An Efficient SQL Injection Detection using Hybrid CNN and RF Algorithm
Generated: 2025-08-29 12:43:21
This PDF mirrors the GitHub-ready Python script `hybrid_cnn_rf_sql_injection.py`.
Copy the .py into your repo, or upload this PDF alongside it.

```
--------------------------------------------------------------------------------

"""
Title: An Efficient SQL Injection Detection using Hybrid CNN and RF Algorithm
Author: Your Name
Repo-ready: Yes (single-file script + saved artifacts)
Python: 3.9+

Overview
--------
This script demonstrates a hybrid approach for SQL injection detection on raw query strings:
1) A lightweight **character-level CNN** in PyTorch learns dense representations of queries.
2) The **pooled CNN embeddings** are exported and consumed by a **RandomForest** classifier from
scikit-learn.
This combines CNN's ability to capture local n-gram patterns with RandomForest's robustness on
tabular features.

What you get
------------
- End-to-end training on a tiny demo dataset (replace with your own data).
- Clean modular code (you can swap the dataset loader easily).
- Metrics: accuracy, precision, recall, F1.
- Saved artifacts: `cnn_encoder.pt`, `rf_model.joblib`, and `label_encoder.joblib`.

Data format
-----------
Expect a CSV with two columns:
- `query`: str — the raw SQL (or HTTP parameter) text
- `label`: int or str — 1 for "injection", 0 for "benign" (strings will be label-encoded)

If you don't have data yet, the script will fall back to a small synthetic sample to illustrate the
pipeline.

Usage
-----
# install
pip install torch scikit-learn joblib pandas numpy

# run
python hybrid_cnn_rf_sql_injection.py --data data/queries.csv --epochs 5 --batch-size 64

# predict (example)
python hybrid_cnn_rf_sql_injection.py --predict "SELECT * FROM users WHERE name='admin' OR '1'='1';"
```

```
Notes
-----
- For a real project, provide thousands of labeled examples.
- Tune CNN width/filters and RF hyperparams.
- Consider class imbalance handling (e.g., class_weight, focal loss, sampling).
"""

import argparse
import os
import math
import random
from typing import List, Tuple, Optional

import numpy as np
import pandas as pd

import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_recall_fscore_support, classification_report
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder
from joblib import dump, load

# --------------------------
# Reproducibility
# --------------------------
def set_seed(seed: int = 42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

# --------------------------
# Character Vocabulary
# --------------------------
def build_char_vocab(texts: List[str]) -> Tuple[dict, dict]:
    chars = set()
    for t in texts:
        chars.update(list(t))
    # Reserve 0 for PAD, 1 for UNK
    itos = ['<PAD>', '<UNK>'] + sorted(chars)
    stoi = {ch: i for i, ch in enumerate(itos)}
    return stoi, itos
```

```
def encode_text(text: str, stoi: dict, max_len: int) -> List[int]:
    ids = [stoi.get(ch, 1) for ch in text][:max_len]  # 1 is UNK
    if len(ids) < max_len:
        ids += [0] * (max_len - len(ids))             # 0 is PAD
    return ids

# --------------------------
# Dataset
# --------------------------
class QueryDataset(Dataset):
    def __init__(self, df: pd.DataFrame, stoi: dict, max_len: int, label_encoder:
Optional[LabelEncoder] = None, fit_le: bool = False):
        assert 'query' in df.columns and 'label' in df.columns, "DataFrame must have 'query' and
'label' columns"
        self.stoi = stoi
        self.max_len = max_len
        self.queries = df['query'].astype(str).tolist()
        self.labels_raw = df['label'].tolist()
        self.le = label_encoder if label_encoder is not None else LabelEncoder()
        if fit_le or label_encoder is None:
            self.labels = self.le.fit_transform(self.labels_raw)
        else:
            self.labels = self.le.transform(self.labels_raw)

    def __len__(self):
        return len(self.queries)

    def __getitem__(self, idx):
        x = torch.tensor(encode_text(self.queries[idx], self.stoi, self.max_len), dtype=torch.long)
        y = torch.tensor(self.labels[idx], dtype=torch.long)
        return x, y

# --------------------------
# CNN Encoder
# --------------------------
class CharCNNEncoder(nn.Module):
    def __init__(self, vocab_size: int, emb_dim: int = 64, num_filters: int = 64, kernel_sizes:
Tuple[int, ...] = (2,3,4,5), out_dim: int = 128, dropout: float = 0.2):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, emb_dim, padding_idx=0)
        self.convs = nn.ModuleList([
            nn.Conv1d(in_channels=emb_dim, out_channels=num_filters, kernel_size=k) for k in
kernel_sizes
        ])
        self.dropout = nn.Dropout(dropout)
        self.proj = nn.Linear(num_filters * len(kernel_sizes), out_dim)
    def forward(self, x):  # x: (B, T)
        emb = self.embedding(x)             # (B, T, E)
        emb = emb.transpose(1, 2)           # (B, E, T)
        conv_outs = []
        for conv in self.convs:
            c = torch.relu(conv(emb))       # (B, F, T')
            p = torch.max(c, dim=-1).values # (B, F)
            conv_outs.append(p)
        h = torch.cat(conv_outs, dim=-1)    # (B, F*len(K))
        h = self.dropout(h)
        z = self.proj(h)                    # (B, out_dim)
        return z

# -------------------------
# Simple Classifier (for pretraining encoder if desired)
# -------------------------
class CNNClassifier(nn.Module):
    def __init__(self, encoder: CharCNNEncoder, num_classes: int):
        super().__init__()
        self.encoder = encoder
        self.fc = nn.Linear(encoder.proj.out_features, num_classes)

    def forward(self, x):
        z = self.encoder(x)
        logits = self.fc(z)
        return logits, z

# -------------------------
# Training utilities
# -------------------------
def train_cnn(encoder: CharCNNEncoder, train_loader: DataLoader, val_loader: DataLoader,
num_classes: int, epochs: int = 5, lr: float = 1e-3, device: str = "cpu"):
    model = CNNClassifier(encoder, num_classes).to(device)
    optim = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()

    best_val = -1
    for ep in range(epochs):
        model.train()
        total_loss = 0.0
        for xb, yb in train_loader:
            xb, yb = xb.to(device), yb.to(device)
            optim.zero_grad()
            logits, _ = model(xb)
            loss = criterion(logits, yb)
            loss.backward()
            optim.step()
            total_loss += loss.item() * xb.size(0)
        avg_loss = total_loss / len(train_loader.dataset)

        # Validate
        model.eval()
        preds, golds = [], []
        with torch.no_grad():
            for xb, yb in val_loader:
                xb = xb.to(device)
                logits, _ = model(xb)
                pred = logits.argmax(dim=-1).cpu().numpy().tolist()
                preds += pred
                golds += yb.numpy().tolist()
        acc = accuracy_score(golds, preds)
        if acc > best_val:
            best_val = acc

        print(f"[Epoch {ep+1}/{epochs}] loss={avg_loss:.4f} val_acc={acc:.4f}")

    return model

def extract_embeddings(encoder: CharCNNEncoder, loader: DataLoader, device: str = "cpu") ->
Tuple[np.ndarray, np.ndarray]:
    encoder.eval()
    X, y = [], []
    with torch.no_grad():
        for xb, yb in loader:
            xb = xb.to(device)
            z = encoder(xb).cpu().numpy()
            X.append(z)
            y += yb.numpy().tolist()
    X = np.concatenate(X, axis=0)
    y = np.array(y)
    return X, y

# ---------------------------
# Data loader helpers
# ---------------------------
def load_or_make_demo(df_path: Optional[str]) -> pd.DataFrame:
    if df_path and os.path.exists(df_path):
        return pd.read_csv(df_path)
    # Tiny synthetic sample; replace with real data for production.
    benign = [
        "SELECT * FROM users WHERE id=5;",
        "SELECT name FROM products WHERE price < 100;",
        "INSERT INTO logs(level, msg) VALUES('INFO','started');",
        "UPDATE accounts SET last_login=NOW() WHERE user='alice';",
        "DELETE FROM temp WHERE created_at < NOW() - INTERVAL '7 days';"
    ]
    injected = [
        "SELECT * FROM users WHERE name='' OR '1'='1'; --",
        "admin'--",
        "id=1; DROP TABLE users; --",
        "SELECT * FROM posts WHERE title LIKE '%a%' OR 1=1;",
        "'; EXEC xp_cmdshell('dir'); --"
    ]
    data = [{'query': q, 'label': 0} for q in benign] + [{'query': q, 'label': 1} for q in injected]
    return pd.DataFrame(data)

# -------------------------
# Main
# -------------------------
def main():
    parser = argparse.ArgumentParser(description="Hybrid CNN + RandomForest for SQL Injection
Detection")
    parser.add_argument('--data', type=str, default=None, help='Path to CSV with columns:
query,label')
    parser.add_argument('--max-len', type=int, default=256, help='Max characters per query')
    parser.add_argument('--epochs', type=int, default=5, help='CNN pre-train epochs')
    parser.add_argument('--batch-size', type=int, default=32, help='Batch size')
    parser.add_argument('--lr', type=float, default=1e-3, help='Learning rate')
    parser.add_argument('--rf-trees', type=int, default=300, help='RandomForest n_estimators')
    parser.add_argument('--rf-max-depth', type=int, default=None, help='RandomForest max_depth')
    parser.add_argument('--predict', type=str, default=None, help='Single-query prediction mode')
    parser.add_argument('--device', type=str, default='cpu', help='cpu or cuda')
    args = parser.parse_args()

    set_seed(42)

    # Load data (or demo)
    df = load_or_make_demo(args.data)

    # Split
    train_df, test_df = train_test_split(df, test_size=0.25, random_state=42, stratify=df['label'])

    # Build vocab on train
    stoi, itos = build_char_vocab(train_df['query'].astype(str).tolist())

    # Label encoder
    le = LabelEncoder()
    le.fit(train_df['label'])

    # Datasets
    train_ds = QueryDataset(train_df, stoi, args.max_len, label_encoder=le, fit_le=False)
    test_ds  = QueryDataset(test_df,  stoi, args.max_len, label_encoder=le, fit_le=False)

    # Dataloaders
    train_loader = DataLoader(train_ds, batch_size=args.batch_size, shuffle=True, drop_last=False)
    val_loader   = DataLoader(test_ds,  batch_size=args.batch_size, shuffle=False, drop_last=False)

    # CNN Encoder
    vocab_size = len(itos)
    encoder = CharCNNEncoder(vocab_size=vocab_size, emb_dim=64, num_filters=64,
kernel_sizes=(2,3,4,5), out_dim=128, dropout=0.25)

    # Pre-train CNN classification head (helps encoder learn)
    model = train_cnn(encoder, train_loader, val_loader, num_classes=len(le.classes_),
epochs=args.epochs, lr=args.lr, device=args.device)

    # Extract embeddings
    X_train, y_train = extract_embeddings(model.encoder, train_loader, device=args.device)
    X_test,  y_test  = extract_embeddings(model.encoder, val_loader, device=args.device)

    # RandomForest on top of embeddings
    rf = RandomForestClassifier(n_estimators=args.rf_trees, max_depth=args.rf_max_depth,
random_state=42, n_jobs=-1)
    rf.fit(X_train, y_train)

    # Evaluate
    y_pred = rf.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    p, r, f1, _ = precision_recall_fscore_support(y_test, y_pred, average='binary', zero_division=0)

    print("Accuracy:", acc)
    print("Precision:", p)
    print("Recall:", r)
    print("F1:", f1)
    print("\nClassification Report:\n", classification_report(y_test, y_pred, target_names=[str(c)
for c in le.classes_], zero_division=0))

    # Save artifacts
    os.makedirs("artifacts", exist_ok=True)
    torch.save(model.encoder.state_dict(), os.path.join("artifacts", "cnn_encoder.pt"))
    dump(rf, os.path.join("artifacts", "rf_model.joblib"))
    dump(le, os.path.join("artifacts", "label_encoder.joblib"))

    # Single prediction mode
    if args.predict is not None:
        x = torch.tensor([encode_text(args.predict, stoi, args.max_len)], dtype=torch.long)
        with torch.no_grad():
            z = model.encoder(x).numpy()
        pred = rf.predict(z)[0]
        label = le.inverse_transform([int(pred)])[0]
        print(f"Prediction for query: {args.predict!r} -> label={label}")
```

```
if __name__ == "__main__":
    main()
```