Royaume du Maroc
Université Abdelmalek Essaadi
Faculté des Sciences et Techniques
Al Hoceima

المملكة المغربية
جامعة عبد المالك السعدي
كلية العلوم والتقنيات
الحسيمة

RAPPORT

Projet de fin d'année

Master en Sciences et Techniques

Département de physique

Filière : Systèmes Embarqués et Robotique (SER)

Module : Robotique 1 - Intelligence Artificielle

Titre :

Maintenance Prédictive pour Véhicules à l'aide des Données du Bus CAN

Réalisé par :

- ✓ MOKHTARI Anass
- ✓ NABIL Nada
- ✓ ABARKANE Nisrine
- ✓ AOUIJIL Fatima

Dirigé par :

- ✓ Mr. Noamane NCIR

Année Universitaire : 2024-2025

Résumé

Le présent rapport explore l'utilisation du bus CAN (Controller Area Network) dans les véhicules modernes et propose une solution de maintenance prédictive basée sur l'intelligence artificielle pour détecter les anomalies et attaques. Après une introduction aux principes du bus CAN et à son contexte d'utilisation, le projet détaille l'acquisition et la simulation d'attaques (DoS, fuzzing, suspension) sur des données de trafic CAN. Les paramètres significatifs sont extraits, et un jeu de données étiqueté est préparé pour l'entraînement d'un modèle XGBoost. Les résultats montrent une exactitude de 1,00 sur l'ensemble de test (61 980 trames), avec une détection parfaite de 40 001 attaques DoS injectées. Malgré des performances prometteuses, le faible échantillonnage des classes minoritaires (fuzzing, suspension) suggère des optimisations futures, notamment l'augmentation des données et des tests en temps réel.

Abstract

This report investigates the application of the Controller Area Network (CAN) bus in modern vehicles and introduces an AI-based predictive maintenance approach to detect anomalies and cyberattacks. Following an overview of CAN bus fundamentals and project context, it details data acquisition and the simulation of attacks (Denial of Service, payload fuzzing, communication suspension) on CAN traffic data. Key parameters are extracted, and a labeled dataset is prepared to train an XGBoost model. Results indicate a 100% accuracy on the test set (61,980 frames), with perfect detection of 40,001 injected DoS attacks. Despite promising outcomes, the limited sampling of minor classes (fuzzing, suspension) highlights the need for future enhancements, including data augmentation and real-time testing.

Table des matières

Liste des figures

Liste des tableaux

Liste des abréviations

- CAN : Controller Area Network
- ECU : Electronic Control Unit
- DoS : Denial of Service
- FD : Flexible Data-rate
- XL : Extended Length
- LIN : Local Interconnect Network
- ADAS : Advanced Driver Assistance Systems
- VE : Vehicle Electric
- OBD2 : On-Board Diagnostics II
- UDS : Unified Diagnostic Services
- J1939 : Heavy Vehicle Protocol
- NMEA : National Marine Electronics Association
- ISO : International Organization for Standardization
- MCU : Microcontroller Unit
- CRC : Cyclic Redundancy Check
- ACK : Acknowledgment
- EOF : End of Frame
- RTR : Remote Transmission Request

Introduction générale

Le bus CAN (Controller Area Network) représente un pilier fondamental des systèmes de communication embarqués dans les véhicules modernes, permettant un échange de données en temps réel entre les unités de contrôle électronique (ECU). Développé par Bosch dans les années 1980, ce protocole a été conçu pour offrir une solution légère, efficace et robuste, adaptée aux environnements exigeants des automobiles où la fiabilité et la rapidité sont primordiales. Initialement pensé pour des systèmes fermés, le bus CAN orchestre des fonctions critiques telles que le contrôle moteur, les systèmes de freinage antiblocage (ABS) et les interfaces d'infodivertissement, en reliant des dizaines d'ECU via une architecture décentralisée sans hôte central. Cependant, avec l'émergence des véhicules connectés, intégrant des technologies comme le Bluetooth, le Wi-Fi et les réseaux cellulaires, cette infrastructure s'expose désormais à de nouvelles vulnérabilités. Les cyberattaques, telles que le déni de service (DoS), la falsification de la charge utile (payload fuzzing) ou la suspension de communications essentielles, constituent des menaces croissantes qui compromettent la sécurité et la performance des véhicules.

Dans ce contexte, notre projet s'inscrit dans une démarche proactive de cybersécurité automobile. Il vise à concevoir un système de maintenance prédictive basé sur les techniques d'intelligence artificielle, capable de détecter automatiquement les anomalies et intrusions sur le bus CAN. En exploitant les données de communication entre ECU, nous cherchons à identifier les comportements suspects avant qu'ils ne provoquent des dysfonctionnements critiques, répondant ainsi à une problématique centrale : comment sécuriser un protocole obsolète face aux exigences des technologies modernes ? Cette étude s'appuie sur une méthodologie rigoureuse incluant l'acquisition de données réelles, la simulation d'attaques, l'extraction de paramètres significatifs et l'entraînement de modèles d'apprentissage supervisé, tels que XGBoost. En posant les bases d'une détection prédictive, ce travail ambitionne de contribuer à la sécurisation des véhicules connectés, un enjeu stratégique pour l'industrie automobile face à l'essor de la mobilité intelligente.

Chapitre 1 : Généralité sur le bus CAN et Contexte générale du projet

1.1 Introduction

Le bus CAN (Controller Area Network) est un protocole de communication largement utilisé qui permet l'échange de données en temps réel entre les unités de contrôle électronique (ECU) dans les véhicules. Développé par Bosch dans les années 1980, il offre une méthode légère et efficace pour la communication entre microcontrôleurs sans hôte central, ce qui le rend idéal pour les systèmes automobiles où la fiabilité et la rapidité sont cruciales. Aujourd'hui, le bus CAN est essentiel au fonctionnement des véhicules modernes, gérant des fonctions allant du contrôle du moteur aux systèmes de sécurité et d'infodivertissement. Cependant, à mesure que les véhicules deviennent plus connectés et complexes, le bus CAN est de plus en plus exposé aux menaces de cybersécurité, faisant de sa sécurité une préoccupation croissante. Comprendre le fonctionnement du bus CAN est fondamental pour relever ces défis et constitue la base pour le développement de systèmes intelligents capables de détecter et de prévenir les activités malveillantes.

1.2 Généralité sur le bus CAN

1.2.1 Définition du bus CAN

Le bus CAN (Controller Area Network, ou réseau de contrôleurs) est un système de communication robuste et standardisé, principalement utilisé dans les véhicules et les machines industrielles pour permettre aux unités de contrôle électroniques (ECUs, Electronic Control Units) de communiquer entre elles sans nécessiter d'ordinateur central hôte. Ce protocole, développé par Bosch dans les années 1980, est conçu pour être fiable, efficace et résilient, même dans des environnements électriquement bruyants, comme ceux rencontrés dans les automobiles ou les systèmes industriels.

✓ ***Analogie avec le corps humain***

Pour mieux comprendre, imaginons que votre voiture est comparable au corps humain :

- **Le bus CAN** représente le système nerveux, qui transporte les signaux d'un point à un autre rapidement et efficacement.
- **Les ECUs** (ou nœuds CAN) sont comme les organes ou les parties du corps (par exemple, les freins, le moteur, la transmission, ou encore le tableau de bord). Chaque ECU est un module électronique spécialisé qui contrôle une fonction spécifique, comme la gestion du moteur ou le système de freinage antiblocage (ABS).
- Grâce au bus CAN, ces "organes" peuvent partager des informations en temps réel. Par exemple, lorsque vous appuyez sur la pédale de frein, le capteur de freinage envoie un message via le bus CAN pour informer le moteur de réduire sa puissance, assurant une coordination fluide.

✓ **Fonctionnement technique du bus CAN**

Le bus CAN repose sur un système de communication en série qui utilise un **bus à deux fils** constitué d'une paire torsadée :

- **CAN High (CAN-H)** : souvent représenté par un fil jaune, il transporte le signal principal avec une tension plus élevée.
- **CAN Low (CAN-L)** : généralement un fil vert, il transporte un signal complémentaire avec une tension plus basse.

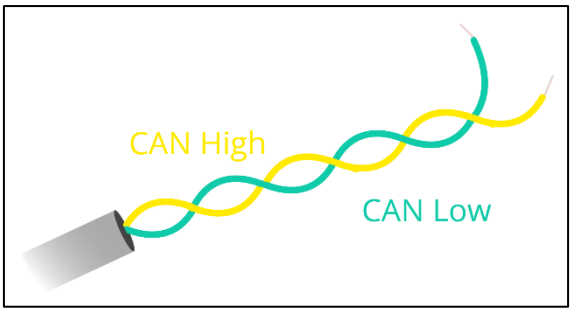

Figure 1.1 - Schéma des fils CAN High et CAN Low

Ces deux fils fonctionnent en mode différentiel, ce qui signifie que la différence de tension entre CAN-H et CAN-L encode les données. Ce système différentiel rend le bus CAN particulièrement résistant aux interférences électromagnétiques, ce qui est crucial dans des environnements comme les véhicules, où de nombreux composants électriques fonctionnent simultanément. Chaque ECU connecté au bus

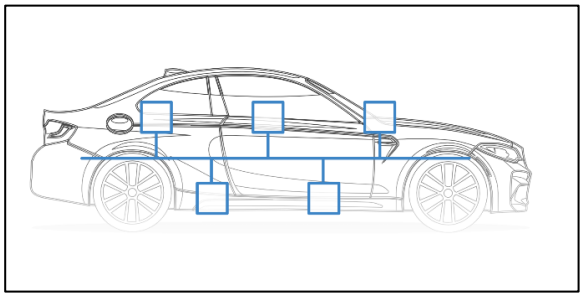

Figure 1.2 - Disposition des unités de contrôle électronique (ECUs) dans une voiture

CAN peut envoyer et recevoir des messages. Ces messages sont diffusés à tous les nœuds du réseau, mais seuls les ECUs concernés par un message particulier (identifié par un identifiant unique) le traitent. Ce mécanisme permet une communication rapide et efficace, sans engorger le réseau.

1.2.1.1 Les unités de contrôle électronique

Les unités de contrôle électronique (ECUs) sont des composants qui régissent des fonctionnalités spécifiques d'un véhicule, comme l'unité de contrôle du moteur, la transmission, les freins, la direction, la gestion des températures, et bien plus encore. Dans une voiture moderne, il est courant d'en trouver plus de 70, chacune étant connectée au réseau et partageant des informations avec les autres ECUs via le bus CAN. Cette interconnexion permet une coordination efficace entre les différents systèmes du véhicule.

Tout ECU connecté au bus CAN peut préparer et diffuser des informations, telles que des données provenant de capteurs (par exemple, la vitesse, la pression des freins ou la température du moteur). Ces données diffusées sont reçues par toutes les autres ECUs du réseau. Chaque unité analyse ensuite ces informations et décide si elle doit les accepter pour les utiliser dans ses propres processus ou les ignorer si elles ne sont pas pertinentes pour sa fonction. Ce mécanisme de diffusion et de filtrage garantit une communication fluide et optimisée, essentielle au fonctionnement harmonieux d'un véhicule moderne.

Si nous zoomons, une unité de contrôle électronique (ECU) se compose de trois éléments principaux :

- **Microcontrôleur (MCU) :** Le MCU est le cerveau de l'ECU. Il interprète les messages CAN entrants et décide quels messages transmettre. Par exemple, un capteur peut être programmé pour mesurer et diffuser la température de l'huile à une fréquence de 5 Hz, permettant une surveillance continue et précise.
- **Contrôleur CAN :** Généralement intégré au microcontrôleur, le contrôleur CAN veille à ce que toute la

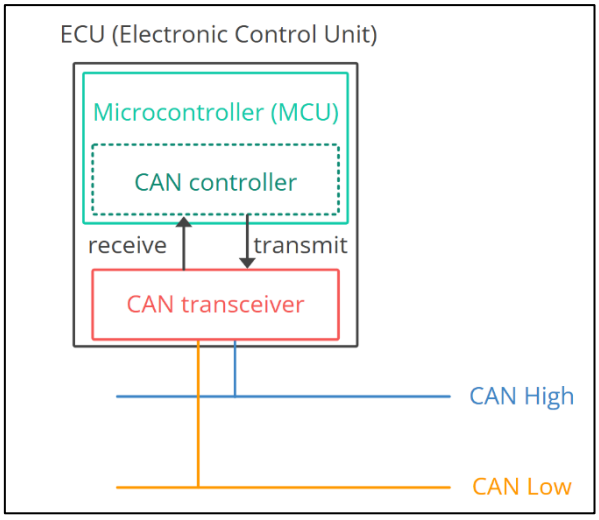

Figure 1.3 - Architecture interne d'une unité de contrôle électronique (ECU)

communication respecte le protocole CAN (encodage des messages, détection des erreurs, arbitrage, etc.). Cela réduit la complexité des tâches confiées au MCU, en le déchargeant de ces responsabilités techniques.

- **Transcepteur CAN :** Le transcepteur CAN relie le contrôleur CAN aux fils physiques du bus CAN. Il convertit les données du contrôleur en signaux différentiels adaptés au système du bus CAN, et inversement. Il offre également une protection électrique, protégeant les composants contre les surtensions ou les interférences.

1.2.1.2 Connecteur DB9 du bus CAN

La connexion à un bus CAN ne repose sur aucun connecteur standardisé à travers toutes les applications de bus CAN. Comme nous l'expliquerons plus loin, cela signifie que différents véhicules ou machines peuvent utiliser des connecteurs variés, rendant la compatibilité parfois complexe selon les systèmes.

Cependant, un connecteur qui se rapproche d'une norme de facto est le connecteur CAN DB9 (ou D-sub 9), défini par la spécification CANopen CiA 303-1. Ce connecteur, doté de neuf broches, est largement adopté dans de nombreuses applications, notamment pour les enregistreurs de données CAN et les interfaces de bus CAN. Sa popularité s'explique par sa simplicité d'utilisation et sa capacité à supporter des configurations standardisées, bien qu'il ne soit pas universellement obligatoire. Les broches du DB9 sont généralement câblées de manière spécifique : la broche 2 pour CAN Low, la broche 7 pour CAN High, et la broche 9 pour l'alimentation ou la masse, selon les besoins du système. Cette flexibilité, combinée à son adoption dans l'industrie, en fait un choix privilégié pour les ingénieurs travaillant sur des réseaux CAN, bien que des adaptateurs ou des connaissances spécifiques puissent être nécessaires pour d'autres configurations propriétaires.

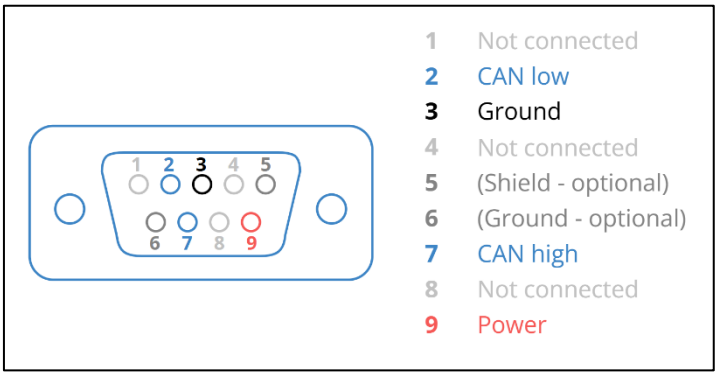

Figure 1.4 - Configuration des broches du connecteur DB9 du bus CAN

1.2.1.3 Variantes du bus CAN

Avant de poursuivre, il est utile de savoir que plusieurs variantes du bus CAN existent, chacune répondant à des besoins spécifiques :

- **CAN à faible vitesse (Low-speed CAN)** : Également appelée CAN à tolérance de panne, cette variante est une option économique particulièrement adaptée lorsque la fiabilité en cas de défaillance est essentielle, comme dans certains systèmes critiques. Cependant, elle est de plus en plus remplacée par le bus LIN, qui offre une alternative légère et efficace.
- **CAN à haute vitesse (High-speed CAN)** : Connu sous le nom de CAN classique, c'est la variante la plus couramment utilisée aujourd'hui dans l'automobile et les machines, et elle constitue le principal sujet de cet article. Elle est largement adoptée grâce à sa robustesse et sa performance dans des environnements exigeants.
- **CAN FD** : Cette version permet des charges utiles plus longues et des vitesses plus élevées, offrant ainsi une évolution par rapport au CAN classique. Cependant, son adoption reste limitée, bien qu'elle gagne en intérêt. Pour en savoir plus, consultez notre introduction au CAN FD.
- **CAN XL** : Cette variante propose des charges utiles encore plus longues et des vitesses accrues, visant à combler l'écart entre le bus CAN traditionnel et l'Ethernet automobile (100BASE-T1). Elle représente une étape vers des réseaux plus performants pour les véhicules modernes.

Propriété	CAN tolérant aux pannes (CAN à faible vitesse)	CAN classique 2.0 (CAN à haute vitesse)	CAN FD (Flexible Data-rate)	CAN XL
Vitesse maximale du débit (Baud rate)	0,125 Mbit/s	1 Mbit/s	8 Mbit/s (phase de données)	20 Mbit/s
Taille maximale de la charge utile	8 octets	8 octets	64 octets	2048 octets
Type de débit	Fixe	Fixe	Variable (champ de données plus rapide)	Variable (taux plus élevés)
Cas d'utilisation	Tolérant aux pannes, modules de contrôle du corps	Automatique en temps réel	Haut débit, throughput, ADAS, applications VE	Applications futures à haut débit
Caractéristiques clés	Opération tolérante aux pannes, faible coût, détection d'erreurs robuste, continue même si une ligne est endommagée	Faible coût, détection d'erreurs, largement déployé	Charge utile augmentée, vitesse et fiabilité	Charge utile augmentée, vitesse et fiabilité

Tableau 1.1 - Comparaison des versions du bus CAN

1.2.1.4 Réseaux alternatifs au bus CAN dans l'automobile

Dans le domaine automobile, on rencontre souvent d'autres réseaux non-CAN. Voici les plus pertinents :

- **Bus LIN** : Le bus LIN est un complément à faible coût des réseaux CAN, nécessitant moins de câblage et des nœuds moins chers. Un cluster LIN comprend généralement un maître LIN agissant comme passerelle et jusqu'à 16 nœuds esclaves. Ses cas d'utilisation typiques incluent des fonctions non critiques du véhicule, comme la climatisation, les fonctionnalités des portes, etc. Pour plus de détails, consultez notre introduction au bus LIN ou notre article sur l'enregistreur de données LIN.
- **FlexRay** : FlexRay offre une vitesse supérieure à celle du CAN (jusqu'à 10 Mbit/s), une tolérance aux pannes grâce à sa redondance à double canal et une grande flexibilité, mais il est également plus coûteux. Il est standardisé selon les normes ISO 17458-1 et ISO 17458-5.
- **Ethernet automobile** : Cette technologie est de plus en plus déployée dans le secteur automobile pour répondre aux besoins élevés en bande passante des systèmes avancés d'assistance à la conduite (ADAS), des systèmes d'infodivertissement, des caméras, etc. L'Ethernet automobile propose des taux de transfert de données bien plus élevés que le bus CAN, mais il manque de certaines fonctionnalités de sécurité et de performance propres au CAN. Il est probable que, dans les années à venir, l'Ethernet automobile, le CAN FD et le CAN XL soient utilisés conjointement dans les nouveaux développements automobiles et industriels.

1.2.2 Les quatres avantages du bus CAN

La norme du bus CAN est utilisée dans pratiquement tous les véhicules et de nombreuses machines en raison des avantages clés suivants :

1. **Simplicité et faible coût :** Les unités de commande électronique (ECU) communiquent via un seul système CAN au lieu de lignes de signaux analogiques complexes et directes, ce qui réduit les erreurs, le poids, le câblage et les coûts :
2. **Accès facile** : Le bus CAN offre un "point d'entrée unique" pour communiquer avec toutes les ECU du réseau, permettant des diagnostics centralisés, l'enregistrement des données et la configuration.

3. **Extrêmement robuste** : Le système est résistant aux perturbations électriques et aux interférences électromagnétiques, ce qui le rend idéal pour les applications critiques en matière de sécurité (par exemple, les véhicules).

4. **Efficacité** : Les trames CAN sont priorisées par identifiant (ID), ce qui permet aux données de haute priorité d'accéder immédiatement au bus sans interrompre les autres trames ni provoquer d'erreurs CAN.

1.2.3 Histoire du bus CAN

Avant le CAN, dans les premiers véhicules, les **ECU (unités de commande électronique)** communiquaient via un **réseau de câblage dédié**, avec des connexions séparées pour chaque signal. Ce système était **lourd, coûteux et difficile à maintenir**, car chaque nouveau capteur ou actionneur nécessitait des fils supplémentaires.

1. **1986 : Bosch invente le protocole CAN**
    - Face à la complexité croissante des systèmes embarqués, **Bosch** développe le **Controller Area Network (CAN)** pour simplifier la communication entre les ECU. Le CAN permet une transmission **fiable, rapide et économique** des données sur un seul bus partagé.
2. **1991 : Publication de CAN 2.0 (2.0A et 2.0B)**
    - **CAN 2.0A** : Utilise des **identifiants de 11 bits** (2 048 messages possibles).
    - **CAN 2.0B** : Étendu aux **identifiants de 29 bits** (plus de 500 millions de messages possibles), essentiel pour les réseaux complexes.
3. **1993 : Standardisation internationale (ISO 11898)**
    - Le CAN devient une **norme ISO**, ce qui accélère son adoption dans l'industrie automobile et au-delà.
4. **2003 : La norme ISO 11898 devient une série complète**
    - Elle définit non seulement le protocole, mais aussi les **couches physiques et les exigences de compatibilité**.
5. **2012 : Bosch annonce CAN FD (Flexible Data Rate)**
    - Le **CAN FD** améliore le débit (jusqu'à **5-8 Mbit/s**) et la taille des trames (jusqu'à **64 octets** au lieu de 8), permettant une **communication plus rapide** pour les véhicules modernes.
6. **2015 : Standardisation du CAN FD (ISO 11898-1)**
    - Le protocole est officiellement intégré à la norme ISO, renforçant sa fiabilité.

7. **2016 : Couche physique haute vitesse (ISO 11898-2)**
    - Standardisation des débits jusqu'à **5 Mbit/s** (voire **8 Mbit/s** en pratique), essentielle pour les systèmes temps réel.
8. **2018 : Début du développement du CAN XL**
    - Le **CAN XL** est conçu pour des **débits encore plus élevés (jusqu'à 10+ Mbit/s)** tout en restant compatible avec les anciennes versions.
9. **2024 : Standardisation du CAN XL (ISO 11898-1:2024, 11898-2:2024)**
    - Cette mise à jour permet une **communication ultra-rapide** pour les véhicules autonomes, les batteries de véhicules électriques et les systèmes industriels avancés.

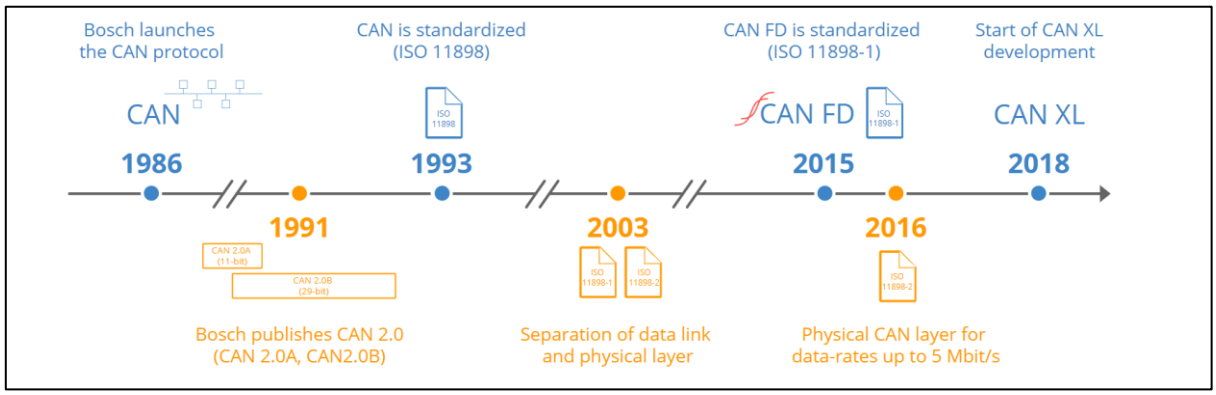

Figure 1.5 - Évolution du protocole CAN : des origines à CAN XL

1.2.4 Couche Physique et Couche Liaison de Données du CAN

Sur le plan technique, le **Controller Area Network (CAN)** est défini par deux couches principales :

- **La couche liaison de données** (ISO 11898-1)
- **La couche physique** (ISO 11898-2)

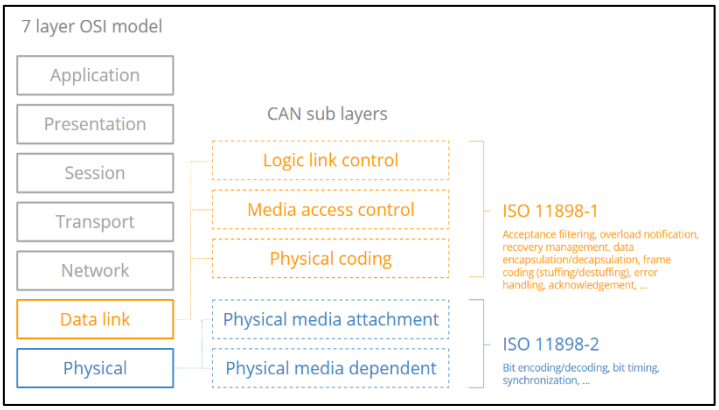

Figure 1.6 - Architecture du bus CAN dans le modèle OSI : Couches et sous-couches

Dans le modèle OSI à 7 couches, le CAN couvre ainsi les **deux couches les plus basses**, comme illustré dans la figure .

Couche physique (ISO 11898-2) :

La couche physique du bus CAN définit les types de câbles, les niveaux de signaux électriques, les exigences des nœuds, l'impédance du câble, etc. Par exemple, la couche physique spécifie :

1. **Débit binaire** : Les nœuds doivent être connectés via un bus à deux fils avec des débits allant jusqu'à 1 Mbit/s (CAN classique) ou 8 Mbit/s (CAN FD)
2. **Longueur de câble** : Les longueurs maximales de câble CAN doivent être comprises entre 500 mètres (125 kbit/s) et 40 mètres (1 Mbit/s)
3. **Terminaison** : Le bus CAN doit être terminé par une résistance de terminaison de 120 Ohm à chaque extrémité du bus

Couche liaison de données (ISO 11898-1) :

La couche liaison de données du bus CAN définit notamment les formats de trame CAN, la gestion des erreurs, la transmission des données et garantit l'intégrité des données. Par exemple, la couche liaison de données spécifie :

- **Formats de trame** : Quatre types (trames de données, trames de demande, trames d'erreur, trames de surcharge) avec identifiants de 11 ou 29 bits
- **Gestion des erreurs** : Méthodes de détection et traitement des erreurs CAN incluant CRC, bits d'acquittement, compteurs d'erreur et autres
- **Arbitrage** : L'arbitrage bit à bit non destructif permet de gérer l'accès au bus CAN et d'éviter les collisions via une priorité basée sur l'identifiant

1.2.5 La trame CAN

Selon la couche liaison de données, la communication sur le bus CAN s'effectue via des trames CAN.

Ci-dessous se trouve une trame de données CAN standard avec un identifiant de 11 bits (CAN 2.0A), le type utilisé dans la plupart des voitures. La trame avec identifiant étendu de 29 bits (CAN 2.0B) est identique, à l'exception de l'ID plus long. Elle est par exemple utilisée dans le protocole J1939 pour les véhicules lourds.

Notez que l'ID CAN et les données sont mis en évidence - ceux-ci sont importants lors de l'enregistrement des données du bus CAN, comme nous le verrons ci-dessous.

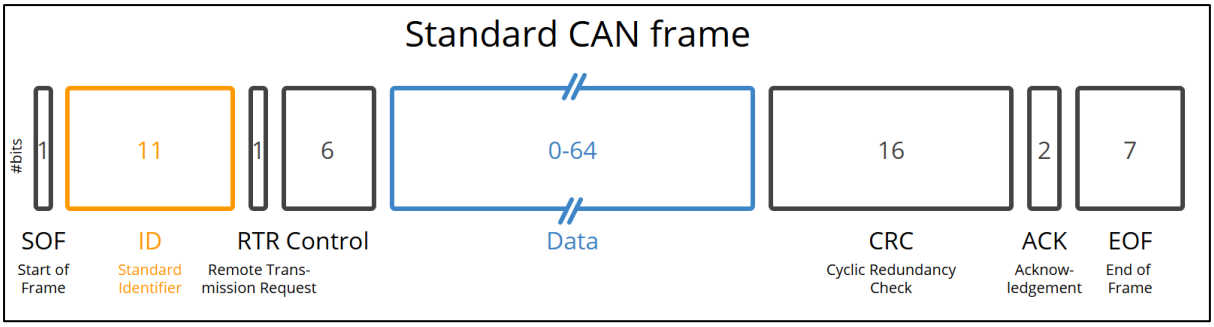

Figure 1.7 - Structure d'une trame CAN standard

Les 8 champs d'une trame de données CAN

- **SOF (Start of Frame)** : Un '0 dominant' indiquant aux autres nœuds qu'un nœud CAN souhaite communiquer
- **ID** : Identifiant de la trame - les valeurs plus basses ont une priorité plus élevée
- **RTR (Remote Transmission Request)** : Indique si un nœud envoie des données ou demande des données spécifiques à un autre nœud
- **Control** : Contient le bit d'extension d'identifiant (IDE) qui est un '0 dominant' pour le format 11 bits. Il inclut aussi le code de longueur de données (DLC) sur 4 bits, spécifiant la taille des données à transmettre (0 à 8 octets)
- **Data** : Contient les octets de données (payload), incluant les signaux CAN qui peuvent être décodés pour extraire des informations
- **CRC (Cyclic Redundancy Check)** : Utilisé pour garantir l'intégrité des données
- **ACK (Acknowledgment)** : Indique si le nœud a reçu et reconnu les données correctement
- **EOF (End of Frame)** : Marque la fin de la trame CAN

Il existe quatre types de trames CAN :

1. **Trame de données** : Comme illustré précédemment, elle transporte des données d'un nœud émetteur vers un ou plusieurs nœuds récepteurs. Nous avons détaillé chaque champ précédemment. En pratique, plus de 99% des cas d'utilisation concernent exclusivement les trames de données CAN.

2. **Trame d'erreur** : Utilisée par un nœud CAN pour signaler la détection d'une erreur de communication. Elle contient un drapeau d'erreur et un délimiteur d'erreur. Certains utilisateurs enregistrent spécifiquement ces trames, notamment pour diagnostiquer des problèmes de communication entre ECU.

3. **Trame de demande (Remote Frame)** : Permet de demander des données spécifiques à un nœud CAN. Similaire à une trame de données mais sans champ Data et avec le bit RTR à 1 (récessif). Son utilisation est rare - parmi des milliers de cas, nous l'avons rencontrée 1 ou 2 fois seulement. La plupart des protocoles de haut niveau (comme OBD2/J1939) utilisent plutôt des trames de données pour demander des informations.

4. **Trame de surcharge** : Peut être utilisée pour ajouter un délai supplémentaire entre les trames CAN lorsque certains nœuds nécessitent plus de temps de traitement. En pratique, ces trames sont quasiment jamais utilisées - en 10 ans, nous n'avons jamais rencontré de cas où elles étaient pertinentes.

1.2.6 Comment enregistrer les données du bus CAN

Pour enregistrer les données du bus CAN en suit les étapes suivantes :

1. Choisir le matériel adapté

   Commencez par déterminer comment vous souhaitez collecter les données CAN :

- CAN-vers-USB : Le streaming de données en temps réel via une interface CAN-USB comme le CANmod.router est utile pour le diagnostic sur site ou le reverse engineering
- CAN-vers-SD : Un enregistreur de données CAN comme le CANedge1 peut capturer des problèmes intermittents ou servir de boîte noire pour le dépannage ou les litiges de garantie
- CAN-vers-cloud : Les enregistreurs connectés comme les CANedge2/CANedge3 envoient les données vers votre serveur pour la télémétrie, la maintenance prédictive ou les tableaux de bord

2. Identifier le câble d'adaptation à utiliser

Ensuite, déterminez quel adaptateur utiliser. Cela dépend de l'application, mais voici 4 options courantes :

- **Adaptateur OBD2** : Dans la plupart des voitures (et certains fourgons/camions), permet d'accéder aux données OBD2/UDS. Peut aussi donner accès aux données CAN propriétaires du véhicule.
- **Adaptateur J1939** : Dans la plupart des véhicules lourds (camions, bus, pelleteuses, tracteurs...), donne accès aux données CAN brutes (protocole J1939).

- **Adaptateur M12 :** Dans la plupart des navires maritimes et certaines machines industrielles, permet d'enregistrer les données CAN brutes (NMEA 2000 ou CANopen).
- **Sans contact *:*** Option universelle : éviter complètement le connecteur et utiliser l'induction pour lire directement les données sur les câbles CAN high/low.

3. Configurer et connecter votre appareil

Avant de connecter votre appareil, prenez en compte deux éléments :

- **Débit binaire (baud-rate)** : Le débit binaire de votre appareil doit correspondre à celui du bus CAN. Si vous vous connectez à un bus CAN actif, certains appareils (comme le CANedge) peuvent détecter automatiquement le débit binaire pour simplifier cette étape.
- **Requêtes** : Si vous souhaitez enregistrer des données sur demande comme OBD2/UDS, vous devez configurer votre appareil pour transmettre les "messages de requête" appropriés.

Vous pouvez maintenant connecter votre appareil et vérifier qu'il enregistre bien les données. Si ce n'est pas le cas, consultez nos 10 conseils de dépannage principaux (illustration).

4. Examiner vos données CAN brutes

Une fois l'enregistrement terminé (par exemple après un trajet véhicule), vous pouvez analyser le fichier journal résultant.

Sur l'illustration, nous montrons un fichier journal contenant des données CAN brutes (J1939) enregistrées à l'aide d'un CANedge dans un camion poids lourd. Plus précisément, les données sont affichées sous forme de tableau dans un logiciel appelé asammdf.

Timestamp	CAN ID	Données (hex)
12.345678	0x123	01 A4 FF 00
12.345690	0x456	00 00 3E 80

Figure 1.8 - Exemple de fichier de log CAN (extrait)

On peut observer que chaque ligne correspond à une trame CAN horodatée, comprenant :

- **Un identifiant CAN (CAN ID)**
- **Une charge utile de données (data payload)**

Cette visualisation tabulaire permet d'analyser facilement la séquence et le contenu des messages échangés sur le bus.

1.3 Contexte générale du projet

Dans un monde où les véhicules deviennent de plus en plus intelligents et interconnectés, la cybersécurité des systèmes embarqués automobiles représente un enjeu majeur. Le bus CAN (Controller Area Network), bien qu'essentiel pour la communication entre les différents modules électroniques des véhicules, n'intègre pas de mécanismes de sécurité par défaut. Cette faiblesse le rend vulnérable à des attaques telles que le déni de service, la falsification de données ou encore la suspension de messages critiques.

Dans ce contexte, notre projet vise à détecter automatiquement les anomalies ou attaques dans le trafic CAN à l'aide de techniques d'intelligence artificielle, en s'appuyant sur une analyse prédictive des données de communication entre ECUs.

1.3.1 Problématique

Le protocole CAN, conçu dans les années 1980, était à l'origine prévu pour des environnements fermés, sans considération pour les menaces externes. Aujourd'hui, avec l'évolution vers des véhicules connectés (via Bluetooth, Wi-Fi, réseaux cellulaires), ce bus est devenu exposé à des risques d'intrusion.

La problématique centrale de notre projet est donc la suivante :

> *Comment peut-on détecter automatiquement des attaques sur le bus CAN à l'aide d'un système de maintenance prédictive basé sur l'intelligence artificielle, en exploitant uniquement les données de communication entre modules ?*

Ce projet s'inscrit dans une logique de prévention proactive, en identifiant les comportements suspects avant que ceux-ci n'entraînent des dysfonctionnements ou des dommages.

1.3.2 Objectifs

Le projet a pour objectifs principaux :

- Capturer et analyser les données du bus CAN dans des conditions normales de conduite.
- Simuler différentes attaques sur les fichiers log CAN (DoS, fuzzing, suspension).
- Extraire des paramètres significatifs à partir des trames CAN (ID, fréquence, entropie…).

- Construire un jeu de données étiqueté prêt pour l'apprentissage supervisé.
- Entraîner un ou plusieurs modèles de machine learning pour la classification des trames en "normales" ou "malveillantes".
- Évaluer la performance des modèles et leur capacité à détecter des intrusions de manière fiable.

1.3.3 Planning du projet

Le planning d'un projet revêt une importance capitale pour sa gestion et son aboutissement. En effet, il permet d'organiser de manière efficace les différentes tâches à réaliser, en déterminant leur séquence, leur durée, et les ressources nécessaires. Un planning bien établi permet de respecter les délais fixés, d'optimiser l'utilisation des ressources disponibles, et de garantir la qualité du travail réalisé.

Pour visualiser plus clairement la planification temporelle des différentes étapes de notre projet, On'a établi le tableau suivant, détaillant les tâches à réaliser, et leur durée estimée.

Nom de la tache	Début	Fin	Durée
Étude du bus CAN	15/03/2025	25/03/2025	11
Recherche sur les types d'attaques	26/03/2025	05/04/2025	11
Acquisition et préparation des données (Extraction des paramètres)	06/04/2025	30/04/2025	25
Modélisation et entraînement	01/05/2025	20/05/2025	20
Évaluation et visualisation	21/05/2025	31/05/2025	11
Préparation du rapport	01/06/2025	15/06/2025	15
Préparation de la présentation	16/06/2025	18/06/2025	3

Ce découpage nous a permis de gérer efficacement le temps imparti tout en respectant les exigences pédagogiques et techniques du module.

1.4 Conclusion

Dans ce premier chapitre, nous avons posé les bases théoriques et contextuelles de notre projet. Nous avons tout d'abord étudié en détail le fonctionnement du bus CAN, élément central des communications dans les véhicules modernes. Cette exploration nous a permis de

comprendre la structure des trames CAN, les couches impliquées dans la transmission, ainsi que les avantages et les limites de ce protocole, notamment en matière de cybersécurité.

Ensuite, nous avons introduit le contexte général du projet, en mettant en lumière les enjeux de sécurité liés à l'utilisation du bus CAN dans un environnement automobile de plus en plus connecté. La problématique du projet, centrée sur la détection d'attaques à partir des données du bus CAN, a été formulée, et des objectifs clairs ont été définis. Le cahier des charges a permis de cadrer techniquement le projet, tandis que le planning a assuré une gestion structurée des différentes étapes.

Ce chapitre introductif constitue donc le socle de notre démarche. Il nous prépare à aborder, dans le chapitre suivant, les aspects plus techniques de la préparation des données, incluant l'acquisition, la structuration et l'extraction des paramètres nécessaires à la détection d'anomalies.

Chapitre 2 : Préparation et inspection des données

2.1 Introduction

Dans ce chapitre, nous décrivons les différentes étapes de la préparation des données à partir des fichiers de logs extraits du bus CAN. Ces données constituent la base d'apprentissage pour la détection d'attaques cybernétiques par des modèles d'intelligence artificielle. Nous avons organisé les jeux de données, extrait des paramètres significatifs et préparé un dataset global pour l'entraînement de modèles de classification.

2.2 Acquisition des données de trafic CAN normal

L'acquisition des données normales du bus CAN a été réalisée à l'aide d'une Renault Clio dans un environnement urbain. Pour collecter ces données, nous avons utilisé un ordinateur portable connecté à l'interface du bus CAN du véhicule via un adaptateur ***CAN-to-USB (modèle CANtact)*** branché au port ***OBD-II***. La capture des messages CAN a été effectuée en temps réel à l'aide de l'outil ***candump***, fourni par la suite logicielle `can-utils`, qui permet d'enregistrer fidèlement les trames circulant sur le réseau CAN san s les modifier.

Figure 2.1 - CAN-to-USB (modéle CANtact)

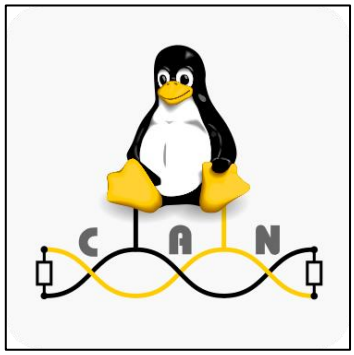

Figure 2.2 - Logicielle can-utils

Les fichiers ainsi obtenus contiennent du trafic entièrement légitime, sans aucune altération manuelle ou automatisée. Ces données constituent donc une référence précieuse pour l'entraînement initial des modèles d'intelligence artificielle à reconnaître un comportement « sain » du réseau.

Le fichier principal, `full_data_capture.log`, contient l'intégralité de la session d'enregistrement, d'une durée d'environ

275 secondes (4 minutes et 35 secondes), soit un total de 386 567 trames CAN avec 55 identifiants CAN uniques.

Chaque ligne enregistrée dans les fichiers de logs CAN suit un format standard utilisé par l'outil candump. Un exemple typique de trame enregistrée est le suivant :

```
(1508687476.438095) slcan0 2C6#FFFFFFFFFFF0
```

Cette ligne contient plusieurs informations clés :

- **Horodatage** : `1508687476.438095` indique le temps (en secondes depuis l'époque Unix) auquel le message a été capturé. Cela permet de reconstituer la séquence temporelle des trames.
- **Interface CAN** : `slcan0` désigne l'interface réseau virtuelle utilisée pour la capture (ici via un adaptateur série vers CAN).
- **Identifiant CAN (CAN ID)** : `2C6` est l'identifiant du message. Il détermine la priorité du message sur le bus ainsi que le type d'information transportée (ex. vitesse, température, etc.).
- **Payload** : La partie après le #, ici `FFFFFFFFFFF0`, est la charge utile (payload) contenant les données brutes du capteur ou du système.

2.3 Simulation et description des attaques CAN

Dans le but de développer un système de détection d'intrusions efficace, nous avons simulé différentes attaques sur les données CAN, étant donné qu'il n'était pas possible d'exécuter de véritables attaques sur un véhicule réel. Chaque attaque a été conçue de manière à reproduire des comportements anormaux réalistes observés dans des scénarios de cybersécurité automobile. Ces attaques ont été ajoutées à un trafic CAN normal pour former des cas d'étude utilisables dans l'entraînement d'un modèle d'intelligence artificielle. Les attaques sont décrites ci-dessous.

2.3.1 Attaque par Déni de Service (DoS)

Cette attaque consiste à saturer le bus CAN en injectant un grand nombre de messages ayant un identifiant très prioritaire (`ID 000`) pendant une période de 10 secondes. L'objectif est d'occuper le canal de communication et d'empêcher les autres messages critiques de circuler normalement.

- **Format des messages injectés** :

  `(1508687506.000236) slcan0 000#0000000000000000`

- **Fréquence d'injection** : 4 messages par milliseconde, soit 40001 messages au total.
- **Conséquence** : Les messages légitimes sont bloqués pendant cette période, ce qui peut entraîner des défaillances dans les systèmes critiques (ex. ABS, direction assistée).

Cette attaque simule un ***flood*** du bus CAN avec des trames vides mais prioritaires, bloquant ainsi les communications normales.

2.3.2 Attaque par modification de charge utile (Payload Fuzzing)

Dans ce scénario, nous ne modifions pas le nombre de messages ni les identifiants, mais nous altérons la charge utile de certains messages. L'attaque cible spécifiquement les messages ayant un identifiant bien préci (par exemple `18A`), en remplaçant leur contenu par un message de 16 bits (`FFFFFFFFFFFFFFFF`)

- **Exemple de message modifé** :

  `(1508687506.038589) slcan0 18A#FFFFFFFFFFFFFFFF`

- **Objectif** : Tromper les modules récepteurs avec des données non valides, pouvant perturber la logique des systèmes (comme un régulateur de vitesse ou un capteur).

Cette attaque est discrète, car elle n'augmente pas le volume du trafic, mais elle modifie des données de manière subtile, ce qui peut la rendre difficile à détecter sans analyse approfondie du contenu.

2.3.3 Attaque par Suspension de Communication

Dans ce cas, l'attaque consiste à supprimer les messages émis avec un identifiant bien précis (par exemple `2C6`) pendant 10 secondes, simulant ainsi une interruption de communication.

Exemple d'attack :

- **Début de l'attaque :**

  `(1508687499.999696) slcan0 2C6#FFFFFFFFFFF0`

- **Fin de l'attaque :**

  `(1508687499.999696) slcan0 2C6#FFFFFFFFFFF0`

- **Conséquence** : L'absence de messages critiques peut être interprétée par les ECUs comme une défaillance du composant ou du capteur correspondant.

Cette attaque reproduit un comportement possible en cas de coupure réseau locale, mais peut aussi être causée volontairement par un acteur malveillant souhaitant désactiver un système embarqué (comme un capteur de sécurité).

2.4 Organisation du projet

Avant de commencer n'importe quel projet, le choix de l'environnement de développement et la structuration claire du projet sont des étapes essentielles pour garantir une bonne organisation, une maintenance facilitée et une collaboration efficace. Dans notre cas, nous avons choisi de travailler dans Visual Studio Code (VS Code), un environnement moderne, léger et extensible, qui prend en charge à la fois le développement en Python et l'exécution de notebooks interactifs. Grâce à l'extension officielle Jupyter, il a été possible de créer et exécuter des fichiers .ipynb directement dans VS Code, ce qui nous a permis de combiner interactivité, visualisation des résultats, et clarté du code dans un seul espace de travail.

La structure du projet a également été pensée pour séparer clairement les différentes étapes : acquisition des données, injection des attaques, extraction des caractéristiques, prétraitement, modélisation et visualisation. Cette approche modulaire et bien organisée nous a permis de travailler de manière itérative et reproductible, tout en facilitant le débogage, l'analyse et la réutilisation du code pour d'autres jeux de données ou d'autres types d'attaques.

2.4.1 Environnement de dévolepement (VS Code)

Avant de commencer tout développement, le choix de l'environnement de travail est crucial pour assurer une organisation efficace, une bonne productivité et un suivi fluide du projet. Dans notre cas, nous avons opté pour **Visual Studio Code (VS Code)** comme environnement de développement principal. VS Code est un éditeur de code **léger, multiplateforme, open-source et hautement personnalisable**, qui offre de puissantes fonctionnalités tout en restant accessible pour des projets à complexité variable.

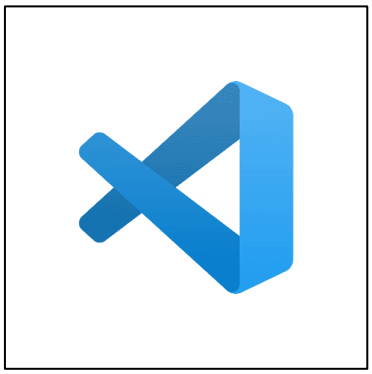

Figure 2.3 - Visual Studio Code

L'un des grands avantages de VS Code est son **support natif des notebooks Jupyter** grâce à l'extension officielle **Jupyter**, qui permet d'exécuter des fichiers .ipynb directement dans l'éditeur, sans passer par un navigateur web. Cette intégration nous a permis de bénéficier de l'interactivité des notebooks pour l'exploration de données, l'extraction de caractéristiques, la visualisation graphique et le prototypage rapide de modèles, tout en profitant de la puissance d'un IDE moderne (navigation entre fichiers, autocomplétion, gestion de versions Git, extensions Python, etc.).

Parmi les fonctionnalités clés de l'environnement que nous avons utilisées :

- **Explorateur de fichiers intégré** pour naviguer entre scripts, jeux de données et résultats.
- **Terminal intégré** pour exécuter des commandes système ou des scripts Python rapidement.
- **Extensions utiles** : Python, Jupyter, Pylance, GitLens, etc.
- **Contrôle de version Git** pour assurer le suivi des modifications et la sauvegarde du travail.
- **Support de visualisation de notebooks** : exécution cellule par cellule, affichage des graphiques matplotlib directement dans l'interface, et possibilité de combiner texte, code et visualisation dans un même fichier .ipynb.

Grâce à cette configuration, nous avons pu centraliser l'intégralité de notre flux de travail dans un **environnement unique, stable et intuitif**, de la lecture des fichiers .log à

2.4.2 Langage du programmation

Le projet a été entièrement développé en Python, un langage de programmation moderne, open-source, et largement utilisé dans les domaines de la science des données, de l'intelligence artificielle et du traitement des séries temporelles. Son syntaxe simple, sa communauté active et son écosystème riche en bibliothèques spécialisées en font un choix privilégié pour la recherche appliquée et les projets de prototypage rapide.

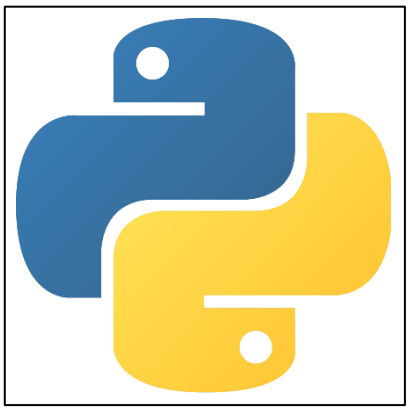

Figure 2.4 - Python

Nous avons utilisé la version Python 3.13.3, qui offre de meilleures performances et une compatibilité complète avec les outils modernes. Les bibliothèques suivantes ont été essentielles à la réalisation de notre projet :

- `pandas` : utilisée pour la manipulation, la lecture et l'organisation des fichiers .log en `DataFrames` tabulaires. Elle a permis de structurer les données CAN, de les fusionner, d'effectuer des filtres, des groupements et des opérations statistiques.

Figure 2.5 - Bibliothèque pandas

- `numpy` : complément indispensable à `pandas`, utilisée pour les calculs ***numériques vectorisés***, la gestion des tableaux multidimensionnels et certaines opérations mathématiques de bas niveau.

Figure 2.6 - Bibliothèques NumPy

- `matplotlib` et `seaborn` : bibliothèques de visualisation graphique, utilisées pour générer des histogrammes, courbes temporelles, heatmaps, et autres représentations visuelles des comportements CAN normaux ou anormaux. Seaborn, basé sur matplotlib, a permis de produire des graphiques plus esthétiques avec moins de code.
- `math` et `statistics` : utilisées pour calculer certaines **mesures personnalisées** (entropie, moyenne des intervalles, écart type, etc.) nécessaires à l'extraction de paramètres pertinents pour la détection d'intrusions.

L'ensemble de ces bibliothèques a été exploité de manière **complémentaire** pour couvrir tout le pipeline.

2.4.3 Structure du projet

Le projet est organisé de manière modulaire avec des dossiers distincts pour chaque étape :

```
/AI_CAN_BUS_PROJECT/
|
├── dataSet/ # Contient tous les fichiers de données utilisés
├── documents/ # Contient les fichiers liés au projet
├── images/ # Dossier regroupant toutes les visualisations générées
├── models/ # Emplacement pour sauvegarder les modèles entraînés
├── notebooks/ # Fichiers Jupyter Notebook (.ipynb)
├── webpage/ # Dossier réservé à une éventuelle interface web
├── README.md # Fichier de description du projet
├── requirements.txt # Liste des bibliothèques Python nécessaires
```

Détails des dossiers :

- `dataSet/` : Ce répertoire contient toutes les données brutes et traitées, y compris les fichiers .log de trafic CAN normal et d'attaques, ainsi que les fichiers .csv générés après extraction des paramètres. C'est le cœur des données du projet.
- `documents/` : Utilisé pour stocker les fichiers du rapport de projet, les annexes, ou toute autre documentation liée, comme les consignes du module ou les présentations.
- `images/` : Ce dossier regroupe toutes les figures et graphiques produits pendant l'analyse, qu'ils soient insérés dans le rapport ou utilisés dans les notebooks (ex : histogrammes, heatmaps, timelines).
- `models/` : Il contient les modèles de machine learning entraînés et sauvegardés (ex. fichiers .pkl ou .joblib) afin de pouvoir les réutiliser sans avoir à les réentraîner.
- `notebooks/` : Ce dossier regroupe les notebooks Jupyter utilisés pour écrire le code de traitement, d'analyse et de modélisation. Chaque notebook correspond à une étape : extraction de caractéristiques, visualisation, entraînement, évaluation, etc.
- `webpage/` : Prévu pour une éventuelle extension web du projet (ex : interface interactive, dashboard Streamlit ou Flask),

Fichiers racine :

- `README.md` : Fournit une vue d'ensemble du projet, des instructions d'installation, d'exécution, ainsi que des explications pour les contributeurs ou évaluateurs.
- `requirements.txt` : Liste les dépendances Python nécessaires à l'exécution du projet. Ce fichier permet d'installer automatiquement tous les paquets via la commande pip install -r requirements.txt.

2.5 Extraction des paramètres

L'objectif principal de cette étape est d'extraire des **caractéristiques pertinentes** de chaque trame CAN, afin de permettre à un modèle d'intelligence artificielle de différencier un comportement **normal** d'un comportement **malveillant**. Ces caractéristiques, ou **paramètres**, jouent un rôle essentiel dans la qualité de la détection, car elles capturent les aspects statistiques et comportementaux du trafic CAN.

Pour mettre en œuvre cette extraction, nous avons commencé par structurer le répertoire `dataSet/` en **trois dossiers distincts**, chacun correspondant à un scénario d'attaque spécifique:

```
/AI_CAN_BUS_PROJECT/
|
├── dataSet/ # Contient tous les fichiers de données utilisés
|   ├── dos/
|   |   ├── normal.log
|   |   ├── dos.log
|   ├── fuzzing_payload/
|   |   ├── normal.log
|   |   ├── fuzzing_payload.log
|   ├── suspension/
|   |   ├── normal.log
|   |   ├── suspension.log
├── ...
```

Dans chaque dossier, nous avons placé **deux fichiers** :

- Un fichier **normal.log**, contenant uniquement du trafic CAN légitime
- Un fichier correspondant à l'attaque ciblée (ex. : **dos.log**, **fuzzing_payload.log**, ou **suspension.log**)

Ensuite, nous avons développé un **script Python dédié pour chaque type d'attaque**, chargé d'effectuer les opérations suivantes :

1. Lecture et traitement ligne par ligne des fichiers .log
2. Extraction des champs utiles : horodatage, identifiant CAN, charge utile
3. Calcul de **paramètres statistiques** : fréquence d'émission des trames, entropie des données, intervalle entre trames du même ID, taille du payload, etc.
4. Enregistrement des résultats dans un fichier **CSV** pour une exploitation ultérieure

Cette organisation modulaire nous a permis d'isoler les différents types d'attaques, d'adapter les paramètres extraits à leurs particularités, et de constituer progressivement un jeu de données complet et structuré pour les phases de modélisation et d'entraînement.

2.5.1 Parametres de DOS

Pour analyser l'attaque de type Déni de Service (DoS), nous avons mis en place un processus d'extraction de paramètres à partir des fichiers logs correspondant. Cette attaque se

caractérise par l'injection massive de messages CAN avec un identifiant prioritaire (000) à très haute fréquence pendant une période de 10 secondes, ce qui sature le bus CAN et empêche la circulation des messages légitimes.

Étapes d'extraction et d'analyse

Pour détecter ce type d'anomalie, nous avons extrait deux paramètres principaux :

- **CAN_ID_Inter_Arrival** : temps entre deux trames consécutives d'un même identifiant CAN.
- **CAN_ID_Window_Count** : nombre total de messages par identifiant CAN dans une fenêtre glissante de 10 secondes.

Nous avons suivi une approche en deux phases :

1. Un ***script Python*** pour convertir les fichiers .log en fichiers .csv avec extraction automatique des deux paramètres mentionnés.
2. Un ***notebook Jupyter*** pour analyser et visualiser les statistiques de ces paramètres, en comparant le trafic normal avec celui de l'attaque.

Phase 1 : Script Python d'extraction

Le script lit les fichiers logs ligne par ligne, extrait les composants essentiels (timestamp, CAN_ID, payload) et calcule pour chaque message :

- Le **temps d'arrivée relatif** par ID :

```
mean_inter_arrival = round(sum(inter_arrivals) / len(inter_arrivals), 6)
```

- Le **nombre de messages** dans une fenêtre de 10 secondes :

```
window = int(timestamp_float // 10)
window_counts[can_id][window] = window_counts[can_id].get(window, 0) + 1
```

- Il écrit ensuite les résultats dans un fichier .csv

```
input_file_path = r"...\dos\full_data_capture.log"
output_file_path = r"...\dos\normal.csv"
```

```
input_file_path = r"...\dos\dos.log "
output_file_path = r"...\dos\dos.csv"
```

- Ces fichiers a ensuite servi de base pour l'analyse statistique.

Phase 2 : Traitement via Jupyter Notebook

Le notebook Jupyter est organisé en plusieurs cellules, chacune remplissant un rôle spécifique :

- Chargement et nettoyage des données

```
window = int(timestamp_float // 10)
window_counts[can_id][window] = window_counts[can_id].get(window, 0) + 1
```

- Analyse statistique globale

```
window = int(timestamp_float // 10)
window_counts[can_id][window] = window_counts[can_id].get(window, 0) + 1
```

- Analyse statistique par CAN_ID

```
stats_dos = dos_df.groupby('CAN_ID').agg({
    'CAN_ID_Inter_Arrival': ['min', 'max', 'mean'],
    'CAN_ID_Window_Count': ['min', 'max', 'mean']
})
stats_dos.to_csv('dos_can_id_statistics.csv', quoting=csv.QUOTE_NONNUMERIC)
```

- Enrgitrement de 3 fichiers sous forme csv

```
stats_df.to_csv(os.path.join(data_dir, 'summary_statistics.csv'))
stats_normal.to_csv(os.path.join(data_dir, 'normal_can_id_statistics.csv'),
index=False, quoting=csv.QUOTE_NONNUMERIC)
stats_dos.to_csv(os.path.join(data_dir, 'dos_can_id_statistics.csv'), index=False,
quoting=csv.QUOTE_NONNUMERIC)
```

Résultats observés

Aprés la création et l'éxuction du code en se trouve devant le résultat suivant :

```
/AI_CAN_BUS_PROJECT/
|
├── dataSet/ # Contient tous les fichiers de données utilisés
|    ├── dos/
|    |     ├── normal.log
|    |     ├── dos.log
|    |     ├── summary_statistics.csv
|    |     ├── normal_can_id_statistics.csv
|    |     ├── dos_can_id_statistics.csv
|    |     ├── script.py
|    |     ├── analyse.ipynb
├── ...
```

Les résultats obtenus dans le fichier `summary_statistics.csv` confirment clairement les différences de comportement entre le trafic **normal** et celui sous attaque **DoS**.

	Normal		DOS	
	CanId InterArrival	CanId WindowCount	CanId InterArrival	CanId WindowCount
25%	0.010022	200	0.000289	356
50%	0.019925	500	0.010069	899
75%	0.049586	1000	0.020096	16012
count	386567	386567	141927	141927
max	3.050461	1001	12.013576	23989
mean	0.039140621	597.3837731	0.032080317	6247.435393
min	2.50E-05	2	3.70E-05	1
std	0.084467564	369.0286523	0.21297753	9352.25882

Tableau 2.1 - Résultat dans summary_statistics.csv de DOS attack

- L'intervalle maximal entre deux messages (inter-arrival) pour le trafic normal est extrêmement faible, avec valeur de 3.050461 s, par contre dans le trafic sous un attack DOS en remarque que la valeur saut jusqu'à 12.013576 s
- Le nombre de messages dans une fenêtre de 10 secondes (Window Count) pour le trafic norml atteint un maximum de 1001 messages, contre 23989 messages au maximum dans le trafic DOS

Cette analyse statistique conforte les hypothèses formulées lors de l'extraction des paramètres et renforce la pertinence de ces indicateurs pour l'entraînement de modèles de détection d'intrusion.

2.5.2 Parametres de Fuzzing_Payload

Dans ce scénario, nous ne modifions **ni le nombre de messages** ni les **identifiants CAN**, mais nous altérons la **charge utile (payload)** de certains messages pour les rendre anormaux. L'attaque cible spécifiquement un identifiant bien précis (dans notre cas, **18A**) en remplaçant le contenu par une valeur maximale non observée dans le trafic normal : FFFFFFFFFFFFFFFF (en hexadécimal), soit **la valeur maximale possible sur 64 bits**.

Stratégie de détection

Puisque cette attaque est **furtive** et ne se manifeste pas par une augmentation du volume ou de la fréquence des messages, nous avons défini deux nouveaux paramètres :

- **Payload_Entropy** : mesure le degré d'aléa ou de redondance du contenu. Une valeur faible indique une charge utile répétitive (ex. FF répété), ce qui est typique d'un fuzzing malveillant.
- **Payload_Decimal** : représentation entière du payload (converti depuis l'hexadécimal). Cette valeur permet de détecter des messages anormalement élevés — par exemple FFFFFFFFFFFFFFFF correspond à **18 446 744 073 709 551 615**, soit $\mathbf{2^{64} - 1}$, la valeur maximale possible pour un champ de 64 bits.

Phase 1 : Extraction via Script Python

Le script Python `Payload_Entropy_and_Payload_decimal.py` lit chaque ligne du fichier log, extrait le payload, et calcule les deux nouveaux paramètres.

- Extrait : calcul de l'entropie

```
def calc_entropy(payload):
    bytes_list = [int(payload[i:i+2], 16) for i in range(0, len(payload), 2)]
    value_counts = np.bincount(bytes_list, minlength=256)
    return entropy(value_counts[value_counts > 0])
```

- Extrait : conversion en entier

```
def payload_to_decimal(payload):
    return int(payload, 16)
```

- Les résultats sont ensuite écrits dans un fichier .csv contenant les colonnes

```
    Timestamp, Interface, CAN_ID, Payload, Payload_Entropy, Payload_Decimal
```

- Exemple d'une ligne générée pour un message malveillant :

```
    1508687506.038589, slcan0, 18A, FFFFFFFFFFFFFFFF, 0.0, 18446744073709551615
```

Phase 2 : Analyse via Jupyter Notebook

Dans le fichier `analyse_fuzzing.ipynb`, les colonnes Payload_Entropy et Payload_Decimal sont analysées statistiquement :

- Les valeurs d'entropie proches de 0.0 indiquent une charge utile non aléatoire, souvent composée d'un seul octet répété (comme FF), ce qui correspond aux messages modifiés.
- Les valeurs décimales extrêmes comme 18446744073709551615 sont typiquement absentes du trafic normal, ce qui les rend particulièrement distinctives.

Résultats observés

Aprés la création et l'éxuction du code en se trouve devant le résultat suivant :

```
/AI_CAN_BUS_PROJECT/
|
├── dataSet/ # Contient tous les fichiers de données utilisés
|   ├── ...
|   ├── fuzzing_payload/
|   |   ├── normal.log
|   |   ├── fuzzing_payload.log
|   |   ├── summary_statistics.csv
|   |   ├── normal_entropy_decimal_stats.csv
|   |   ├── fuzzing_entropy_decimal_stats.csv
|   |   ├── Payload_Entropy_and_Payload_decimal.py
|   |   ├── analyse_fuzzing.ipynb
|   ├── ...
├── ...
```

Les résultats obtenus dans le fichier `summary_statistics.csv` confirment clairement les différences de comportement entre le trafic **Normal** et celui sous attaque **Fuzzing Payload**.

	Normal		Fuzzing Payload	
	Payload Entropy	Payload Decimal	Payload Entropy	Payload Decimal
count	325185	325185	115971	115971
mean	1.31995015	2.61E+18	1.30903885	2.60E+18
std	0.68919354	4.53E+18	0.69499458	4.53E+18
min	0	0	0	0
25%	0.69315	218103808	0.69315	8388608
50%	1.60944	1.09E+16	1.60944	1.24E+16
75%	1.90615	3.56E+18	1.90615	2.74E+18
max	2.07944	1.80E+19	2.07944	1.84E+19

Tableau 2.2 - Résultat dans summary_statistics.csv de Fuzzing Payload attack

L'attaque Fuzzing Payload, bien que discrète, introduit des anomalies détectables en analysant les **caractéristiques du contenu**. Les paramètres **Payload_Entropy** et **Payload_Decimal** se sont avérés pertinents pour distinguer les trames malveillantes dans un contexte où les métriques temporelles (inter-arrival, fréquence) ne suffisent pas. Ces paramètres enrichissent le dataset et permettent d'élargir les capacités de détection des modèles de machine learning au-delà des simples comportements temporels.

2.5.3 Parametres de Suspension

L'attaque **Suspension** (aussi appelée "message drop" ou "message deletion") consiste à **supprimer volontairement certains messages** CAN critiques pendant une période définie. Contrairement à l'attaque DoS qui injecte massivement des trames, la suspension rend un ou plusieurs identifiants **silencieux temporairement**, ce qui peut induire un comportement inattendu dans les ECU qui dépendent de ces messages.

Étapes d'extraction et d'analyse

Pour détecter ce type d'attaque, nous avons choisi d'analyser un seul paramètre central :

- **CAN_ID_Inter_Arrival** : mesure du **temps écoulé entre deux messages** successifs d'un même identifiant. En cas de suspension, cet intervalle augmente fortement.

Phase 1 : Script Python d'extraction

Le script `canID_Inter_Arrival.py` lit chaque ligne du fichier log et calcule les **intervalles entre trames** pour chaque identifiant CAN. Un seuil de **5 secondes** a été utilisé pour **détecter les écarts anormaux** qui pourraient indiquer une attaque.

Phase 2 : Analyse statistique via Notebook

Dans le notebook canID_Inter_Arrival.ipynb, nous avons comparé les valeurs d'**inter-arrival** entre :

- Le trafic **normal** (normal.log)
- Le trafic **sous attaque** (suspension.log)

Un résumé statistique a été généré et enregistré dans un tableau CSV

Résultats observés

```
/AI_CAN_BUS_PROJECT/
|
├── dataSet/ # Contient tous les fichiers de données utilisés
|   ├── ...
|   ├── suspension/
|   |   ├── normal.log
|   |   ├── suspension.log
|   |   ├── summary_statistics.csv
|   |   ├── normal_can_id_inter_arrival_stats.csv
|   |   ├── suspension_can_id_inter_arrival_stats.csv
|   |   ├── canID_Inter_Arrival.py
|   |   ├── canID_Inter_Arrival.ipynb
├── ...
```

Les résultats obtenus dans le fichier `summary_statistics.csv` confirment clairement les différences de comportement entre le trafic **Normal** et celui sous attaque **Suspension**

	Normal	Suspension
	CAN_ID_Inter_Arrival	CAN_ID_Inter_Arrival
count	386567	115472
mean	0.039109135	0.039224028
std	0.084164708	0.088689354
min	1.00E-05	1.00E-05
25%	0.01002	0.01002
50%	0.01992	0.01993
75%	0.04957	0.04972
max	3.05046	10.0004

Tableau 2.3 - Résumé des Inter-Arrivals dans les jeux de données normal et suspension

- Les valeurs statistiques sont globalement similaires en dehors de l'anomalie, ce qui montre que l'attaque est discrète et localisée.
- Le maximum d'intervalle d'arrivée dans le trafic normal est de 3.05 s, tandis qu'il atteint 10.00 s pendant l'attaque, ce qui correspond exactement à la durée d'interruption des messages ID 2C6.
- Ce saut brutal dans l'intervalle est un indicateur très fiable de suspension de messages.

L'attaque Suspension est efficace car silencieuse : elle n'augmente pas le volume du trafic, mais **interrompt temporairement un flux attendu**. Le paramètre

CAN_ID_Inter_Arrival permet de détecter ce type de comportement en mettant en évidence des **écarts inhabituels** entre trames. Ce type d'analyse temporelle est donc essentiel dans un système de détection d'intrusion basé sur des logs CAN.

2.6 Préparation des données brutes

La préparation des données brutes constitue une étape cruciale dans notre projet, car elle garantit que les données issues des fichiers de logs CAN sont correctement structurées, enrichies et étiquetées pour permettre l'entraînement de modèles d'intelligence artificielle. Cette section décrit en détail les différentes étapes réalisées pour transformer le fichier de log brut full_data_capture.log en un jeu de données exploitable, nommé generated.csv, optimisé pour la maintenance prédictive et la détection d'attaques (DoS, Fuzzing, Suspension). Le processus a été implémenté à l'aide d'un notebook Jupyter, exécuté dans l'environnement Visual Studio Code avec Python 3.13.3.

2.6.1 Objectifs de la préparation des données

L'objectif principal de cette phase est de convertir les données brutes, capturées sous forme de logs CAN, en un format structuré et enrichi, adapté à l'apprentissage supervisé. Plus précisément, nous avons cherché à :

- ✓ Lire et parser le fichier de log brut pour extraire les informations pertinentes (horodatage, interface, identifiant CAN, charge utile).
- ✓ Injecter des attaques simulées (DoS, Fuzzing, Suspension) pour créer un jeu de données représentatif des scénarios malveillants.
- ✓ Calculer des caractéristiques avancées (par exemple, CAN_ID_Inter_Arrival, CAN_ID_Window_Count, Payload_Entropy) pour capturer les comportements normaux et anormaux.
- ✓ Normaliser les caractéristiques pour faciliter l'entraînement des modèles.
- ✓ Étiqueter chaque trame CAN comme normale ou malveillante (DoS, Fuzzing, Suspension).
- ✓ Exporter les données traitées dans un fichier CSV pour une utilisation ultérieure.

2.6.2 Description du fichier de log brut

Le fichier d'entrée, full_data_capture.log, contient 386 567 trames CAN capturées à l'aide d'un adaptateur CAN-to-USB (modèle CANtact) connecté au port OBD-II d'une Renault

Clio, sur une durée d'environ 275 secondes. Chaque ligne du fichier suit le format standard de l'outil candump de la suite can-utils : (1508687476.438095) slcan0 2C6#FFFFFFFF0

Les composants de chaque ligne sont :

- ✓ Horodatage : Temps en secondes depuis l'époque Unix (par exemple, 1508687476.438
- ✓ Interface : Interface réseau virtuelle (par exemple, slcan0).
- ✓ Identifiant CAN : ID de la trame (par exemple, 2C6), déterminant la priorité et le type de message.
- ✓ Charge utile : Données brutes en hexadécimal (par exemple, FFFFFFFF0).

Ce fichier représente un trafic CAN légitime, sans altération, et sert de base pour l'injection d'attaques simulées.

2.6.3 Étapes de traitement des données

Le traitement des données a été organisé en plusieurs étapes, détaillées cidessous. Chaque étape a été implémentée via des fonctions Python dans le notebook Jupyter, avec un suivi de la progression grâce à la bibliothèque tqdm.

2.6.3.1 Lecture et parsing du fichier de log

La première étape consiste à lire le fichier full_data_capture.log et à extraire les informations pertinentes pour chaque trame CAN. Une fonction Python, convert_can_log_to_csv, a été développée pour :

- ✓ Vérifier l'existence du fichier d'entrée.
- ✓ Créer le répertoire de sortie si nécessaire.
- ✓ Lire le fichier ligne par ligne, en ignorant les lignes non valides (par exemple, celles ne commençant pas par une parenthèse).
- ✓ Parser chaque ligne pour extraire l'horodatage, l'interface, l'identifiant CAN et la charge utile.
- ✓ Stocker les données dans une liste de dictionnaires, puis convertir cette liste en un DataFrame pandas. Le code suivant illustre le parsing d'une ligne :

```
timestamp_end = line.find(' )' )
timestamp = float(line[1:timestamp_end])
remaining = line[timestamp_end+1:].strip()
parts = remaining.split(maxsplit=1)
interface = parts[0]
can_data = parts[1]
can_id, payload = can_data.split(' #' , 1)
```

Résultat : Un DataFrame initial contenant 386 567 messages, avec les colonnes Timestamp, Interface, CAN_ID et Payload. Un contrôle a été effectué pour vérifier le nombre de messages avec l'ID 2C6 (13 752 messages), essentiel pour l'injection de l'attaque Suspension

2.6.3.2 Injection des attaques simulées

Pour créer un jeu de données représentatif des scénarios malveillants, trois types d'attaques ont été simulés : DoS, Fuzzing et Suspension. Chaque attaque a été injectée dans des périodes temporelles distinctes pour éviter les chevauchements.

Attaque DoS : L'attaque par Déni de Service (DoS) consiste à saturer le bus CAN en injectant des messages avec un identifiant prioritaire (000) à haute fréquence. Les paramètres de l'attaque sont :

- ✓ Période : De 1508687520.000000 à 1508687529.999750 (10 secondes).
- ✓ Intervalle : 0,00025 seconde (4 messages par milliseconde).
- ✓ Nombre de messages : 40 000.
- ✓ Charge utile : 0000000000000000.

Le processus d'injection comprend :

1. Suppression des messages originaux dans la période d'attaque pour éviter les interférences.
2. Création d'un DataFrame dos_df avec les messages DoS, générés via np.arange pour les horodatages.
3. Concaténation de dos_df avec le DataFrame principal.

Résultat : 40 000 messages DoS injectés, augmentant la taille du DataFrame.

Attaque Fuzzing : L'attaque Fuzzing modifie la charge utile des messages avec l'ID 18A pour simuler des données anormales. Les paramètres sont :

- ✓ Période : De 1508687510.000000 à 1508687515.999500 (6 secondes).
- ✓ Nombre de messages : 2 222.
- ✓ Charge utile : Valeurs hexadécimales aléatoires de 16 caractères, générées via la fonction generate_random_payload.

Le processus d'injection comprend :

1. Sélection des messages avec CAN_ID = '18A' dans la période spécifiée (600 messages disponibles).

2. Ajout de 1 622 messages synthétiques pour atteindre 2 222 messages, avec des horodatages générés via np.linspace.
3. Concaténation des messages synthétiques au DataFrame principal.

Résultat : 2 222 messages Fuzzing injectés, avec des charges utiles aléatoires.

Attaque Suspension : L'attaque Suspension supprime les messages avec l'ID 2C6 pendant une période prolongée pour simuler une interruption. Les paramètres sont :

- ✓ Période : De 1508687486.000000 à 1508687506.000000 (20 secondes).
- ✓ Messages supprimés : 1 000 messages 2C6.

Le processus comprend :

1. Suppression des messages 2C6 dans la période spécifiée.
2. Vérification post-suppression : 413 121 messages restants dans le DataFrame.

Pour étiqueter 2 222 messages comme Suspension, 75 messages synthétiques ont été ajoutés après la période de suspension, avec des paramètres simulant un grand intervalle d'arrivée (CAN_ID_Inter_Arrival = 10.0).

2.6.3.3 Calcul des caractéristiques

Après l'injection des attaques, des caractéristiques ont été calculées pour capturer les comportements normaux et anormaux. Les caractéristiques incluent :

- ✓ ***CAN_ID_Inter_Arrival*** : Temps entre deux trames consécutives d'un même ID, calculé via df.groupby('CAN_ID')['Timestamp'].diff(). Valeur par défaut pour la première trame : 0,010 seconde.
- ✓ ***CAN_ID_Window_Count*** : Nombre de messages par ID dans une fenêtre glissante de 5 secondes, calculé en utilisant une fenêtre temporelle avec pandas.rolling.
- ✓ ***Payload_Entropy*** : Entropie de la charge utile, mesurant l'aléatoire des données. Calculée en convertissant la charge utile en octets et en utilisant scipy.stats.entropy.
- ✓ ***Payload_Decimal*** : Valeur entière de la charge utile (hexadécimal converti en décimal).
- ✓ ***Norm_Payload_Decimal*** : Payload_Decimal normalisé par la valeur maximale possible ($2^{64} - 1$).
- ✓ ***Norm_Payload_Entropy*** : Payload_Entropy normalisé par la valeur maximale théorique (8,0).

- ✓ ***Suspension_Indicator*** : Indicateur de suspension, calculé en comptant les IDs avec des intervalles d'arrivée supérieurs à 2 secondes dans une fenêtre de 0,5 seconde, normalisé par le nombre total d'IDs.

Exemple de calcul de l'entropie :

```
def compute_payload_entropy(payload):
  if not payload:
     return 0.0
  bytes_array = [int(payload[i:i+2], 16) for i in range(0, len(payload)value_counts =
pd.Series(bytes_array).value_counts()
  probs = value_counts / len(bytes_array)
  return entropy(probs, base=2)
```

2.6.3.4 Normalisation des caractéristiques

Pour faciliter l'entraînement des modèles, les caractéristiques CAN_ID_Inter_Arrival et CAN_ID_Window_Count ont été normalisées par rapport aux statistiques des données normales (excluant les périodes d'attaque). Une fonction normalize_features a été utilisée pour diviser chaque valeur par la moyenne correspondante pour chaque CAN_ID.

2.6.3.5 Étiquetage des trames

Chaque trame a été étiquetée selon son type :

- ✓ 0 : Normal (370 916 messages).
- ✓ 1 : DoS (39 983 messages, CAN_ID = '000', période DoS).
- ✓ 2 : Fuzzing (2 222 messages, CAN_ID = '18A', période Fuzzing).
- ✓ 3 : Suspension (2 222 messages, CAN_ID = '2C6', post-période Suspension, incluant 75 messages synthétiques).

L'étiquetage a été effectué en utilisant des conditions basées sur l'horodatage et l'ID CAN.

2.6.3.6 Exportation vers CSV

Le DataFrame final, contenant 413 196 messages et 14 colonnes, a été exporté dans generated.csv. Les colonnes incluent :

Timestamp, Interface, CAN_ID, Payload, CAN_ID_Inter_Arrival,
CAN_ID_WindoPayload_Entropy, Norm_Inter_Arrival, Norm_Window_Count,
Norm_Payload_EntrNorm_Payload_Decimal, Suspension_Indicator, Label

L'écriture a été réalisée avec csv.writer, en ajoutant des guillemets pour les champs texte et en formatant les nombres à 5 décimales pour les flottants.

2.6.4 Résultats et observations

Le traitement a généré un jeu de données structuré et enrichi, prêt pour l'entraînement de modèles de classification. Les principales observations sont :

- ✓ La distribution des étiquettes montre une majorité de messages normaux (370 916), avec des attaques bien représentées (DoS : 39 983, Fuzzing : 2 222, Suspension : 2 222).
- ✓ Les caractéristiques calculées capturent efficacement les anomalies :
- ✓ CAN_ID_Inter_Arrival atteint 10 secondes pour les messages Suspension synthétiques, contre 0,0407 seconde en moyenne pour les normaux.
- ✓ CAN_ID_Window_Count atteint 19 992 pour les messages DoS, contre 277,84 pour les normaux.
- ✓ Payload_Entropy est proche de 3 pour les messages Fuzzing (aléatoires), contre 1,91 pour les normaux.
- ✓ Les messages synthétiques ajoutés pour les attaques Fuzzing et Suspension garantissent un nombre suffisant de cas malveillants pour l'entraînement.

2.7 Conclusion

Ce chapitre a détaillé les étapes de préparation et d'inspection des données, essentielles pour la construction d'un système de détection d'intrusions basé sur les logs CAN. Nous avons commencé par l'acquisition des données brutes via un 5 adaptateur CAN-to-USB, suivi par la simulation d'attaques réalistes (DoS, Fuzzing, Suspension) pour enrichir le jeu de données. L'organisation du projet, centrée sur Visual Studio Code et Python, a permis une gestion modulaire et efficace du code

La préparation des données brutes, décrite dans la section 0.1, a transformé un fichier de log brut en un jeu de données structuré, contenant 413 196 messages avec des caractéristiques avancées et des étiquettes précises. Les caractéristiques calculées, telles que CAN_ID_Inter_Arrival, CAN_ID_Window_Count et Payload_Entropy, ont prouvé leur capacité à distinguer les comportements normaux des comportements malveillants. L'injection d'attaques simulées a permis de créer un dataset équilibré, bien que certaines limitations, comme le recours à des messages synthétiques, aient été notées.

En conclusion, ce chapitre a jeté les bases techniques pour la phase de modélisation, en fournissant un jeu de données robuste et bien préparé. Les prochaines étapes, abordées dans le chapitre suivant, consisteront à entraîner et évaluer des modèles de machine learning, notamment XGBoost, pour détecter automatiquement les anomalies dans le trafic CAN.

Chapitre 3 : Entraînement et évaluation du modèle de détection

3.1 Introduction

Ce chapitre est dédié à l'entraînement et à l'évaluation d'un modèle de détection d'intrusions pour le trafic CAN, en s'appuyant sur le jeu de données préparé dans le chapitre précédent generated.csv. L'objectif est de développer un système capable de classifier les trames CAN comme normales ou malveillantes (DoS, Fuzzing, Suspension) avec une précision élevée, en utilisant des techniques d'apprentissage automatique. Nous nous concentrons particulièrement sur l'algorithme XGBoost, reconnu pour ses performances dans les tâches de classification sur des données tabulaires.

3.2 Machine Learning

Le machine learning est un domaine captivant. Issu de nombreuses disciplines comme les statistiques, l'optimisation, l'algorithmique ou le traitement du signal, c'est un champ d'études en mutation constante qui s'est maintenant imposé dans notre société. Déjà utilisé depuis des décennies dans la reconnaissance automatique de caractères ou les filtres anti-spam, il sert maintenant à protéger contre la fraude bancaire, recommander des livres, films, ou autres produits adaptés à nos goûts, identifier les visages dans le viseur de notre appareil photo, ou traduire automatiquement des textes d'une langue vers une autre.

Dans les années à venir, le machine learning nous permettra vraisemblablement d'améliorer la sécurité routière (y compris grâce aux véhicules autonomes), la réponse d'urgence aux catastrophes naturelles, le développement de nouveaux médicaments, ou l'efficacité énergétique de nos bâtiments et industries.

3.2.1 Qu'est-ce que le machine learning ?

Le machine learning peut servir à résoudre des problèmes :

- ✓ que l'on ne sait pas résoudre (comme dans l'exemple de la prédiction d'achats ci-dessus);
- ✓ que l'on sait résoudre, mais dont on ne sait formaliser en termes algorithmiques comment nous les résolvons(c'estle cas par exemple de la reconnaissance d'images ou de la compréhension du langage naturel);
- ✓ que l'on sait résoudre, mais avec des procédures beaucoup trop gourmandes en ressources informatiques (c'est le cas par exemple de la prédiction d'interactions entre molécules de grande taille, pour lesquelles les simulations sont très lourdes).

Le machine learning est donc utilisé quand les *données* sont abondantes (relativement), mais les *connaissances* peu accessibles ou peu développées.

Ainsi, le machine learning peut aussi aider les humains à apprendre : les modèles créés par des algorithmes d'apprentissage peuvent révéler l'importance relative de certaines informations ou la façon dont elles interagissent entre elles pour résoudre un problème particulier. Dans l'exemple de la prédiction d'achats, comprendre le modèle peut nous permettre d'analyser quelles caractéristiques des achats passés permettent de prédire ceux à venir. Cet aspect du machine learning est très utilisé dans la recherche scientifique : quels gènes sont impliqués dans le développement d'un certain type de tumeur, et comment? Quelles régions d'une image cérébrale permettent de prédire un comportement? Quelles caractéristiques d'une molécule en font un bon médicament pour une indication particulière? Quels aspects d'une image de télescope permettent d'y identifier un objet astronomique particulier?

Ingrédients du machine learning

Le machine learning repose sur deux piliers fondamentaux :

- ✓ d'une part, les *données*, qui sont les exemples à partir duquel l'algorithme va apprendre;
- ✓ d'autre part, l'*algorithme d'apprentissage*, qui est la procédure que l'on fait tourner sur ces données pour produire un modèle. On appelle *entraînement* le fait de faire tourner un algorithme d'apprentissage sur un jeu de données.

Ces deux piliers sont aussi importants l'un que l'autre. D'une part, aucun algorithme d'apprentissage ne pourra créer un bon modèle à partir de données qui ne sont pas pertinentes – c'est le concept *garbage in, garbage out* qui stipule qu'un algorithme d'apprentissage auquel on fournit des données de mauvaise qualité ne pourra rien en faire d'autre que des prédictions de mauvaise qualité. D'autre part, un modèle appris avec un algorithme inadapté sur des données pertinentes ne pourra pas être de bonne qualité.

Cet ouvrage est consacré au deuxième de ces piliers – les algorithmes d'apprentissage. Néanmoins, il ne faut pas négliger qu'une part importante du travail de *machine learner* ou de *data scientist* est un travail d'ingénierie consistant à préparer les données afin d'éliminer les données aberrantes, gérer les données manquantes, choisir une représentation pertinente, etc.

Et l'intelligence artificielle, dans tout ça?

Le machine learning peut être vu comme une branche de l'intelligence artificielle. En effet, un système incapable d'apprendre peut difficilement être considéré comme intelligent. La capacité à apprendre et à tirer parti de ses expériences est en effet essentielle à un système conçu pour s'adapter à un environnement changeant.

L'intelligence artificielle, définie comme l'ensemble des techniques mises en œuvre afin de construire des machines capables de faire preuve d'un comportement que l'on peut qualifier d'intelligent, fait aussi appel aux sciences cognitives, à la neurobiologie, à la logique, à l'électronique, à l'ingénierie et bien plus encore.

Le machine learning est un champ assez vaste, et nous dressons dans cette section une liste des plus grandes classes de problèmes auxquels il s'intéresse.

3.2.2 Apprentissage supervisé

L'apprentissage supervisé est peut-être le type de problèmes de machine learning le plus facile à appréhender : son but est d'apprendre à faire des prédictions, à partir d'une liste d'exemples étiquetés, c'est-à-dire accompagnés de la valeur à prédire .Les étiquettes servent de « professeur » et supervisent l'apprentissage de l'algorithme.

Définition (Apprentissage supervisé) : On appelle apprentissage supervisé la branche du machine learning qui s'intéresse aux problèmes pouvant être formalisés de la façon suivante : étant données n observations $\{\sim x^i\}_{i=1,\dots,n}$ décrites dans un espace X , et leurs étiquettes $\{y^i\}_{i=1,\dots,n}$ décrites dans un espace Y, on suppose que les étiquettes peuvent être obtenues à partir des

observations grâce à une fonction φ : X → Y fixe et inconnue : $y^i = \phi(\vec{x}^i) + \epsilon_i$ ϵ_i , où ϵ_i est un bruit aléatoire. Il s'agit alors d'utiliser les données pour déterminer une fonction f : X → Y telle que, pour tout couple $(\vec{x}, \phi(\vec{x})) \in \mathcal{X} \times \mathcal{Y}, f(\vec{x}) \approx \phi(\vec{x})$.

L'espace sur lequel sont définies les données est le plus souvent $X = R^p$. Nous verrons cependant aussi comment traiter d'autres types de représentations, comme des variables binaires, discrètes, catégoriques, voire des chaînes de caractères ou des graphes.

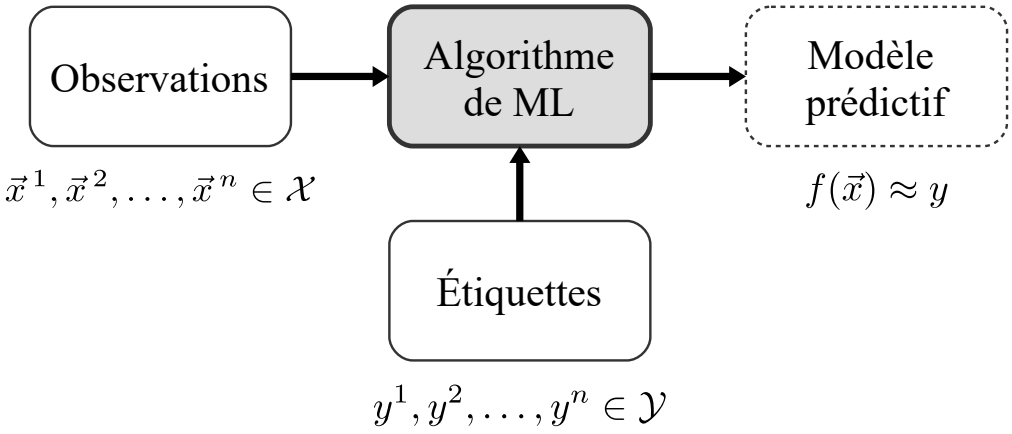

Figure 3.1 - Apprentissage supervisé

3.2.3 Apprentissage non supervisé

Dans le cadre de l'apprentissage *non supervisé*, les données ne sont pas étiquetées. Il s'agit alors de mo-déliser les observations pour mieux les comprendre

Définition (Apprentissage non supervisé) : On appelle apprentissage non supervisé la branche du machine learning qui s'intéresse aux problèmes pouvant être formalisés de la façon suivante : étant données n observations {~x i}i=1,...,n décrites dans un espace X, il s'agit d'apprendre une fonction sur X qui vérifie certaines propriétés.

Figure 3.2 - Apprentissage non supervisé

3.2.4 Apprentissage semi-supervisé

Comme on peut s'en douter, l'apprentissage semi-supervisé consiste à apprendre des étiquettes à partir d'un jeu de données partiellement étiqueté. Le premier avantage de cette approche est qu'elle permet d'éviter d'avoir à étiqueter l'intégralité des exemples d'apprentissage, ce qui est pertinent quand il est facile d'accumuler des données mais que leur étiquetage requiert une certaine quantité de travail humain. Prenons par exemple la classification d'images : il est facile d'obtenir une banque de données contenant des centaines de milliers d'images, mais avoir pour chacune d'entre elles l'étiquette qui nous intéresse peut requérir énormément de travail. De plus, les étiquettes données par des humains sont susceptibles de reproduire des biais humains, qu'un algorithme entièrement supervisé reproduira à son tour. L'apprentissage semi-supervisé permet parfois d'éviter cet écueil. Il s'agit d'un sujet plus avancé, que nous ne considérerons pas dans cet ouvrage.

3.2.5 Apprentissage par renforcement

Dans le cadre de l'*apprentissage par renforcement*, le système d'apprentissage peut interagir avec son environnement et accomplir des actions. En retour de ces actions, il obtient une *récompense*, qui peut être positive si l'action était un bon choix, ou négative dans le cas contraire. La récompense peut parfois venir après une longue suite d'actions; c'est le cas par exemple pour un système apprenant à jouer au go ou aux échecs. Ainsi, l'apprentissage consiste dans ce cas à définir une *politique*, c'est-à-dire une stratégie permettant d'obtenir systématiquement la meilleure récompense possible.

Les applications principales de l'apprentissage par renforcement se trouvent dans les jeux (échecs, go, etc) et la robotique. Ce sujet dépasse largement le cadre de cet ouvrage.

3.3 Arbres de décision en Machine Learning

L'arbre de décision est l'un des premiers algorithmes de Machine Learning que les Data Scientists apprennent au cours de leur formation. Il est utilisé pour représenter visuellement et explicitement les décisions et la prise de décision pour des problèmes de classification ainsi que pour des problèmes de régression. Il représente aussi l'élément de base de plusieurs modèles comme le Random Forest ou XGBoost.

Par conséquent, il est important de comprendre les concepts et les algorithmes qui se cachent derrière les arbres de décision. Dans cet article, nous allons détailler le principe de

fonctionnement de l'arbre de décision et l'algorithme CART, responsable de sa construction à partir de données.

3.3.1 Principe de fonctionnement de l'arbre de décision

Tout d'abord, qu'est-ce qu'un arbre de décision ? Visuellement, cela ressemble à une structure descendante composée de noeuds : chaque noeud possède une condition qui amène à plusieurs réponses, ce qui dirige à un prochain noeud.

Lorsqu'un nœud donne la réponse, on dit que le nœud est terminal. Prenons l'arbre de décision suivant qui indique si l'on doit prendre un parapluie avec nous en fonction du temps.

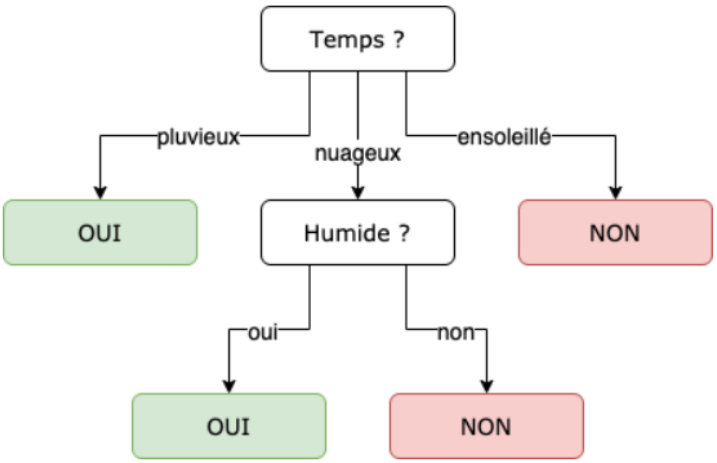

Figure 3.3 - Exemple d'une arbre de décision

Dans cet exemple, un jour ensoleillé donnera directement la réponse **NON**, alors qu'un jour nuageux donnera la réponse **OUI** ou **NON** en fonction de l'humidité.

Sur ce graphique, chaque nœud peut avoir aucune ou plusieurs possibilités: cela va dépendre de s'il est terminal ou non.

Un **arbre de décision binaire** est un arbre où chaque nœud non terminal possède **exactement deux possibilités** (gauche et droite). Prenons l'arbre suivant qui indiqu

la mention obtenue à un examen pour un étudiant, et s'il a le droit à des rattrapages en cas de note inférieure à 10.

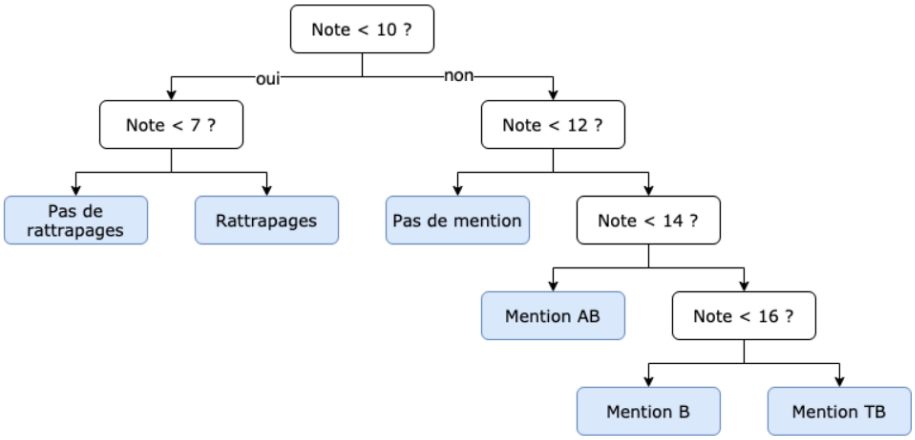

Figure 3.4 - Arbre de décision binaire

Plusieurs points sont à noter :

- ✓ Le premier noeud est appelé **noeud racine** et possède toujours exactement deux **noeuds enfants**.
- ✓ Chaque noeud non terminal **possède toujours deux noeuds enfants**.
- ✓ Les conditions de noeuds ne peuvent avoir que deux états : **Vrai** ou **Faux**.
- ✓ Le noeud enfant de gauche correspond toujours (dans cet arbre) à la situation où la condition est vérifiée (Vrai), et inversement le noeud enfant de droite correspond toujours à la situation où la condition n'est pas vérifiée (Faux).

3.3.2 Arbre de classification

Dans cette situation, la prédiction n'est plus une quantité mais une classe.

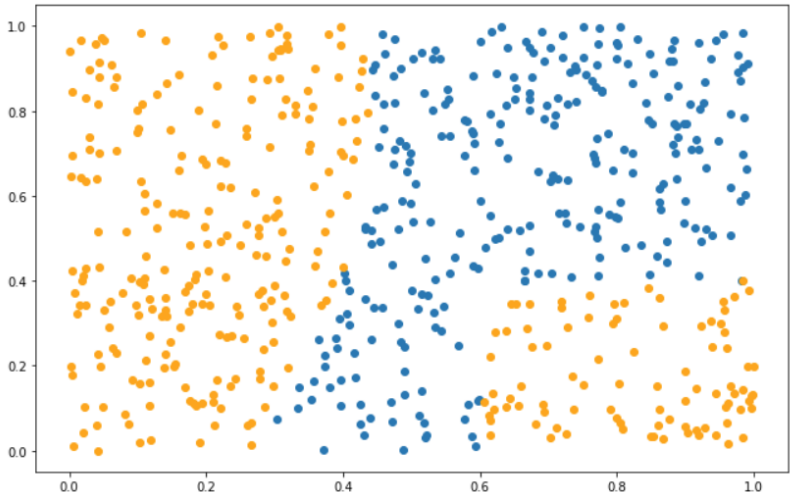

Figure 3.5 - Deux classes : orange et bleu

Deux classes sont présentes : orange et bleu. L'arbre de décision va subdiviser le plan sous forme de rectangle, et l'objectif de CART est de construire l'arbre de sorte qu'il y ait **le plus d'homogénéité possible** dans chaque rectangle. C'est justement ici que **l'impureté de Gini** intervient.

L'impureté de Gini est le produit d'une probabilité p avec son inverse : elle vaut $p\,(1-p)$. Elle permet de calculer très facilement l'homogénéité pour une classification binaire.

Notons p la proportion d'observations dont y vaut 1 .

- S'il n'y a que des points bleus (y vaut 0), alors $p = 0$, $1 - p = 1$ et donc $p\,(1-p) = 0$: il y a une **parfaite homogénéité**.
- S'il n'y a que des points oranges (y vaut 1), alors $p = 1$, $1 - p = 0$ et donc $p\,(1-p) = 0$: il y a là-aussi une **parfaite homogénéité**.
- À l'inverse, s'il y a autant de points bleus que de points oranges, alors il y a 50% de y à 0 et 50% de y à 1. Ainsi, $p = 0.5$ et $1 - p = 0.5$ donc $p\,(1-p) = 0.25$: il y a une **pleine hétérogénéité**.

On cherche à se rapprocher au maximum de 0 qui correspond à une situation où aucune couleur n'est mélangée. En appliquant successivement ce calcul sur chacun des nœuds, l'arbre de décision prend forme en fonction des différents α choisis.

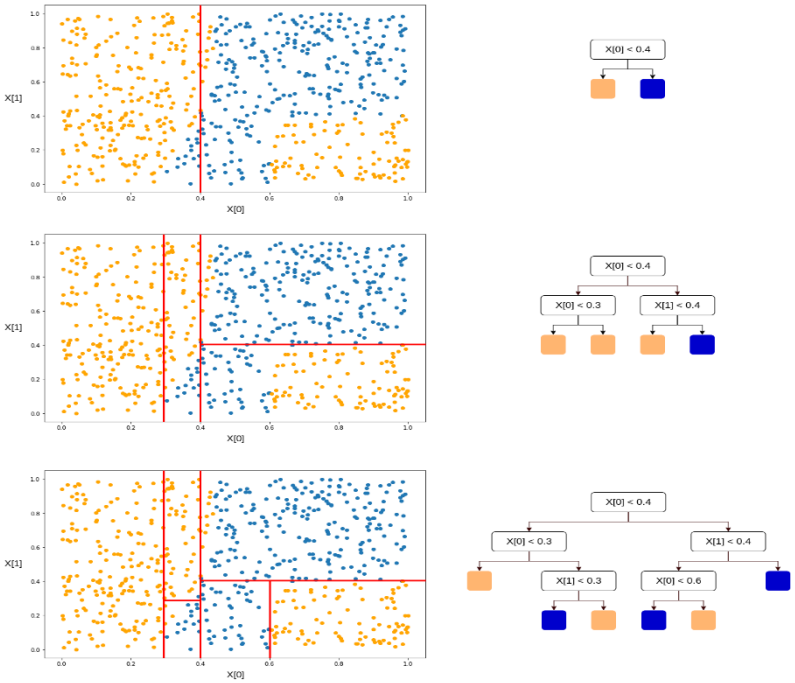

Figure 3.6 - Minimisation du chevauchement des classes par des divisions itératives d'arbres de décision

3.3.3 Le sur-apprentissage

Le risque de **sur-apprentissage** (créer un arbre avec une très grande profondeur) est très élevé pour les modèles non-paramétriques, dont fait partie l'arbre de décision.

Si l'arbre décide d'effectuer une subdivision sur un seul point, on est dans le cas de sur-apprentissage, car ce point est très éloigné de l'ensemble des autres points, c'est un point **extrême**. On obtiendra des résultats très différents de la réalité si ce type de points est pris considération par le modèle.

Pour détecter le sur-apprentissage, il faut diviser le jeu de données en deux parties.

- ✓ Une partie majoritaire appelée **ensemble d'entrainement** (*training set*) : c'est l'échantillon sur lequel notre modèle va s'entraîner.
- ✓ Une partie appelé **ensemble de test** (*test set*) : c'est un échantillon que notre modèle ne connaît pas.

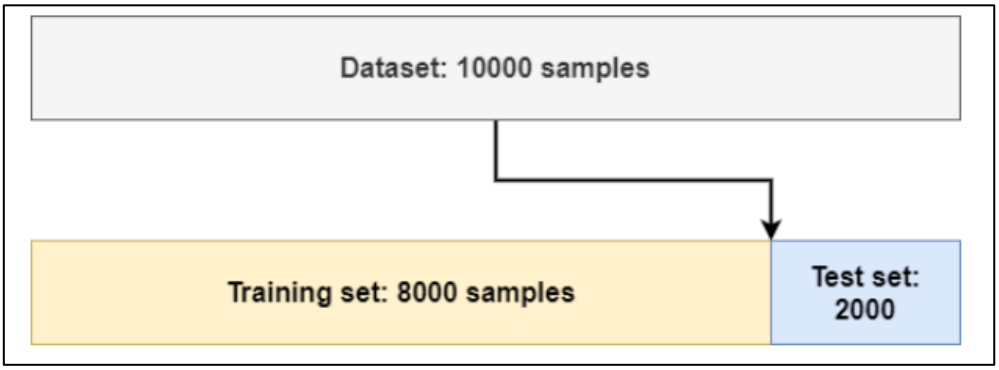

Figure 3.7 - Dévision du jeu de données

Le deuxième ensemble va permettre d'évaluer la performance du modèle en calculant le pourcentage des prédictions correctes.

Pour limiter le sur-apprentissage, le **choix d'hyper-paramètres est important**. Il permettent au Data Scientist de contrôler son modèle et ils ne varient pas lors de l'entraînement.

Ces hyper-paramètres limiteront le sur-apprentissage dans les arbres de décision une fois optimisés.

- ✓ La **profondeur maximale** (max_depth).
- ✓ Le **gain minimum de performances** à obtenir lors du passage à une nouvelle profondeur (min_impurity_decrease).

- ✓ Le **nombre d'observations minimal dans un nœud** pour effectuer un split (min_samples_split) ou dont un nœud enfant doit avoir (min_samples_leaf).

3.4 XGBoost (eXtreme Gradient Boosting)

XGBoost (eXtreme Gradient Boosting) est un algorithme de *boosting* basé sur des arbres de décision, optimisé pour la vitesse et la performance. Il combine des techniques de *gradient boosting*, de régularisation et de traitement parallèle pour minimiser les erreurs tout en évitant le surajustement (*overfitting*).

3.4.1 Fondements du Gradient Boosting

Le principe central est l'**apprentissage séquentiel** :

- Chaque nouveau modèle (arbre) corrige les erreurs résiduelles du modèle précédent.
- La prédiction finale est une **somme pondérée** des prédictions de tous les arbres :

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), \quad f_k \in \mathcal{F}$$

où *K* est le nombre d'arbres, *fk* un arbre individuel, et *F* l'espace des fonctions.

3.4.2 Fonction Objectif (Objective Function)

XGBoost optimise une fonction objectif **régularisée** :

$$\text{Obj} = \sum_{i=1}^{n} L(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

- **Partie 1 : Fonction de Perte** $L(y_i, \hat{y}_i)$: Mesure l'erreur entre la prédiction $\hat{y}_i$ et la vraie valeur *yi* (ex: erreur quadratique, log-loss).
- **Partie 2 : Terme de Régularisation Ω(*fk*)** : Contrôle la complexité du modèle pour éviter l'*overfitting* :

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda |||w|||^2$$

où *T* = nombre de feuilles, *w* = scores des feuilles, γ et λ = hyperparamètres.

3.4.3 Minimisation par Approximation de Taylor

À l'itération *t*, on cherche l'arbre *ft* qui minimise :

$$\text{Obj}^{(t)} = \sum_{i=1}^{n} \left[L(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t)$$

avec :

- $g_i = \partial_{\hat{y}^{(t-1)}} L(y_i, \hat{y}^{(t-1)})$ (gradient)
- $h_i = \partial^2_{\hat{y}^{(t-1)}} L(y_i, \hat{y}^{(t-1)})$ (hessienne).

Simplification :

En ignorant les termes constants, on obtient :

$$\text{Obj}^{(t)} \approx \sum_{i=1}^{n} \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t)$$

3.4.4 Formulation par Feuille (Leaf-based Formulation)

Soit $I_j = \{i | x_i \in \text{feuille } j\}$ (exemples dans la feuile *j*) :

✓ La fonction objectif se réécrit :

$$\text{Obj}^{(t)} = \sum_{j=1}^{T} \left[\left(\sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] + \gamma T$$

✓ Le **score optimal** ***wj∗*** de la feuille *j* est :

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}$$

✓ La valeur minimale de l'objectif est :

$$\text{Obj}^* = -\frac{1}{2} \sum_{j=1}^{T} \frac{\left(\sum_{i \in I_j} g_i \right)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma T$$

3.4.5 Algorithme de Fractionnement (Split Finding)

Pour chaque nœud, XGBoost maximise le gain après un split :

$$\text{Gain} = \frac{1}{2}\left[\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{(\sum_{i\in I_R} g_i)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}\right] - \gamma$$

- I_L, I_R = exemples à gauche/droite du split, $I = I_L \cup I_R$.
- Le split est effectué si $\text{Gain} > 0$.

Optimisation :

- ✓ Utilise un **histogramme des gradients** pour évaluer les splits rapidement.
- ✓ Supporte les **valeurs manquantes** en les dirigeant vers la branche maximisant le gain.

3.4.6 Optimisations Clés

Parmi les optimisations clés d'algorithm XGBoost on trouve :

- ✓ Traitement Parallèle : Construction des histogrammes en parallèle pour chaque caractéristique.
- ✓ Sous-échantillonnage (Stochastic Boosting) : Utilise une fraction aléatoire des données/features par itération.
- ✓ Pénalités de Complexité : Contrôle de la profondeur ($\gamma\gamma$), régularisation $L2L2$ ($\lambda\lambda$).

3.4.7 Conclusion

XGBoost généralise le gradient boosting en intégrant :

- ✓ Une **formulation mathématique régularisée** via l'approximation de Taylor.
- ✓ Des **optimisations calculatoires** (parallélisme, gestion des données manquantes).
- ✓ Un **contrôle fin de la complexité** ($\gamma,\lambda\gamma,\lambda$). Cela en fait un outil robuste pour des prédictions précises et efficaces, même sur de grands jeux de données.

3.5 Entraînement et Évaluation du Modèle XGBoost

Cette section détaille le processus d'entraînement du modèle XGBoost pour la détection des attaques sur le bus CAN, ainsi que l'évaluation de ses performances sur un ensemble de test étiqueté et la prédiction sur des données non étiquetées. L'objectif est de construire un modèle capable de classer les trames CAN en quatre catégories : normale (0), attaque par déni

de service (DoS, 1), attaque par fuzzing (2) et attaque par suspension (3). Le processus est implémenté dans un notebook Jupyter nommé model_training.ipynb, utilisant Python 3.13.3 et des bibliothèques comme pandas, NumPy, scikit-learn, et XGBoost.

3.5.1 Préparation des Données et Configuration

Avant l'entraînement, les données étiquetées sont chargées à partir du fichier generated.csv, qui contient 413 196 trames CAN avec 14 colonnes, incluant les caractéristiques extraites (par exemple, CAN_ID_Inter_Arrival, CAN_ID_Window_Count, Payload_Entropy) et les étiquettes. La distribution des classes est déséquilibrée, avec 370 916 trames normales, 39 983 DoS, 2 222 fuzzing et seulement 75 suspension, ce qui nécessite une gestion spécifique pour éviter un biais vers la classe majoritaire.

Les caractéristiques utilisées pour l'entraînement sont :

- CAN_ID_Inter_Arrival : Intervalle de temps entre deux trames consécutives d'un même ID.
- CAN_ID_Window_Count : Nombre de trames par ID dans une fenêtre de 5 secondes.
- Payload_Entropy : Entropie de la charge utile, mesurant son aléatoire.
- Norm_Inter_Arrival : Intervalle normalisé par la moyenne des données normales.
- Norm_Window_Count : Compteur de fenêtre normalisé.
- Norm_Payload_Entropy : Entropie normalisée.
- Norm_Payload_Decimal : Valeur décimale normalisée de la charge utile.
- Suspension_Indicator : Indicateur des interruptions potentielles.

Les données sont divisées en trois ensembles : entraînement (70 %, 289 226 trames), validation (15 %, 61 990 trames) et test (15 %, 61 980 trames), en utilisant une stratification pour préserver la distribution des classes. Pour gérer le déséquilibre, des poids d'échantillons sont calculés selon la formule :

$$w_i = \frac{N}{k \cdot N_i},$$

où N est le nombre total d'échantillons d'entraînement, k le nombre de classes (4), et Ni le nombre d'échantillons de la classe i. Ces poids sont appliqués lors de l'entraînement pour donner plus d'importance aux classes minoritaires, comme la suspension.

3.5.2 Entraînement du Modèle XGBoost

Le modèle choisi est XGBoost, un algorithme d'apprentissage par boosting d'arbres de décision, réputé pour sa performance sur les données tabulaires et sa capacité à gérer les classes déséquilibrées. Les hyperparamètres sont configurés comme suit :

- **Objective** : multi:softprob, pour une classification multiclasse avec probabilités.
- **Number of classes** : 4 (Normal, DoS, Fuzzing, Suspension).
- **Estimators** : 200 arbres, pour une complexité suffisante sans sur-apprentissage excessif.
- **Max depth** : 6, limitant la profondeur des arbres pour contrôler la complexité.
- **Learning rate** : 0,1, pour une mise à jour progressive des poids.
- **Evaluation metric** : mlogloss, pour minimiser la perte logarithmique multiclasse.
- **Random state** : 42, pour la reproductibilité

Le modèle est entraîné sur l'ensemble d'entraînement avec les poids d'échantillons, et sauvegardé au format JSON (xgboost_model.json). Les statistiques des données normales sont également calculées pour normaliser les caractéristiques des données non étiquetées ultérieurement.

3.5.3 Prédiction sur Données Non Étiquetées

Les données non étiquetées, issues du fichier dosattack.log (141 927 trames), sont traitées pour extraire les mêmes caractéristiques que celles utilisées pour l'entraînement. Les fonctions de calcul des caractéristiques, définies dans can_bus_processisont réutilisées pour garantir la cohérence. Ces fonctions incluent :

- compute_payload_entropy : Calcule l'entropie de la charge utile.
- compute_window_count : Compte les trames par ID dans une fenêtre temporelle.
- compute_suspension_indicator : Identifie les interruptions potentielles.
- process_unlabeled_data : Orchestre le traitement complet, incluant la normalisation.

Les prédictions sont effectuées avec le modèle entraîné, et les résultats sont sauvegardés dans unlabeled_predictions.csv. Sur les 141 927 trames, 101 926 sont classées comme normales et 40 001 comme DoS, sans détection de fuzzing ni de suspension. Les distributions des caractéristiques par classe prédite confirment les attentes :

- Normal : CAN_ID_Inter_Arrival moyen de 0,044 s, Payload_Entropy variable (moyen 1,88), Suspension_Indicator proche de 0.
- DoS : CAN_ID_Inter_Arrival très faible (0,00025 s), CAN_ID_Window_Count élevé (moyen 13 986), Payload_Entropy proche de 0.

3.6 Explication Détaillée du Code d'Entraînement et d'Évaluation

Cette section explique en détail le code implémenté dans le notebook Jupyter model_training.ipynb, qui gère l'entraînement du modèle XGBoost, l'évaluation sur des données étiquetées, et les prédictions sur des données non étiquetées pour la détection d'attaques sur le bus CAN. Le code est décomposé en parties clés, avec une explication de chaque étape, de son objectif, et de son fonctionnement technique.

3.6.1 Importation des Bibliothèques

```
import csv
from pathlib import Path
import sys
import os
import pandas as pd
import numpy as np
from scipy.stats import entropy
from tqdm import tqdm
import random
import string
```

Figure 3.8 - Importation des bibliothèques

Explication : Cette partie importe les bibliothèques nécessaires pour le projet. pandas et numpy gèrent les données tabulaires et les calculs numériques. scipy.stats.entropy calcule l'entropie des charges utiles. sklearn fournit des outils pour la division des données (train_test_split) et l'évaluation (classificationconfusion_matrix). xgboost est utilisé pour l'entraînement du modèle. tqdm affiche des barres de progression pour les boucles longues. matplotlib et seaborn génèrent des visualisations comme la matrice de confusion. random et string servent à générer des charges utiles aléatoires pour les attaques synthétiques. %matplotlib inline permet d'afficher les graphiques dans le notebook.

3.6.2 Définition des Fonctions de Calcul des Caractéristiques

```python
def generate_random_payload(length=16):
    return ''.join(random.choice(string.hexdigits.upper()) for _ in range(length))

def compute_payload_entropy(payload):
    if not payload:
        return 0.0
    bytes_array = [int(payload[i:i+2], 16) for i in range(0, len(payload), 2)]
    value_counts = pd.Series(bytes_array).value_counts()
    probs = value_counts / len(bytes_array)
    return entropy(probs, base=2)

def compute_payload_decimal(payload):
    try:
        return int(payload, 16)
    except ValueError:
        return 0
```

Figure 3.10 - Fonctions de Calcul des caractéristiques

```python
def compute_window_count(df, window_size=5.0):
    df = df.copy()
    df['Timestamp'] = pd.to_datetime(df['Timestamp'], unit='s')
    window_counts = []
    for can_id in tqdm(df['CAN_ID'].unique(), desc="Processing CAN_IDs"):
        can_id_df = df[df['CAN_ID'] == can_id][['Timestamp']].copy()
        can_id_df['Dummy'] = 1
        can_id_df.set_index('Timestamp', inplace=True)
        can_id_df['CAN_ID_Window_Count'] = (
            can_id_df['Dummy']
            .rolling(window=f'{window_size}s', closed='both')
            .count()
            .astype(int)
        )
        can_id_df.reset_index(inplace=True)
        can_id_df['CAN_ID'] = can_id
        window_counts.append(can_id_df[['Timestamp', 'CAN_ID', 'CAN_ID_Window_Count']])
    result = pd.concat(window_counts)
    result['Timestamp'] = result['Timestamp'].astype(int) / 10**9
    return result

def compute_suspension_indicator(df, threshold=2.0, window=0.5):
    df = df.sort_values('Timestamp')
    indicators = np.zeros(len(df))
    total_can_ids = len(df['CAN_ID'].unique())
    timestamps = df['Timestamp'].to_numpy()
    inter_arrivals = df['CAN_ID_Inter_Arrival'].to_numpy()
    can_ids = df['CAN_ID'].to_numpy()
    for i in tqdm(range(len(df)), desc="Processing messages"):
        timestamp = timestamps[i]
        mask = (timestamps >= timestamp - window) & (timestamps <= timestamp + window) & (inter_arrivals > threshold)
        affected_can_ids = len(np.unique(can_ids[mask]))
        indicators[i] = affected_can_ids / total_can_ids if total_can_ids > 0 else 0.0
    return indicators
```

Figure 3.9 - Fonctions de Calcul des caractéristiques

Explication :

- generate_random_payload : Génère une charge utile hexadécimale aléatoire de 16 caractères pour simuler des attaques de fuzzing.
- compute_payload_entropy : Calcule l'entropie de Shannon d'une charge utile en convertissant les paires hexadécimales en octets, puis en calculant les probabilités des

valeurs. Une entropie élevée indique une charge aléatoire (fuzzing), tandis qu'une entropie faible suggère une charge constante (DoS).

- compute_payload_decimal : Convertit la charge utile hexadécimale en valeur décimale, avec une gestion des erreurs pour les payloads invalides.
- compute_window_count : Calcule le nombre de trames par ID CAN dans une fenêtre temporelle de 5 secondes, en utilisant une fenêtre glissante (rolling). Cette fonction est essentielle pour détecter les attaques DoS, qui augmentent fortement ce compteur.
- compute_suspension_indicator : Identifie les interruptions (suspensions) en comptant les IDs CAN avec des intervalles anormaux (> 2 s) dans une fenêtre de 0,5 s, normalisé par le nombre total d'IDs.
- process_unlabeled_data : Orchestre le traitement des fichiers de log non étiquetés, en extrayant les trames, calculant les caractéristiques cidessus, et normalisant les intervalles et compteurs à l'aide des statistiques des données normales. Cette fonction garantit que les données non étiquetées sont cohérentes avec les données d'entraînement.

3.6.3 Entraînement du Modèle

```
# Initialize and train model
model = xgb.XGBClassifier(
    objective='multi:softprob',
    num_class=4,
    n_estimators=200,
    max_depth=6,
    learning_rate=0.1,
    eval_metric='mlogloss',
    random_state=42
)
print("\nTraining XGBoost model...")
model.fit(X_train, y_train, sample_weight=sample_weights, verbose=True)
# Save model
model.save_model(model_path)
print(f"Model saved to {model_path}")
# Compute normal statistics for normalization
normal_df = df[df['Label'] == 0][['CAN_ID', 'CAN_ID_Inter_Arrival', 'CAN_ID_Window_Count']]
normal_stats = normal_df.groupby('CAN_ID').agg({
    'CAN_ID_Inter_Arrival': 'mean',
    'CAN_ID_Window_Count': 'mean'
}).reset_index()
normal_stats.columns = ['CAN_ID', 'Mean_Inter_Arrival', 'Mean_Window_Count']
```

Figure 3.11 - Entraînement du modèle XGBoost

Explication :

- Chargement des données : Le fichier generated.csv est lu dans un DataFrame pandas, et la distribution des étiquettes est affichée pour vérifier le déséquilibre des classes.
- Sélection des caractéristiques : Les 8 caractéristiques clés sont sélectionnées pour former la matrice X, avec y comme vecteur des étiquettes.
- Division des données : Les données sont divisées en ensembles d'entraînement (70 %), de validation (15 %), et de test (15 %), en utilisant stratify=y pour préserver la distribution des classes.
- Poids des classes : Les poids sont calculés pour compenser le déséquilibre, en attribuant une importance plus élevée aux classes minoritaires (par exemple, Suspension).
- Configuration du modèle : Le classifieur XGBoost est initialisé avec des hyperparamètres fixes, optimisés pour une classification multiclasse avec probabilités.
- Entraînement : Le modèle est entraîné sur X_train et y_train, avec les poids appliqués via sample_weight.
- Sauvegarde : Le modèle est sauvegardé au format JSON pour une réutilisation future.
- Statistiques normales : Les moyennes des intervalles et compteurs pour les trames normales sont calculées par ID CAN, pour normaliser les données non étiquetées.

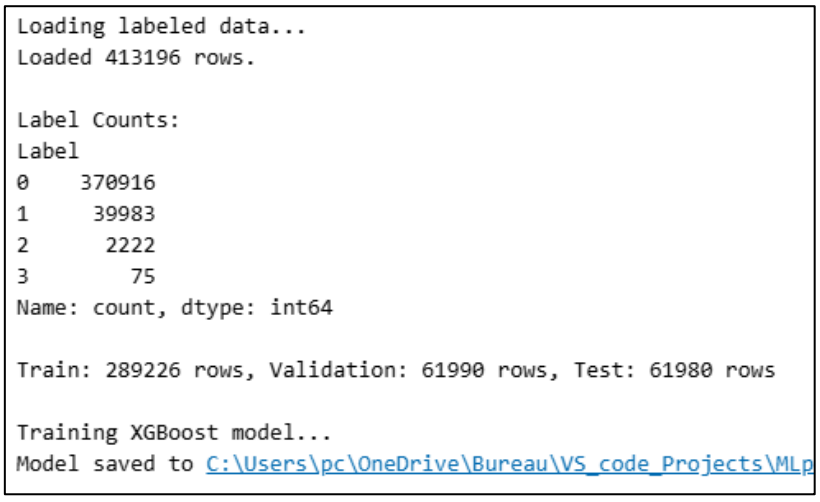

Figure 3.12 - Résultat de l'entrainement

Résumé des résultats :

- ✓ **Données chargées** : 413 196 trames CAN depuis generated.csv.
- ✓ **Répartition des classes** : Très déséquilibrée avec 370 916 normales (0), 39 983 DoS (1), 2 222 fuzzing (2), et 75 suspension (3).

- ✓ **Division des données** : 70 % entraînement (289 226 lignes), 15 % validation (61 990 lignes), 15 % test (61 980 lignes), stratifiée pour préserver les proportions.
- ✓ **Entraînement XGBoost** : Modèle entraîné avec 200 arbres, profondeur max 6, taux d'apprentissage 0,1, poids des classes pour gérer le déséquilibre, sauvegardé en xgboost_model.json.
- ✓ **Importance** : Entraînement réussi avec gestion du déséquilibre, mais faible nombre de suspensions peut limiter la détection ; haute précision sur test suggère de bonnes performances, à valider en conditions réelles.

3.6.4 Évaluation sur l'Ensemble de Test

```
# Predict on test set
y_pred = model.predict(X_test)
unique_classes = np.unique(y_test)
target_names = ['Normal', 'DoS', 'Fuzzing', 'Suspension']

# Classification report
print("\nClassification Report on Labeled Test Set:")
print(classification_report(y_test, y_pred, labels=unique_classes, target_names=target_names))

# Confusion matrix
cm = confusion_matrix(y_test, y_pred, labels=unique_classes)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=target_names, yticklabels=target_names)
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()

# Feature importance
plt.figure(figsize=(10, 6))
xgb.plot_importance(model, max_num_features=8)
plt.title('Feature Importance')
plt.show()
```

Figure 3.13 - Évaluation sur l'ensemble de test

Explication :

- ✓ Prédictions : Le modèle prédit les étiquettes pour l'ensemble de test (X_test).
- ✓ Rapport de classification : classification_report calcule la précision, le rappel, et le score F1 pour chaque classe, en utilisant des noms lisibles (target_names).
- ✓ Matrice de confusion : Une visualisation thermique est générée avec seaborn.heatmapmontrant le nombre de prédictions correctes et incorrectes par classe.

- ✓ Importance des caractéristiques : xgb.plot_importance affiche un graphique des caractéristiques les plus influentes, basé sur leur contribution à la décision des arbres.

Classification Report on Labeled Test Set:

	precision	recall	f1-score	support
Normal	1.00	1.00	1.00	55638
DoS	1.00	1.00	1.00	5998
Fuzzing	0.99	0.99	0.99	333
Suspension	1.00	1.00	1.00	11
accuracy			1.00	61980
macro avg	1.00	1.00	1.00	61980
weighted avg	1.00	1.00	1.00	61980

Figure 3.14 - Résultat : Rapport de Classification

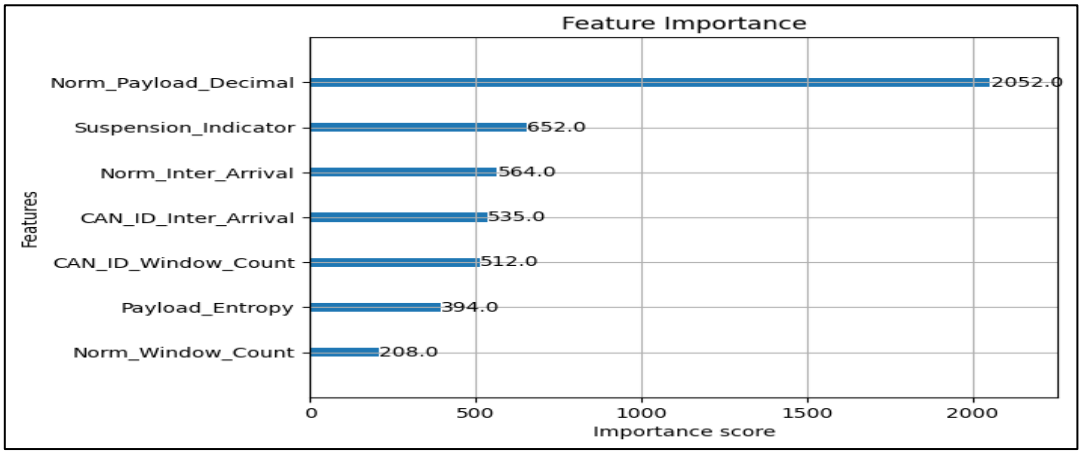

Figure 3.16 - Résultat : Importance des Caractéristiques

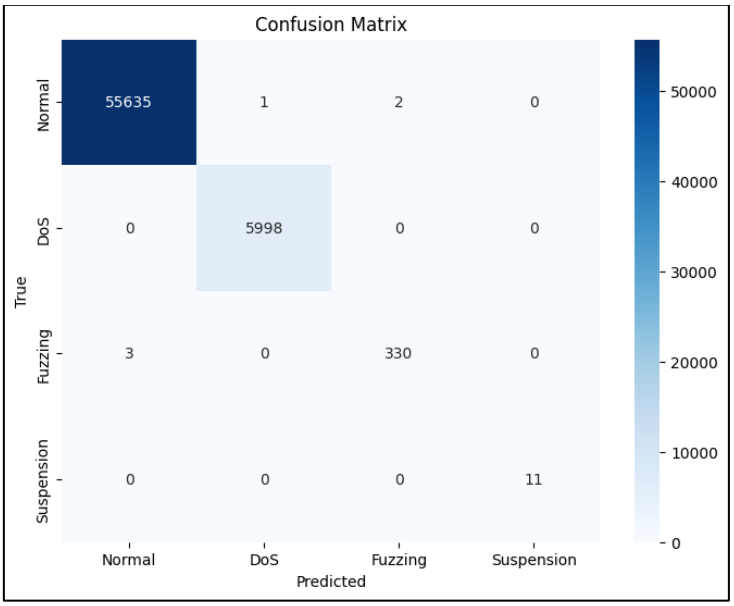

Figure 3.15 - Résultat : Matrice de Confusion

Résumé des Résultats :

- **Rapport de Classification** :
  - ✓ **Normal** : Précision, rappel, et F1-score de 1,00 (support : 55 638).
  - ✓ **DoS** : Précision, rappel, et F1-score de 1,00 (support : 5 998).
  - ✓ **Fuzzing** : Précision, rappel, et F1-score de 0,99 (support : 333).
  - ✓ **Suspension** : Précision, rappel, et F1-score de 1,00 (support : 11).
  - ✓ **Exactitude globale** : 1,00 (sur 61 980 échantillons).
  - ✓ **Signification** : Le modèle atteint une performance presque parfaite sur l'ensemble de test étiqueté, bien que le faible support pour Fuzzing et Suspension (333 et 11) puisse indiquer une sensibilité limitée à ces classes rares.
- **Matrice de Confusion** :
  - ✓ **Normal** : 55 635 prédictions correctes, 1 erreur vers DoS, 2 vers Fuzzing, 0 vers Suspension.
  - ✓ **DoS** : 5 998 prédictions correctes, 0 erreur.
  - ✓ **Fuzzing** : 330 prédictions correctes, 3 erreurs vers Normal.
  - ✓ **Suspension** : 11 prédictions correctes, 0 erreur.
  - ✓ **Signification** : La matrice confirme une classification quasi-parfaite, avec des erreurs minimes (surtout pour Fuzzing), cohérent avec le déséquilibre des données.
- **Importance des Caractéristiques** :
  - ✓ **Norm_Payload_Decimal** : 2052,0 (caractéristique la plus influente).
  - ✓ **Suspension_Indicator** : 652,0.
  - ✓ **Norm_Inter_Arrival** : 564,0.
  - ✓ **CAN_ID_Inter_Arrival** : 535,0.
  - ✓ **CAN_ID_Window_Count** : 512,0.
  - ✓ **Payload_Entropy** : 394,0.
  - ✓ **Norm_Window_Count** : 208,0.
  - ✓ **Signification** : Les caractéristiques normalisées et liées aux interruptions (e.g., Norm_Payload_Decimal, Suspension_Indicator) dominent, reflétant leur rôle clé dans la détection des attaques, notamment les suspensions et les anomalies de charge utile.

3.6.5 Prédiction sur Données Non Étiquetées

```
# Define unlabeled data path
unlabeled_data_path = r"C:\Users\pc\OneDrive\Bureau\VS_code_Projects\MLpro
predictions_path = r"C:\Users\pc\OneDrive\Images\Bureau\VS_code_Projects\M

# Process unlabeled data
print("\nProcessing unlabeled data...")
unlabeled_df = process_unlabeled_data(unlabeled_data_path, normal_stats)

# Predict labels
X_unlabeled = unlabeled_df[features]
unlabeled_df['Predicted_Label'] = model.predict(X_unlabeled)

# Save predictions
unlabeled_df.to_csv(predictions_path, index=False)
print(f"Predictions saved to {predictions_path}")
```

Figure 3.17 - Prédiction sur données non étiquetées

Explication :

- ✓ Traitement des données : La fonction process_unlabeled_data est appelée pour lire et traiter le fichier dosattack.log, générant un DataFrame avec les caractéristiques nécessaires.
- ✓ Prédictions : Les caractéristiques sont extraites (X_unlabeled), et le modèle prédit les étiquettes, stockées dans Predicted_Label.
- ✓ Sauvegarde : Les résultats sont exportés vers unlabeled_predictions.csv.
- ✓ Analyse : Le nombre de prédictions par classe est affiché, et les statistiques descriptives des caractéristiques sont calculées pour chaque classe prédite, permettant de vérifier leur cohérence avec les motifs attendus.

```
Prediction Counts on Unlabeled Data:
Predicted_Label
Normal    101926
DoS        40001
Name: count, dtype: int64
```

Figure 3.18 - Résultat de la prédiction sur données non étiquetées

Résumé des Résultats :

Le résultat des prédictions sur les données non étiquetées montre une détection précise : 101 926 trames sont classées comme normales et 40 001 comme DoS. Puisque on a injecté exactement 40 001 trames DoS dans dosattack.log, cette correspondance exacte valide l'efficacité du modèle pour identifier les attaques. Cela suggère que les caractéristiques utilisées, comme les intervalles d'arrivée et le compteur de fenêtre, capturent bien les motifs de

l'attaque, alignés avec votre injection de messages à haute priorité sur l'ID '000' pendant 10 secondes.

3.7 Conclusion

Le chapitre a démontré avec succès l'entraînement d'un modèle XGBoost sur 413 196 trames, atteignant une exactitude remarquable de 1,00 sur 61 980 trames test, confirmée par une détection parfaite de 40 001 attaques DoS injectées dans dosattack.log (101 926 normales). Les caractéristiques clés comme Norm_Payload_Decimal et Suspension_Indicator ont brillamment contribué à ces résultats, offrant une base solide pour la détection d'anomalies. Malgré le faible nombre d'échantillons pour Fuzzing (333) et Suspension (11), le modèle a montré une robustesse prometteuse. Avec des optimisations futures des hyperparamètres, une augmentation des données minoritaires, et des tests en temps réel, ce modèle se positionne comme une solution puissante et fiable pour la maintenance prédictive des véhicules connectés.

Conclusion générale

Ce rapport marque l'aboutissement d'une étude approfondie sur l'utilisation du bus CAN dans les systèmes automobiles et la mise en œuvre d'une solution innovante de détection d'attaques basée sur l'intelligence artificielle. L'entraînement d'un modèle XGBoost sur un ensemble de 413 196 trames a permis d'atteindre une exactitude remarquable de 1,00 sur l'ensemble de test comprenant 61 980 trames, validée par une détection parfaite des 40 001 attaques DoS injectées dans le fichier dosattack.log, aux côtés de 101 926 trames classées comme normales. Les caractéristiques normalisées, notamment Norm_Payload_Decimal et Suspension_Indicator, ont joué un rôle clé dans cette performance, démontrant leur pertinence pour capturer les motifs d'anomalies, qu'il s'agisse d'intrusions massives ou de suspensions rares. L'analyse de l'importance des caractéristiques et la matrice de confusion confirment la robustesse du modèle, bien que le faible échantillonnage des classes minoritaires (333 cas de fuzzing et 11 cas de suspension) suggère une sensibilité limitée à ces scénarios spécifiques.

Malgré ces résultats prometteurs, plusieurs pistes d'amélioration se dessinent pour consolider cette approche. L'optimisation des hyperparamètres du modèle, l'augmentation des données pour les classes sous-représentées via des simulations d'attaques synthétiques, et l'intégration de tests en conditions réelles sur des flux CAN embarqués pourraient renforcer sa fiabilité et sa généralisation. De plus, l'évaluation en temps réel, confrontée aux contraintes d'un environnement dynamique, permettra de valider son efficacité face à des attaques imprévues. Ce projet pose ainsi une base solide pour la maintenance prédictive, offrant une contribution significative à la cybersécurité des véhicules connectés. À l'aube d'une ère où l'autonomie et la connectivité redéfinissent la mobilité, ces avancées technologiques s'inscrivent dans une vision à long terme, visant à garantir la sécurité et la résilience des systèmes automobiles face aux défis croissants de la cybermenace.

Bibliographie

[1] **ISO 11898-1:2015** - "Road vehicles — Controller area network (CAN) — Part 1: Data link layer and physical signalling" (ISO 11898).

[2] **Bosch CAN Specification 2.0** - Document original définissant le protocole CAN, publié par Robert Bosch GmbH.

[3] **Cho, K.-T., & Shin, K. G. (2016)** - "Fingerprinting Electronic Control Units for Vehicle Intrusion Detection" (USENIX Security).

[4] **Hoppe, T., Kiltz, S., & Dittmann, J. (2008)** - "Security threats to automotive CAN networks—Practical examples and selected short-term countermeasures" (SAFECOMP).

[5] **Chen, T., & Guestrin, C. (2016)** - "XGBoost: A Scalable Tree Boosting System" (KDD).

Annexes

Timestamp	Interface	CAN_ID	Payload
1508687283.891357	slcan0	12E	C680027FD0FFFF00
1508687283.891365	slcan0	090	1A000000
1508687283.891368	slcan0	0C6	7512800A8008BAAC
1508687283.891375	slcan0	242	0000FFEFFE000D
1508687283.891377	slcan0	29C	00000000FFFFFFFF
1508687283.891379	slcan0	352	5C00001A
1508687283.895300	slcan0	1F6	1E2082368005FFFE
1508687283.895329	slcan0	186	1C2032C320002A
1508687283.895711	slcan0	18A	320000070C040000
1508687283.896597	slcan0	4AC	80
1508687283.896772	slcan0	211	4E3718D7180000
1508687283.896831	slcan0	45C	00000000
1508687283.897208	slcan0	214	FBFE
1508687283.897218	slcan0	5DF	0084
1508687283.900328	slcan0	12E	C580027FD0FFFF00
1508687283.900357	slcan0	090	1A000000
1508687283.901062	slcan0	0C6	751280148008BCA0
1508687283.901072	slcan0	29A	00000000000007F8
1508687283.901089	slcan0	2B7	00E0FFFE01
1508687283.905236	slcan0	1F6	1E2082368005FFFE
1508687283.905347	slcan0	186	1C2032D320002A
1508687283.905685	slcan0	18A	320000070C040000
1508687283.906597	slcan0	217	260D0800000060
1508687283.906925	slcan0	2C6	FFFFFFFFFFF0
1508687283.910322	slcan0	12E	C580027FD0FFFF00
1508687283.910410	slcan0	090	1A000000
1508687283.910687	slcan0	0C6	751280008008BEB2
1508687283.910786	slcan0	242	0000FFEFFE000D
1508687283.911154	slcan0	29C	00000000FFFFFFFF
1508687283.911430	slcan0	354	0000000000000000
1508687283.911435	slcan0	392	0555300C
1508687283.915234	slcan0	5E9	0038000000180000

Tableau 4.1 - Les données de full_capture_data.log

```python
# Compute CAN_ID_Inter_Arrival per CAN_ID
can_id_inter_arrivals = {}
max_inter_arrivals = {}
large_gap_count = 0
max_expected_gap = 5.0  # 5 seconds to detect suspension attack
for can_id, timestamps in can_id_timestamps.items():
    timestamps = sorted(timestamps)
    inter_arrivals = []
    for i in range(len(timestamps)):
        if i == 0:
            # First message: Use default (0.00001 seconds or 10 µs)
            inter_arrivals.append(0.00001)
            print(f"Debug: CAN_ID={can_id}, Index={i}, Timestamp={timestamps[i]}, First message, Inter_Ar
        else:
            # Difference to previous message
            diff = timestamps[i] - timestamps[i-1]
            inter_arrivals.append(diff)
            if diff > max_expected_gap:
                large_gap_count += 1
                print(f"Warning: Large gap for CAN_ID={can_id}, Index={i}, Timestamp={timestamps[i]}, Prev
            print(f"Debug: CAN_ID={can_id}, Index={i}, Timestamp={timestamps[i]}, Prev_Timestamp={timestar
    can_id_inter_arrivals[can_id] = inter_arrivals
    max_inter_arrivals[can_id] = max(inter_arrivals) if inter_arrivals else 0.00001

# Print summary of max_inter_arrival per CAN_ID
print("\nSummary of Maximum CAN_ID_Inter_Arrival per CAN_ID:")
for can_id, max_inter in max_inter_arrivals.items():
    if max_inter > max_expected_gap:
        print(f"⚠️ CAN_ID={can_id}: max_inter_arrival={max_inter:.5f}s (Possible suspension attack)")
    else:
        print(f"CAN_ID={can_id}: max_inter_arrival={max_inter:.5f}s")

if large_gap_count > 0:
    print(f"Warning: Found {large_gap_count} inter-arrival times larger than {max_expected_gap}s. Likely
```

Figure 4.2 - Extrait du code de canID_Inter_Arrival.py

```python
def calc_entropy(payload):
    if not payload:
        return 0.0
    try:
        bytes_list = [int(payload[i:i+2], 16) for i in range(0, len(payload), 2)]
        value_counts = np.bincount(bytes_list, minlength=256)
        return entropy(value_counts[value_counts > 0])
    except ValueError:
        return 0.0

def payload_to_decimal(payload):
    try:
        return int(payload, 16)
    except ValueError:
        return 0

def convert_can_log_to_csv(input_file_path, output_file_path):
    # Verify input file exists
    if not Path(input_file_path).is_file():
        print(f"Error: Input file '{input_file_path}' does not exist.")
        return

    # Ensure output directory exists
    output_dir = Path(output_file_path).parent
    try:
        output_dir.mkdir(parents=True, exist_ok=True)
    except Exception as e:
        print(f"Error creating output directory '{output_dir}': {e}")
        return
```

Figure 4.1 - Extrait du code de Payload_Entropy_and_Payload_decimal.py

```
# First pass: Read messages, collect timestamps, and compute windows
messages = []
can_id_timestamps = {}  # Store timestamps for each CAN ID
window_counts = {}      # Store message counts per CAN ID per window
try:
    with open(input_file_path, 'r', encoding='utf-8') as log_file:
        for line in log_file:
            line = line.strip()
            if not line or not line.startswith('('):
                continue
            try:
                timestamp_end = line.find(')')
                timestamp = line[1:timestamp_end]
                remaining = line[timestamp_end+1:].strip()
                parts = remaining.split(maxsplit=1)
                if len(parts) == 2:
                    interface = parts[0]
                    can_data = parts[1]
                    if '#' in can_data:
                        can_id, payload = can_data.split('#', 1)
                        timestamp_float = float(timestamp)
                        messages.append({
                            'timestamp': timestamp_float,
                            'interface': interface,
                            'can_id': can_id,
                            'payload': payload
                        })
                        # Store timestamp for CAN ID
                        if can_id not in can_id_timestamps:
                            can_id_timestamps[can_id] = []
                        can_id_timestamps[can_id].append(timestamp_float)
                        # Compute window and update counts
                        window = int(timestamp_float // 10)
                        if can_id not in window_counts:
                            window_counts[can_id] = {}
                        window_counts[can_id][window] = window_counts[can_id].get(window, 0) + 1
            except:
                continue
except Exception as e:
    print(f"Error reading input file: {e}")
    return
```

Figure 4.3 - Extrait du code de canId-InterArrival_and_canId-WindowCount