désormais, chaque fois que vous accédez à « /var/log/secure », le noyau génère un événement. Par défaut, le système d'audit entrepose les entrées de journal dans le fichier suivant « /var/log/audit/audit.log » et le format des entrées rend leur utilisation difficile. Principalement parce que les évènements peuvent être sur une ou plusieurs lignes et qu’ils peuvent arriver dans le désordre.

Un autre problème avec ce système d'audit est qu’il n'existe pas encore d'espace de noms dans le noyau. C’est-à-dire que les données auditées par le système ne contiennent pas de détails sur le conteneur, celles-ci contiennent des détails sur tous les appels système effectués sur un hôte par les applications conteneurisées ou non, mais il n’y a aucune mention du conteneur qui a effectué l’appel système. Nous avons donc décidé de réimplémenter le daemon dans l’espace utilisateur afin de pouvoir plus facilement auditer les conteneurs en nous **connectant**

directement au noyau via le *netlink audit socket* (Morris, 2009). De plus, notre implémentation peut envoyer des données, en format JSON, à des bases de données externes.

3.2.3 Implémentation

Dans cette section, nous allons expliquer comment nous avons réalisé notre **réimplémentation** partielle du Linux Auditing System afin que celui-ci soit utilisable par Cosmos. Le daemon de conteneur, habituellement Docker, génère un *containerd-shim* qui agit comme un adaptateur vers le gestionnaire de conteneur. Il agit également comme un daemon qui surveille le conteneur réel[33]. Le processus *shim* est le processus parent des processus conteneurisés réels dans l'arborescence des processus. Ainsi, chaque conteneur a un processus de shim de conteneur en cours d'exécution.

La figure suivante illustre l'arborescence des processus pour les conteneurs Docker que nous venir de décrire :

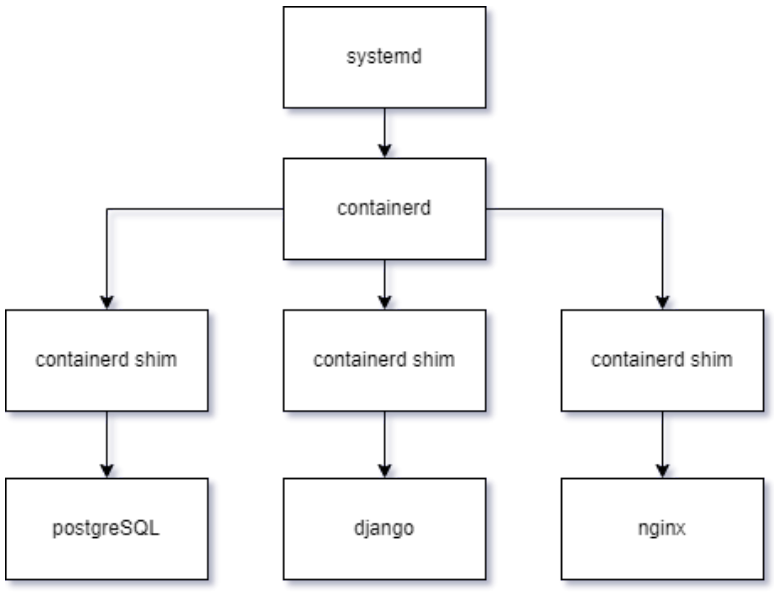

Figure 21- Arbre de Processus pour les Conteneurs Docker

Comme nous pouvons le remarquer dans la figure 21, chaque conteneur a un processus *containerd-shim* en cours d'exécution. En remontant l'arborescence des processus, il est donc possible de savoir si une application est conteneurisée ou non. En effet, si nous remontons

[33] https://iximiuz.com/en/posts/implementing-container-runtime-shim/

jusqu'au processus avec l'identifiant 1, c'est-à-dire *systemd,* le processus qui est responsable du démarrage et de l'arrêt du système d'exploitation sans avoir trouvé de shim, alors l'application n'est pas conteneurisée. Cependant, si nous remontons et trouvons un processus shim, alors l'application est conteneurisée. Notre réimplémentation utilise cette propriété afin de différencier les différents processus qui sont en cours d'exécution sur l'hôte.

Les étapes suivantes représentent l'algorithme que nous avons créé pour y parvenir :

```
1   Input: PID
2   Function: EstUnConteneur {
3       if (PID == 1) return false
4       else if (PID == containerShimProcesses ) return true
5       else EstUnConteneur(PPID.parent)
6   }
7   Output: Booléen
```

Figure 22 - Psedocode pour savoir si un processus est conteneurisé ou non

Tel que :

- **PID :** le numéro d'identification d'un processus ;
- **PID.parent :** le numéro d'identification du processus parent ;
- **containerShimProcesses :** une liste contenant les numéro d'identification des containerd-shim.

Pour Cosmos, puisque nous nous intéressons aux environnements conteneurisés, les applications non conteneurisées ne sont pas auditées. De plus, remonter l'arborescence au complet à chaque fois n'est pas efficace au niveau des ressources, nous avons donc ajouté un système de mise en cache pour réduire cette utilisation.

Maintenant que notre module de collecte d'appels système est capable de faire cette distinction, nous pouvons nous concentrer sur les étapes pour la collecte de données. La journalisation commence au démarrage du conteneur et se termine lorsque le conteneur termine son exécution. Les étapes de collecte de ces données sont décrites ci-dessous :

1. A la création d'un nouveau conteneur, les appels système faits par celui-ci sont **audités** avec notre module de collecte d'appels système ;
2. Les appels système sont **filtrés** pour ne consigner que les appels système du ou des conteneurs choisis ;
3. Si la connexion est configurée et disponible, les appels système effectués par le conteneur vers le système hôte sont **envoyés** à une base de données externe, sinon ils sont entreposés dans un fichier sur la machine hôte ;
4. Les données envoyées à la base de données externe ou à la machine hôte sont **traitées** pour créer une liste complète des appels système récupérés à partir du conteneur.
5. Quand le conteneur termine son exécution, le système **libère** les ressources allouées à son audit.

Une fois les appels système collectés en temps réel, ceux-ci sont envoyés vers le module de prédiction qui, étant donné une nouvelle séquence d'appels système, calcule la probabilité du prochain appel dans une séquence en fonction des appels précédents.

3.3 Module de Prédiction

Le prochain module est le module de prédiction. La prédiction du mot suivant, dans notre cas de l'appel système suivant, également appelé la modélisation du langage, est une tâche qui consiste à prédire le prochain mot d'une phrase. C'est l'une des tâches fondamentales dans le traitement du langage naturel. Cette tâche peut être basée sur un réseau de neurones pour déterminer la probabilité qu'une séquence donnée de mots se produise dans une phrase (Jozefowicz, Vinyals, Schuster, Shazeer, & Wu, 2016).

Dans notre contexte, nous pouvons imaginer une séquence d'appels système comme étant un langage. En effet, pour pouvoir communiquer avec le système d'exploitation, une application devra être en mesure de « parler » cette langue. Nous pouvons aussi considérer ce langage comme étant universel, car peu importe si l'application est écrite en C/C++, Java ou encore Python, ils utiliseront les mêmes appels système. En écoutant une application communiquer avec le système d'exploitation, il nous sera possible de comprendre le comportement de celle-ci et de prédire ses prochains mots.

Sur cette base, nous pouvons effectuer diverses tâches pour détecter des séquences d'appels système anormales. Par exemple, il est possible d'estimer la probabilité relative de différents appels système dans des contextes différents.

L'image suivante est une séquence d'appels système, de mots, et peut être interprétée comme étant une phrase dite par une application dans ce langage :

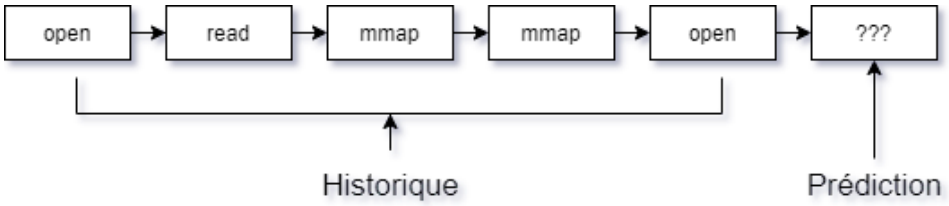

Figure 23 – Prédiction du prochain appel système basé sur l'historique

Considérons un langage produit par un programme arbitraire, par exemple un serveur web HTTP comme Apache Tomcat, en d'autres mots sa séquence d'appels système. On peut raisonnablement présumer que la mémoire d'une machine, dans le monde réel, n'est pas une ressource infinie et donc que le système hôte possède un **nombre fini d'appels système**. En ce moment, Linux x86_64 en a un total de 332[34].

Un modèle de langage attribue des probabilités à des séquences de symboles arbitraires de telle sorte que plus une séquence $(w_1, w_2, \dots, w_n)$ est susceptible d'exister dans ce langage, plus sa probabilité est élevée. La plupart des modèles de langage estiment cette probabilité comme un produit de la probabilité de chaque symbole compte tenu de ses symboles précédents :

$$P(w_1, w_2, \dots, w_n) = p(w_1)p(w_2|w_1)p(w_3|w_1, w_2) \dots p(w_n|w_1, w_2, w_3, \dots, w_{n-1})$$

$$= \prod_{i=1}^{n} p(w_i|w_1, \dots, w_{i-1})$$

Étant donné qu'un processus typique exécute une longue chaine d'appels système, le nombre d'appels système requis pour bien comprendre la signification d'une séquence d'appels système est assez important. De plus, les appels système sont entrelacés les uns avec les autres de manière compliquée. Les limites entre les séquences d'appels système sont également vagues. À

[34] https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md

cet égard, l'apprentissage de la dépendance à long terme est crucial pour concevoir des systèmes de détection d'intrusion efficaces (Gao & Lin, 2021). Afin de remédier à ce problème, nous utilisons l'architecture de réseau de neurones récurrents artificiels suivant : mémoire à long et cours terme. Ce type de réseau est particulièrement bien adapté à la modélisation du langage, car il permet la modélisation exacte de la probabilité d'une séquence de mots (Sundermeyer, Schluter, & Ney, 2012). Ceci aide à atténuer le problème de **rareté des données** qui se produit dans d'autres types de modèles plus classiques.

En supposant notre deuxième assertion qui était que les arguments d'appel système utilisée dans l'exécution d'activités anormales diffèrent des valeurs utilisées dans l'exécution d'activités normales, étant donné une nouvelle séquence d'appel système, celle-ci peut être classé comme étant une séquence normale, qui n'a pas été compromise, si elle est semblable avec celles vues durant la phase d'entraînement. Dans le cas contraire, la séquence est classifiée comme étant une séquence anormale, qui a été compromise. Une fois la vraisemblance calculée, celle-ci est envoyée à notre dernier module, le module de seuil, qui décide s'il s'agit d'une séquence de nature normale ou anormale.

3.4 Module de seuil

La plupart des systèmes de détection d'intrusion, incluant Cosmos, utilisent un classificateur de seuil (Almseidin, Alzubi, Kovacs, & Alkasassbeh, 2017). La valeur seuil décrit une valeur limite qui est utilisée afin de déterminer si une donnée doit être considérée comme étant normale ou non. Si le score d'anomalie d'un point de données dépasse le seuil prédéfini, il est marqué comme anormal. La valeur seuil détermine la sensibilité du système à réagir aux conditions anormales et représente un hyperparamètre qui peut être ajusté. Le module de seuil est configuré à l'aide de l'hyperparamètre θ et fonctionne grâce à la fonction suivante :

$$F(x) = \begin{cases} normal, & x \leq \theta \\ anormal\,, & autrement \end{cases}$$

Les appels système effectués par un conteneur vers le système hôte sont comparés au modèle d'apprentissage machine pour prédire la nature de ces derniers. Celle-ci peut être normale, sans des données d'attaque ou anormale, avec des données d'attaque. Si le modèle rapporte un score

de probabilité élevée, alors la séquence est similaire à la séquence utilisée pour entrainer le modèle. Sinon, la séquence d'appels système est signalée comme une anomalie. Notre système évalue une séquence d'appels système avec un score d'anomalie, qui sert de base à la prise de décision. Sur la base d'un seuil prédéfini, les observations sont classées comme normales ou anormales par rapport à leur score d'anomalie.

3.5 Exemple d'utilisation de Cosmos

Dans cette section, nous allons voir comment Cosmos fonctionne à l'aide d'un exemple pédagogique. Supposons que nous avons une application conteneurisée qui désire enregistrer un fichier. Celle-ci va (1) envoyer les appels système *open*, *write, writre, …, close* au noyau du système d'exploitation hôte. Ensuite, puisque Cosmos est connecté au *netlink audit socket,* il sera en mesure (2) d'intercepter cet appel système. Par la suite, il va (3) l'ajouter à la *séquence d'appels système.* Cette séquence d'appel système sera (4) envoyée au module de prédiction. Celui-ci va (5) calculer la vraisemblance de cette séquence par rapport aux séquences d'entrainement et l'envoyer au module de seuil. Finalement, (6) ce score sera évalué par rapport à la valeur de seuil pour déterminer s'il y a eu une intrusion. Puisque dans cet exemple, enregistrer un fichier n'est pas une activité suspecte, Cosmos évaluera la séquence comme étant non suspecte.

La figure suivante aide à visualiser les différentes séquences qui auront lieu durant l'utilisation de Cosmos :

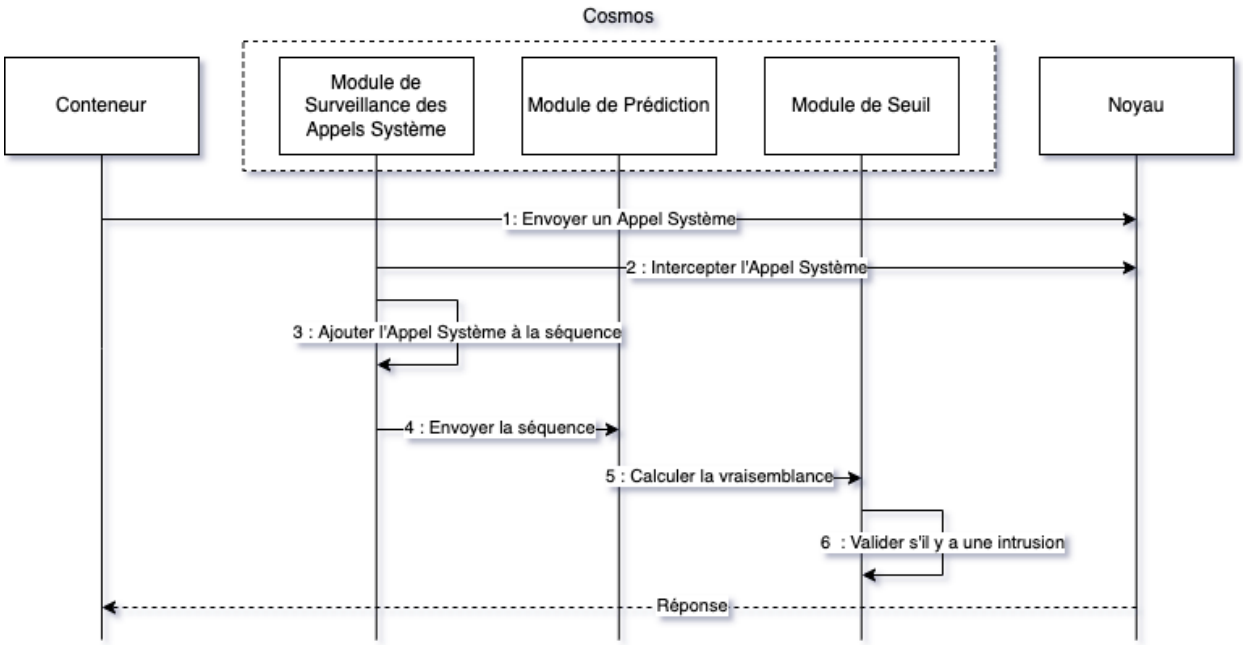

Figure 24 - Diagramme de séquence

Dans ce chapitre, nous avons présenté l'architecture de Cosmos en incluant le fonctionnement de ses différents modules. Nous avons vu les deux assertions nécessaires au fonctionnement de Cosmos qui étaient que les traces d'attaques apparaitraient dans les arguments des appels système et qu'il serait possible d'utiliser ses appels système afin de déterminer si une séquence provenait d'une sans attaque. Finalement nous avons exploré le fonctionnement de Cosmos à l'aide d'un exemple pédagogique.

Chapitre 4 – La Mise en Œuvre du Système

Dans ce chapitre, nous donnons les détails de la configuration que nous avons utilisée pour nos expériences, les technologies que nous avons choisies afin de réaliser l'implémentation du projet, les ensembles de données utilisées et comment nous avons entrainé le modèle qui sera utilisé par le système. La section sur nos ensembles de données explique pourquoi nous en avons choisis deux, ce qu'elles contiennent et comment elles ont été générées.

4.1 Configuration de l'Environnement

Notre système et les données générées afin de réaliser nos expériences ont été faits sur la même machine qui avait la configuration suivante :

- Système d'exploitation : Ubuntu 20.14 LTS (Long Term Support)
- Processeur : Intel Core i7 8700K
- Mémoire vive : 16 Go
- Disque dur : 500 Go

Nous avons décidé d'utiliser Ubuntu pour être notre système d'exploitation, car il s'agit d'une distribution Linux moderne, actuellement installée par de nombreux utilisateurs dans le monde[35]. Il s'agit aussi de la distribution la plus utilisée pour créer des environnements conteneurisés hébergés dans l'infonuagique, ce qui ajoute plus de réalisme à nos expériences. De plus, nous avons entièrement mis à jour le système d'exploitation avant nos expérimentations et nous avons gardé ses paramètres par défaut.

4.2 Implémentation du Système

Dans cette section, nous allons voir les technologies qui ont été sélectionnées afin de réaliser l'implémentation de Cosmos. Nous allons aussi expliquer en quoi celles-ci vont nous aider à atteindre nos objectifs.

[35] https://ubuntu.com/desktop/developers

4.2.1 Go - Portabilité et Performance

Un de nos objectifs était de développer un système qui n'**affecterait pas de façon négative les performances du système hôte**. Pour y parvenir, les trois modules de Cosmos, c'est-à-dire le module de collecte d'appels système, le module de prédiction et le module de seuil ont été implémentés en *Go*. Ce langage de programmation est un langage compilé, statiquement typé et concurrent inspiré par C conçu par Google. Après la compilation, Go produit un fichier binaire unique qui contient toutes les dépendances de l'application, ce qui facilite grandement la distribution de celui-ci. La compilation dynamique est possible avec *CGO* mais par défaut Go utilise la compilation statique. Bien que la taille soit clairement au détriment des binaires Go, il existe des moyens de minimiser ses effets.

L'image suivante montre la différence entre une application qui utilise des librairies statiques et une application qui utilise des libraires partagées (dynamique) :

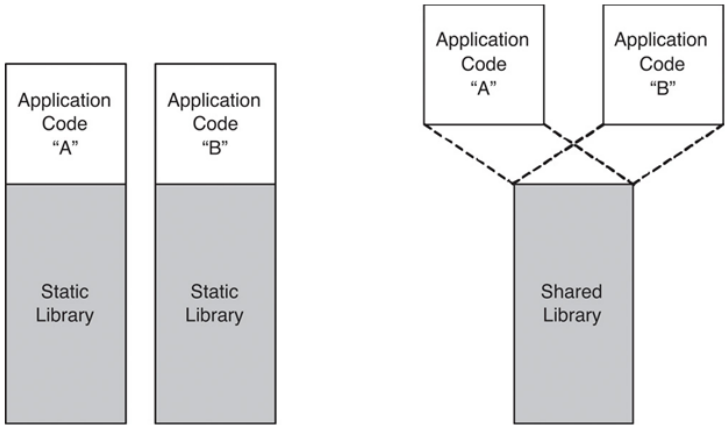

Figure 25 - Librairies Satiques vs Dynamiques

Dans le contexte éphémère des environnements conteneurisés, une application qui utilise des librairies statiques facilite grandement la distribution. Effectivement, nous avons été en mesure de compiler Cosmos pour le système d'exploitation Linux et de distribuer le même fichier binaire sur différentes distributions.

La plupart des environnements conteneurisés sont hébergés dans l'infonuagique. De façon générale, les fournisseurs de service infonuagique utilisent le modèle de facturation *payez pour ce que vous utilisez*. La pratique est similaire à celle des factures de services publics, en utilisant

uniquement les ressources nécessaires. Notre système doit être donc petite et performante afin de minimiser le coût. Dans ce contexte, nous avons décidé de compresser Cosmos à l'aide d'*UPX*. En ce qui concerne les performances, Go atteint un niveau qui est assez proche du C et qui ne serait pas possible à atteindre avec des langages interprétés comme Python[36]. Afin d'avoir de meilleures performances, nous aurions pu faire Cosmos en C ou C++, cependant nous aurions dû gérer son utilisation de la mémoire, car ces langages, contrairement à Go, n'ont pas de ramasse-miettes pour libérer les allocations supprimées (Engheim, 2021).

Nous pouvons maintenant nous concentrer sur un autre de nos objectifs qui était d'utiliser l'apprentissage machine afin que Cosmos performe mieux dans la détection d'anomalie que les systèmes traditionnels.

4.2.2 TensorFlow - Apprentissage Machine

La partie apprentissage machine de Cosmos a été réalisé en Python, plus précisément avec la librairie *TensorFlow*. Il s'agit d'une librairie développée par Google, devenue très populaire auprès de la communauté, car les APIs qu'elle offre facilitent grandement le développement d'applications qui utilisent l'apprentissage automatique. Nous avons choisi cette librairie principalement pour diminuer le temps requis pour entrainer notre modèle. En effet, TensorFlow permet de distribuer la phase d'entrainement sur plusieurs machines[37]. Cette tendance de distribuer la phase d'apprentissage a commencé en 2017 (Goyal, et al., 2017), lorsque Facebook a publié un article montrant un ensemble de méthodes pour réduire le temps d'apprentissage d'un modèle. Un moteur distribué fait partie du répertoire standard de TensorFlow et fonctionne exceptionnellement bien avec les opérations et fonctionnalités existantes.

Le figure 26 montre les gains en temps quand la phase d'entrainement est distribuée sur plusieurs cartes graphiques. Ces gains en temps ne sont pas uniques à Cosmos et s'appliquent sur les phases d'entrainement en général :

[36] https://getstream.io/blog/switched-python-go/
[37] https://ai.googleblog.com/2016/04/announcing-tensorflow-08-now-with.html

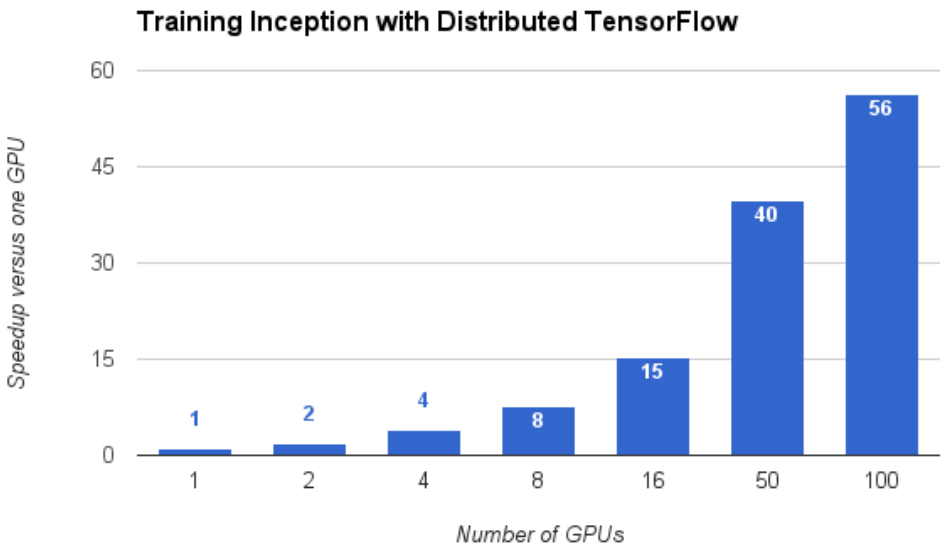

Figure 26 - Gains de Temps pour l'entrainement Distribué[38]

La phase d'entrainement des modèles demande une grande puissance de calculs. Il y a présentement une tendance à l'échelle de l'industrie vers la spécialisation du matériel dans le but de réduire le temps nécessaire à la réalisation de cette phase (Emma Wang, Wei, & Brooks, 2019). Il est possible de grandement réduire le temps d'entrainement si celui-ci est fait sur une carte graphique, mais dans le contexte actuel de la pandémie, celles-ci sont rares[39]. Puisque nous n'avons pas de carte graphique à notre disposition, TensorFlow était un choix logique, car il offre la possibilité de connecter notre environnement local à Google Cloud afin de faire cette phase sur l'infrastructure de Google. De plus, Google Cloud nous offre la possibilité d'utiliser des *Tensor Prosessing Unit*. Il s'agit d'un circuit intégré créé comme accélérateur développé par Google spécifiquement pour l'apprentissage automatique. L'intégration avec TensorFlow est transparente[40]. Selon Google, un Tensor Prosessing Unit fourni des performances 15 à 30 fois supérieures et à celles des CPU et des GPU[41].

Naturellement, cette réduction du temps d'entrainement va nous permettre de faire des itérations plus rapides et donc de passer de la recherche et l'expérimentation à la production

[38] https://ai.googleblog.com/2016/04/announcing-tensorflow-08-now-with.html
[39] https://www.pcmag.com/news/inside-the-gpu-shortage-why-you-still-cant-buy-a-graphics-card
[40] https://cloud.google.com/tpu/docs/tpus
[41] https://cloud.google.com/blog/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu

plus rapidement. Pour faciliter l'accès aux données d'appels système collecté par Cosmos, afin de réaliser nos expériences, celles-ci ont été entreposées dans une base de données externe qui n'était pas installée sur l'hôte.

4.2.3 Elasticsearch - Base de Données Externe

Par défaut, les données capturées par le module de collecte d'appels système sont enregistrées dans un fichier texte qui se trouve sur l'hôte où le conteneur est exécuté. Cependant, Cosmos offre aussi la possibilité d'envoyer les données collectées vers une base de données externe. Nous avons décidé d'utiliser Elasticsearch. Cette base de données permet de stocker, rechercher et analyser de gros volumes de données en temps quasi réel. La principale différence entre Elasticsearch et les bases de données de type SQL est qu'en plus d'entreposé les données, celui-ci peut les indexer, c'est-à-dire qu'il les organise afin de réduire le temps des requêtes, ce qui le rend idéal pour l'analyse des données de journalisation comme celles que Cosmos collecte. Cependant, bien qu'Elasticsearch et que SQL utilisent des termes différents pour la façon dont les données sont organisés (et une sémantique différente), leur objectif est essentiellement le même.

Le tableau suivant compare la correspondance des concepts d'ElasticSearch pour un utilisateur de base de données relationnel[42] :

Tableau 5 - Concepts d'ElasticSearch et de SQL

ElasticSearch	SQL
Index	Base de données
Shard	Partition
Type	Table
Document	Rangé
Field	Colonne
Implicite	Schéma
- (Tout est indexé)	Index

[42] https://www.elastic.co/guide/en/elasticsearch/reference/current/sql_and_elasticsearch.html

QUERY DSL	SQL

Bien que la correspondance entre les concepts ne soit pas exactement un à un et que la sémantique soit quelque peu différente, il y a plus de choses en commun que de différences entre ces deux types de bases de données. Puisque nous venons d'explorer où les données allaient être entreposées, il est temps de se concentrer sur les données en tant que telles.

4.3 Ensembles de Données

Afin de réaliser nos expériences, nous avons utilisé deux ensembles de données. Le premier ensemble est l'ensemble de données de *ADFA-LD*[43]. Nous l'avons utilisé afin de vérifier que Cosmos est capable d'utiliser les appels système faits entre une application et le noyau de l'hôte afin de détecter des anomalies et valider notre première assertion qui était que les activités anormales apparaitront dans les arguments des appels système. Cet ensemble de données a été choisi, car nous n'en avons pas trouvé qui avait été générés directement à partir d'un environnement conteneurisé. Le deuxième ensemble de données a été généré à partir d'une **base de données MySQL conteneurisée** et de l'application *mysqlslap*. Il s'agit d'une application qui fait l'émulation des charges clients sur un serveur MySQL. Nous avons utilisé ce programme pour émuler différentes charges client. Cet ensemble de données correspond aux appels système qu'un conteneur, contenant un serveur MySQL, effectue vers le système d'exploitation hôte. Dans les sous-sections suivantes, nous allons explorer l'ensemble de données ADFA-LD et les étapes suivies afin de générer notre ensemble de données pour une application conteneurisée.

4.3.1 Ensemble de Données ADFA

ADFA est un ensemble de données utilisé dans le contexte des systèmes de détection d'intrusion publiée par l'Académie Nationale de Défense Australienne. Celle-ci est largement utilisée pour tester divers produits de détection d'intrusion et contient divers appels système qui ont été caractérisés. Les différents vecteurs d'attaques utilisés pour générer cet ensemble sont visibles dans le tableau 7. L'ensemble comprend des données de deux systèmes d'exploitation distincts, Linux (ADFA-LD) et Windows (ADFA-WD), qui enregistrent respectivement diverses séquences

[43] https://research.unsw.edu.au/projects/adfa-ids-datasets

d'appels système de ceux-ci. Les séquences d'appels système sont largement considérées comme le moyen le plus précis pour détecter les intrusions (Kadar, Tverdyshev, & G, 2012). Nous allons uniquement nous intéresser à celles de Linux puisque les conteneurs sont presque exclusivement exécutés sur ce système d'exploitation.

L'ADFA-LD représente un digne successeur de l'ensemble de données *KDD*, une référence bien connue dans la recherche de techniques de détection d'intrusion (Kumar Sharma, 2015). Ce nouvel ensemble de données utilise une version du système d'exploitation Linux moderne et les méthodes utilisées pour le compromettre sont aussi récentes, mais surtout accessibles au public (Creech & H, 2013). L'ensemble de données est divisé en trois groupes que nous pouvons voir dans le tableau 6. Chaque groupe contient la séquence d'appel système d'origine. Les données de l'ensemble d'entrainement et de l'ensemble de validation ont été collectées pendant le fonctionnement normal de l'hôte. Les séquences d'appels système ont été collectées grâce au Linux Auditing System, le même moyen que nous avons décidé d'utiliser.

Le tableau souvent montre comment les données d'ADFA-LD ont été divisées afin de faciliter les diverses phases nécessaires à la réalisation d'un projet d'apprentissage automatique :

Tableau 6 - Nombre d'appels système dans différentes catégories d'ADFA-LD

	Trace	**Appels Système**
Données d'entrainement	833	308 077
Données de validation	4372	2 122 085
Données d'attaque	746	317 388
Total	5951	2 747 550

Les types d'attaques utilisées pour générer ADFA-LD sont des techniques couramment utilisées par les testeurs d'intrusion et les pirates informatiques ce qui augmente le réalisme de celle-ci. Bien que les attaques à distance soient très puissantes et désastreuses pour la victime, elles sont difficiles à exécuter contre un système d'exploitation moderne. Un exemple de ce type d'attaque,

souvent présenté dans les cours d'introduction à la sécurité informatique, est *MS08-067*[44]. Cette vulnérabilité permet l'exécution de code à distance. Un attaquant pourrait exploiter cette vulnérabilité sans s'authentifier pour exécuter du code arbitraire.

Nous avons choisi d'utiliser l'ensemble ADFA-LD pour vérifier notre assertion que les appels système pouvaient être utilisés dans la détection d'anomalie. Afin de vérifier que l'hypothèse est aussi valide pour les applications conteneurisées, nous avons décidé de générer notre propre ensemble de données qui proviendrait d'une application conteneurisée.

4.3.2 Ensemble de Données Générées

Initialement, puisque nous n'avions pas trouvé d'ensembles de données qui contenaient des appels système pour des applications conteneurisées, nous avons effectué plusieurs expériences afin de collecter les données du conteneur à différentes étapes d'exécution. L'idée était de collecter des données sur la base d'une utilisation normale et d'utiliser les données pour créer un profil d'utilisation normal. Finalement, notre ensemble de données a été généré à partir d'un conteneur Docker exécutant la base de données officielle MySQL. Il s'agit essentiellement d'une installation de base de données dans un système d'exploitation Linux, cependant il faut mapper le port du conteneur à un port du système hôte.

L'image ci-dessous montre à quoi ressemble un vecteur d'appels système que Cosmos a capturé :

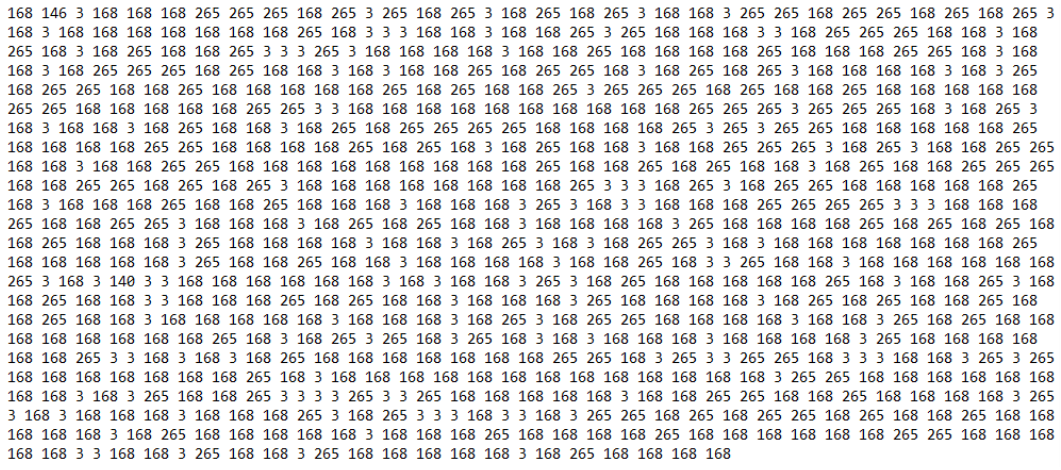

Figure 27 – Séquence d'appels système

[44] https://docs.microsoft.com/en-us/security-updates/securitybulletins/2008/ms08-067

Pour nous aider à générer notre ensemble de données **d'activités normales**, c'est-à-dire sans trace d'attaque, nous avons utilisé l'utilitaire *mysqlslap*. Il s'agit d'un programme qui émule des requêtes pour les serveurs MySQL. Ces requêtes sont simulées comme si plusieurs clients accédaient simultanément au serveur, ce qui ajoute du réalisme. L'outil dispose de plusieurs options qui permettent de sélectionner le niveau de concurrence et le nombre d'itérations pour exécuter le test de charge. Il offre aussi la possibilité de personnaliser la base de données créée, par exemple en spécifiant le nombre de colonnes à utiliser lors de la création de la base de données. De plus, il est possible de choisir le nombre d'insertions et de requêtes à effectuer sur la base de données. Pendant que mysqlslap simule les charges client sur la base de données exécutée dans le conteneur, Cosmos est utilisé pour capturer tous les appels système que le conteneur effectue vers le système d'exploitation hôte.

La figure suivante montre comment nous avons généré les données d'activités normales :

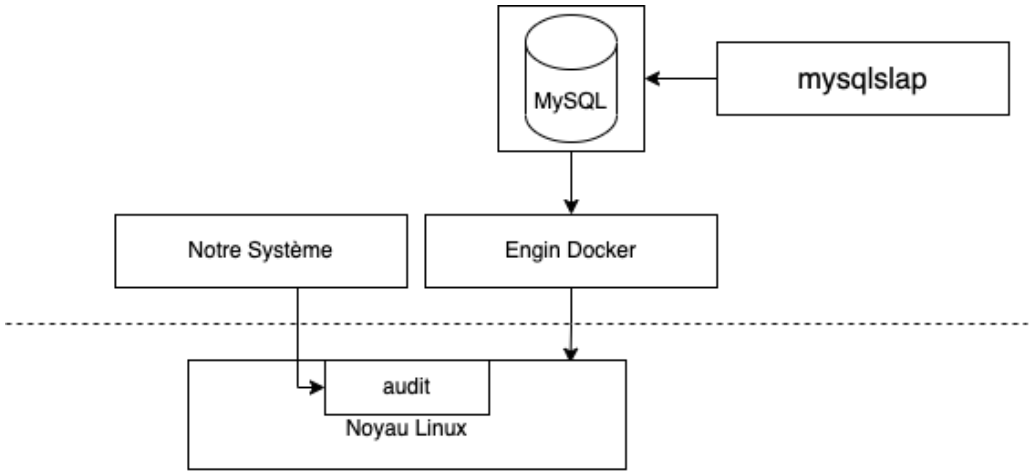

Figure 28 - Génération de données normales

Nous devons maintenant générer les données contenant des **activités anormales**, c'est-à-dire avec des traces d'attaques. Pour simuler une attaque sur le conteneur, nous avons utilisé *sqlmap*. Il s'agit d'un outil d'*injection SQL* automatique couramment utilisé à des fins de test d'intrusion.

L'injection SQL est une technique d'injection de code qui peut potentiellement altérer ou encore détruire une base de données. Cela se produit généralement lorsqu'un utilisateur doit entrer des

informations, comme son nom d'utilisateur et son mot de passe, mais au lieu d'un nom et mot de passe celui-ci donne une instruction SQL qui vont s'exécuterez sur la base de données. En exploitant une vulnérabilité, un attaquant peut l'utiliser pour contourner à la fois les mécanismes d'authentification et d'autorisation d'une application afin de récupérer des informations vitales de la base de données (Ojagbule, Wimmer, & Haddad, 2018).

Par exemple, La figure suivante est un pseudocode exécuté sur un serveur Web qui s'occupe d'authentifier un utilisateur grâce à son nom d'utilisateur et son mot de passe.

```
1   uname = request.POST['username']
2   passwd = request.POST['password']
3
4   sql = "SELECT id FROM users WHERE username='" + uname + "' AND password='" + passwd + "'"
5
6   database.execute(sql)
```

Figure 29 - pseudocode pour authentifier un utilisateur

Ce code est vulnérable à l'injection SQL. Un attaquant pourrait utiliser des commandes SQL dans l'entrée d'une manière qui modifierait l'instruction SQL exécutée par le serveur de base de données. Il pourrait utiliser le mot de passe suivant : « password' OR 1=1 ». Par conséquent, le serveur de base de données exécutera la requête SQL suivante :

```
1
2    SELECT id FROM users WHERE username='username' AND password='password' OR 1=1'
3
```

Figure 30 - Requête SQL

En raison de l'instruction « OR 1=1 », la clause « WHERE » renvoie le premier identifiant de la table des utilisateurs, quels que soient le nom d'utilisateur et le mot de passe.

Nous avons choisi ce type d'attaque, car OWASP (Open Web Application Security Project) classe l'injection SQL au premier rang de sa liste de risques liés aux applications web les plus critiques (OWASP Top Ten, 2021). Pendant que sqlmap est utilisé pour attaquer notre serveur de base de données conteneurisé, générer un ensemble de données de comportement malveillant en attaquant la base de données, Cosmos est utilisé pour capturer tous les appels système que le conteneur effectue vers le système d'exploitation hôte.

La figure suivante montre comment nous avons généré les données contenant des attaques. La différence entre celle-ci et la figure 31 est que sqlmap et mysqlslap sont exécutés en simultané :

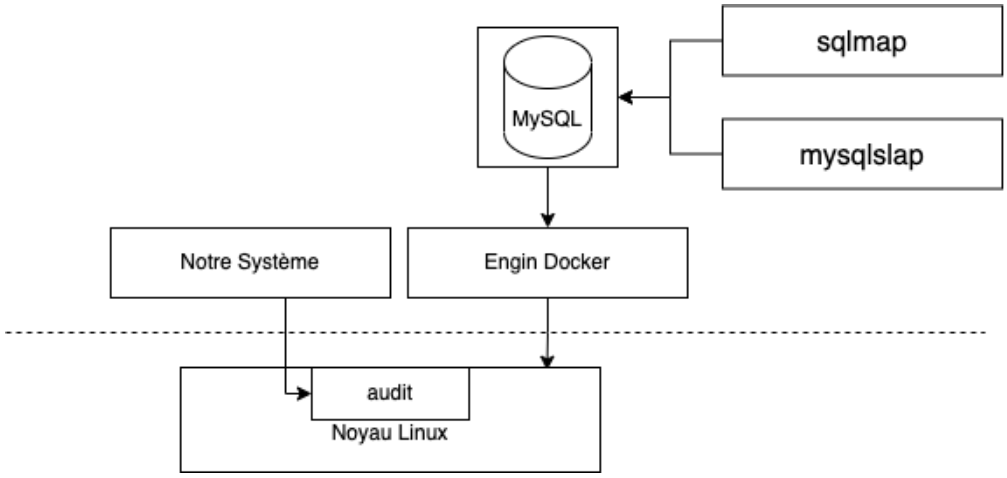

Figure 31 - Génération de données anormales

Maintenant que nous avons deux ensembles de données, la prochaine étape de la mise en œuvre du projet est de les utiliser afin d'entraîner le modèle qui sera utilisé par Cosmos pour faire la détection d'anomalie.

Dans ce chapitre, nous avons exploré les technologies que nous avons sélectionnées afin de réaliser l'implémentation de Cosmos. Nous avons aussi vu les ensembles de données utilisées pour entraîner le modèle et comment celles-ci ont été générées.

Chapitre 5 – Expériences et Résultats

Dans ce chapitre, nous présentons les algorithmes sélectionnés afin de comparer leurs performances à celles de Cosmos, les métriques d'évaluation, nos expériences et finalement une analyse des résultats obtenus. La première expérience avait pour but de démontrer qu'il était possible d'utiliser les appels système faits entre une application et le noyau hôte pour détecter des anomalies dans le système. Les autres avaient pour but de démontrer l'efficacité de notre système sur une application conteneurisée.

5.1 Algorithmes Sélectionnés pour les Expériences

Afin d'évaluer les performances de Cosmos, nous allons utiliser quatre algorithmes d'apprentissage machine sur nos deux ensembles de données. Chacun de ces algorithmes provient d'une catégorie différente :

- **Arbres de Décision (DTs)** : Plus précisément J48. L'arbre de décision J48 est l'implémentation de l'algorithme ID3 (Iterative Dichotomiser 3) développé par l'équipe projet WEKA. Nous l'avons sélectionné, car il s'agit d'un des meilleurs algorithmes d'apprentissage automatique pour examiner les données catégorielles (Saravanan & Gayathri, 2018) ;
- **K-Moyennes (KMC)** : Nous avons selectionné cet algorithme de partitionnement de données, car il est largement utilisé dans la détection d'anomalies (Xi, Tang, & Hu, 2011)
- **Naïve Bayésienne (NB)** : Nous l'avons sélectionné, car il est couramment utilisé pour prédire la probabilité d'appartenance à une classe.
- **Machine à Vecteur de Support (SVM)** : Parfois utilisé dans la détection d'anomalie, il a été démontré qu'il peut non seulement atteindre une performance satisfaisante, mais aussi réduire considérablement le coût de calcul (Xie, Hu, & Jill Slay, 2014).

Nous avons décidé d'utiliser ces algorithmes parce qu'ils sont largement acceptés dans la littérature et qu'ils permettent une détection plus élevée de précision que les autres méthodes d'apprentissage automatique.

5.2 Métriques d'Évaluation

Une fois qu'un modèle a été entrainé, il est important de mesurer la validité avec laquelle le modèle fait des prédictions, si cette étape est ignorée, nous ne serons pas en mesure de savoir si les résultats du modèle sont valides.

Afin de savoir si notre modèle possède des performances satisfaisantes, nous devons choisir une mesure de précision des résultats compatibles avec nos objectifs. Dans notre cas, puisque nous désirons savoir si les activités d'une application conteneurisée sont normales ou non. Quatre résultats sont possibles (Munaiah, Meneely, Wilson, & Short) :

- **Vrai Positif (VP)** : Nombre de valeurs d'événement correctement prédites ;
- **Faux Positif (FP)** : Nombre de valeurs d'événement prédites de manière incorrecte ;
- **Vrai Négatif (VN)** : Nombre de valeurs sans événement correctement prédites ;
- **Faux Négatif (FN)** : Nombre de valeurs sans attaque mal prédites.

Nous avons utilisé ces métriques d'évaluation, car elles sont largement utilisées dans le domaine de la recherche d'informations (Manning, 2008). Il est possible de les utiliser afin de créer une *matrice de confusion.* Une matrice de confusion est un tableau souvent utilisé pour décrire les performances d'un modèle de classification sur un ensemble de données pour lesquelles les vraies valeurs sont connues.

Les quatre cas sont illustrés dans la matrice de confusion ci-dessous :

Tableau 7 - Matrice de Confusion

		Valeur réelle		
		Attaque	**Normal**	**Total**
Valeur prédite	**Attaque**	VP	FP	VP + FP
	Normal	FN	VN	FN + VN
	Total	VP + FN	FP + VN	

Un cas d'utilisation pour cette matrice est quand l'ensemble de données est déséquilibré, comme dans notre cas (Goseva-Popstojanova, Anastasovski, & Pantev, 2012). Par exemple, dans le

contexte de la détection d'intrusion, il y aura beaucoup plus de donnés normales, sans attaque, que de données anormales, avec attaque. Sans elle, Cosmos pourrait utiliser la stratégie suivante : uniquement prédire la classe majoritaire, c'est-à-dire sans attaque, et sa précision serait élevée, car il aura raison la grande majorité du temps. Cependant, le système sera inutile.

À partir des valeurs de la matrice de confusion, nous pouvons calculer diverses métriques du modèle, comme :

Précision : C'est la fraction d'instances de données prédites comme positives qui sont réellement positives :

$$Précision = \frac{VP}{VP + FP}$$

L'exactitude : Mesure la justesse d'un système de détection d'intrusion dans la détection d'une activité normale ou anormale. Elle est décrite comme le pourcentage de toutes ces instances correctement prédites par rapport à toutes les instances :

$$Exactitude = \frac{VP + VN}{VP + FP + VN + FN}$$

Rappel : Parfois appelé *taux de vrais positifs*, il est calculé comme le rapport entre le nombre d'attaques correctement prédites et le nombre total d'attaques. Si toutes les intrusions sont détectées alors le rappel est de 1 ce qui est extrêmement rare pour un système de détection d'intrusion. Le rappel peut être exprimé mathématiquement comme :

$$Rappel = \frac{VP}{VP + FN}$$

Cette métrique mesure la partie manquante de la Précision ; à savoir, le pourcentage des instances d'attaque réelles couvertes par le classificateur. Par conséquent, il est souhaitable que Cosmos ait une valeur de rappel élevée.

Taux de Faux Positifs : Il est calculé comme le rapport entre le nombre d'instances normales classées à tort comme une attaque et le nombre total d'instances normales :

$$TFP = \frac{FP}{FP + VN}$$

Valeur-F : Est une valeur qui combine précision et rappel en une seule mesure. Elle est calculée comme moyenne harmonique de précision et de rappel.

$$Valeur\ F = \frac{2}{\frac{1}{Précision} + \frac{1}{Rappel}}$$

La Valeur-F est préférable lorsqu'une seule métrique de précision est souhaitée comme critère d'évaluation

Courbe des Caractéristiques de Fonctionnement du Récepteur (ROC) : il s'agit d'un graphique du taux de vrais positifs par rapport aux taux de faux positifs. Il représente les performances du classificateur binaire lorsque son seuil de discrimination est varié.

Aire Sous la Courbe ROC (AUC) : mesure de la capacité de discrimination du modèle, ce qui signifie la capacité d'un modèle à classer correctement un nouvel ensemble de données de test fourni.

5.3 Expériences

Le modèle a été entrainé exclusivement avec des données normales, c'est-à-dire, aucune séquence avec des données d'attaque. Une fois cette phase terminée, le modèle est utilisé par Cosmos qui est installé sur le système hôte. Le modèle est utilisé par le module de prédiction et lui permet de calculer la probabilité qu'une nouvelle séquence d'appels système, qui a été capturée par le module de surveillance des appels système, soit normale. Finalement, le module de seuil prend la décision et détermine si cette séquence est normale ou non.

Nos expériences ont été réalisées sur deux ensembles de données, et les résultats sont discutés ci-dessous :

- Utilisation de l'ensemble de données ADFA-LD ;
- Utilisation de l'ensemble de données généré à partir du serveur MySQL dans le conteneur Docker.

5.3.1 Expérience ADFA-LD

Afin de prouver qu'il est possible de détecter des anomalies dans l'hôte grâce aux appels système, notre première expérience est effectuée à l'aide de l'ensemble de données de ADFA-LD. Le modèle a été entrainé avec des séquences d'appels système normales, sans trace d'attaque. Puisqu'il a été entrainé ainsi, le modèle va calculer un score de probabilité élevé pour ces séquences. Cependant, pour les séquences d'appels système anormales, avec des traces d'attaque, le modèle va calculer un score de probabilité faible.

Le tableau suivant donne les résultats obtenus par les différents algorithmes d'apprentissage machine sur l'ensemble de données ADFA-LD :

Tableau 8 - Résultats de l'expérience ADFA-LD

	TFP	Précision	Rappel	Exactitude	Valeur-F	AUC
Notre Modèle	0.124	0.96	0.96	0.969	0.96	0.948
DTs	0.154	0.959	0.96	0.959	0.96	0.927
KMC (k=2)	0.858	0.749	0.722	0.706	0.735	0.423
Naïve Bayes	0.083	0.902	0.699	0.699	0.75	0.894
SVM	0.844	0.885	0.879	0.878	0.826	0.517

Nous proposons maintenant une comparaison visuelle pour les résultats de performance obtenue par les différents algorithmes :

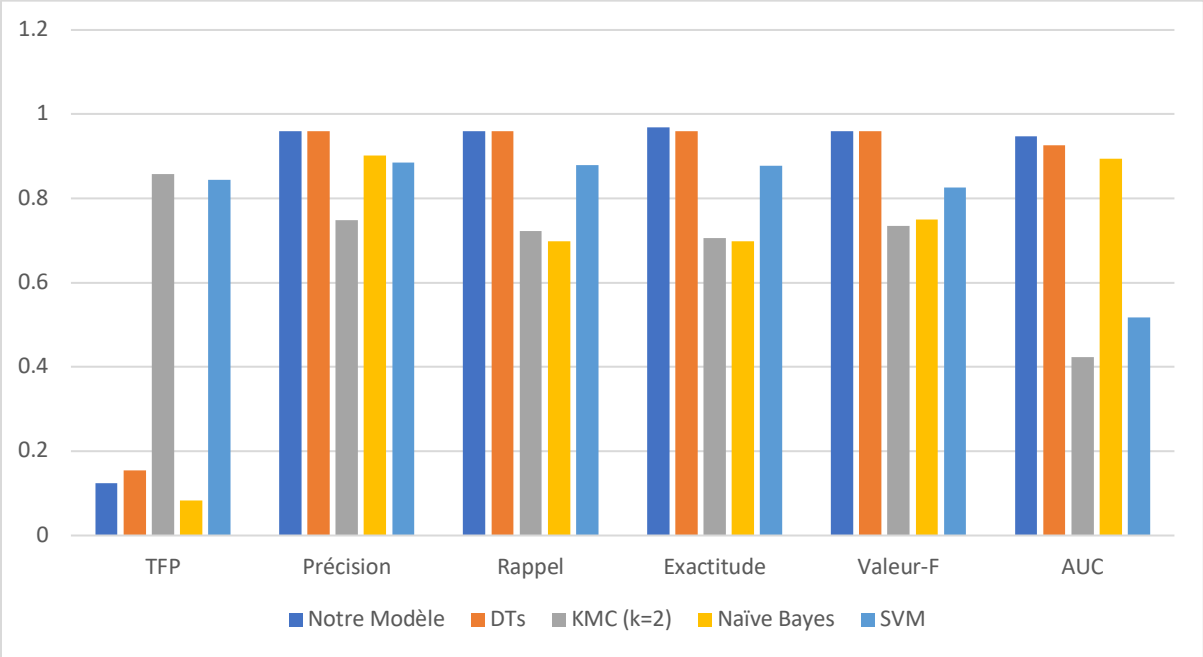

Figure 32 – Résultats de l'expérience ADFA-LD

On peut observer que tous les algorithmes fonctionnent relativement bien sur cet ensemble de données. Cependant, Cosmos et DTs ont obtenu les meilleurs résultats. En effet, d'après les résultats de cette première expérience, ils ont des performances similaires et performent mieux que les autres algorithmes sélectionnés.

Sur la base des classifications obtenues à partir de l'ensemble de données de ADFA-LD, la courbe ROC, illustrée à la ci-dessous, a été générée pour Cosmos :

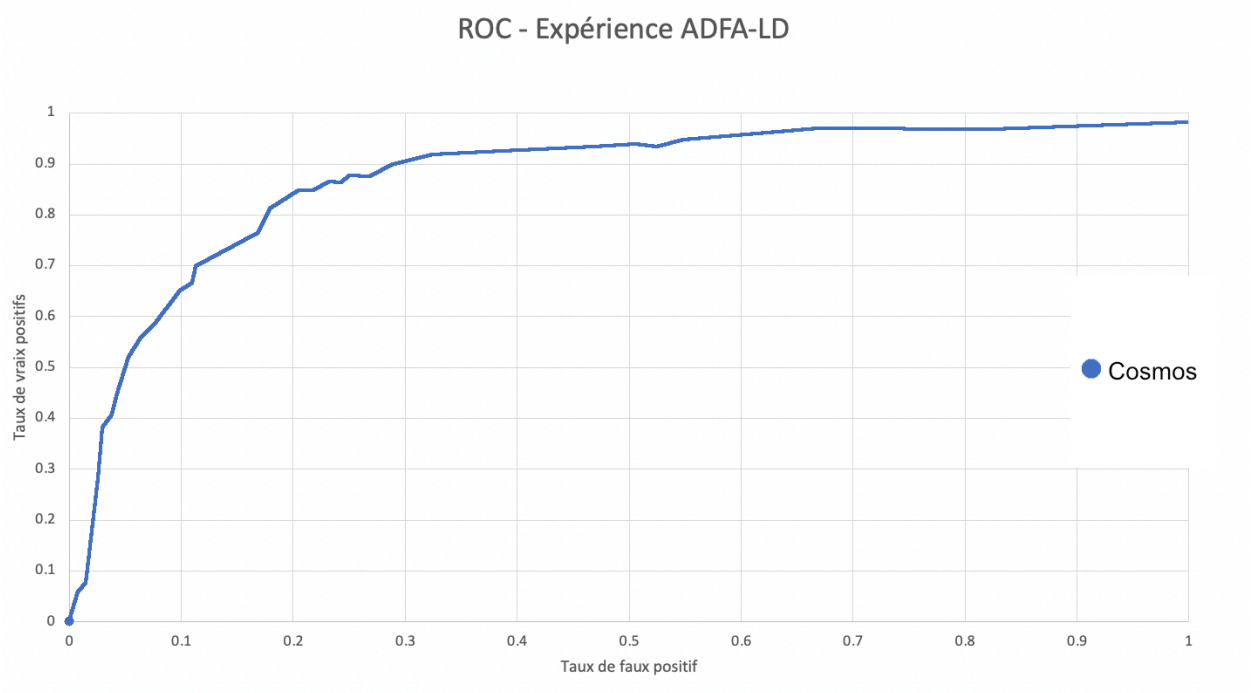

Figure 33 – ROC pour l'expérience ADFA-LD

L'aire sous la courbe (AUC) la plus élevée obtenue à partir de notre modèle est de 0.948. Le système classe donc la plupart des séquences d'appels système dans leurs classes respectives. C'est-à-dire qu'il est capable de classer la plupart des séquences d'appels normaux et anormaux dans l'ensemble de données.

Puisque le système est capable de classifier de façon exacte la plupart des séquences d'appels système de l'ensemble de données ADFA-LD, nous pouvons étendre nos tests sur l'ensemble de données généré à partir du serveur MySQL conteneurisé.

5.3.2 Expérience MySQL Conteneurisé

En ayant la confiance que Cosmos est capable de détecter des activités anormales dans le système, basé sur nos résultats obtenus sur l'ensemble de données ADFA-LD, nous avons effectué des tests sur des séquences d'appels système collectés à partir d'un conteneur exécutant un serveur de base de données MySQL. Même si cette fois l'application est exécutée dans un conteneur, Cosmos devrait encore être en mesure de détecter s'il y a des traces d'attaque dans une séquence d'appels système reçue.

Le modèle a été entrainé exclusivement avec des données normales, lors d'une utilisation normale du serveur de base de données conteneurisé sous différentes charges d'utilisation. Lors du test du modèle, nous utilisons de nouvelles séquences d'appels système générés et qui peuvent contenir des séquences contenant des données d'attaques. Étant donné que le modèle est généré en tenant compte de l'utilisation normale de la base de données, une séquence normale d'appels système renverrait une probabilité plus élevée que les appels système collectés lors d'une activité anormale.

5.3.3 Expérience #1

Pour notre première expérience avec une application conteneurisée, nous avons utilisé les paramètres suivants dans mysqlslap afin de simuler un achalandage faible :

- **Nombre de clients :** 100
- **Nombre de requêtes moyen par utilisateur :** 20

Les séquences d'appels système sont collectées en temps réel grâce à notre module de surveillance des appels système avant d'être entreposées dans une base de données externe. Parfois, une séquence anormale est produite, celle-ci a été simulée à l'aide de sqlmap.

Pour cette expérience, l'apprentissage du modèle est effectué à l'aide d'appels système collectés à partir du conteneur lors d'un évènement d'utilisation faible à modéré du système. Nous utilisons uniquement des appels système normaux pour entrainer le modèle.

Le tableau suivant donne les résultats obtenus par les différents algorithmes d'apprentissage machine sur l'ensemble de données généré pour l'expérience à charge d'application faible :

Tableau 9 - Résultats de l'expérience #1

	TFP	Précision	Rappel	Exactitude	Valeur-F	AUC
Notre Modèle	0.107	0.902	0.762	0.761	0.819	0.858
DTs	0.289	0.897	0.911	0.91	0.90	0.841
KMC (k=2)	0.862	0.755	0.868	0.828	0.80	0.481
Naïve Bayes	0.052	0.408	0.182	0.182	0.203	0.697
SVM	0.179	0.378	0.368	0.367	0.285	0.634

Une comparaison visuelle pour les résultats de performance obtenue par les différents algorithmes est illustrée dans l'illustration suivante :

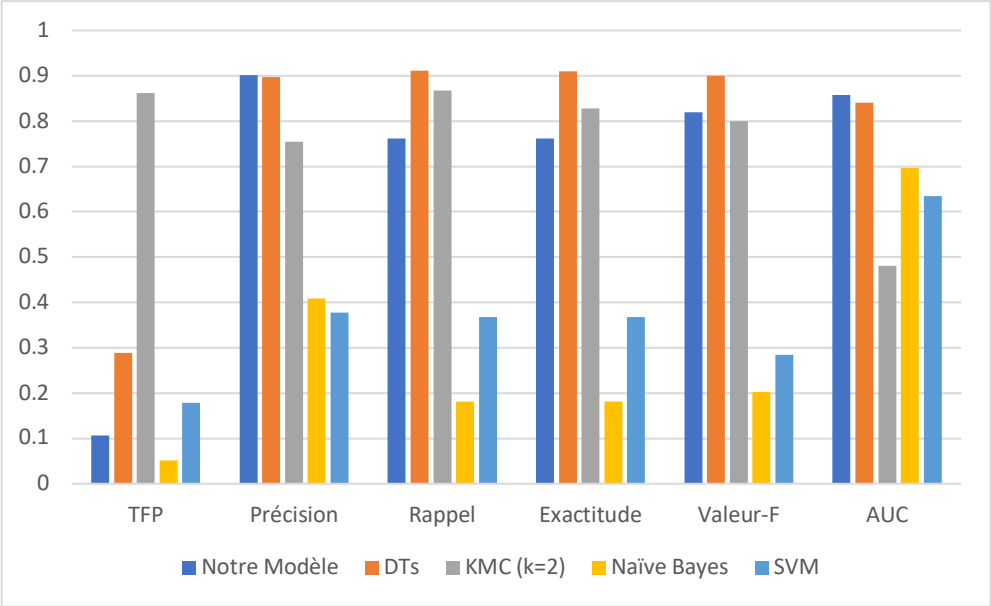

Figure 34 – Résultats de l'expérience #1

Pour cette expérience, on peut observer qu'il y a maintenant une plus grande disparité dans les performances des différents algorithmes. Cependant, on peut encore observer que Cosmos et DTs ont encore obtenu globalement les meilleurs résultats. DTs performe mieux que notre modèle quand il s'agit de la précision, du rappel, de l'exactitude et la valeur-F, mais produit un taux de faux positifs plus élevé.

À partir de la figure suivante, générée pour notre modèle, nous pouvons observer que la plupart des séquences d'appels système anormales ont correctement été classifiées :

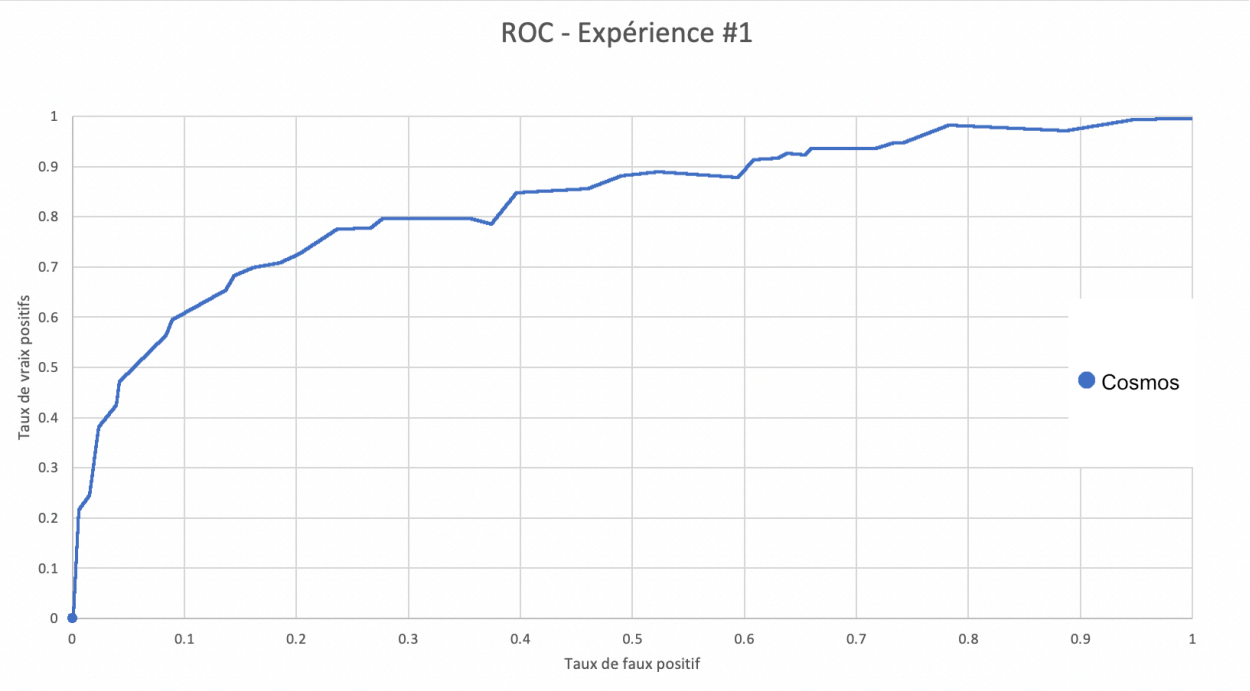

Figure 35 – ROC de l'expérience #1

L'aire sous la courbe la plus élevée atteinte est de 0.858. Le modèle classe donc encore la plupart des séquences d'appels système dans leurs classes respectives. Cependant, ces performances ont diminué par rapport à ceux obtenus dans l'expérience sur l'ensemble de données ADFA-LD.

Une raison possible pour cette perte de performance est peut-être l'occurrence de séquences d'appels système normales non présentes dans l'ensemble de données d'entrainement. Créez un bon ensemble de données et quelque chose de difficile, bien que les séquences d'appels système elles-mêmes puissent être faciles à acquérir, la collecte ou la génération d'une quantité suffisante de séquences significative pour l'évaluation des systèmes de détection d'intrusion est une tâche ardue.

5.3.4 Expérience #2

Pour cette expérience, nous avons utilisé des séquences d'appels système qui ont été collectées lors d'une utilisation modérée du serveur de base de données :

- **Nombre de clients :** 200
- **Nombre de requêtes moyen par utilisateur :** 40

Afin d'être constant avec l'expérience #1, des séquences d'appels système anormales ont été introduites, toujours à l'aide de sqlmap. En répétant la simulation plusieurs fois, plusieurs séquences d'appels système normaux sont collectées.

Le tableau suivant donne les résultats obtenus par les différents algorithmes d'apprentissage machine sur l'ensemble de données généré pour l'expérience à charge d'application modérée :

Tableau 10 - Résultats de l'expérience #2

	TFP	Précision	Rappel	Exactitude	Valeur-F	AUC
Notre Modèle	0.101	0.896	0.722	0.732	0.775	0.828
DTs	0.24	0.887	0.86	0.851	0.88	0.821
KMC (k=2)	0.173	0.126	0.234	0.233	0.156	0.53
Naïve Bayes	0.051	0.457	0.2	0.20	0.226	0.681
SVM	0.169	0.392	0.391	0.391	0.319	0.654

Les résultats obtenus pour cette expérience peuvent être observés dans l'illustration suivante :

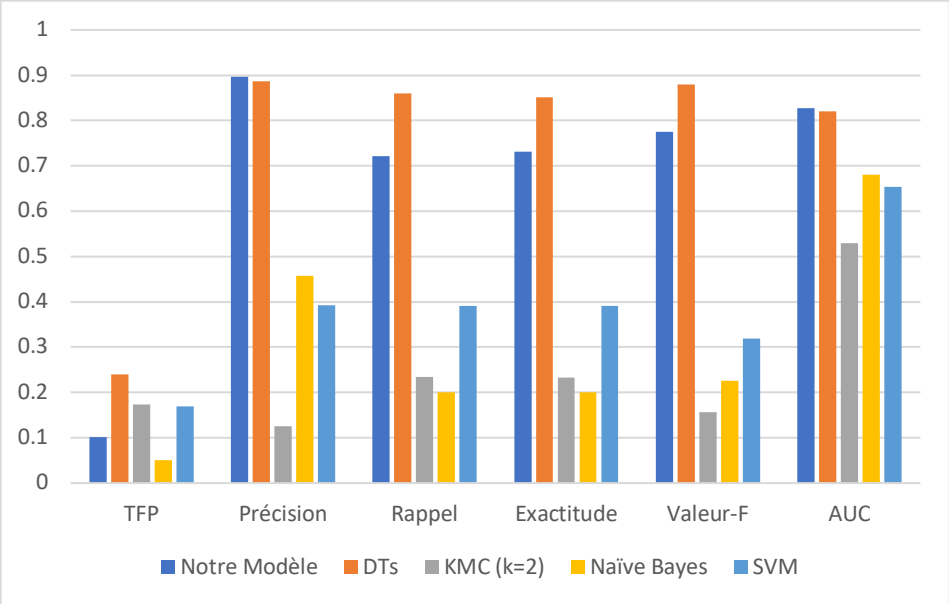

Figure 36 - Résultats de l'expérience #2

Cette deuxième expérience sur notre ensemble de données généré montre de nouveau que notre modèle, ainsi que DTs ont de meilleures performances que les autres algorithmes. Encore une fois, les métriques de performances sont globalement à la baisse. Cependant, cette fois-ci, notre modèle performe mieux que DTs pour la précision.

À partir de la courbe ROC suivante, nous pouvons remarquer une augmentation du nombre de faux positifs, c'est-à-dire de séquences normales classées à tort comme anormales :

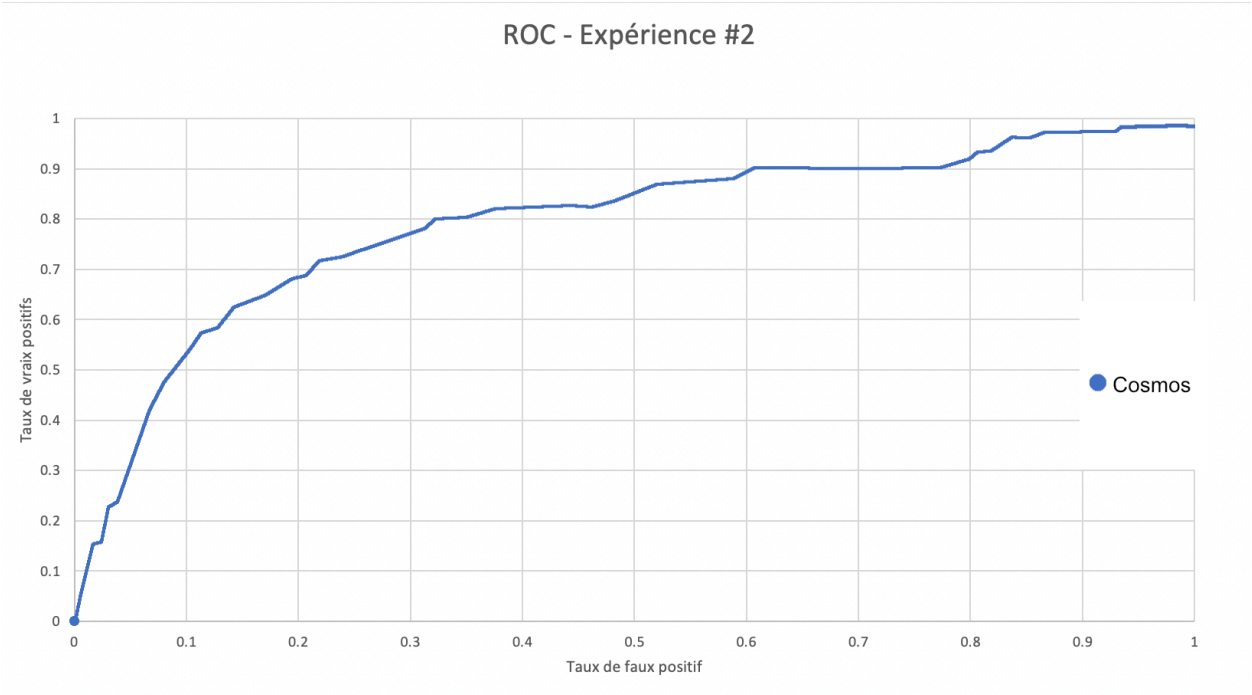

Figure 37 – ROC de l'expérience #2

Comme nous avons pu l'observer à partir de la courbe ROC, l'expérience a entrainé une baisse des performances par rapport à l'expérience précédente avec une faible utilisation du système. L'aire sous la courbe la plus élevée atteinte est 0.828, ce qui est inférieur par rapport aux expériences précédentes. La baisse des performances est peut-être due à la similitude croissante entre les séquences d'appel système qui contiennent des traces d'attaques avec celles qui n'en ont pas.

Encore une fois, nous pensons que la raison de la baisse des performances pourrait être l'occurrence d'appels système qui n'ont pas été rencontrés lors de la phase d'apprentissage et une plus grande similitude entre les séquences d'appels système avec et sans traces d'attaque à mesure que l'utilisation de l'application augmente.

5.3.5 Expérience #3

Pour notre troisième et dernière expérience, nous avons utilisé les paramètres suivants dans mysqlslap afin de simuler un achalandage élevé :

- **Nombre de clients :** 500
- **Nombre de requêtes moyen par utilisateur :** 60

Les appels système pour l'ensemble de tests sont collectés en simulant une utilisation élevée de l'application et en effectuant une utilisation normale et anormale de l'application. Tout comme les autres expériences, la génération de séquences anormales a été simulée à l'aide de l'outil de test de pénétration sqlmap.

L'apprentissage du modèle est effectué à l'aide d'appels système collectés à partir du conteneur lors d'un évènement d'utilisation élevée du système. Pour être consistent avec les expériences précédentes, nous utilisons des appels système sans trace d'attaque pour entrainer le modèle.

Le tableau suivant donne les résultats obtenus par les différents algorithmes d'apprentissage machine sur l'ensemble de données généré pour l'expérience à charge d'application élevée :

Tableau 11 - Résultats de l'expérience #3

	TFP	**Précision**	**Rappel**	**Exactitude**	**Valeur-F**	**AUC**
Notre Modèle	0.144	0.833	0.720	0.724	0.761	0.79
DTs	0.251	0.829	0.8	0.742	0.78	0.788
KMC (k=2)	0.24	0.119	0.258	0.257	0.138	0.509
Naïve Bayes	0.05	0.472	0.21	0.21	0.237	0.641
SVM	0.172	0.351	0.369	0.369	0.294	0.598

Les résultats obtenus pour cette dernière expérience peuvent être observés dans l'illustration suivante :

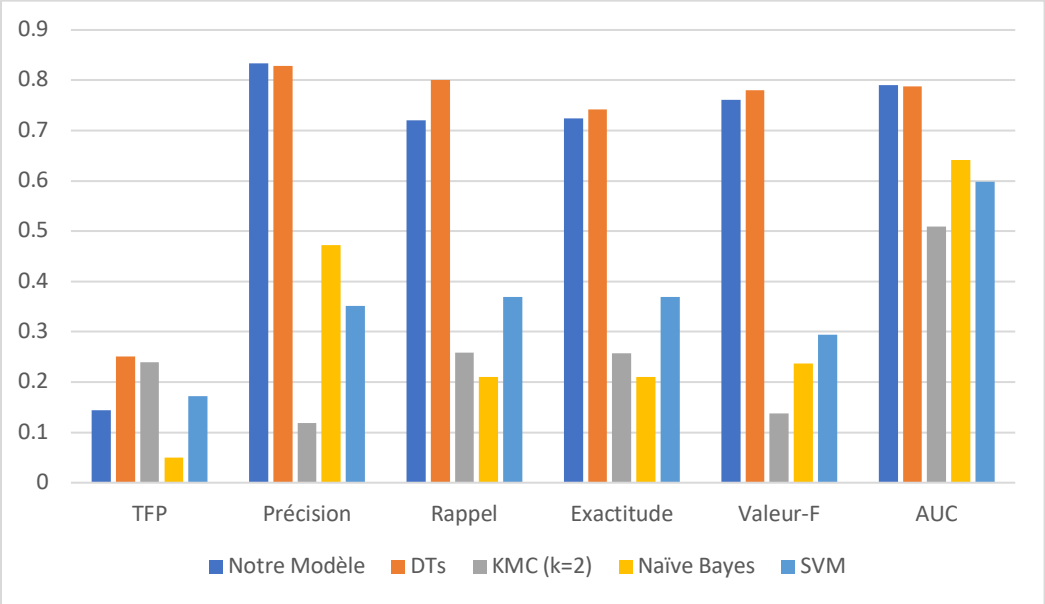

Figure 38 - Résultats de l'expérience #3

Les performances de notre modèle et de DTs sont supérieures aux autres algorithmes sélectionnés. Encore une fois ces deux algorithmes ont presque les mêmes scores. En effet, toutes les métriques sauf TFP montrent que DTs semble être un peu plus de performance que notre modèle. Cela n'est pas vraiment surprenant, car l'algorithme est l'un des meilleurs algorithmes d'apprentissage automatique pour examiner les données de manière catégorique et continue. Cependant, il faut noter qu'il consomme plus d'espace mémoire, et donc affecte négativement les performances du système hôte.

La courbe ROC pour l'expérience montre une augmentation du nombre de faux positifs, par rapport aux expériences précédentes :

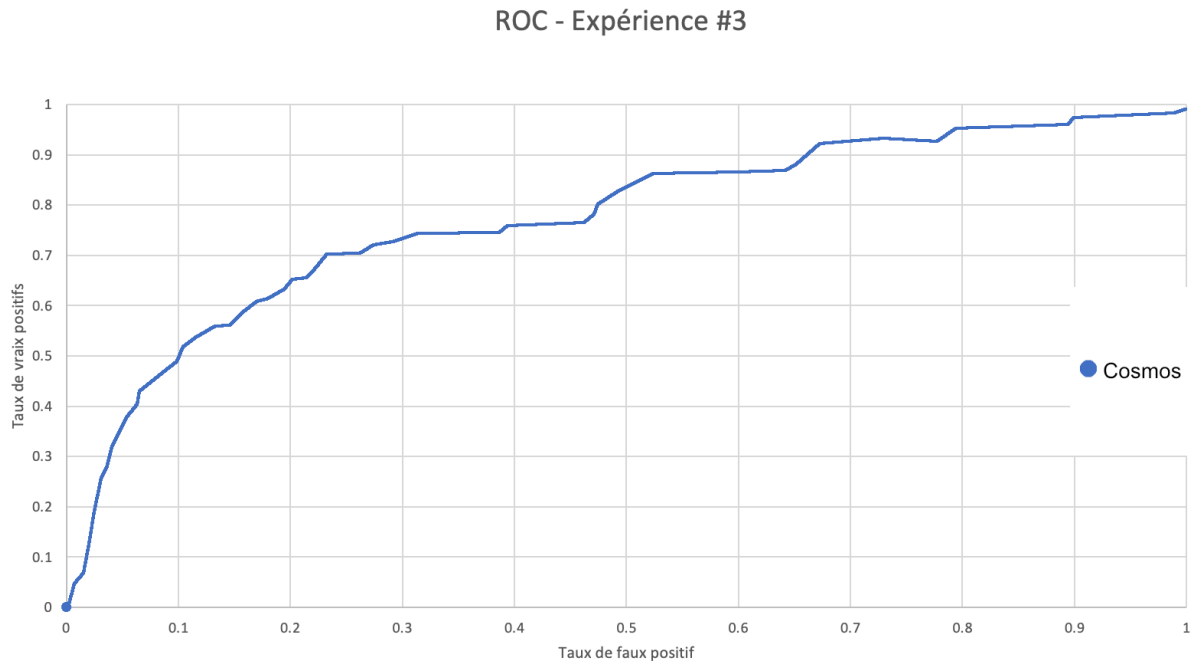

Figure 39 – ROC de l'expérience #3

L'aire sous la courbe la plus élevée pour cette dernière expérience était de 0.79. À partir du graphique, on peut observer l'augmentation du nombre de faux positifs, c'est-à-dire un plus grand nombre de séquences d'appels système marquées comme une activité anormale. Cela est probablement dû à l'augmentation du nombre d'appels système non rencontrés pendant la phase d'apprentissage. Cependant, une autre raison pourrait être un niveau de similitude élevé entre les appels système normaux et anormaux.

5.4 Analyse des Résultats

Durant les trois expériences, on observe que tous les algorithmes fonctionnent raisonnablement bien. Cependant, Cosmos et DTs ont obtenu les meilleurs résultats dans toutes les expériences. Nous pouvons également remarquer que l'augmentation du nombre de faux positifs, les séquences normales classées à tort comme anormales, est proportionnelle à la charge d'utilisation de la base de données augmente. Quand la charge d'utilisation augmente, le nombre de faux positifs aussi.

La courbe ROC ci-dessous est la combinaison des trois expériences faites sur la base de données conteneurisée :

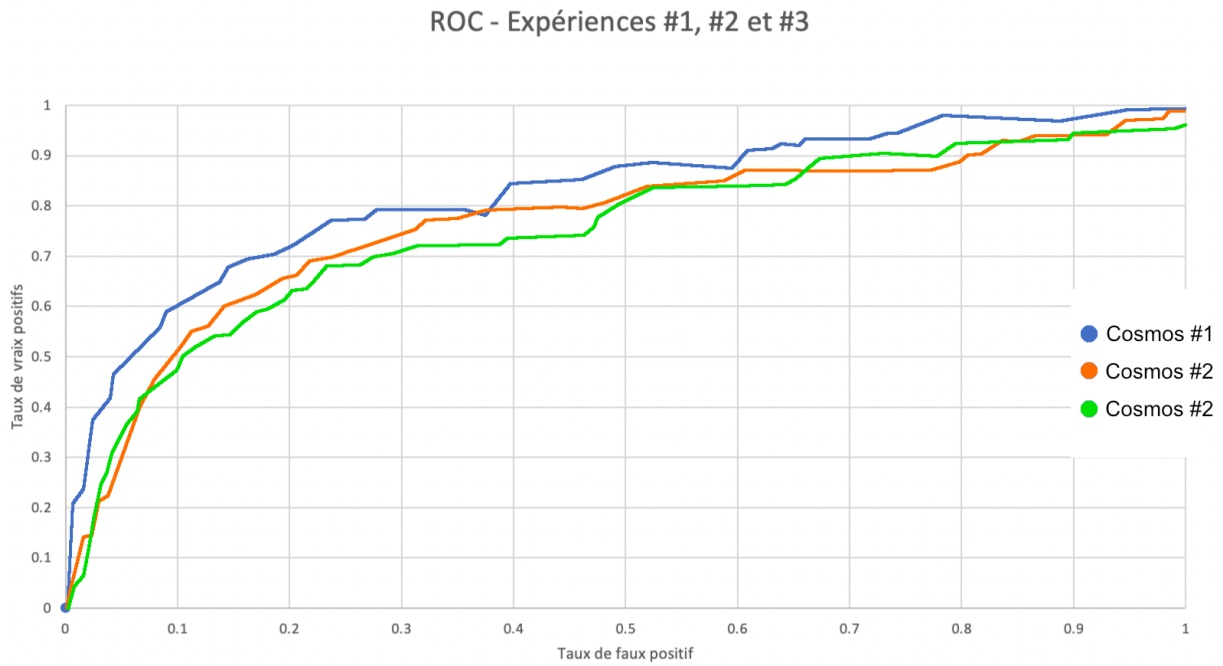

Figure 40 - ROC pour les expériences #1, #2 et #3

Les raisons probables pour cette diminution des performances quand la charge d'utilisation augmente pourraient être :

- Une fréquence plus élevée d'appels système non observés pendant la phase d'apprentissage et une plus grande similitude entre les séquences ;
- Une similitude entre les traces d'appels système des attaques lancés via sqlmap ;
- La taille du vecteur d'appels système donné en entrée est peut-être trop petite. Une taille plus grande permettrait d'avoir accès à davantage d'informations sur la séquence d'appels système.

D'après les résultats de nos trois expériences, on peut remarquer que Cosmos à de meilleures performances par rapport aux algorithmes de comparaison sélectionnée. De plus, dans le contexte des environnements conteneurisés, qui sont presque exclusivement hébergés dans l'infonuagique, la consommation de ressources doit être prise en considération. Si cette consommation n'est pas prise en considération lors du choix du modèle, l'entreprise devra débourser plus d'argent. Même si DTs a eu des performances similaires à Cosmos, cependant sa consommation plus de ressources, mémoire vive et temps de calcul, est beaucoup plus élevée et

engendra donc un coût supplémentaire. Pour ses deux raisons, les performances et la consommation de ressource, nous concluons que Cosmos est un système de détection d'intrusion idéal pour les applications conteneurisées.

Chapitre 6 - Conclusion et travaux futurs

L'objectif principal de ce travail était de développer un système de détection d'intrusion, pour les environnements conteneurisés, qui tirerait avantage de l'apprentissage machine. Cosmos a été en mesure de classer avec succès des séquences contenant des traces d'attaques, visibles dans leurs appels système, avec une AUC d'environ 0.859. Cependant, à mesure que le nombre de requêtes fait augmente, nous avons remarqué une augmentation constante de faux positifs.

Nous voulions aussi proposer un système qui permet d'auditer les activités des conteneurs sans modifier le contenu de ceux-ci ou encore de connaitre leur nature. Dans le chapitre 3, nous avons présenté la capture des appels système envoyée entre une application conteneurisée et son hôte grâce au Linux Audit Framework. Nous avons aussi montré comment il était possible d'utiliser l'arborescence des processus pour savoir si un processus est conteneurisé ou non.

Nous avons conçu un système adaptatif et faiblement couplé qui peut être facilement connecté à d'autres applications. Le format des données en sortie de Cosmos est en JSON, un format de fichier standard utilisé pour échanger des données. Il est également possible de connecter une base de données externe directement à Cosmos afin que les données ne soient pas entreposées directement dans un fichier sur l'hôte.

En ce qui concerne l'apprentissage machine, nous avons été en mesure d'entrainer un modèle qui performe mieux dans la détection d'anomalie que les systèmes traditionnels. En effet, celui-ci a réussi à classer les séquences d'appels système contenant des traces d'attaques et celles qui n'en contenaient pas en atteignant une aire sous la courbe d'environ 0.858 durant l'expérience #1. De plus, durant toutes nos expériences, notre modèle a eu de meilleures performances que les autres algorithmes sélectionnés.

Cosmos devait aussi être portable et ne devait pas affecter négativement les performances du système hôte. Dans le chapitre 4, nous avons présenté l'implémentation de nos modules. Utiliser Go, nous a permis de compiler notre application en un fichier binaire unique qui contient toutes ces dépendances. De plus, notre modèle, contrairement à J48, ne consomme pas beaucoup d'espace mémoire, et donc n'affecte presque pas les performances du système hôte.

Finalement, en ce qui concerne les travaux futurs, nous aimerions généraliser Cosmos afin qu'il puisse être utilisé dans d'autres environnements, par exemple modifier le module de surveillance des appels système afin qu'il soit compatible avec Windows. Cosmos peut aussi être étendue pour utiliser différents types de données pour faire ses prédictions. Par exemple il pourrait utiliser la fréquence des appels système. Pendant nos expériences, auditer une application pouvait consommer jusqu'à 40% de ressources supplémentaires. Trouver un moyen de réduire cette empreinte rendrait la solution plus viable pour une utilisation réelle. De plus, notre modèle doit refaire la phase d'entrainement si une nouvelle version de l'application à auditer est disponible, il serait intéressant de permettre au modèle l'apprentissage en continu des applications qui doivent être auditées.

Bibliographie

A Nichols, J., Chan, H. W., & Baker, M. (2019, 02). Machine learning: applications of artificial intelligence to imaging and diagnosis. Retrieved 09 16, 2021, from https://www.ibm.com/cloud/learn/supervised-learning

Aalam, Z., Kumar, V., & Gour, S. (2021). A review paper on hypervisor and virtual machine security.

Adams, K., & Agesen, O. (2007). A Comparison of Software and Hardware Techniques for x86.

Almseidin, M., Alzubi, M., Kovacs, S., & Alkasassbeh, M. (2017, 10 16). Evaluation of machine learning algorithms for intrusion detection system. IEEE.

Anderson, J. P. (1980, 02). Computer Security Threat Monitoring and Surveilance.

Benkhelifa, E., Welsh, T., & Hamouda, W. (2018, 05). A Critical Review of Practices and Challenges in Intrusion Detection Systems for IoT: Toward Universal and Resilient Systems.

Bernstein, D. (2014). Containers and Cloud: From LXC to Docker to Kubernetes. In *IEEE Cloud Computing* (pp. 81-84).

Bhatia, G., & Choudhary, A. (2017, 08). The Road To Docker: A Survey.

Combe, T., Martin, A., & Di Pietro, R. (2016). To Docker or Not to Docker: A Security Perspective. *IEEE Cloud Computing, 3*(5), 54 - 62.

Creech, G., & H, J. (2013, 06 15). Generation of a new IDS test dataset: Time to retire the KDD collection.

Denning, D. (1987, 02). An Intrusion-Detection Model. *Transactions on Software Engineering*.

Dilhara, M., Ketkar, A., & Dig, D. (2021). Understanding Software-2.0: A Study of Machine Learning.

Dolev, S., & Lodha, S. (2017). Cyber Security Cryptography and Machine Learning.

Dreger, H., Feldmann, A., Paxson, V., & Sommer, R. (2008). Predicting the Resource Consumption of Network Intrusion Detection Systems.

Eder, M. (2016, 07). Hypervisor- vs. Container-based Virtualization.

Emma Wang, Y., Wei, G.-Y., & Brooks, D. (2019, 10 22). Benchmarking TPU, GPU, and CPU Platforms for Deep Learning.

Engheim, E. (2021, 12 24). *Go Does Not Need a Java Style GC*. Retrieved from https://itnext.io/go-does-not-need-a-java-style-gc-ac99b8d26c60

Gao, J., & Lin, C. (2021, 01). Introduction to the Special Issue on Statistical Language Modeling .

Goseva-Popstojanova, K., Anastasovski, G., & Pantev, R. (2012, 11). Using Multiclass Machine Learning Methods to Classify Malicious Behaviors Aimed at Web Systems. IEEE.

Goyal, P., Dollàr, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., . . . He, K. (2017, 05 08). Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour.

Harini, C., & Fancy, C. (2020). A Study on the Prevention Mechanisms for Kernel Attacks. In *Artificial Intelligence Techniques for Advanced Computing Applications* (pp. 11-17). Springer.

He, J., Dai, T., & Gu, X. (2019, 06). A Study on Container Vulnerability Exploit Detection.

Hua, Z., Yu, Y., Gu, J., Xia, Y., Chen, H., & Zang, B. (2021, 08). TZ-Container: protecting container from untrusted OS with ARM TrustZone.

Islam Shamim, S. (2021, 08). Mitigating security attacks in kubernetes manifests for security best practices violation.

Jiang, K., Wang, W., Wang, A., & Wu, H. (2020, 02). Network Intrusion Detection Combined Hybrid Sampling With Deep Hierarchical Network.

Jozefowicz, R., Vinyals, O., Schuster, M., Shazeer, M., & Wu, Y. (2016, 02 11). Modeling, Exploring the Limits of Language.

Jun Lee, T., Gottschlich, J., Tatbul, N., Metcalf, E., & Zdonik, S. (2018). System, Greenhouse: A Zero-Positive Machine Learning.

Kadar, M., Tverdyshev, S., & G, F. (2012). System Calls Instrumentation for Intrusion Detection in Embedded Mixed-Criticality Systems.

Karn, R. R., Kudva, P., Huang, H., Suneja, S., & Elfadel, I. M. (2020, 04). Cryptomining Detection in Container Clouds Using System Calls and Explainable Machine Learning.

Karpathy, A. (2017, 10 11). *Software 2.0*. Retrieved from https://karpathy.medium.com/software-2-0-a64152b37c35

Kerrisk, M. (2021, 10 12). *Namespaces in operation, part 1: namespaces overview*. Retrieved 09 21, 2021, from https://lwn.net/Articles/531114/

Kharraz, A., Arshad, S., & Mulliner, C. (2016). UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware. In *USENIX Security Symposium* (pp. 757-772).

Khraisat, A., & Alazab, A. (2021, 03). A critical review of intrusion detection systems in the internet of things: techniques, deployment strategy, validation strategy, attacks, public datasets and challenges.

Kovács, Á. (2017, 07). Comparison of different Linux containers.

Kumar Sharma, S. (2015, 11). Analysis of KDD Dataset Attributes - Class wise for Intrusion Detection.

Kwon, S., & Lee, J.-H. (2020, 02). DIVDS: Docker Image Vulnerability.

Laureano, M., Maziero, C., & Jamhour, E. (2004, 07). Intrusion detection in virtual machine environments.

Lazarevic, A., Kumar, V., & Srivastava, J. (2005, 01). ntrusion Detection: A Survey.

Li, Y., & Wu, H. (03). A Clustering Method Based on K-Means Algorithm. *2012*.

Lin, X., Lei, L., Wang, Y., Jing, J., Sun, K., & Zhou, Q. (2018, 12). A Measurement Study on Linux Container Security: Attacks and Countermeasures.

Ma, S., Zhai, J., Kwon, Y., Hyung Lee, K., Zhang, X., Ciocarlie, G., . . . Yegneswaran, V. (2018, 07 12). Kernel-Supported Cost-Effective Audit Logging for Causality Tracking.

Manning, C. R. (2008). *Introduction to Information Retrieval.* Cambridge: Cambridge University Press.

Martin, A., Raponi, S., Combe, T., & Pietro Di, R. (2018). Docker ecosystem – Vulnerability Analysis.

Merkel, D. (2014). Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal, 2014*(239), 2.

Morris, J. (2009, 11 12). *Linux Kernel Security Overview*. Retrieved from http://namei.org/presentations/linux-kernel-security-kca09.pdf

Munaiah, N., Meneely, A., Wilson, R., & Short, B. (n.d.). Are Intrusion Detection Studies Evaluated Consistently? A Systematic Literature Review. 26.

Ojagbule, O., Wimmer, H., & Haddad, R. (2018, 04 19). Vulnerability Analysis of Content Management Systems to SQL Injection Using SQLMAP.

Olah, C. (2015, 08 27). *Understanding LSTM Networks*. Retrieved 09 20, 2021, from http://colah.github.io/posts/2015-08-Understanding-LSTMs/

OWASP Top Ten. (2021, 11 15). Retrieved from https://owasp.org/www-project-top-ten/

P. Bennett, K., & Demiriz, A. (1999). Semi-Supervised Support Vector Machines .

Pike, R., Presotto, D., Thompson, K., Trickey, H., & Winterbottom, P. (1992). The Use of Name Spaces in Plan 9. Retrieved from http://doc.cat-v.org/plan_9/4th_edition/papers/names

Rajdeep, D., A Reddy, R., & Dharmesh, K. (2014, 04). Virtualization vs Containerization to Support PaaS.

Rege, M., & Blanch K. Mbah, R. (2018). Machine Learning for Cyber Defense and Attack.

Rodriguez, M. Z., Comin, C., Casanova, D., M. Bruno, O., R. Amancio, D., Costa, L., & A. Rodrigues, F. (2019, 01). Clustering algorithms: A comparative approach. Retrieved 09 16, 2021, from https://developer.ibm.com/tutorials/learn-clustering-algorithms-using-python-and-scikit-learn/

Sahoo, J., Mohapatra, S., & Lath, R. (2010, 04). Virtualization: A Survey On Concepts, Taxonomy And Associated Security Issues.

Saravanan, N., & Gayathri, V. (2018). Performance and Classification Evaluation of J48 Algorithm and Kendall's Based J48 Algorithm (KNJ48) .

Sarker, I. H. (2021, 04). Machine Learning: Algorithms, Real-World Applications and Research Directions. Retrieved 09 16, 2021, from https://www.ibm.com/cloud/learn/unsupervised-learning

Shu, R., Gu, X., & Enck, W. (2017, 04). A Study of Security Vulnerabilities on Docker Hub.

Sultan, S., Ahmad, I., & Dimitriou, T. (2019, 04). Container Security: Issues, Challenges, and the Road Ahead.

Sundermeyer, M., Schluter, R., & Ney, H. (2012, 09 09). LSTM Neural Networks for Language Modeling.

Threat matrix for Kubernetes. (2021, 10 22). Retrieved from https://www.microsoft.com/security/blog/2020/04/02/attack-matrix-kubernetes/

Tunde-Onadele, O., He, J., Dai, T., & Gu, X. (2019, 06). A Study on Container Vulnerability Exploit.

Wang, G., Chau, M., & Chen, H. (2017, May 23). Intelligence and Security Informatics: 12th Pacific Asia Workshop. Springer.

Weiwei, H., & Tan, Y. (2017, 02). Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN.

Wenhao, J., & Zheng, L. (2020, 09). Vulnerability Analysis and Security Research of Docker Container.

Xi, K., Tang, Y., & Hu, J. (2011). Correlation keystroke verification scheme for user access control in cloud computing environment. *The Computer Journal*, 1632–1644.

Xie, M., Hu, J., & Jill Slay, J. (2014). Evaluating Host-based Anomaly Detection Systems: Application of the One-class SVM Algorithm to ADFA-LD.

Xin, L., Lei, L., Wang, Y., Jing, J., Sun, K., & Zhou, Q. (2018). A Measurement Study on Linux Container Security: Attacks and Countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference* (pp. 418-429).

Yang, X., & Hui, Z. (2015, 05). Intrusion Detection Alarm Filtering Technology Based on Ant Colony Clustering Algorithm.

Yin, C., Zhu, Y., Fei, J., & He, X. (2017, 10). A Deep Learning Approach for Intrusion Detection Using Recurrent Neural Networks.

Zeng, L., Xiao, Y., & Chen, H. (2015, 06 8). Linux Auditing: Overhead and Adaptation. *2015 IEEE International Conference on Communications (ICC)*.

Zerouali, A., Mens, T., Robles, G., & Gonzalez-Barahona, J. (2018, 11). On The Relation Between Outdated Docker Containers, Severity Vulnerabilities and Bugs.