AYI NEDJIMI Consultants

IA Frameworks pour l'Analyse de Malwares - Deep Learning

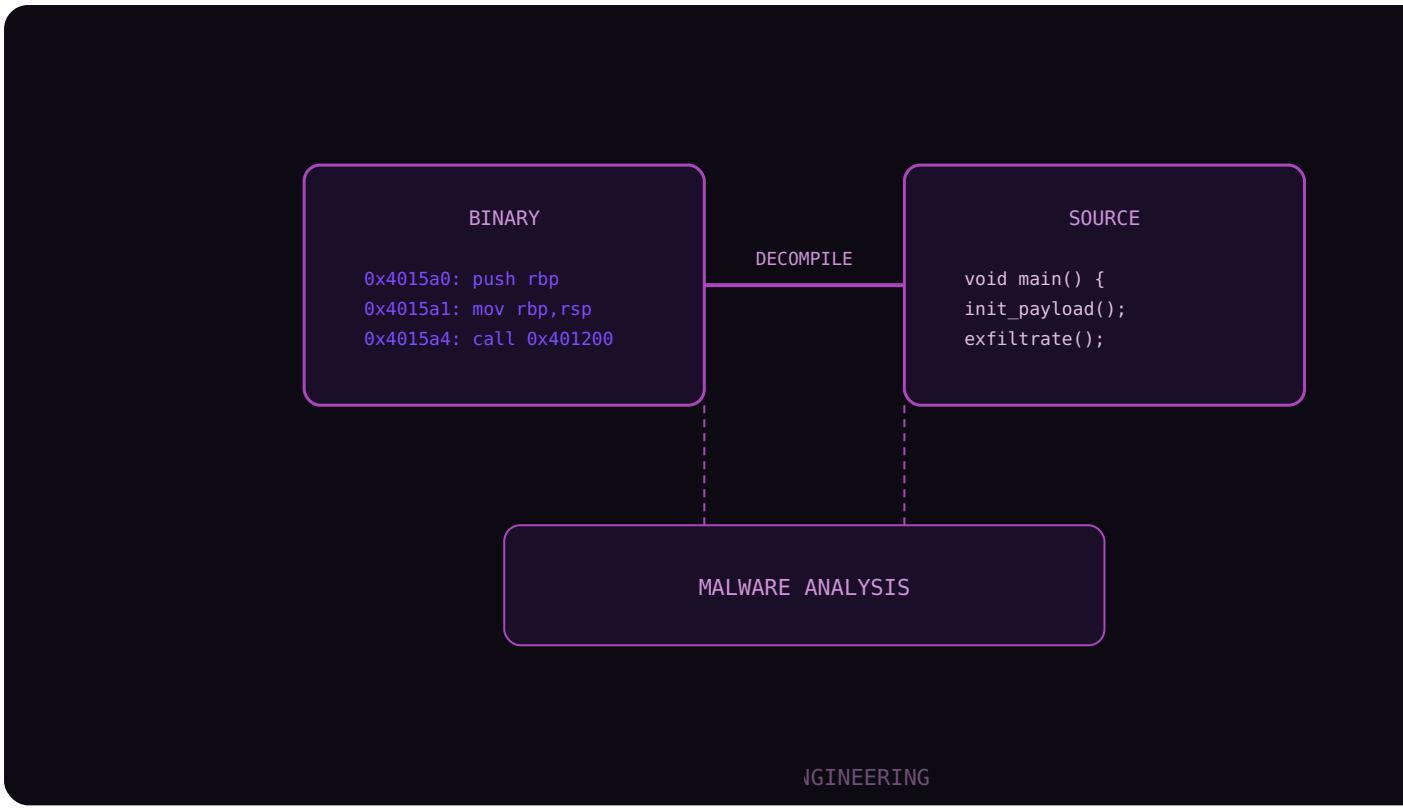
Catégorie : Retro-Ingenierie Lecture : 5 min Publié le : 08/03/2026 Auteur : Ayi NEDJIMI

Frameworks IA pour l'analyse automatisée de malwares : MalConv++, GNN, SOREL-20M, SHAP, clustering Emotet/LockBit, pipeline MLOps. Decouvrez les.

IA Frameworks pour l'Analyse de Malwares - Deep Learning constitue un enjeu majeur pour les professionnels de la sécurité informatique et les équipes techniques. Frameworks IA pour l'analyse automatisée de malwares : MalConv++, GNN, SOREL-20M, SHAP, clustering Emotet/ LockBit, pipeline MLOps. Decouvrez les. Ce guide détaillé sur ia frameworks analyse malwares propose une méthodologie structurée, des outils éprouvés et des recommandations opérationnelles directement applicables. L'objectif est de fournir aux praticiens — consultants, ingénieurs sécurité, administrateurs systèmes — les connaissances et les techniques nécessaires pour aborder ce sujet avec rigueur. Chaque section s'appuie sur des retours d'expérience terrain et intègre les évolutions les plus récentes du domaine. Les recommandations présentées sont adaptées aux environnements d'entreprise et tiennent compte des contraintes opérationnelles réelles.

Avertissement : Les techniques présentées dans cet article sont destinées exclusivement à des fins éducatives et de tests autorisés. Toute utilisation malveillante est illégale et contraire à l'éthique professionnelle.

1. Introduction

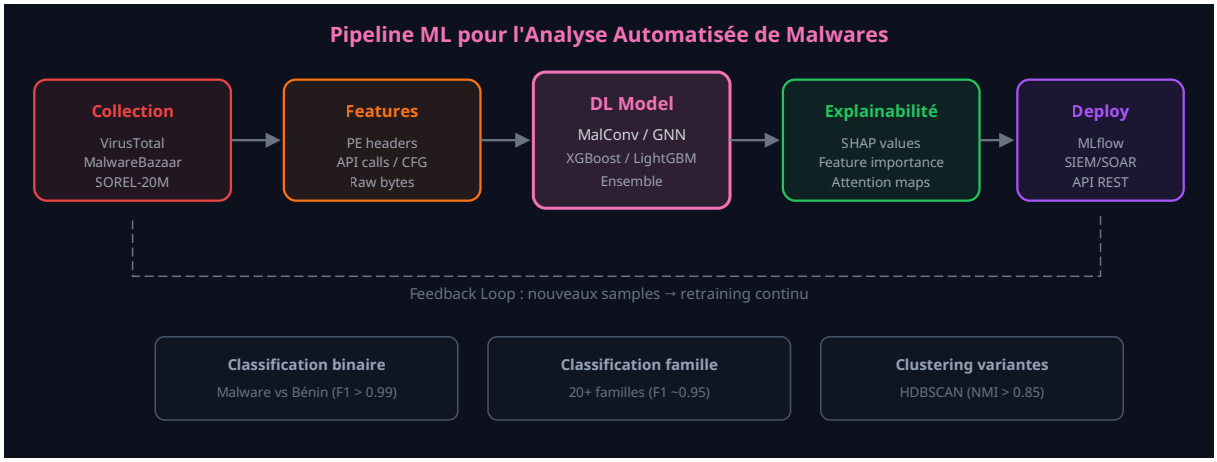


L'analyse manuelle de malwares atteint ses limites face au volume : **AV-TEST** enregistre plus de **450 000 nouveaux malwares par jour** en 2025. Aucune équipe d'analystes, aussi compétente soit-elle, ne peut traiter ce volume. L'intelligence artificielle, et en particulier le **deep learning**, offre la promesse d'une analyse automatisée, scalable et continue des menaces. Ce guide approfondi examine en detail les aspects fondamentaux et avances de IA Frameworks pour l'Analyse de Malwares, en proposant une analyse structuree et documentee des enjeux actuels. Les professionnels y trouveront des recommandations concretes, des methodologies eprouvees et des retours d'experience terrain directement applicables en environnement de production. L'analyse integre les dernieres evolutions technologiques, les tendances emergentes du secteur et les meilleures pratiques recommandees par les experts du domaine. Cet article s'adresse aux ingenieurs, architectes et responsables securite souhaitant approfondir leurs connaissances et renforcer leur posture de securite.

Points cles :

- 1. Introduction
- 2. Feature Engineering pour Malwares
- 3. MalConv et Architectures CNN
- 4. Graph Neural Networks pour CFG
- 5. Dataset SOREL-20M

Cet article explore les frameworks ML/DL les plus avancés pour l'analyse de malwares : des architectures CNN opérant sur les raw bytes (MalConv) aux Graph Neural Networks analysant les Control Flow Graphs, en passant par les techniques d'explainabilité (SHAP) et le clustering non supervisé de familles de malwares. Chaque section inclut du code PyTorch fonctionnel et des résultats expérimentaux sur des datasets réels. Pour approfondir, consultez notre article sur Ghidra Reverse Engineering Guide Debutant. Pour plus d'informations, consultez les ressources de ANSSI.



2. Feature Engineering pour Malwares

Le choix et l'extraction des features sont déterminants pour la performance des modèles ML de détection de malwares. On distingue trois catégories principales : features statiques, dynamiques, et raw bytes. Pour approfondir, consultez notre article sur Fileless Malware Analyse Detection Memoire. Pour plus d'informations, consultez les ressources de MITRE ATT&CK.

```
"""
Extracteur de features statiques pour fichiers PE
Compatible avec le format SOREL-20M et EMBER
"""
import pefile
import hashlib
import math
import numpy as np
from collections import Counter

class PEFeatureExtractor:
    """Extraction de features statiques depuis un PE"""

    def __init__(self, pe_data: bytes):
        self.data = pe_data
        self.pe = pefile.PE(data=pe_data)

    def extract_all(self) -> dict:
        """Extraire toutes les features"""
        features = {}
        features.update(self._header_features())
        features.update(self._section_features())
        features.update(self._import_features())
        features.update(self._entropy_features())
        features.update(self._string_features())
        return features

    def _header_features(self) -> dict:
        """Features du PE header"""
        h = self.pe.FILE_HEADER
        o = self.pe.OPTIONAL_HEADER
        return {
            'machine': h.Machine,
            'num_sections': h.NumberOfSections,
            'timestamp': h.TimeDateStamp,
            'characteristics': h.Characteristics,
            'image_size': o.SizeOfImage,
            'code_size': o.SizeOfCode,
            'entry_point': o.AddressOfEntryPoint,
            'image_base': o.ImageBase,
            'subsystem': o.Subsystem,
            'dll_characteristics': o.DllCharacteristics,
            'stack_reserve': o.SizeOfStackReserve,
            'heap_reserve': o.SizeOfHeapReserve,
            'num_rva_sizes': o.NumberOfRvaAndSizes,
            # Ratio entry point / code size
            'ep_ratio': o.AddressOfEntryPoint / max(o.SizeOfCode, 1),
        }

    def _section_features(self) -> dict:
        """Features par section"""
        features = {}
        for i, section in enumerate(self.pe.sections):
            name = section.Name.decode('utf-8',
                errors='replace').strip('\x00')
            raw = section.SizeOfRawData
            virtual = section.Misc_VirtualSize

            features[f'section_{i}_name'] = name
            features[f'section_{i}_entropy'] = section.get_entropy()
            features[f'section_{i}_raw_size'] = raw
            features[f'section_{i}_virtual_size'] = virtual
```

```
            features[f'section_{i}_ratio'] = raw / max(virtual, 1)
            features[f'section_{i}_characteristics'] = \
                section.Characteristics

        # Métriques agrégées
        entropies = [s.get_entropy() for s in self.pe.sections]
        features['max_section_entropy'] = max(entropies) if entropies else 0
        features['mean_section_entropy'] = np.mean(entropies) if entropies else 0
        features['has_high_entropy_section'] = int(
            any(e > 7.0 for e in entropies))
        return features

    def _import_features(self) -> dict:
        """Features des imports"""
        features = {
            'num_imports': 0,
            'num_import_dlls': 0,
            'suspicious_imports': 0,
        }

        suspicious_apis = {
            'VirtualAlloc', 'VirtualProtect', 'WriteProcessMemory',
            'CreateRemoteThread', 'NtUnmapViewOfSection',
            'SetWindowsHookEx', 'GetAsyncKeyState',
            'InternetOpenA', 'URLDownloadToFile',
            'ShellExecuteA', 'WinExec', 'CreateService',
            'RegSetValueEx', 'CryptEncrypt', 'CryptDecrypt',
        }

        if hasattr(self.pe, 'DIRECTORY_ENTRY_IMPORT'):
            features['num_import_dlls'] = len(
                self.pe.DIRECTORY_ENTRY_IMPORT)
            for entry in self.pe.DIRECTORY_ENTRY_IMPORT:
                for imp in entry.imports:
                    features['num_imports'] += 1
                    if imp.name and imp.name.decode(
                        'utf-8', errors='ignore') in suspicious_apis:
                        features['suspicious_imports'] += 1

        return features

    def _entropy_features(self) -> dict:
        """Entropie globale du fichier"""
        byte_counts = Counter(self.data)
        total = len(self.data)
        entropy = 0.0
        for count in byte_counts.values():
            p = count / total
            if p > 0:
                entropy -= p * math.log2(p)

        return {
            'file_entropy': entropy,
            'file_size': total,
            'is_packed': int(entropy > 7.0),
        }

    def _string_features(self) -> dict:
        """Features basées sur les strings"""
        import re
        strings = re.findall(rb'[\x20-\x7e]{4,}', self.data)
```

```
        urls = [s for s in strings if b'http' in s.lower()]
        ips = [s for s in strings
               if re.match(rb'\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}', s)]

        return {
            'num_strings': len(strings),
            'avg_string_len': np.mean([len(s) for s in strings])
                              if strings else 0,
            'num_urls': len(urls),
            'num_ips': len(ips),
            'has_urls': int(len(urls) > 0),
        }
```

Votre sandbox d'analyse est-elle suffisamment isolée pour exécuter des échantillons malveillants en toute sécurité ?

3. MalConv et Architectures CNN

MalConv (Raff et al., 2018) est une architecture CNN pionnière qui opère directement sur les **raw bytes** du fichier PE, sans extraction de features manuelle. Le modèle apprend à identifier les patterns malveillants directement depuis la séquence d'octets. Pour approfondir, consultez notre article sur Reverse Engineering Dotnet Decompilation Analyse.

3.1 Architecture MalConv++ en PyTorch

```
"""
MalConv++ : Architecture CNN améliorée pour classification
de malwares sur raw bytes
"""
import torch
import torch.nn as nn
import torch.nn.functional as F

class MalConvPlusPlus(nn.Module):
    """
    MalConv++ avec :
    - Embedding 8D pour les 256 valeurs de bytes
    - Gated convolutions multi-échelles
    - Global max pooling + attention
    - Classification binaire (malware/bénin)
    """
    def __init__(self, max_length=2000000, embed_dim=8,
                 num_filters=128, kernel_sizes=[500, 1000, 2000],
                 num_classes=2):
        super().__init__()

        self.max_length = max_length

        # Embedding : 257 valeurs (256 bytes + padding)
        self.embed = nn.Embedding(257, embed_dim, padding_idx=256)

        # Gated convolutions multi-échelles
        self.conv_layers = nn.ModuleList()
        self.gate_layers = nn.ModuleList()

        for ks in kernel_sizes:
            self.conv_layers.append(
                nn.Conv1d(embed_dim, num_filters, ks, stride=ks//2)
            )
            self.gate_layers.append(
                nn.Conv1d(embed_dim, num_filters, ks, stride=ks//2)
            )

        total_filters = num_filters * len(kernel_sizes)

        # Attention mechanism
        self.attention = nn.Sequential(
            nn.Linear(total_filters, total_filters // 4),
            nn.Tanh(),
            nn.Linear(total_filters // 4, 1)
        )

        # Classifier
        self.classifier = nn.Sequential(
            nn.Linear(total_filters, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(256, 64),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(64, num_classes)
        )

    def forward(self, x):
        # x : [batch, seq_len] (long tensor of byte values)
```

```
        # Padding/truncation to max_length
        if x.size(1) > self.max_length:
            x = x[:, :self.max_length]
        elif x.size(1) < self.max_length:
            pad = torch.full(
                (x.size(0), self.max_length - x.size(1)),
                256, dtype=torch.long, device=x.device)
            x = torch.cat([x, pad], dim=1)

        # Embedding : [batch, seq_len, embed_dim]
        emb = self.embed(x)
        # Transpose for Conv1d : [batch, embed_dim, seq_len]
        emb = emb.transpose(1, 2)

        # Multi-scale gated convolutions
        pooled_outputs = []
        for conv, gate in zip(self.conv_layers, self.gate_layers):
            conv_out = conv(emb)
            gate_out = torch.sigmoid(gate(emb))
            gated = conv_out * gate_out  # Element-wise gating

            # Global max pooling over time dimension
            pooled = F.adaptive_max_pool1d(gated, 1).squeeze(-1)
            pooled_outputs.append(pooled)

        # Concatenate multi-scale features
        combined = torch.cat(pooled_outputs, dim=1)

        # Classification
        logits = self.classifier(combined)
        return logits

# Entraînement
def train_malconv(model, train_loader, epochs=20, lr=1e-3):
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
        optimizer, T_max=epochs)

    for epoch in range(epochs):
        model.train()
        total_loss = 0
        correct = 0
        total = 0

        for batch_x, batch_y in train_loader:
            batch_x = batch_x.cuda()
            batch_y = batch_y.cuda()

            optimizer.zero_grad()
            logits = model(batch_x)
            loss = criterion(logits, batch_y)
            loss.backward()

            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            optimizer.step()

            total_loss += loss.item()
            _, predicted = logits.max(1)
            correct += predicted.eq(batch_y).sum().item()
            total += batch_y.size(0)
```

```
        scheduler.step()
        acc = 100. * correct / total
        print(f"Epoch {epoch+1}/{epochs} | "
              f"Loss: {total_loss/len(train_loader):.4f} | "
              f"Acc: {acc:.2f}%")
```

Notre avis d'expert
Les techniques d'anti-analyse deviennent de plus en plus sophistiquées. Les packers, le code polymorphe et les checks d'environnement virtuel compliquent considérablement le travail d'analyse. La maîtrise des outils de désobfuscation est devenue indispensable.

4. Graph Neural Networks pour CFG

Les **Graph Neural Networks (GNN)** sont particulièrement adaptés à l'analyse de malwares car le **Control Flow Graph (CFG)** d'un binaire est naturellement un graphe. Chaque noeud est un basic block, chaque arête un transfert de contrôle. Pour approfondir, consultez notre article sur Anti Retro Ingenierie Apt.

4.1 Représentation du CFG en graphe

```
"""
Construction d'un graphe de CFG pour analyse GNN
Utilise angr pour l'extraction et PyTorch Geometric pour le GNN
"""
import angr
import networkx as nx
import numpy as np
import torch
from torch_geometric.data import Data

class CFGGraphBuilder:
    """Construit un graphe PyG depuis un binaire"""

    # Opcodes groupés par catégorie sémantique
    OPCODE_CATEGORIES = {
        'arithmetic': ['add', 'sub', 'mul', 'div', 'inc', 'dec',
                       'imul', 'idiv', 'neg'],
        'logic': ['and', 'or', 'xor', 'not', 'shl', 'shr',
                  'sar', 'rol', 'ror'],
        'transfer': ['mov', 'lea', 'push', 'pop', 'xchg',
                     'movzx', 'movsx', 'cmov'],
        'control': ['jmp', 'je', 'jne', 'jg', 'jl', 'jge',
                    'jle', 'call', 'ret', 'loop'],
        'compare': ['cmp', 'test'],
        'string': ['rep', 'movs', 'stos', 'lods', 'cmps', 'scas'],
        'system': ['int', 'syscall', 'sysenter', 'cpuid',
                   'rdtsc', 'in', 'out'],
        'nop': ['nop'],
        'crypto': ['aesenc', 'aesdec', 'pclmulqdq'],
    }

    def __init__(self, binary_path):
        self.proj = angr.Project(binary_path,
                                 auto_load_libs=False)

    def build_graph(self, max_nodes=500):
        """Construire le graphe depuis le CFG"""
        cfg = self.proj.analyses.CFGFast()
        nx_graph = cfg.graph

        # Limiter le nombre de noeuds
        nodes = list(nx_graph.nodes())[:max_nodes]
        node_to_idx = {n: i for i, n in enumerate(nodes)}

        # Features de chaque noeud (basic block)
        node_features = []
        for node in nodes:
            features = self._extract_block_features(node)
            node_features.append(features)

        # Matrice de features
        x = torch.tensor(np.array(node_features),
                         dtype=torch.float)

        # Arêtes
        edges = []
        for u, v in nx_graph.edges():
            if u in node_to_idx and v in node_to_idx:
                edges.append([node_to_idx[u], node_to_idx[v]])

        if not edges:
            edge_index = torch.zeros((2, 0), dtype=torch.long)
```

```
        else:
            edge_index = torch.tensor(edges,
                dtype=torch.long).t().contiguous()

        return Data(x=x, edge_index=edge_index)

    def _extract_block_features(self, block, feature_dim=32):
        """Extraire les features d'un basic block"""
        features = np.zeros(feature_dim)

        try:
            # Nombre d'instructions
            num_insns = block.instructions
            features[0] = min(num_insns / 50.0, 1.0)

            # Taille du bloc
            features[1] = min(block.size / 500.0, 1.0)

            # Distribution des catégories d'opcodes
            if hasattr(block, 'capstone') and block.capstone:
                insns = list(block.capstone.insns)
                for insn in insns:
                    mnemonic = insn.mnemonic.lower()
                    for cat_idx, (cat, opcodes) in enumerate(
                        self.OPCODE_CATEGORIES.items()):
                        if any(mnemonic.startswith(op)
                               for op in opcodes):
                            if cat_idx + 2 < feature_dim:
                                features[cat_idx + 2] += 1.0
                            break

                # Normaliser les compteurs
                total = len(insns) or 1
                features[2:11] /= total

            # Nombre de successeurs (out-degree)
            features[11] = min(len(list(block.successors())) / 5.0,
                               1.0)

        except Exception:
            pass

        return features
```

4.2 Architecture GNN pour classification de malwares

```
"""
Graph Attention Network (GAT) pour classification
de malwares basée sur le CFG
"""
import torch
import torch.nn as nn
from torch_geometric.nn import GATConv, global_mean_pool
from torch_geometric.nn import global_max_pool

class MalwareGAT(nn.Module):
    """
    Graph Attention Network pour classification de malwares
    Input : graphe du CFG (noeuds = basic blocks)
    Output : classification multi-classes (familles)
    """
    def __init__(self, input_dim=32, hidden_dim=64,
                 num_heads=4, num_classes=10, dropout=0.3):
        super().__init__()

        # GAT layers avec multi-head attention
        self.gat1 = GATConv(input_dim, hidden_dim,
                            heads=num_heads, dropout=dropout)
        self.gat2 = GATConv(hidden_dim * num_heads, hidden_dim,
                            heads=num_heads, dropout=dropout)
        self.gat3 = GATConv(hidden_dim * num_heads, hidden_dim,
                            heads=1, concat=False, dropout=dropout)

        # Batch normalization
        self.bn1 = nn.BatchNorm1d(hidden_dim * num_heads)
        self.bn2 = nn.BatchNorm1d(hidden_dim * num_heads)

        # Readout + classifier
        self.classifier = nn.Sequential(
            nn.Linear(hidden_dim * 2, 128),  # *2 pour mean+max pool
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(128, num_classes)
        )

    def forward(self, data):
        x, edge_index, batch = data.x, data.edge_index, data.batch

        # GAT layer 1
        x = self.gat1(x, edge_index)
        x = self.bn1(x)
        x = nn.functional.elu(x)

        # GAT layer 2
        x = self.gat2(x, edge_index)
        x = self.bn2(x)
        x = nn.functional.elu(x)

        # GAT layer 3
        x = self.gat3(x, edge_index)
        x = nn.functional.elu(x)

        # Dual readout : mean + max pooling
        x_mean = global_mean_pool(x, batch)
        x_max = global_max_pool(x, batch)
        x = torch.cat([x_mean, x_max], dim=1)
```

```
        # Classification
        return self.classifier(x)
```

5. Dataset SOREL-20M

SOREL-20M (Sophos/ReversingLabs, 2020) est le plus grand dataset public de malwares avec labels, contenant **20 millions de samples PE** avec features pré-extraites et métadonnées. C'est le benchmark de référence pour les modèles ML de détection.

```
"""
Chargement et entraînement sur SOREL-20M
avec LightGBM pour une baseline performante
"""
import lightgbm as lgb
import numpy as np
from sklearn.metrics import (
    classification_report, roc_auc_score, f1_score
)

class SORELTrainer:
    """Pipeline d'entraînement sur SOREL-20M"""

    def __init__(self, data_dir):
        self.data_dir = data_dir

    def load_features(self, split='train', max_samples=1000000):
        """
        Charger les features pré-extraites SOREL-20M
        Format : features EMBER-like (2381 dimensions)
        """
        features_path = f"{self.data_dir}/{split}_features.npz"
        labels_path = f"{self.data_dir}/{split}_labels.npz"

        X = np.load(features_path)['features'][:max_samples]
        y = np.load(labels_path)['labels'][:max_samples]

        return X, y

    def train_lightgbm(self, X_train, y_train, X_val, y_val):
        """Entraîner un modèle LightGBM"""

        params = {
            'objective': 'binary',
            'metric': ['binary_logloss', 'auc'],
            'boosting_type': 'gbdt',
            'num_leaves': 2048,
            'learning_rate': 0.05,
            'feature_fraction': 0.8,
            'bagging_fraction': 0.8,
            'bagging_freq': 5,
            'min_child_samples': 50,
            'reg_alpha': 0.1,
            'reg_lambda': 0.1,
            'num_threads': -1,
            'verbose': -1,
        }

        train_data = lgb.Dataset(X_train, label=y_train)
        val_data = lgb.Dataset(X_val, label=y_val,
                               reference=train_data)

        model = lgb.train(
            params,
            train_data,
            num_boost_round=5000,
            valid_sets=[train_data, val_data],
            valid_names=['train', 'val'],
            callbacks=[
                lgb.early_stopping(50),
                lgb.log_evaluation(100)
            ]
```

```
        )

        return model

    def evaluate(self, model, X_test, y_test):
        """Évaluation complète du modèle"""
        y_pred_proba = model.predict(X_test)
        y_pred = (y_pred_proba > 0.5).astype(int)

        metrics = {
            'auc_roc': roc_auc_score(y_test, y_pred_proba),
            'f1': f1_score(y_test, y_pred),
            'report': classification_report(
                y_test, y_pred,
                target_names=['Benign', 'Malware']
            ),
        }

        # Faux positifs à différents seuils
        for threshold in [0.01, 0.001, 0.0001]:
            fp_rate = np.mean(y_pred_proba[y_test == 0] > 0.5)
            metrics[f'fp_rate'] = fp_rate

        return metrics

# Usage typique
# trainer = SORELTrainer("/data/sorel-20m/")
# X_train, y_train = trainer.load_features('train')
# X_val, y_val = trainer.load_features('val', max_samples=200000)
# model = trainer.train_lightgbm(X_train, y_train, X_val, y_val)
# X_test, y_test = trainer.load_features('test', max_samples=200000)
# metrics = trainer.evaluate(model, X_test, y_test)
# print(f"AUC-ROC: {metrics['auc_roc']:.4f}")
# print(f"F1: {metrics['f1']:.4f}")
# print(metrics['report'])
```

Cas concret
L'analyse du wiper HermeticWiper, déployé contre des organisations ukrainiennes en février 2022, a révélé l'utilisation d'un driver légitime de partitionnement pour corrompre le MBR et les partitions NTFS. La rétro-ingénierie rapide par ESET et SentinelOne a permis de publier des indicateurs de compromission en moins de 24 heures.

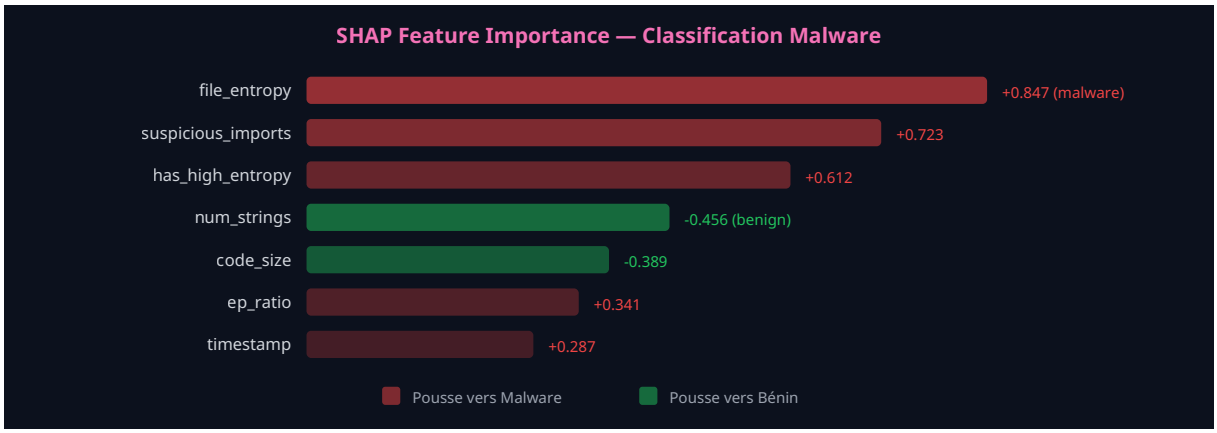
Disposez-vous en interne des compétences de rétro-ingénierie nécessaires pour analyser un malware ciblant votre organisation ?

6. Explainabilité avec SHAP/LIME

L'explainabilité est cruciale en analyse de malwares : un analyste doit comprendre **pourquoi** un modèle classifie un fichier comme malveillant, pas seulement le résultat. **SHAP** (SHapley Additive exPlanations) fournit des valeurs de contribution pour chaque feature.

```
"""
Explainabilité des décisions de classification malware
avec SHAP et visualisation
"""
import shap
import numpy as np
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

class MalwareExplainer:
    """Expliquer les décisions du modèle de détection"""

    FEATURE_NAMES = [
        'file_size', 'file_entropy', 'num_sections',
        'max_section_entropy', 'has_high_entropy',
        'num_imports', 'suspicious_imports',
        'num_strings', 'num_urls', 'has_urls',
        'ep_ratio', 'code_size', 'image_size',
        'timestamp', 'dll_characteristics',
        'num_import_dlls', 'stack_reserve',
        # ... (2381 features au total pour EMBER/SOREL)
    ]

    def __init__(self, model, X_background):
        """
        model : modèle LightGBM ou sklearn
        X_background : échantillon de référence (100-500 samples)
        """
        self.explainer = shap.TreeExplainer(model)
        self.X_background = X_background

    def explain_sample(self, sample, feature_names=None):
        """Expliquer la classification d'un sample"""
        if feature_names is None:
            feature_names = self.FEATURE_NAMES

        shap_values = self.explainer.shap_values(
            sample.reshape(1, -1))

        # Top features contributives
        if isinstance(shap_values, list):
            sv = shap_values[1][0]  # Classe malware
        else:
            sv = shap_values[0]

        top_indices = np.argsort(np.abs(sv))[-10:][::-1]

        explanation = {
            'base_value': self.explainer.expected_value,
            'prediction': sv.sum() + (
                self.explainer.expected_value[1]
                if isinstance(self.explainer.expected_value, list)
                else self.explainer.expected_value
            ),
            'top_features': [
                {
                    'name': feature_names[i]
                            if i < len(feature_names)
                            else f'feature_{i}',
                    'value': float(sample[i]),
                    'shap_value': float(sv[i]),
```

```
                    'direction': 'malware' if sv[i] > 0 else 'benign'
                }
                for i in top_indices
            ]
        }

        return explanation

    def generate_report(self, sample, explanation):
        """Générer un rapport textuel d'explication"""
        report = []
        report.append("=== Rapport d'Analyse ML ===\n")

        pred = explanation['prediction']
        label = "MALWARE" if pred > 0 else "BÉNIN"
        confidence = abs(pred)

        report.append(f"Verdict : {label} "
                      f"(confiance : {confidence:.2f})\n")
        report.append("\nFacteurs déterminants :")

        for feat in explanation['top_features'][:5]:
            direction = "+" if feat['direction'] == 'malware' \
                        else "-"
            report.append(
                f"  {direction} {feat['name']}: "
                f"{feat['value']:.4f} "
                f"(impact: {feat['shap_value']:+.4f})"
            )

        return "\n".join(report)
```



7. Clustering de Familles de Malwares

```
"""
Clustering non supervisé de familles de malwares
UMAP pour la réduction de dimension + HDBSCAN pour le clustering
"""
import numpy as np
import umap
import hdbscan
from sklearn.metrics import normalized_mutual_info_score

class MalwareFamilyClustering:
    """Pipeline de clustering de familles de malwares"""

    def __init__(self, n_neighbors=15, min_dist=0.1,
                 min_cluster_size=10):
        self.reducer = umap.UMAP(
            n_neighbors=n_neighbors,
            min_dist=min_dist,
            n_components=2,
            metric='cosine',
            random_state=42
        )
        self.clusterer = hdbscan.HDBSCAN(
            min_cluster_size=min_cluster_size,
            min_samples=5,
            metric='euclidean',
            cluster_selection_method='eom'
        )

    def fit_predict(self, features, true_labels=None):
        """
        features : [N, D] matrice de features
        true_labels : labels réels (optionnel, pour évaluation)
        """
        # Réduction de dimension
        embedding = self.reducer.fit_transform(features)

        # Clustering
        clusters = self.clusterer.fit_predict(embedding)

        results = {
            'embedding': embedding,
            'clusters': clusters,
            'num_clusters': len(set(clusters)) - (1 if -1 in clusters else 0),
            'noise_ratio': np.mean(clusters == -1),
        }

        if true_labels is not None:
            # Filtrer le bruit pour le NMI
            mask = clusters != -1
            results['nmi'] = normalized_mutual_info_score(
                true_labels[mask], clusters[mask])

        return results

    def generate_cluster_signatures(self, features,
                                    clusters, feature_names):
        """
        Générer des signatures par cluster
        (features discriminantes)
        """
        unique_clusters = sorted(set(clusters) - {-1})
        signatures = {}
```

```
        global_mean = np.mean(features, axis=0)

        for cluster_id in unique_clusters:
            mask = clusters == cluster_id
            cluster_features = features[mask]
            cluster_mean = np.mean(cluster_features, axis=0)

            # Features les plus discriminantes
            diff = np.abs(cluster_mean - global_mean)
            top_indices = np.argsort(diff)[-10:][::-1]

            signatures[cluster_id] = {
                'size': int(mask.sum()),
                'top_features': [
                    {
                        'name': feature_names[i]
                                if i < len(feature_names)
                                else f'feature_{i}',
                        'cluster_mean': float(cluster_mean[i]),
                        'global_mean': float(global_mean[i]),
                        'diff': float(diff[i])
                    }
                    for i in top_indices
                ]
            }

        return signatures
```

8. Étude de Cas : LockBit Ransomware

LockBit est le ransomware le plus actif de 2024-2025, responsable de plus de 30% de toutes les attaques ransomware documentées. Son modèle RaaS (Ransomware-as-a-Service) génère de nombreuses variantes, rendant le clustering ML particulièrement pertinent.

```
"""
Pipeline ML complet pour l'analyse de LockBit
De la collecte des samples à la génération de signatures
"""
import os
import json

class LockBitAnalysisPipeline:
    """Pipeline d'analyse LockBit avec ML"""

    # Marqueurs LockBit connus
    LOCKBIT_MARKERS = {
        'mutex': [
            'Global\\{BEF590BE-11A6-442A-A85B-656C78A67A4C}',
            'Global\\{3E5FC7F9-9A51-4367-9063-A120244FBEC7}',
        ],
        'extensions': [
            '.lockbit', '.lock2', '.abcd',
            '.lockbit3', '.lockbit30',
        ],
        'ransom_note': [
            'Restore-My-Files.txt',
            'LockBit-note.hta',
            'lockbit-decryption-note.txt',
        ],
        'registry': [
            r'SOFTWARE\LockBit',
            r'SOFTWARE\Policies\Microsoft\Windows Defender',
        ],
    }

    def classify_variant(self, sample_features):
        """
        Classifier la version de LockBit
        - LockBit 1.0 : chiffrement AES + RSA, note .abcd
        - LockBit 2.0 : multithreaded, auto-propagation
        - LockBit 3.0 (Black) : anti-debug, config chiffrée
        """
        version_features = {
            '1.0': {
                'has_mutex_v1': False,
                'extension': '.abcd',
                'encryption': 'AES-256-ECB',
            },
            '2.0': {
                'has_mutex_v2': False,
                'extension': '.lockbit',
                'has_worm': True,
                'encryption': 'AES-256 + ECDH',
            },
            '3.0': {
                'has_antidbg': True,
                'extension': '.lockbit30',
                'config_encrypted': True,
                'encryption': 'ChaCha20 + RSA-2048',
            }
        }

        scores = {}
        for version, markers in version_features.items():
            score = 0
            for key, expected in markers.items():
```

```
                if key in sample_features:
                    if sample_features[key] == expected:
                        score += 1
            scores[version] = score

        best_version = max(scores, key=scores.get)
        return best_version, scores

    def extract_config(self, decrypted_config):
        """Extraire la configuration LockBit 3.0"""
        config = {}

        try:
            # LockBit 3.0 stocke sa config en JSON chiffré
            # après déchiffrement RC4 (clé dans .rdata)
            parsed = json.loads(decrypted_config)

            config['c2_urls'] = parsed.get('urls', [])
            config['extension'] = parsed.get('ext', '.lockbit30')
            config['ransom_note'] = parsed.get('note', '')
            config['kill_services'] = parsed.get('svc', [])
            config['kill_processes'] = parsed.get('prc', [])
            config['skip_dirs'] = parsed.get('skip_dirs', [])
            config['skip_files'] = parsed.get('skip_files', [])
            config['wallpaper'] = parsed.get('wp', False)

        except json.JSONDecodeError:
            config['error'] = 'Config not JSON format'

        return config
```

9. Étude de Cas : Emotet

Emotet est le botnet/loader le plus résilient de la décennie, ayant survécu à plusieurs takedowns coordonnés par Europol. Son polymorphisme extrême (nouvelles variantes toutes les heures) en fait un cas d'école pour le tracking par ML.

```
"""
Tracking d'Emotet par clustering temporel
Identifier les nouvelles campagnes et variantes
"""
from datetime import datetime, timedelta
import numpy as np
from sklearn.ensemble import IsolationForest

class EmotetTracker:
    """Suivi des variantes Emotet par ML"""

    def __init__(self):
        self.known_clusters = {}
        self.anomaly_detector = IsolationForest(
            n_estimators=200,
            contamination=0.1,
            random_state=42
        )
        self.history = []

    def process_new_sample(self, features, metadata):
        """
        Traiter un nouveau sample Emotet
        Returns: cluster_id, is_new_variant, confidence
        """
        # 1. Vérifier si c'est une variante connue
        best_match = None
        best_score = -1

        for cluster_id, cluster_data in self.known_clusters.items():
            centroid = cluster_data['centroid']
            similarity = self._cosine_similarity(features, centroid)
            if similarity > best_score:
                best_score = similarity
                best_match = cluster_id

        # 2. Seuil de nouveauté
        is_new = best_score < 0.85

        if is_new:
            # Nouveau cluster
            new_id = f"emotet_v{len(self.known_clusters) + 1}"
            self.known_clusters[new_id] = {
                'centroid': features,
                'samples': [metadata],
                'first_seen': datetime.now(),
                'count': 1
            }
            cluster_id = new_id
        else:
            # Mettre à jour le cluster existant
            cluster = self.known_clusters[best_match]
            # Moving average du centroid
            n = cluster['count']
            cluster['centroid'] = (
                cluster['centroid'] * n + features) / (n + 1)
            cluster['count'] += 1
            cluster['samples'].append(metadata)
            cluster_id = best_match

        self.history.append({
            'timestamp': datetime.now().isoformat(),
```

```
            'cluster': cluster_id,
            'is_new': is_new,
            'similarity': float(best_score),
            'sha256': metadata.get('sha256', 'unknown')
        })

        return cluster_id, is_new, best_score

    def detect_campaign_shift(self, window_hours=24):
        """Détecter un changement de campagne"""
        cutoff = datetime.now() - timedelta(hours=window_hours)
        recent = [h for h in self.history
                  if datetime.fromisoformat(h['timestamp']) > cutoff]

        if len(recent) < 5:
            return None

        new_count = sum(1 for h in recent if h['is_new'])
        new_ratio = new_count / len(recent)

        if new_ratio > 0.5:
            return {
                'alert': 'CAMPAIGN_SHIFT',
                'new_variants': new_count,
                'total_samples': len(recent),
                'new_ratio': new_ratio,
                'message': f"Possible nouvelle campagne Emotet : "
                           f"{new_count}/{len(recent)} variants "
                           f"inconnues ({new_ratio:.0%})"
            }
        return None

    @staticmethod
    def _cosine_similarity(a, b):
        dot = np.dot(a, b)
        norm = np.linalg.norm(a) * np.linalg.norm(b)
        return dot / max(norm, 1e-8)
```

10. Déploiement et MLOps

```
# mlflow_config.yaml
# Configuration MLflow pour le pipeline de détection malware

experiment_name: "malware-detection-v3"

model:
  name: "malware-classifier"
  type: "lightgbm"
  version: "3.2.1"

training:
  dataset: "sorel-20m"
  train_size: 10000000
  val_size: 2000000
  test_size: 2000000
  features: "ember_v2"
  feature_dim: 2381

hyperparameters:
  num_leaves: 2048
  learning_rate: 0.05
  num_boost_round: 5000
  early_stopping_rounds: 50
  feature_fraction: 0.8
  bagging_fraction: 0.8

serving:
  endpoint: "/api/v1/predict"
  batch_size: 100
  timeout_ms: 500
  gpu: false

monitoring:
  drift_check_interval: "6h"
  retrain_threshold:
    auc_drop: 0.02
    fp_rate_increase: 0.005
  alerts:
    slack_channel: "#ml-malware-alerts"
    email: "soc@company.com"

retraining:
  schedule: "weekly"
  min_new_samples: 50000
  auto_deploy: false  # Require human approval
  validation_gate:
    min_auc: 0.995
    max_fp_rate: 0.001
```

```
#!/bin/bash
# Script de déploiement du modèle de détection malware

# 1. Entraîner et logger avec MLflow
mlflow run . \
    --experiment-name "malware-detection-v3" \
    -P dataset=sorel-20m \
    -P num_leaves=2048

# 2. Enregistrer le meilleur modèle
mlflow models register \
    --name "malware-classifier" \
    --source "runs:/<run_id>/model"

# 3. Promouvoir en production (après validation)
mlflow models transition \
    --name "malware-classifier" \
    --version 3 \
    --stage Production

# 4. Servir via REST API
mlflow models serve \
    --model-uri "models:/malware-classifier/Production" \
    --port 5001 \
    --host 0.0.0.0

# 5. Tester l'endpoint
curl -X POST http://localhost:5001/invocations \
    -H "Content-Type: application/json" \
    -d '{"dataframe_split": {
        "columns": ["f0","f1","f2"],
        "data": [[7.2, 12, 0.85]]
    }}'
```

1. Introduction	2. Feature Engineering pour Malwares	3. MalConv et Architectures CNN
Implementation	Renforcement de la securite globale	Complexite de mise en oeuvre
Monitoring	Detection proactive des menaces	Ressources necessaires
Conformite	Alignement aux referentiels	Cout de certification

Pour approfondir ce sujet, consultez notre outil open-source reverse-engineering-scripts qui facilite l'assistance à la rétro-ingénierie de binaires.

Questions frequentes

Comment mettre en place IA Frameworks pour l'Analyse de Malwares dans un environnement de production ?

La mise en place de IA Frameworks pour l'Analyse de Malwares en production necessite une planification rigoureuse, incluant l'evaluation des prerequis techniques, la definition d'une architecture cible, des tests de validation approfondis et un plan de deploiement progressif avec des points de controle a chaque etape.

Pourquoi IA Frameworks pour l'Analyse de Malwares est-il essentiel pour la securite des systemes d'information ?

IA Frameworks pour l'Analyse de Malwares constitue un element fondamental de la securite des systemes d'information car il permet de reduire significativement la surface d'attaque, d'ameliorer la detection des menaces et de renforcer la posture globale de securite de l'organisation face aux cybermenaces actuelles.

Quelles sont les bonnes pratiques pour IA Frameworks pour l'Analyse de Malwares en 2026 ?

Les bonnes pratiques pour IA Frameworks pour l'Analyse de Malwares en 2026 incluent l'adoption d'une approche Zero Trust, l'automatisation des controles de securite, la mise en place d'une veille continue sur les vulnerabilites et l'integration des recommandations des organismes de reference comme l'ANSSI et le NIST.

Sources et références : MITRE ATT&CK · CERT-FR

11. Conclusion

L'intelligence artificielle transforme fondamentalement l'analyse de malwares, passant d'une discipline artisanale à une ingénierie scalable. Les frameworks présentés dans cet article — **MalConv++** pour les raw bytes, **GAT** pour les CFG, **LightGBM** sur SOREL-20M, et **HDBSCAN** pour le clustering — constituent un arsenal complet pour l'analyste moderne.

Les tendances émergentes à surveiller :

- **LLM pour la RE** : GPT-4 et Claude démontrent des capacités remarquables en décompilation et explication de code assembleur. L'avenir verra probablement des assistants IA intégrés directement dans IDA Pro et Ghidra
- **Analyse autonome** : des agents IA capables d'exécuter un pipeline complet (triage → analyse statique → analyse dynamique → rapport) sans intervention humaine
- **Adversarial ML** : les attaquants utilisent déjà des techniques adversariales (feature-space attacks, gradient-based evasion) pour contourner les modèles de détection. La robustesse adversariale devient un enjeu critique
- **Federated Learning** : partage de modèles entre organisations sans partager les samples malveillants (contraintes de confidentialité et légales)

Recommandation : Commencez par une baseline LightGBM sur les features EMBER/SOREL avant d'investir dans des architectures deep learning plus complexes. Dans la majorité des cas, le gradient boosting avec un bon feature engineering surpasse les réseaux de neurones, avec un coût d'inférence 100x inférieur.

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

ayinedjimi-consultants.fr · ayi@ayinedjimi-consultants.fr