République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

École Nationale Supérieure en Informatique
- Sidi Bel Abbès -

Rapport De Projet

Option : Ingénierie des Systèmes Informatiques

Détection d'Attaques BOTNET avec l'Intelligence Artificielle

Réalisé par :

BOUCHELAREM LYDIA HOUYEM
HAKKAS YACINE
HALIMI ABDELKRIM
LEBAILI MOHAMED OUAIL

Encadré par :

M. CHAIB Souleyman

Année Universitaire : 2024/2025

Membres de l'Équipe

— **BOUCHELAREM LYDIA HOUYEM**
— **HAKKAS YACINE**
— **HALIMI ABDELKRIM**
— **LEBAILI MOHAMED OUAIL**

Répartition des tâches :

— Recherche et sélection du dataset
— Prétraitement des données
— Développement des modèles d'IA
— Mise en place de l'environnement de test
— Automatisation du processus

Introduction

Les attaques BOTNET représentent une menace croissante dans le paysage de la cybersécurité moderne. Un BOTNET est un réseau d'appareils infectés par des logiciels malveillants et contrôlés à distance par un attaquant, souvent utilisé pour mener des attaques DDoS, voler des données ou diffuser des spams. La détection de ces attaques devient de plus en plus complexe avec l'évolution des techniques d'attaque et l'expansion de l'Internet des Objets (IoT).

Dans ce projet, nous proposons une approche innovante pour détecter les attaques BOTNET en utilisant des techniques avancées d'intelligence artificielle. Notre solution vise à identifier les comportements malveillants dans le trafic réseau avec une grande précision, tout en minimisant les faux positifs.

L'objectif principal de ce travail est de développer un système de détection capable de :

— Analyser le trafic réseau en temps réel

— Identifier les patterns caractéristiques des BOTNETs

— S'adapter aux nouvelles formes d'attaques

— Fonctionner dans divers environnements (IoT, réseaux traditionnels)

Pour réaliser cela, nous avons adopté une méthodologie en plusieurs étapes, depuis la sélection et le prétraitement des données jusqu'au développement et à l'évaluation de modèles d'apprentissage profond.

Chapitre 1

Approche et Méthodologie

1.1 Sélection du Dataset

Pour ce projet, nous avons choisi le dataset CIC IoT 2023 comme base de données principale. Ce dataset présente plusieurs avantages :

— Il est récent et reflète les dernières tendances en matière d'attaques

— Il contient une variété d'attaques BOTNET réalistes

— Il inclut des données provenant d'appareils IoT, pertinentes pour notre étude

— Il est librement accessible à des fins de recherche

Cependant, lors de l'analyse initiale, nous avons constaté que le dataset était déséquilibré avec plus de 90% des données représentant des attaques. Cette caractéristique a influencé notre approche méthodologique.

1.2 Première Approche : Détection d'Anomalies

Contexte général

Dans le cadre de la détection d'anomalies de trafic réseau (notamment pour repérer des attaques de type botnet) à partir du jeu de données **CICIoT**, deux éléments clés ont été envisagés :

— **TimeEval** : Framework d'évaluation d'algorithmes de détection d'anomalies sur séries temporelles

— **Isolation Forest (iForest)** : Algorithme non supervisé d'isolation d'anomalies

Cette approche n'a finalement pas été retenue au profit d'une solution de **classification supervisée**.

Présentation de TimeEval

Caractéristiques principales

— Framework open-source pour benchmarker des algorithmes de détection d'anomalies

— Fonctionnalités clés :

  — Préparation des données en fenêtres temporelles

  — Implémentations d'algorithmes prêts à l'emploi

  — Métriques temporelles spécifiques

  — Pipeline modulaire et reproductible

Architecture

1. Chargement et prétraitement des données
2. Découpage en fenêtres glissantes
3. Extraction de caractéristiques statistiques
4. Normalisation des données
5. Entraînement de l'algorithme
6. Validation et prédiction
7. Benchmarking comparatif

Isolation Forest (iForest)

Principe fondamental

— Algorithme basé sur le principe que "les anomalies sont rares et faciles à isoler"
— Ne modélise pas la distribution normale
— Utilise des arbres d'isolation aléatoires

Construction des arbres

— Sélection aléatoire de sous-échantillons
— Partition récursive des données
— Critères de division :
  — Choix aléatoire d'une feature
  — Choix aléatoire d'un seuil
— Limitation de la profondeur maximale

Calcul du score d'anomalie

$$s(x, m) = 2^{-\frac{E[h(x)]}{c(m)}}$$

— $E[h(x)]$: Profondeur moyenne d'isolation
— $c(m)$: Longueur de chemin moyenne normalisée
— Score proche de 1 → anomalie probable

TABLE 1.1 – Avantages et limites d'Isolation Forest

Avantages	**Limites**
Non supervisé	Ne capture pas les dépendances temporelles
Complexité linéaire	Score continu nécessitant un seuil
Peu de paramètres	Difficulté avec anomalies subtiles

Application sur CICIoT

Préparation des données

— Découpage en fenêtres temporelles (ex : 30 secondes)
— Extraction de caractéristiques par fenêtre
— Normalisation (StandardScaler ou MinMaxScaler)
— Division train/test (70%/30%)

Configuration dans TimeEval

```
model:
  name: "Isolation Forest"
  parameters:
    n_estimators: 100
    max_samples: 256
    contamination: 0.01
```

Voici les commandes Docker que nous avons utilisées pour configurer l'environnement TIMEEVAL :

```
# Construction des images de base
cd TimeEval-algorithms/0-base-images
docker build -t ghcr.io/timeeval/python3-base:0.3.0 python3-base

# Construction des images intermediaires
cd ../1-intermediate-images
# Pour pyod (necessaire pour LOF)
docker build -t ghcr.io/timeeval/pyod:0.3.0 pyod
# Pour scikit-learn (necessaire pour Isolation Forest)
docker build -t ghcr.io/timeeval/sklearn:0.3.0 sklearn

# Construction des algorithmes
cd ..
# Pour Isolation Forest
docker build -t ghcr.io/timeeval/iforest:0.3.0 ./iforest
# Pour LOF
docker build -t ghcr.io/timeeval/lof:0.3.0 ./lof
# Pour d'autres algorithmes (ex: matrix profile)
docker build -t ghcr.io/timeeval/matrix-profile:0.3.0 ./matrix_profile
```

Listing 1.1 – Configuration des images Docker pour TIMEEVAL

Et voici le script bash pour exécuter l'algorithme Isolation Forest :

```
#!/bin/bash

# Definition de l'algorithme a utiliser
TIMEEVAL_ALGORITHM="iforest"

# Creation du repertoire des resultats
mkdir -p results

# Execution de l'algorithme
docker run --rm \
    -v $(pwd)/data:/data:ro \
    -v $(pwd)/results:/results:rw \
    -e LOCAL_UID=$(id -u) \
    -e LOCAL_GID=$(id -g) \
    ghcr.io/timeeval/${TIMEEVAL_ALGORITHM}:latest execute-algorithm '{
        "executionType": "execute",
        "dataInput": "/data/dataset.csv",
        "dataOutput": "/results/anomaly_scores.ts",
        "modelInput": "/results/model.pkl",
        "modelOutput": "/results/model.pkl",
        "customParameters": {
            "n_estimators": 100,
            "max_samples": "auto",
```

```
            "contamination": 0.5,
            "random_state": 42
        }
    }'

echo "Detection d'anomalies terminee. Resultats sauvegardes dans results/anomaly_scores.ts"
```

Listing 1.2 – Script d'execution de l'algorithme Isolation Forest

Résultats observés

- Bonne détection des anomalies brutales
- Difficulté avec les attaques progressives
- Taux de faux positifs modéré mais difficile à ajuster

Raisons du non-retenu

Besoin de classification supervisée

- Nécessité de distinguer plusieurs types d'attaques
- Labels détaillés disponibles dans CICIoT
- Objectif pédagogique de montrer des modèles supervisés

Limites temporelles

- Traitement indépendant des fenêtres
- Perte du contexte séquentiel
- Approches alternatives préférées (LSTM, CNN 1D)

Complexité du réglage

- Choix délicat du seuil de contamination
- Compromis difficile entre faux positifs/négatifs
- Meilleur contrôle avec modèles supervisés

Explicabilité pédagogique

- Visualisations plus parlantes (matrice de confusion)
- Métriques spécifiques à la classification
- Importance des features plus claire

Conclusion

- TimeEval excellent pour benchmarking d'algorithmes
- Isolation Forest adapté pour détection non supervisée
- Approche non retenue en raison :
  - Du besoin de classification multi-classe
  - De la disponibilité des labels détaillés
  - Des exigences pédagogiques du projet
- Solution finale basée sur des approches supervisées

1.3 Seconde Approche : Classification Supervisée

Face à les limitations du Détection d'Anomalies, nous avons décidé de repartir de zéro avec une approche de classification supervisée. Notre encadrant nous a orientés vers deux architectures avancées :

1.3.1 Prétraitement des Données

Avant l'entraînement, nous avons effectué un prétraitement complet des données :

— Nettoyage des valeurs manquantes et aberrantes

— Normalisation des features numériques

— Encodage des features catégorielles

— Rééquilibrage partiel du dataset

— Séparation en ensembles d'entraînement, validation et test

Les objets utilisés pour l'encodage et la normalisation (`encoder.pkl` et `scaler.pkl`) ont été enregistrés après le prétraitement afin d'assurer une transformation cohérente des futures données.

1.3.2 Modèle LSTM

Les réseaux LSTM (Long Short-Term Memory) sont particulièrement adaptés pour :

— Analyser des séquences temporelles (comme le trafic réseau)

— Capturer des dépendances à long terme

— Gérer des données avec des patterns complexes

Notre implémentation LSTM comprend :

— Une couche d'embedding pour représenter les features

— Deux couches LSTM de 128 et 64 unités respectivement

— Une couche dense avec activation ReLU

— Une couche de sortie avec softmax pour la classification multi-classe

1.3.3 Modèle LLM avec DeepSeek pour la détection d'attaques réseau

Objectif

Affiner un modèle de langage basé sur des transformers (DeepSeek-R1-Distill-Llama-8B) sur le jeu de données CICIoT pour classer le trafic réseau comme bénin ou malveillant, en utilisant une stratégie d'adaptation efficace (LoRA) sur du matériel limité (Google Colab avec GPU T4).

Architecture du modèle

Nous avons choisi **DeepSeek-R1-Distill-Llama-8B**, une version distillée et optimisée de LLaMA 8B, offrant un bon compromis entre performance et efficacité des ressources.

- **Basé sur Transformer** : Capacité à comprendre les séquences longues - crucial pour les motifs complexes du trafic réseau
- **Distillé** : Plus léger et plus rapide, adapté aux environnements contraints
- **Compatibilité Unsloth** : Interface optimisée pour intégrer LoRA, la quantification et des flux de travail d'affinage rapide

Optimisation via LoRA

La **Low-Rank Adaptation (LoRA)** est une technique qui affine les grands modèles de langage en gelant les poids originaux et en injectant de petites matrices entraînables de faible rang.

- **Couches ciblées** :
  - Projections d'attention (q_proj, k_proj, v_proj)
  - Projections feedforward (o_proj, gate_proj, up_proj, down_proj)
- **Avantages** :
  - Mémoire efficace (moins de paramètres entraînés)
  - Modèle de base réutilisable
  - Entraînement plus rapide

Configuration technique

```
model_lora = FastLanguageModel.get_peft_model(
    model,
    r=16,
    target_modules=["q_proj", "k_proj", "v_proj", "o_proj",
                    "gate_proj", "up_proj", "down_proj"],
    lora_alpha=32,
    lora_dropout=0,
    bias="none",
    use_gradient_checkpointing="unsloth",
    random_state=3407,
)
```

TABLE 1.2 – Paramètres LoRA et leur justification

Paramètre	Justification
r=16	Compromis expressivité/mémoire
target_modules	Couvrir toutes les projections majeures
lora_alpha=32	Facteur d'échelle pour stabiliser l'apprentissage
lora_dropout=0	Maximiser la performance sur petit jeu de données
bias="none"	Garder le fine-tuning minimal

Optimisations supplémentaires

— **Quantification 4-bit** : Réduction de l'utilisation de la VRAM
— **Précision mixte (fp16)** : Calcul plus rapide avec empreinte mémoire réduite
— **Gradient checkpointing** : Économie de mémoire pendant la rétropropagation
— **Optimiseur AdamW 8-bit** : Version mémoire-efficace d'AdamW
— **Planificateur de taux d'apprentissage cosinus** : Meilleure convergence

Ingénierie des prompts

Conversion des échantillons tabulaires en texte via des templates :

```
Below is a network traffic sample: feature1 | feature2 | ... | featureN
Classify this traffic as either 'benign' or an attack type.
Answer with your label enclosed in <label> and </label> tags.
```

Conclusion

Cette approche combine :
— Un modèle LLM puissant mais efficient (DeepSeek-LLaMA)
— Une méthode de fine-tuning optimisée (LoRA)
— Des techniques d'accélération matérielle (quantification 4-bit)
— Une adaptation des données tabulaires au format texte

1.3.4 Modèle Transformer

Architecture du modèle

Ce projet utilise un **réseau de neurones basé sur Transformer** conçu pour la **classification multi-classes** du trafic réseau afin de détecter les **attaques botnet** en utilisant le jeu de données **CICIoT2023**.

Bien que les Transformers aient été initialement conçus pour le traitement du langage naturel (NLP), des recherches récentes ont démontré leur capacité à modéliser les dépendances dans les données tabulaires structurées.

Avantages des Transformers pour données tabulaires

— Capture les **interactions entre caractéristiques** de manière globale
— Gère à la fois les relations **linéaires et non-linéaires**
— Adapté aux jeux de données avec de nombreuses caractéristiques de types mixtes
— Offre une bonne scalabilité et robustesse au bruit

Composants principaux

Couche de projection d'entrée

— Convertit le vecteur de caractéristiques en une représentation dense de haute dimension ($d_{model} = 64$)
— Dimensions : entrée $[batch\_size, num\_features]$ → sortie $[batch\_size, 1, d_{model}]$

```
self.input_proj = nn.Linear(input_dim, d_model)
```

Bloc encodeur Transformer Composé de plusieurs couches **TransformerEncoderLayer** ($num\_layers = 3$), chaque couche contenant :

- Attention multi-têtes ($nhead = 4$)
- Réseau feedforward ($dim\_feedforward = 128$)
- Normalisation par couche et dropout ($dropout = 0.1$)

```
encoder_layer = nn.TransformerEncoderLayer(
    d_model=d_model,
    nhead=nhead,
    dim_feedforward=dim_feedforward,
    dropout=dropout,
    batch_first=True
)
self.transformer_encoder = nn.TransformerEncoder(encoder_layer,
                                                 num_layers=num_layers)
```

Tête de classification Réseau MLP à 2 couches :

- Couche dense avec activation ReLU pour la régularisation
- Couche finale projetant vers $num\_classes$

```
self.classifier = nn.Sequential(
    nn.Linear(d_model, dim_feedforward),
    nn.ReLU(),
    nn.Dropout(dropout),
    nn.Linear(dim_feedforward, num_classes)
)
```

Passe avant (Forward Pass)

```
def forward(self, x):
    x = x.unsqueeze(1)               # [B, 1, F]
    x = self.input_proj(x)           # [B, 1, d_model]
    x = self.transformer_encoder(x)  # [B, 1, d_model]
    x = x[:, 0, :]                   # [B, d_model]
    return self.classifier(x)        # [B, num_classes]
```

Stratégie d'entraînement

- Fonction de perte : `CrossEntropyLoss`
- Optimiseur : `AdamW` avec décroissance de poids
- Taille de batch : 32 (pour éviter le débordement mémoire)
- Nombre d'époques : 10 avec sauvegarde des checkpoints
- Meilleur modèle sélectionné sur la précision de validation

Évaluation

— Métriques utilisées :
  — Précision globale
  — Matrice de confusion
  — Rapport de classification
— Performance élevée obtenue sur plusieurs types d'attaques

TABLE 1.3 – Résumé des caractéristiques du modèle

Élément	**Détail**
Type de modèle	Transformer pour données tabulaires
Projection d'entrée	Couche linéaire (input_dim → d_model)
Encodeur	3 couches Transformer (4 têtes, FF=128)
Tête de classification	MLP (ReLU, Dropout)
Fonction de perte	CrossEntropy
Optimiseur	AdamW

Points clés pour la présentation

— Les Transformers capturent les dépendances globales dans les données tabulaires
— Architecture adaptée pour la détection d'intrusions en temps réel
— Bonnes performances sur le benchmark CICIoT2023
— Solution légère et efficace pour la classification du trafic réseau

1.3.5 Modèle BERT pour la détection d'attaques botnet

Définition du modèle BERT

BERT (Bidirectional Encoder Representations from Transformers) est un modèle de langage pré-entraîné développé par Google en 2018. Ses caractéristiques principales sont :

— **Bidirectionnalité** : Comprend chaque token en tenant compte de son contexte gauche et droit simultanément
— **Pré-entraînement en deux phases** :
  — Masked Language Modeling (MLM) : Prédiction de tokens masqués
  — Next Sentence Prediction (NSP) : Prédiction de l'ordre des phrases

Dans notre application, BERT est utilisé comme **encodeur universel** pour modéliser les relations entre caractéristiques réseau formatées sous forme textuelle (`colonne:valeur`).

Architecture interne

Tokenizer WordPiece

— Découpe le texte en sous-mots (ex : `"duration:0.5"` → `["dur", "##ation", ":", "0.5"]`)
— Ajout automatique des tokens spéciaux `[CLS]` (début) et `[SEP]` (fin)

Couche d'embedding

— **Token Embedding** : Vecteur dense de dimension 768 pour chaque token

— **Position Embedding** : Encode la position du token dans la séquence

— **Segment Embedding** : Non utilisé dans notre cas (séquence unique)

Encodeurs Transformer (12 couches) Chaque couche contient :

— **Attention multi-têtes** (12 têtes) : Capture les relations entre paires de tokens

— **Réseau feed-forward** : Transformation non-linéaire

— **Connexions résiduelles** et normalisation par couche

Tête de classification

— Le vecteur du token `[CLS]` (synthèse de la séquence) est projeté vers l'espace des classes

— Fonction d'activation softmax pour obtenir les probabilités

Application aux données tabulaires

Prétraitement des données

— Conversion des labels textuels en entiers via `LabelEncoder`

— Division stratifiée des données (80% train, 20% test)

Textualisation des features Chaque échantillon est converti en chaîne de caractères :

```
"duration:0.5 packet_count:10 protocol:tcp ..."
```

Processus complet

1. Tokenization et padding/troncation à 256 tokens
2. Création des embeddings (token + position)
3. Passage à travers les 12 couches Transformer
4. Extraction du vecteur `[CLS]`
5. Classification via couche dense + softmax

Entraînement et évaluation

Configuration d'entraînement

— Taux d'apprentissage : 2e-5

— Batch size : 16

— Précision mixte (fp16)

— Weight decay : 0.01

— Optimiseur : AdamW

TABLE 1.4 – Configuration du fine-tuning BERT

Paramètre	**Valeur**
Modèle de base	bert-base-uncased
Longueur max des séquences	256 tokens
Taux d'apprentissage	2e-5
Batch size	16
Précision	fp16

Métriques d'évaluation

- Accuracy globale
- Rapport de classification (précision, rappel, F1-score)
- Matrice de confusion

Avantages et limitations

Avantages

- Capture automatique des relations complexes entre features
- Flexibilité pour ajouter/supprimer des colonnes
- Bonne généralisation même avec peu d'exemples

Limitations

- Traitement des valeurs numériques comme texte
- Contrainte de longueur maximale (256 tokens)
- Coût computationnel élevé

Plan détaillé pour le rapport

1. Introduction à BERT et son adaptation aux données tabulaires
2. Architecture et modifications apportées
3. Pipeline complet de prétraitement
4. Stratégie de fine-tuning
5. Résultats et analyse des performances
6. Discussion des perspectives d'amélioration

Chapitre 2

Implémentation et Tests

2.1 Environnement de Développement

Nous avons utilisé les technologies suivantes :

— Python 3.10 avec les bibliothèques TensorFlow et PyTorch

— Scikit-learn pour le prétraitement

— Docker pour la création des environnements de test

— Google Colab pour l'entraînement des modèles

2.2 Phase de Test Initiale

Pour valider nos modèles, nous avons d'abord créé une simulation IoT basée sur Docker :

— Plusieurs conteneurs simulant des appareils IoT

— Un conteneur pour le trafic normal

— Un conteneur injectant des attaques BOTNET

— Un système de collecte et analyse du trafic

Les résultats ont été très encourageants avec :

— Un taux de détection supérieur à 95%

— Un taux de faux positifs inférieur à 2%

— Une latence acceptable pour une analyse en temps quasi-réel

2.3 Test sur Machines Réelles

Suite à ces résultats positifs, nous avons testé notre solution dans des environnements plus réalistes :

— Machines virtuelles avec différents systèmes d'exploitation

— Appareils IoT physiques (caméras, capteurs)

— Postes de travail standards

Les performances sont restées stables avec :

— Une adaptation automatique aux différentes architectures

— Une robustesse face aux variations du trafic normal

— Une détection efficace des attaques même dans des environnements bruyants

Chapitre 3

Automatisation du Processus

La dernière phase de notre projet a consisté à automatiser l'ensemble du processus :

- Déploiement automatique des sondes de capture
- Prétraitement en flux continu des données
- Inférence en temps réel avec les modèles entraînés
- Génération d'alertes et rapports

Nous avons développé une architecture modulaire comprenant :

- Un module de capture du trafic
- Un pipeline de traitement des données
- Le moteur d'inférence IA
- Un système de visualisation et d'alerte

Cette automatisation permet :

- Une détection en temps réel des attaques
- Une intégration facile dans des environnements existants
- Une maintenance simplifiée
- Une évolutivité horizontale

Chapitre 4

La réalisation de la plateforme

4.1 Frontend

4.1.1 Architecture du Dashboard

Notre interface frontend est développée avec une stack moderne composée de :

- **React 18** pour la construction des composants UI
- **Tailwind CSS** pour le styling utilitaire
- **Chart.js** avec react-chartjs-2 pour les visualisations
- **WebSocket** pour les mises à jour temps réel

4.1.2 Structure des fichiers

```
src/
 components/
    AttackStats.jsx       # Statistiques globales
    DeviceGrid.jsx        # Grille des devices
    AttackChart.jsx       # Graphiques
    RecentAttacks.jsx     # Tableau des attaques
 pages/
    Dashboard.jsx         # Page principale
    DeviceDetail.jsx      # Détail device
 App.jsx                   # Configuration
```

4.1.3 Connexion Temps Réel

```
// Connexion WebSocket dans Dashboard.jsx
useEffect(() => {
  const ws = new WebSocket('ws://backend:8000/ws/labels/');

  ws.onmessage = (event) => {
    const data = JSON.parse(event.data);
    setAttackData(prev => [...prev, data]);
  };

  return () => ws.close();
}, []);
```

4.1.4 Composants Principaux

Carte de Statistiques

```
function AttackStats({ data }) {
  return (
    <div className="grid grid-cols-4 gap-4 mb-6">
      <div className="bg-white p-4 rounded-lg shadow">
        <h3>Attaques totales</h3>
        <p className="text-2xl font-bold">{data.total}</p>
      </div>
    </div>
  );
}
```

Grille des Devices

```
function DeviceGrid({ devices }) {
  return (
    <div className="grid grid-cols-2 md:grid-cols-4 gap-4">
      {devices.map(device => (
        <div key={device.ip} className={`p-4 rounded-lg ${
          device.status === 'attack' ? 'bg-red-100' : 'bg-green
    -100'
        }`}>
          <p>{device.ip}</p>
        </div>
      ))}
    </div>
  );
}
```

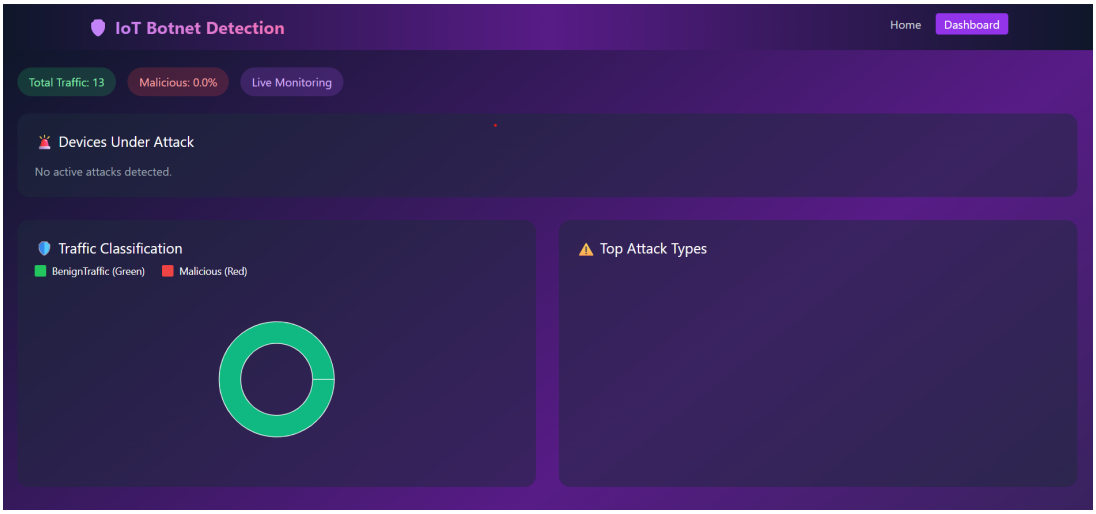

FIGURE 4.1 – Capture du dashboard de surveillance

4.1.5 Description de l'Interface

La figure 4.1 montre les principales zones :

1. Barre de navigation avec indicateur WebSocket
2. Cartes de statistiques globales
3. Graphiques de type d'attaques (PieChart et LineChart)

4. Grille visuelle des devices avec codes couleur
5. Tableau des dernières alertes avec timestamp

4.1.6 Styles avec Tailwind

Exemple de styling utilitaire :

```
<div className="bg-white rounded-lg shadow-md p-6
                hover:shadow-lg transition-shadow">
  <h2 className="text-xl font-semibold text-gray-800">
    Titre du composant
  </h2>
</div>
```

4.1.7 Flux de Données

1. Le backend envoie des updates via WebSocket
2. Le contexte React distribue les données
3. Les composants enfants se mettent à jour
4. Les graphiques s'animent fluidement

4.2 Backend

4.2.1 Introduction

Ce document décrit l'architecture complète et le flux de données d'un système de détection de botnet IoT en temps réel. Le trafic est capturé, les caractéristiques sont extraites, les attaques sont classées par un modèle d'apprentissage profond, et les alertes sont diffusées instantanément vers un tableau de bord basé sur React via WebSockets.

4.2.2 Composants du système

Capture de paquets et extraction de caractéristiques (capture.py)

Bibliothèque : PyShark (wrapper pour tshark)
Fonctionnalités :

- Détection automatique de l'interface IPv4 locale et du sous-réseau (ex : 192.168.20.0/24)
- Capture des paquets en direct et regroupement en "flux" bidirectionnels par (IP source, IP destination, ports, protocole)
- Maintient l'état par flux (heure de début/fin, temps des paquets, tailles, flags TCP, compteurs de direction, indicateurs de protocole)
- Calcule environ 40 caractéristiques statistiques par paquet/flux (durée, nombre d'octets, taux, compteurs de flags, taille min/moy/max/écart-type, temps inter-arrivée, covariance, etc.)
- Écrit les paquets bruts dans capture.pcap et les caractéristiques dans network_traffic.csv

Inférence du modèle et routage des événements (bg-flowmeter.py)

Bibliothèques : TensorFlow (Keras), scikit-learn (joblib), pandas, Django ORM, Django Channels
Fonctionnalités :

- Surveille network_traffic.csv pour les nouvelles lignes de caractéristiques
- Charge un modèle Keras pré-entraîné (model.h5), un scaler (scaler.pkl) et un encodeur de labels (encoder.pkl)
- Prétraite les caractéristiques (mise à l'échelle, remodelage) et exécute l'inférence pour obtenir les probabilités de classe
- Applique un post-traitement :
  - Seuil de confiance : toute prédiction non bénigne avec une confiance < 0.90 est reclassée en BenignTraffic
  - Filtrage : supprime tous les événements DNS_Spoofing
- Enregistre les résultats complets (caractéristiques + prédiction + confiance) dans network_predictions.csv
- Écrit les correspondances distillées (IP destination, label final, horodatage) dans ip_mappings.csv
- Persiste chaque correspondance dans le modèle LabelEvent de Django (SQLite)

— Diffuse chaque correspondance au groupe de canaux "labels" Redis via Django Channels

Passerelle WebSocket (Django Channels + Redis)

Serveur ASGI : Daphne
Couche de canaux : channels_redis.core.RedisChannelLayer (broker Redis sur le port 6379)
Consommateur :

```
class LabelConsumer(AsyncWebsocketConsumer):
    async def connect(self):
        await self.channel_layer.group_add("labels", self.channel_name)
        await self.accept()

    async def new_label(self, event):
        await self.send(text_data=json.dumps(event))
```

Point de terminaison : ws ://127.0.0.1 :8000/ws/labels/

4.2.3 Flux de données global

```
flowchart LR
  subgraph Capture
    A[LAN Interface] --> B[PyShark Sniffer]
    B --> C[capture.pcap]
    B --> D[network_traffic.csv]
  end

  subgraph Inference
    D --> E[bg-flowmeter.py]
    E --> F[network_predictions.csv]
    E --> G[ip_mappings.csv]
    E --> H[LabelEvent DB]
    E --> I[Redis "labels"]
  end

  subgraph WebSocket
    I --> J[LabelConsumer]
    J --> K[Browser WS Clients]
  end

  subgraph Frontend
    K --> L[React Dashboard]
  end
```

4.2.4 Déploiement et exécution

— Installer les dépendances :

```
pip install django channels channels-redis pyshark psutil numpy pandas jo
```

— Démarrer Redis :

```
redis-server
```

— Lancer la capture de paquets :

```
python web_app/botnet_attack/capture.py
```

— Lancer l'inférence et le streaming :

```
python web_app/botnet_attack/bg-flowmeter.py
```

— Démarrer le serveur ASGI :

```
daphne -b 127.0.0.1 -p 8000 core.asgi:application
```

— Ouvrir l'interface frontend dans le navigateur ; les graphiques et tableaux se mettent à jour en temps réel.

4.2.5 Extensions et notes

— Évolutivité : remplacer SQLite par PostgreSQL ; exécuter plusieurs workers Daphne derrière un load-balancer ; clusteriser Redis.
— Sécurité : ajouter une authentification/autorisation sur les WebSockets ; utiliser TLS en production.
— Visualisation : étendre avec des cartes de géolocalisation, des statistiques récapitulatives ou des seuils d'alerte dans le tableau de bord.

Chapitre 5

Conteneurisation avec Docker

Pour faciliter le déploiement, la portabilité et la reproduction de notre système, nous avons encapsulé l'ensemble de l'application dans un conteneur Docker.

5.1 Architecture du Dossier

L'organisation des fichiers du projet est la suivante :

```
botnet_detection/

 Dockerfile
 requirements.txt
 encoder.pkl
 scaler.pkl
 model.h5
 capture.py
 run_all.sh
 bg-flowmeter.py
```

5.2 Contenu du Dockerfile

Voici le contenu du fichier `Dockerfile` utilisé pour construire l'image Docker :

```
FROM python:3.10-slim

ENV DEBIAN_FRONTEND=noninteractive

# Install required tools
RUN apt-get update && \
    apt-get install -y tshark libpcap-dev curl && \
    rm -rf /var/lib/apt/lists/*

# Add user to allow capturing if needed (optional)
RUN groupadd -r wireshark && useradd -r -g wireshark dockeruser
RUN chgrp wireshark /usr/bin/dumpcap && chmod +x /usr/bin/dumpcap
   && setcap cap_net_raw,cap_net_admin=eip /usr/bin/dumpcap

# Create working directory
WORKDIR /app

# Copy everything into container
COPY . .

# Make the runner script executable
RUN chmod +x run_all.sh

```

```
# Install Python dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Run the pipeline
CMD ["bash", "./run_all.sh"]
```

5.3 Fichier `requirements.txt`

Ce fichier contient les bibliothèques nécessaires au fonctionnement du projet :

```
numpy
pandas
tensorflow
joblib
watchdog
pyshark
psutil
scikit-learn
```

5.4 Explication des Fichiers

- `capture.py` : Script responsable de la capture du trafic réseau brut.
- `bg-flowmeter.py` : Convertit le trafic en flux de caractéristiques utilisables par le modèle.
- `encoder.pkl` et `scaler.pkl` : Objets de prétraitement pour l'encodage et la normalisation.
- `model.h5` : Modèle LSTM entraîné pour la détection des attaques.
- `runall.sh` : shell script pour lancer les deux fichier bg-flowmeter.py , capture.py en parallel

5.5 Commandes Utiles

- Construction de l'image :

```
    docker build -t botnet-detector .

```

- Lancement du conteneur :

```
      docker run --rm -it --net=host --cap-add=NET_ADMIN botnet
   -detector

```

- Accès au terminal pour débogage :

```
    docker run -it --entrypoint /bin/bash botnet-detector

```

Cette approche basée sur Docker nous permet de garantir la portabilité, de simplifier le déploiement sur différentes machines ou serveurs, et d'assurer une cohérence entre les environnements de développement et de production.

Conclusion

Ce projet nous a permis d'explorer de manière approfondie le potentiel de l'intelligence artificielle appliquée à la cybersécurité, et plus précisément à la détection d'attaques de type BOTNET. À travers une démarche méthodique, nous avons mis en œuvre, testé et comparé différentes approches, allant de la détection d'anomalies à des modèles d'apprentissage supervisé avancés basés sur des architectures LSTM et Transformer.

L'efficacité de ces modèles, validée par des phases de test rigoureuses en environnement simulé et réel, atteste de la pertinence de notre démarche. L'intégration de techniques de fine-tuning de modèles de langage (LLM) comme DeepSeek, ainsi que l'automatisation complète de la chaîne de détection, montrent notre capacité à proposer une solution opérationnelle, robuste et adaptable.

Enseignements clés de ce travail :

- La qualité et l'équilibre du dataset sont fondamentaux pour des résultats fiables.
- L'adéquation entre les caractéristiques des données et les modèles utilisés est déterminante.
- Les architectures séquentielles comme LSTM et Transformer se révèlent particulièrement performantes pour l'analyse du trafic réseau.
- L'automatisation du pipeline, du prétraitement à la détection, constitue un véritable levier pour une mise en production efficace.

Perspectives d'amélioration futures :

- Intégrer des techniques d'apprentissage par renforcement pour un système adaptatif.
- Développer une compatibilité fine avec des protocoles spécifiques aux objets connectés (IoT).
- Mettre en place un module de réponse automatique face aux menaces détectées.
- Étendre notre solution à la détection d'autres types d'attaques réseaux (ransomware, APT, etc.).

En somme, ce projet constitue une base solide pour la création de solutions intelligentes, dynamiques et évolutives dans le domaine de la cybersécurité. Il reflète notre engagement, notre rigueur et notre capacité à mobiliser des compétences multidisciplinaires pour répondre à un enjeu technologique majeur.

Bibliographie

[1] Canadian Institute for Cybersecurity (2023).
CIC IoT 2023 Dataset.
University of New Brunswick.

[2] Liu, F. T., Ting, K. M., Zhou, Z. H. (2008, December).
Isolation forest.
In 2008 Eighth IEEE International Conference on Data Mining (pp. 413-422). IEEE.

[3] Hochreiter, S., Schmidhuber, J. (1997).
Long short-term memory.
Neural computation, 9(8), 1735-1780.

[4] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017).
Attention is all you need.
Advances in neural information processing systems, 30.

[5] Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., ... Zhou, Y. (2020).
Understanding the Mirai Botnet.
In USENIX Security Symposium (pp. 1093-1110).

[6] Bertino, E., Islam, N. (2022).
Botnets and internet of things security.
Computer, 50(2), 76-79.

[7] Goodfellow, I., Bengio, Y., Courville, A. (2019).
Deep learning.
MIT press.

[8] Schmidl, S., Wenig, P., Papenbrock, T. (2022).
TIMEEVAL : A benchmarking toolkit for time series anomaly detection algorithms.
Proceedings of the VLDB Endowment, 15(8), 1293-1297.

[9] Khraisat, A., Gondal, I., Vamplew, P., Kamruzzaman, J. (2021).
Survey of intrusion detection systems : techniques, datasets and challenges.
Cybersecurity, 4(1), 1-22.