Supplementary File 2: Custom-Developed Code

Running Pipeline:

```
"""
run_pipeline.py
===============
End-to-end reproduction pipeline for the Forensic-Aware Blockchain-Integrated
Extended BiLSTM IDS. Executes the stages in the order described in the
manuscript Protocol:

   1. Load dataset          (raw CSV  -> dataframe)
   2. Preprocessing         (preprocessing.py : clean, impute, denoise,
                             normalize, encode, window)          [Algorithm 1]
   3. Feature selection     (use the selected feature list, e.g. Table 12)
   4. Model build           (model.py : Extended BiLSTM backbone)   [Eqs. 6-12]
   5. Training              (train.py : weighted BCE, Adam, early stop) [Table 4]
   6. Evaluation            (held-out test metrics + confusion matrix)
   7. (optional) Blockchain   (blockchain_client.py : sign + log intrusions)

Run:
   python run_pipeline.py --config ../config.json --data /path/to/dataset.csv

This script is dataset-agnostic: point --data at the preprocessed CSV for any of
the three benchmarks (UNSW-NB15, CIC-IDS-2017, Bot-IoT) and set the matching
feature list / categorical columns in config.json.
"""
from __future__ import annotations
import argparse, json, os, sys
import numpy as np

# local modules
import preprocessing as P
import model as M
import train as T


def load_config(path: str) -> dict:
   with open(path) as f:
      return json.load(f)


def stage_load(data_path: str, label_col: str):
   """Stage 1: load raw data."""
   import pandas as pd
   print(f"[1/6] Loading dataset: {data_path}")
   df = pd.read_csv(data_path)
```

```
    print(f"     rows={len(df)}  cols={df.shape[1]}")
    if label_col not in df.columns:
        raise ValueError(f"label column '{label_col}' not found in dataset")
    return df


def stage_preprocess(df, cfg):
    """Stage 2: preprocessing (Algorithm 1). Returns windowed X, y."""
    print("[2/6] Preprocessing (clean -> impute -> denoise -> normalize -> encode -> window)")
    pcfg = P.PreprocessConfig(
        label_column=cfg["preprocessing"]["label_column"],
        node_column=cfg["preprocessing"].get("node_column", "node_id"),
        categorical_columns=cfg["preprocessing"].get("categorical_columns", []),
        window_length=cfg["preprocessing"]["window_length"],
        stride=cfg["preprocessing"]["stride"],
        ema_alpha=cfg["preprocessing"]["ema_alpha"],
        test_split=cfg["preprocessing"]["test_split"],
    )
    X, y = P.preprocess(df, pcfg)
    print(f"     windows={X.shape[0]}  T={X.shape[1]}  features={X.shape[2]}")
    return X, y


def stage_feature_select(df, cfg):
    """Stage 3: restrict to the selected feature set (Table 12).

    AQU-IMF-RFE produces the selected list offline (see Supplementary File for
    the selection pseudocode); here we apply the resulting list so the pipeline
    reproduces the reported feature space deterministically.
    """
    feats = cfg["feature_selection"]["selected_features"]
    label = cfg["preprocessing"]["label_column"]
    node  = cfg["preprocessing"].get("node_column")
    keep = [c for c in feats if c in df.columns]
    print(f"[3/6] Feature selection: keeping {len(keep)} features -> {keep}")
    cols = keep + [label] + ([node] if node and node in df.columns else [])
    return df[cols].copy()


def stage_train(X, y, cfg):
    """Stages 4-5: build + train the Extended BiLSTM (binary detection)."""
    print("[4/6] Building Extended BiLSTM model")
    print("[5/6] Training (weighted BCE, Adam, early stopping)")
    tcfg = T.TrainConfig(
        learning_rate=cfg["training"]["learning_rate"],
        batch_size=cfg["training"]["batch_size"],
        epochs=cfg["training"]["epochs"],
```

```
        patience=cfg["training"]["patience"],
        val_split=cfg["training"]["val_split"],
        test_split=cfg["training"]["test_split"],
        threshold=cfg["training"]["threshold"],
        seed=cfg["training"]["seed"],
        n_classes=1,
    )
    model, history, test_data = T.train(X, y, tcfg)
    return model, history, test_data


def stage_evaluate(model, test_data, threshold, out_dir):
    """Stage 6: compute metrics + binary confusion matrix, persist them."""
    print("[6/6] Evaluation (held-out test set)")
    os.makedirs(out_dir, exist_ok=True)
    X_test, y_test = test_data
    probs = model.predict(X_test, verbose=0).reshape(-1)
    preds = (probs >= threshold).astype(int)
    y_true = y_test.astype(int)

    TP = int(((preds == 1) & (y_true == 1)).sum())
    TN = int(((preds == 0) & (y_true == 0)).sum())
    FP = int(((preds == 1) & (y_true == 0)).sum())
    FN = int(((preds == 0) & (y_true == 1)).sum())
    N = TP + TN + FP + FN
    acc = (TP + TN) / N if N else 0.0
    prec = TP / (TP + FP) if (TP + FP) else 0.0
    rec = TP / (TP + FN) if (TP + FN) else 0.0
    f1 = 2 * prec * rec / (prec + rec) if (prec + rec) else 0.0
    fpr = FP / (FP + TN) if (FP + TN) else 0.0
    results = {"accuracy": acc, "precision": prec, "recall": rec,
               "f1": f1, "fpr": fpr, "TP": TP, "TN": TN, "FP": FP, "FN": FN}

    with open(os.path.join(out_dir, "test_metrics.json"), "w") as f:
        json.dump(results, f, indent=2)
    # confusion_matrix.csv for figures.py  (rows: Normal, Intrusion)
    import csv as _csv
    with open(os.path.join(out_dir, "confusion_matrix.csv"), "w", newline="") as f:
        w = _csv.writer(f)
        w.writerow(["True\\Pred", "Normal", "Intrusion"])
        w.writerow(["Normal", TN, FP])
        w.writerow(["Intrusion", FN, TP])
    for k, v in results.items():
        print(f"      {k:10s}: {v}")
    print(f"      saved -> {out_dir}/test_metrics.json, confusion_matrix.csv")
    return results
```

```
def main():
    ap = argparse.ArgumentParser(description="Forensic-Aware IoMT IDS pipeline")
    ap.add_argument("--config", required=True, help="path to config.json")
    ap.add_argument("--data", required=True, help="path to dataset CSV")
    ap.add_argument("--out", default="../outputs", help="output directory")
    args = ap.parse_args()

    cfg = load_config(args.config)
    label = cfg["preprocessing"]["label_column"]

    df = stage_load(args.data, label)
    df = stage_feature_select(df, cfg)
    X, y = stage_preprocess(df, cfg)
    model, history, test_data = stage_train(X, y, cfg)
    stage_evaluate(model, test_data, cfg["training"]["threshold"], args.out)

    # persist training history for figure generation (Figures 2-3)
    import csv
    hist_path = os.path.join(args.out, "training_history.csv")
    keys = list(history.history.keys())
    with open(hist_path, "w", newline="") as f:
        w = csv.writer(f); w.writerow(["epoch"] + keys)
        for i in range(len(history.history[keys[0]])):
            w.writerow([i + 1] + [history.history[k][i] for k in keys])
    print(f"[done] training history -> {hist_path}")
    print("[done] pipeline complete.")


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

Preprocessing Phase:

```
"""
   "A Forensic-Aware Blockchain-Integrated Extended BiLSTM Framework for
    Real-Time Intrusion Detection in Healthcare IoT Systems"

This module reproduces the preprocessing pipeline described in the Protocol
section (Algorithm 1, Equations 1-5):

  - per-node stream extraction
  - corrupt-record removal
  - missing-value imputation (feature mean)
  - denoising (median / EMA filter over time)
  - min-max normalization to [0, 1]            (Eq. in Alg. 1)
  - categorical encoding (one-hot / label)
  - feature concatenation -> final feature vector x*
  - temporal windowing with window length T and stride s

The output X = {X(n_j)} is a set of windowed sequences of shape
(num_windows, T, d') ready for ingestion by the Extended BiLSTM model.

"""


from __future__ import annotations

import argparse
from dataclasses import dataclass, field
from typing import Iterable

import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder



# ---------------------------------------------------------------------------- #
# Configuration
# ---------------------------------------------------------------------------- #
@dataclass
class PreprocessConfig:
    """Preprocessing hyperparameters (see Algorithm 1)."""
    node_column: str = "node_id"         # column identifying the IoMT node n_j
    label_column: str = "label"          # ground-truth column (0 = benign, 1 = intrusion)
    categorical_columns: list[str] = field(default_factory=list)
    window_length: int = 20              # T  (temporal window)
    stride: int = 1                  # s  (1 <= s <= T)
    epsilon: float = 1e-8                # numerical-stability constant for min-max
    ema_alpha: float = 0.3               # smoothing factor for the EMA denoise filter
```

```
    drop_corrupt: bool = True
    test_split: float = 0.2             # held-out fraction; stats fit on the rest (train)


# ---------------------------------------------------------------------- #
# Helper operations (the named sub-routines used inside Algorithm 1)
# ---------------------------------------------------------------------- #
def is_corrupt(row: pd.Series) -> bool:
    """IsCorrupt(S[i]): flag rows that are entirely NaN or contain +/- inf."""
    if row.isna().all():
        return True
    numeric = pd.to_numeric(row, errors="coerce")
    return bool(np.isinf(numeric.to_numpy(dtype="float64", na_value=0.0)).any())


def impute_missing(df: pd.DataFrame) -> pd.DataFrame:
    """Missing-value imputation: x[f] <- Mean_f(S) for numeric features."""
    return df.fillna(df.mean(numeric_only=True))


def denoise(df: pd.DataFrame, alpha: float) -> pd.DataFrame:
    """Denoise(x): per-feature exponential moving average (EMA) over time.

    Recursive (adjust=False) form:  y_t = alpha * x_t + (1 - alpha) * y_{t-1}
    """
    return df.ewm(alpha=alpha, adjust=False).mean()


def min_max_normalize(df: pd.DataFrame, eps: float) -> pd.DataFrame:
    """Min-max normalization to [0, 1]:

        x'[f] = (x[f] - min_f) / (max_f - min_f + eps)
    """
    mn = df.min()
    mx = df.max()
    return (df - mn) / (mx - mn + eps)


def encode_categorical(df: pd.DataFrame, columns: Iterable[str]) -> pd.DataFrame:
    """phi(c): label-encode categorical fields.

    """
    out = df.copy()
    for col in columns:
        if col in out.columns:
            out[col] = LabelEncoder().fit_transform(out[col].astype(str))
    return out
```

```
def sanity_check(window: np.ndarray) -> bool:
    """SanityCheck(W_t): reject windows containing NaN/inf."""
    return bool(np.isfinite(window).all())


# ---------------------------------------------------------------------------- #
# Temporal windowing
# ---------------------------------------------------------------------------- #
def make_windows(
    features: np.ndarray,
    labels: np.ndarray,
    T: int,
    s: int,
) -> tuple[np.ndarray, np.ndarray]:
    """Temporal segmentation:

        for t = 1 to |C| - T + 1 step s:
            W_t = [C[t], C[t+1], ..., C[t+T-1]]

    The window label is taken as the label of the last time step (the decision
    point), which matches the per-window classification in Algorithm 2.
    """
    windows, win_labels = [], []
    n = features.shape[0]
    if n < T:
        return np.empty((0, T, features.shape[1])), np.empty((0,))
    for start in range(0, n - T + 1, s):
        w = features[start : start + T]
        if sanity_check(w):
            windows.append(w)
            win_labels.append(labels[start + T - 1])
    return np.asarray(windows, dtype="float32"), np.asarray(win_labels, dtype="float32")


# ---------------------------------------------------------------------------- #
# Main pipeline (Algorithm 1)
# ---------------------------------------------------------------------------- #
def preprocess(df: pd.DataFrame, cfg: PreprocessConfig) -> tuple[np.ndarray, np.ndarray]:
    """Run Algorithm 1 end-to-end.

    Returns
    -------
    X : np.ndarray, shape (total_windows, T, d')
    Y : np.ndarray, shape (total_windows,)
    """
```

```
all_windows, all_labels = [], []

# Determine node grouping; if no node column, treat the whole frame as one stream.
if cfg.node_column in df.columns:
   groups = df.groupby(cfg.node_column, sort=False)
   node_frames = [g for _, g in groups]
else:
   node_frames = [df]

for stream in node_frames:                          # for each node n_j
   stream = stream.reset_index(drop=True)

   if cfg.drop_corrupt:
      keep = ~stream.apply(is_corrupt, axis=1)
      stream = stream[keep].reset_index(drop=True)
   if stream.empty:
      continue

   labels = stream[cfg.label_column].to_numpy(dtype="float32")

   feat = stream.drop(columns=[cfg.label_column, cfg.node_column],
                      errors="ignore")

   feat = encode_categorical(feat, cfg.categorical_columns)    # z <- phi(c)
   feat = feat.apply(pd.to_numeric, errors="coerce")

   # --- Leakage-free statistics: fit on TRAINING ROWS ONLY ----------
   # The first (1 - test_split) fraction of each stream (time-ordered)
   # is treated as the training portion. Imputation means and min-max
   # bounds are computed from this portion only, then applied to all
   # rows, so no test-set information leaks into preprocessing.
   n_rows = len(feat)
   n_train = max(1, int(round(n_rows * (1.0 - cfg.test_split))))

   train_means = feat.iloc[:n_train].mean(numeric_only=True)  # Mean_f(train)
   feat = feat.fillna(train_means)                          # impute w/ train means

   feat = denoise(feat, cfg.ema_alpha)                      # Denoise(x)

   train_min = feat.iloc[:n_train].min()                    # min-max bounds
   train_max = feat.iloc[:n_train].max()                    #   from train only
   feat = (feat - train_min) / (train_max - train_min + cfg.epsilon)
   # Unseen values: validation/test samples may fall outside the training
   # range, producing normalized values <0 or >1. They are clipped to the
   # [0, 1] interval so inference inputs stay bounded.
   feat = feat.clip(lower=0.0, upper=1.0)
   feat = feat.fillna(0.0)
```

```
        x_star = feat.to_numpy(dtype="float32")                # x* = Concat(x', z)

        w, wl = make_windows(x_star, labels, cfg.window_length, cfg.stride)
        if len(w):
            all_windows.append(w)
            all_labels.append(wl)

    if not all_windows:
        raise ValueError("No valid windows produced. Check window length / data.")

    X = np.concatenate(all_windows, axis=0)
    Y = np.concatenate(all_labels, axis=0)
    return X, Y


# ---------------------------------------------------------------------- #
# CLI
# ---------------------------------------------------------------------- #
#
# ======================================================================
#
#  SETTINGS  --  EDIT THESE IF NEEDED
#  Configured for the UNSW-NB15 training set. Change the dataset
#
# ======================================================================
#
INPUT_CSV   = "UNSW_NB15_training-set.csv"   # <-- your data file (same folder as this script)
OUTPUT_NPZ  = "preprocessed.npz"             # <-- output produced for training

LABEL_COLUMN = "label"                  # UNSW-NB15: 0 = normal, 1 = attack
NODE_COLUMN  = "node_id"                    # not present in UNSW-NB15; harmless if absent

# Columns that are categorical / text and must be encoded (UNSW-NB15):
CATEGORICAL_COLUMNS = ["proto", "service", "state"]

# textual attack category, because 'attack_cat' would leak the label.
DROP_COLUMNS = ["id", "attack_cat"]

WINDOW_LENGTH = 20    # T
STRIDE        = 1     # s
#
# ======================================================================
#


def main() -> None:
```

```
    # Allow optional command-line overrides, but everything has a default
    # so the script runs with a plain:  python preprocessing.py
    p = argparse.ArgumentParser(description="IoMT IDS preprocessing (Algorithm 1)")
    p.add_argument("--input", default=INPUT_CSV, help="CSV of raw traffic records")
    p.add_argument("--output", default=OUTPUT_NPZ, help="output .npz (X, Y)")
    p.add_argument("--node-column", default=NODE_COLUMN)
    p.add_argument("--label-column", default=LABEL_COLUMN)
    p.add_argument("--categorical", nargs="*", default=CATEGORICAL_COLUMNS)
    p.add_argument("--window", type=int, default=WINDOW_LENGTH)
    p.add_argument("--stride", type=int, default=STRIDE)
    args = p.parse_args()

    cfg = PreprocessConfig(
        node_column=args.node_column,
        label_column=args.label_column,
        categorical_columns=args.categorical,
        window_length=args.window,
        stride=args.stride,
    )

    print(f"[preprocessing] reading: {args.input}")
    df = pd.read_csv(args.input)
    print(f"[preprocessing] loaded shape = {df.shape}")

    # Drop leakage / id columns if present
    to_drop = [c for c in DROP_COLUMNS if c in df.columns]
    if to_drop:
        df = df.drop(columns=to_drop)
        print(f"[preprocessing] dropped columns: {to_drop}")

    X, Y = preprocess(df, cfg)
    np.savez_compressed(args.output, X=X, Y=Y)
    print(f"[preprocessing] X shape = {X.shape}, Y shape = {Y.shape}")
    print(f"[preprocessing] saved -> {args.output}")

    # ------------------------------------------------------------------ #
    #  FINAL SAMPLE COUNT SUMMARY
    # ------------------------------------------------------------------ #
    total = int(Y.shape[0])
    n_intrusion = int((Y == 1).sum())
    n_benign = int((Y == 0).sum())
    print("\n================  FINAL SAMPLE COUNT (after preprocessing) ================")
    print(f"  Dataset file ..... {args.input}")
    print(f"  Window length (T) ..... {args.window}")
    print(f"  Features per step (d) ... {X.shape[2]}")
    print(f"  Retained windows (N) .... {total}")
```

```
    print(f"     - benign (label 0) ... {n_benign}")
    print(f"     - intrusion (label 1)  {n_intrusion}")
    print(f"  80/20 split ..... train ~{int(round(total*0.8))}, test ~{int(round(total*0.2))}")

print("=========================================================================
=======")


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

Model Building:

```
"""
============================================================
Reference implementation of the Extended BiLSTM detector (Algorithm 2,
Equations 6-12) from:

   "A Forensic-Aware Blockchain-Integrated Extended BiLSTM Framework for
    Real-Time Intrusion Detection in Healthcare IoT Systems"

Architecture (per the Protocol section):

   Input  X(n_j) in R^{T x d}
     |
     v
   [Eq. 6]  Temporal feature extraction : U = phi(Conv1D(X))
     |
     v
   [Eq. 7]  Stacked Bidirectional LSTM  : h_t = [h_t_fwd ; h_t_bwd]
     |
     v
   [Eq. 9]  Residual + LayerNorm        : h_t = LayerNorm(h_t + u_t)
     |
     v
   [Eq.10]  Attention scoring           : alpha_t = softmax(W_a h_t)
   [Eq.11]  Context vector              : c = sum_t alpha_t h_t
     |
     v
   [Eq.12]  Dense + sigmoid             : y_hat = sigma(W_c c + b_c)

A threshold tau (Eq. 13) converts y_hat into a benign/intrusion decision; the
thresholding lives in the inference/pipeline code, not in the model graph.

Framework: TensorFlow / Keras.
"""

from __future__ import annotations

from dataclasses import dataclass

import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.utils import register_keras_serializable


@dataclass
class ModelConfig:
```

```
    """Hyperparameters for the Extended BiLSTM (align with Table 4)."""
    timesteps: int = 20          # T  (window length)
    n_features: int = 0          # d  (set at build time from the data)
    conv_filters: int = 64       # Conv1D channels for temporal feature extraction
    conv_kernel: int = 3
    lstm_units: int = 64         # hidden size h per direction
    n_lstm_layers: int = 2       # "stacked" bidirectional LSTM
    dropout: float = 0.3
    recurrent_dropout: float = 0.0
    dense_units: int = 64
    n_classes: int = 1           # 1 = binary detection (sigmoid); >=3 = multi-class
                                 # attack categorization head (softmax)


# ---------------------------------------------------------------------------- #
# Attention layer  (Equations 10-11)
# ---------------------------------------------------------------------------- #
@register_keras_serializable(package="ExtendedBiLSTM")
class TemporalAttention(layers.Layer):
    """Additive/scoring attention over time.

        score_t  = W_a h_t                  (Eq. 10, numerator argument)
        alpha_t  = softmax_t(score_t)       (Eq. 10)
        c        = sum_t alpha_t * h_t      (Eq. 11)

    Returns the context vector c and (optionally) the attention weights alpha,
    which provide the interpretability discussed in the manuscript.
    """

    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def get_config(self):
        # Enables model.save()/load_model() to round-trip this custom layer.
        return super().get_config()

    def build(self, input_shape):
        hidden = int(input_shape[-1])
        # W_a : trainable scoring vector (Eq. 10)
        self.W_a = self.add_weight(
            name="W_a",
            shape=(hidden, 1),
            initializer="glorot_uniform",
            trainable=True,
        )
        super().build(input_shape)
```

```
    def call(self, h, return_attention: bool = False):
        # h : (batch, T, hidden)
        scores = tf.matmul(h, self.W_a)              # (batch, T, 1)
        alpha = tf.nn.softmax(scores, axis=1)        # (batch, T, 1), sum_t alpha_t = 1
        context = tf.reduce_sum(alpha * h, axis=1)   # (batch, hidden)  -> Eq. 11
        if return_attention:
            return context, tf.squeeze(alpha, axis=-1)
        return context


# ---------------------------------------------------------------------------- #
# Model builder
# ---------------------------------------------------------------------------- #
def build_extended_bilstm(cfg: ModelConfig) -> tf.keras.Model:
    """Construct the Extended BiLSTM-Attention classifier (Algorithm 2)."""
    if cfg.n_features <= 0:
        raise ValueError("ModelConfig.n_features must be set to d (>0).")

    inp = layers.Input(shape=(cfg.timesteps, cfg.n_features), name="window_input")

    # --- Eq. 6 : Temporal feature extraction U = phi(Conv1D(X)) -------------
    u = layers.Conv1D(
        filters=cfg.conv_filters,
        kernel_size=cfg.conv_kernel,
        padding="same",
        activation="relu",        # phi(.) nonlinearity
        name="temporal_conv1d",
    )(inp)

    # --- Eq. 7-8 : Stacked Bidirectional LSTM -------------------------------
    h = u
    for i in range(cfg.n_lstm_layers):
        h = layers.Bidirectional(
            layers.LSTM(
                cfg.lstm_units,
                return_sequences=True,         # need per-timestep states for attention
                dropout=cfg.dropout,
                recurrent_dropout=cfg.recurrent_dropout,
            ),
            name=f"bilstm_{i + 1}",
        )(h)

    # --- Eq. 9 : Residual connection + LayerNorm ----------------------------
    # Project u to match the BiLSTM output width (2 * lstm_units) so the
    # residual add is dimensionally valid, then h = LayerNorm(h + u).
    u_proj = layers.Dense(2 * cfg.lstm_units, name="residual_projection")(u)
    h = layers.Add(name="residual_add")([h, u_proj])
```

```
    h = layers.LayerNormalization(name="layer_norm")(h)

    # --- Eq. 10-11 : Attention -> context vector --------------------------
    context = TemporalAttention(name="temporal_attention")(h)

    # --- Shared dense hidden representation -------------------------------
    z = layers.Dense(cfg.dense_units, activation="relu", name="dense_hidden")(context)
    z = layers.Dropout(cfg.dropout, name="head_dropout")(z)

    # --- Output head ------------------------------------------------------
    # Stage 1 (detection): single sigmoid unit -> P(intrusion), threshold tau.
    # Stage 2 (categorization): softmax over C attack classes -> argmax label.
    if cfg.n_classes <= 1:
        out = layers.Dense(1, activation="sigmoid",
                           name="intrusion_prob")(z)          # y_hat in [0,1]  (Eq. 12)
        model_name = "Extended_BiLSTM_Detection"
    else:
        out = layers.Dense(cfg.n_classes, activation="softmax",
                           name="attack_class_probs")(z)      # softmax over C classes
        model_name = "Extended_BiLSTM_Categorization"

    return models.Model(inputs=inp, outputs=out, name=model_name)


def build_detection_model(cfg: ModelConfig) -> tf.keras.Model:
    """Stage 1: binary intrusion detection (sigmoid + threshold tau)."""
    import dataclasses
    return build_extended_bilstm(dataclasses.replace(cfg, n_classes=1))


def build_categorization_model(cfg: ModelConfig, n_classes: int) -> tf.keras.Model:
    """Stage 2: multi-class attack categorization (softmax over n_classes).

    Shares the same backbone architecture as Stage 1; only the output head
    and loss differ (categorical cross-entropy, argmax decision).
    """
    import dataclasses
    if n_classes < 2:
        raise ValueError("categorization requires n_classes >= 2")
    return build_extended_bilstm(dataclasses.replace(cfg, n_classes=n_classes))


def build_attention_inspector(model: tf.keras.Model) -> tf.keras.Model:
    """Return a model that also outputs attention weights for interpretability.

    Useful for the forensic/explainability discussion: it exposes alpha_t so an
    analyst can see which timesteps drove a given alert.
```

```
    """
    attn_layer = model.get_layer("temporal_attention")
    ln_out = model.get_layer("layer_norm").output
    context, alpha = attn_layer(ln_out, return_attention=True)
    return models.Model(inputs=model.input, outputs=[model.output, alpha],
                        name="Extended_BiLSTM_with_attention")


if __name__ == "__main__":
    demo = build_extended_bilstm(ModelConfig(timesteps=20, n_features=30))
    demo.summary()
```

Training Phase:

```
"""
============================================================
Reference implementation of Algorithm 3 and Equations 18-26 from:

    "A Forensic-Aware Blockchain-Integrated Extended BiLSTM Framework for
     Real-Time Intrusion Detection in Healthcare IoT Systems"

Covers:
  - 80/20 train/test split (Results section)
  - weighted binary cross-entropy loss           (Eqs. 18-21)
  - Adam optimizer with bias-corrected moments      (Eqs. 24-26)
  - mini-batch training with early stopping         (Algorithm 3)
  - learning rate 0.001, batch size 64, 50 epochs   (Results / Table 4)

Run:
    python train.py --data preprocessed.npz --out-model model.keras
"""

from __future__ import annotations

import argparse
from dataclasses import dataclass

import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split

from model import ModelConfig, build_extended_bilstm


@dataclass
class TrainConfig:
    """Training hyperparameters (align with Table 4 / Results section)."""
    learning_rate: float = 1e-3      # eta (Adam)
    beta_1: float = 0.9              # Eq. 24 first-moment decay
    beta_2: float = 0.999            # Eq. 24 second-moment decay
    epsilon: float = 1e-7            # Eq. 26 numerical stability
    batch_size: int = 64             # B
    epochs: int = 50
    patience: int = 5                # K consecutive epochs w/o val improvement
    val_split: float = 0.2           # validation fraction within the training set
    test_split: float = 0.2          # 20% held out for testing
    threshold: float = 0.5           # tau (Eq. 13) for reporting metrics
    n_classes: int = 1               # 1 = Stage 1 binary detection;
                                     # >=3 = Stage 2 multi-class categorization
```

```
    seed: int = 42


# ---------------------------------------------------------------------------- #
# Weighted binary cross-entropy  (Equations 18-21)
# ---------------------------------------------------------------------------- #
def compute_class_weights(y: np.ndarray) -> tuple[float, float]:
    """Class weights (Eqs. 19-20):

        w_p = N / (2 * N_p)       (positive / intrusion)
        w_n = N / (2 * N_n)       (negative / benign)
    """
    N = len(y)
    N_p = float(np.sum(y == 1))
    N_n = float(np.sum(y == 0))
    w_p = N / (2.0 * N_p) if N_p > 0 else 1.0
    w_n = N / (2.0 * N_n) if N_n > 0 else 1.0
    return w_p, w_n


def make_weighted_bce(w_p: float, w_n: float):
    """Weighted BCE (Eq. 21):

        L = -1/N sum_i [ w_p y_i log(y_hat_i) + w_n (1 - y_i) log(1 - y_hat_i) ]

    The unweighted special case (w_p = w_n = 1) recovers Eq. 18.
    """
    w_p = tf.constant(w_p, dtype=tf.float32)
    w_n = tf.constant(w_n, dtype=tf.float32)

    def loss(y_true, y_pred):
        y_true = tf.cast(y_true, tf.float32)
        eps = tf.keras.backend.epsilon()
        y_pred = tf.clip_by_value(y_pred, eps, 1.0 - eps)
        term_pos = w_p * y_true * tf.math.log(y_pred)
        term_neg = w_n * (1.0 - y_true) * tf.math.log(1.0 - y_pred)
        return -tf.reduce_mean(term_pos + term_neg)

    return loss


# ---------------------------------------------------------------------------- #
# Training (Algorithm 3)
# ---------------------------------------------------------------------------- #
def train(X: np.ndarray, Y: np.ndarray, tcfg: TrainConfig):
    tf.random.set_seed(tcfg.seed)
    np.random.seed(tcfg.seed)
```

```
# 80/20 train/test split (stratified to preserve class ratio)
X_train, X_test, y_train, y_test = train_test_split(
   X, Y, test_size=tcfg.test_split, random_state=tcfg.seed,
   stratify=Y if len(np.unique(Y)) > 1 else None,
)

# Build the Extended BiLSTM sized to the data
mcfg = ModelConfig(timesteps=X.shape[1], n_features=X.shape[2],
                   n_classes=tcfg.n_classes)
model = build_extended_bilstm(mcfg)

# Adam optimizer (Eqs. 24-26)
optimizer = tf.keras.optimizers.Adam(
   learning_rate=tcfg.learning_rate,
   beta_1=tcfg.beta_1,
   beta_2=tcfg.beta_2,
   epsilon=tcfg.epsilon,
)

if tcfg.n_classes <= 1:
   # Stage 1: binary detection -- weighted BCE (Eqs. 19-21)
   w_p, w_n = compute_class_weights(y_train)
   print(f"[train] class weights -> w_p={w_p:.4f}, w_n={w_n:.4f}")
   model.compile(
      optimizer=optimizer,
      loss=make_weighted_bce(w_p, w_n),
      metrics=[
         tf.keras.metrics.BinaryAccuracy(name="accuracy", threshold=tcfg.threshold),
         tf.keras.metrics.Precision(name="precision", thresholds=tcfg.threshold),
         tf.keras.metrics.Recall(name="recall", thresholds=tcfg.threshold),
         tf.keras.metrics.AUC(name="auc"),
      ],
   )
else:
   # Stage 2: multi-class attack categorization -- categorical cross-entropy.
   # Labels y are integer class indices; argmax of the softmax gives the
   # predicted attack category. class_weight balances categories by frequency.
   import numpy as _np
   classes = _np.arange(tcfg.n_classes)
   counts = _np.bincount(y_train.astype(int), minlength=tcfg.n_classes)
   total = counts.sum()
   class_weight = {int(c): float(total / (tcfg.n_classes * max(counts[c], 1)))
                   for c in classes}
   print(f"[train] categorization class weights -> {class_weight}")
   model.compile(
      optimizer=optimizer,
```

```
        loss=tf.keras.losses.SparseCategoricalCrossentropy(),
        metrics=[tf.keras.metrics.SparseCategoricalAccuracy(name="accuracy")],
      )

    # Early stopping: terminate if val loss does not improve for K epochs;
    # restore the best weights (theta* with minimum validation loss).
    early_stop = tf.keras.callbacks.EarlyStopping(
      monitor="val_loss",
      patience=tcfg.patience,
      restore_best_weights=True,
      verbose=1,
    )

    fit_kwargs = dict(
      validation_split=tcfg.val_split,
      epochs=tcfg.epochs,
      batch_size=tcfg.batch_size,
      callbacks=[early_stop],
      verbose=2,
    )
    if tcfg.n_classes > 1:
      fit_kwargs["class_weight"] = class_weight

    history = model.fit(X_train, y_train, **fit_kwargs)

    # Held-out evaluation
    print("\n[train] held-out test evaluation:")
    results = model.evaluate(X_test, y_test, verbose=0, return_dict=True)
    for k, v in results.items():
      print(f"    {k:10s}: {v:.4f}")

    # Derived F1 (precision/recall come from thresholded metrics)
    p = results.get("precision", 0.0)
    r = results.get("recall", 0.0)
    f1 = 2 * p * r / (p + r) if (p + r) > 0 else 0.0
    print(f"    {'f1':10s}: {f1:.4f}")

    return model, history, (X_test, y_test)


# ---------------------------------------------------------------------------- #
# CLI
# ---------------------------------------------------------------------------- #
def main() -> None:
    ap = argparse.ArgumentParser(description="Train Extended BiLSTM IDS (Algorithm 3)")
    ap.add_argument("--data", default="preprocessed.npz",
                    help=".npz with arrays X and Y (from preprocessing.py)")
```

```
    ap.add_argument("--out-model", default="extended_bilstm.keras")
    ap.add_argument("--epochs", type=int, default=50)
    ap.add_argument("--batch-size", type=int, default=64)
    ap.add_argument("--lr", type=float, default=1e-3)
    ap.add_argument("--patience", type=int, default=5)
    args = ap.parse_args()

    blob = np.load(args.data)
    X, Y = blob["X"], blob["Y"]
    print(f"[train] loaded X={X.shape}, Y={Y.shape}")

    tcfg = TrainConfig(
        learning_rate=args.lr,
        batch_size=args.batch_size,
        epochs=args.epochs,
        patience=args.patience,
    )
    model, _, _ = train(X, Y, tcfg)
    model.save(args.out_model)
    print(f"[train] saved model -> {args.out_model}")


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

Figures Development:

```
"""
figures.py
==========
Regenerates the data-driven figures from saved result files:

  Figure 2  - training/validation accuracy convergence  (from training_history.csv)
  Figure 3  - training/validation loss convergence       (from training_history.csv)
  Figure 13 - binary confusion matrix (Normal vs Intrusion) (from confusion_matrix.csv)

Comparative bar charts (Figures 4-6, 8-12) are produced from the per-model
metric values in the corresponding result CSVs; a generic helper is provided.

Usage:
    python figures.py --history ../outputs/training_history.csv --out ../outputs/figures
    python figures.py --confusion ../outputs/confusion_matrix.csv --out ../outputs/figures

All plots use Matplotlib defaults and are exported at 300 dpi.
"""
from __future__ import annotations
import argparse, csv, os
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
import numpy as np


def _read_csv(path):
    with open(path) as f:
        rows = list(csv.reader(f))
    header, data = rows[0], rows[1:]
    cols = {h: [] for h in header}
    for r in data:
        for h, v in zip(header, r):
            cols[h].append(v)
    return cols


def plot_history(history_csv, out_dir):
    cols = _read_csv(history_csv)
    epochs = [int(e) for e in cols["epoch"]]

    # Figure 2 : accuracy
    if "accuracy" in cols and "val_accuracy" in cols:
```

```
        plt.figure(figsize=(6, 4))
        plt.plot(epochs, [float(x) for x in cols["accuracy"]], marker="o", label="Training")
        plt.plot(epochs, [float(x) for x in cols["val_accuracy"]], marker="s", label="Validation")
        plt.xlabel("Epoch"); plt.ylabel("Accuracy")
        plt.title("Training and Validation Accuracy"); plt.legend(); plt.grid(True, alpha=0.3)
        plt.tight_layout(); plt.savefig(os.path.join(out_dir, "Figure2_accuracy.png"), dpi=300)
        plt.close()

    # Figure 3 : loss
    if "loss" in cols and "val_loss" in cols:
        plt.figure(figsize=(6, 4))
        plt.plot(epochs, [float(x) for x in cols["loss"]], marker="o", label="Training")
        plt.plot(epochs, [float(x) for x in cols["val_loss"]], marker="s", label="Validation")
        plt.xlabel("Epoch"); plt.ylabel("Loss")
        plt.title("Training and Validation Loss"); plt.legend(); plt.grid(True, alpha=0.3)
        plt.tight_layout(); plt.savefig(os.path.join(out_dir, "Figure3_loss.png"), dpi=300)
        plt.close()
    print(f"[figures] convergence plots saved to {out_dir}")


def plot_confusion(confusion_csv, out_dir):
    """confusion_matrix.csv : header row + 2 data rows (Normal, Intrusion)."""
    cols = _read_csv(confusion_csv)
    labels = ["Normal", "Intrusion"]
    cm = np.array([
        [float(cols["Normal"][0]), float(cols["Intrusion"][0])],
        [float(cols["Normal"][1]), float(cols["Intrusion"][1])],
    ])
    plt.figure(figsize=(4.5, 4))
    plt.imshow(cm, cmap="Blues")
    plt.xticks([0, 1], labels); plt.yticks([0, 1], labels)
    plt.xlabel("Predicted"); plt.ylabel("True")
    plt.title("Confusion Matrix (Binary)")
    for i in range(2):
        for j in range(2):
            plt.text(j, i, int(cm[i, j]), ha="center", va="center",
                     color="white" if cm[i, j] > cm.max() / 2 else "black")
    plt.colorbar(); plt.tight_layout()
    plt.savefig(os.path.join(out_dir, "Figure13_confusion.png"), dpi=300)
    plt.close()
    print(f"[figures] confusion matrix saved to {out_dir}")


def plot_bar_comparison(metric_csv, out_dir, name="Figure_comparison"):
    """Generic per-model bar chart. CSV: header 'model,metric' rows."""
    cols = _read_csv(metric_csv)
    models = cols[list(cols.keys())[0]]
```

```
    values = [float(v) for v in cols[list(cols.keys())[1]]]
    plt.figure(figsize=(7, 4))
    plt.bar(models, values)
    plt.ylabel(list(cols.keys())[1]); plt.xticks(rotation=30, ha="right")
    plt.tight_layout(); plt.savefig(os.path.join(out_dir, f"{name}.png"), dpi=300)
    plt.close()
    print(f"[figures] bar chart saved -> {name}")


def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--history"); ap.add_argument("--confusion")
    ap.add_argument("--bars");                                  ap.add_argument("--bars-name",
default="Figure_comparison")
    ap.add_argument("--out", default="../outputs/figures")
    a = ap.parse_args()
    os.makedirs(a.out, exist_ok=True)
    if a.history:   plot_history(a.history, a.out)
    if a.confusion: plot_confusion(a.confusion, a.out)
    if a.bars:      plot_bar_comparison(a.bars, a.out, a.bars_name)
    if not any([a.history, a.confusion, a.bars]):
        print("Nothing to plot. Provide --history, --confusion, or --bars.")


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

Blockchain Client:

```
"""
================================================================================
Reference implementation of the off-chain side of Algorithm 4 (Equations
13, 14, 17, 27, 28) from:

   "A Forensic-Aware Blockchain-Integrated Extended BiLSTM Framework for
    Real-Time Intrusion Detection in Healthcare IoT Systems"

This module connects the trained Extended BiLSTM detector to the PoA smart
contract (IoMTIntrusionLedger.sol). For each input window W_i:

   y_hat   = model(W_i)                    # Extended BiLSTM inference
   delta_t = 1 if y_hat >= tau else 0      # Eq. 13 decision
   if delta_t == 1:                        # Eq. 27
      build D_i, sign H_i, submit recordIntrusion(...)   # Eq. 14 / Eq. 17
      -> contract emits BlockCreated / NodeIsolated / AdminAlert

It targets a Proof-of-Authority chain (e.g., Geth Clique or Besu IBFT) reached
over web3. A DRY-RUN mode runs the full pipeline without a live chain so the
logic can be exercised offline.

Dependencies: web3>=6, eth-account, numpy, tensorflow
"""

from __future__ import annotations

import argparse
import hashlib
import json
import time
from dataclasses import dataclass

import numpy as np


# ---------------------------------------------------------------------------- #
# Configuration
# ---------------------------------------------------------------------------- #
@dataclass
class ChainConfig:
   rpc_url: str = "http://127.0.0.1:8545"     # PoA node RPC endpoint
   contract_address: str = ""                 # deployed IoMTIntrusionLedger
   abi_path: str = "IoMTIntrusionLedger.abi.json"
   gateway_key: str = ""                      # private key of an authorized gateway
   chain_id: int = 1337                       # local PoA dev chain id
```

```
    threshold: float = 0.5                  # tau (Eq. 13)
    dry_run: bool = True                    # if True, do not touch a real chain


# ---------------------------------------------------------------------------- #
# Feature digest & signing helpers (D_i, H_i, Sig_i)
# ---------------------------------------------------------------------------- #
def feature_digest(window: np.ndarray, node_id: str) -> bytes:
    """Digest of selected event features D_i (Eq. 14).

    Hashes a compact summary of the window so the on-chain record is bound to
    the observed traffic without storing raw patient data on-chain.
    """
    summary = {
        "node": node_id,
        "mean": float(np.mean(window)),
        "std": float(np.std(window)),
        "last": window[-1].tolist(),
    }
    blob = json.dumps(summary, sort_keys=True).encode()
    return hashlib.sha256(blob).digest()


def to_bytes32(value: str) -> bytes:
    """Right-pad / hash an arbitrary string to a 32-byte field."""
    b = value.encode()
    if len(b) <= 32:
        return b.ljust(32, b"\x00")
    return hashlib.sha256(b).digest()


# ---------------------------------------------------------------------------- #
# Blockchain client
# ---------------------------------------------------------------------------- #
class LedgerClient:
    """Thin web3 wrapper around IoMTIntrusionLedger; supports DRY-RUN."""

    def __init__(self, cfg: ChainConfig):
        self.cfg = cfg
        self._local_chain: list[dict] = []      # used in dry-run
        self._prev_hash = b"\x00" * 32          # genesis PrevHash

        if cfg.dry_run:
            self.w3 = None
            self.contract = None
            self.account = None
            return
```

```
    # ---- live PoA connection ----
    from web3 import Web3
    from web3.middleware import geth_poa_middleware   # required for PoA chains
    from eth_account import Account

    self.w3 = Web3(Web3.HTTPProvider(cfg.rpc_url))
    # PoA chains use an extended extraData field; this middleware handles it.
    self.w3.middleware_onion.inject(geth_poa_middleware, layer=0)
    if not self.w3.is_connected():
        raise ConnectionError(f"cannot reach PoA node at {cfg.rpc_url}")

    with open(cfg.abi_path) as fh:
        abi = json.load(fh)
    self.contract = self.w3.eth.contract(
        address=Web3.to_checksum_address(cfg.contract_address), abi=abi
    )
    self.account = Account.from_key(cfg.gateway_key)

# ----- Eq. 14 / 17 / 27 : record an intrusion block ------------------
def record_intrusion(
    self,
    node_id: str,
    patient_id: str,
    attack_class: int,
    probability: float,
    digest: bytes,
    signature: bytes,
    isolate: bool,
) -> str:
    prob_bp = int(round(max(0.0, min(1.0, probability)) * 10000))

    if self.cfg.dry_run:
        ts = int(time.time())
        h = hashlib.sha256(
            digest + ts.to_bytes(8, "big") + prob_bp.to_bytes(2, "big") + self._prev_hash
        ).digest()
        self._local_chain.append({
            "index": len(self._local_chain),
            "hashId": h.hex(),
            "timestamp": ts,
            "nodeId": node_id,
            "patientId": patient_id,
            "attackClass": attack_class,
            "probabilityBp": prob_bp,
            "prevHash": self._prev_hash.hex(),
            "isolated": isolate,
```

```
            })
            self._prev_hash = h
            return h.hex()

        # ---- live transaction ----
        fn = self.contract.functions.recordIntrusion(
            to_bytes32(node_id),
            to_bytes32(patient_id),
            attack_class,
            prob_bp,
            digest,
            signature,
            isolate,
        )
        tx = fn.build_transaction({
            "from": self.account.address,
            "nonce": self.w3.eth.get_transaction_count(self.account.address),
            "gas": 500_000,
            "gasPrice": self.w3.eth.gas_price,
            "chainId": self.cfg.chain_id,
        })
        signed = self.account.sign_transaction(tx)
        tx_hash = self.w3.eth.send_raw_transaction(signed.rawTransaction)
        receipt = self.w3.eth.wait_for_transaction_receipt(tx_hash)
        return receipt.transactionHash.hex()

    def verify_chain(self) -> bool:
        if self.cfg.dry_run:
            prev = "00" * 32
            for b in self._local_chain:
                if b["prevHash"] != prev:
                    return False
                prev = b["hashId"]
            return True
        return self.contract.functions.verifyChain().call()


# ---------------------------------------------------------------------- #
# End-to-end pipeline (Algorithm 4)
# ---------------------------------------------------------------------- #
CLASS_NAMES = {0: "Normal", 1: "DDoS", 2: "Spoofing"}


def run_pipeline(model, X, node_ids, patient_ids, cfg: ChainConfig,
                 attack_classes=None, audit_safe: bool = False):
    """Stream windows through detection + blockchain mitigation (Algorithm 4).
```

```
Parameters
----------
model      : trained Extended BiLSTM (Keras) or any callable -> prob in [0,1]
X          : array (num_windows, T, d)
node_ids    : per-window IoMT node identifiers
patient_ids : per-window patient/device identifiers
attack_classes : optional per-window class index for multi-class logging
audit_safe  : if True, also append a lightweight benign tx (AuditSafe=1)
"""
client = LedgerClient(cfg)
t0 = time.time()
probs = model.predict(X, verbose=0).reshape(-1)   # detection delay T_d

results = []
for i, p in enumerate(probs):
    delta_t = 1 if p >= cfg.threshold else 0      # Eq. 13
    node_id = str(node_ids[i])
    patient_id = str(patient_ids[i])

    if delta_t == 1:                          # Eq. 27 (delta_t = 1 branch)
        digest = feature_digest(X[i], node_id)     # D_i
        # Sig_i: ECDSA (secp256k1) signature of D_i by the gateway key,
        # EIP-191 prefixed; the contract recovers the signer via ecrecover
        # and checks it against the authorized gateway (msg.sender).
        from eth_account.messages import encode_defunct
        _msg = encode_defunct(primitive=digest)
        signature = client.account.sign_message(_msg).signature  # Sig_i
        attack_class = int(attack_classes[i]) if attack_classes is not None else 1
        tx = client.record_intrusion(
            node_id=node_id,
            patient_id=patient_id,
            attack_class=attack_class,
            probability=float(p),
            digest=digest,
            signature=signature,
            isolate=True,                     # NodeIsolation(n_j)
        )
        label = CLASS_NAMES.get(attack_class, "Malicious")
        results.append((i, node_id, label, float(p), tx))
    elif audit_safe:                          # AuditSafe=1 light benign record
        digest = feature_digest(X[i], node_id)
        client.record_intrusion(node_id, patient_id, 0, float(p), digest, b"", False)
        results.append((i, node_id, "Safe", float(p), None))
    else:
        results.append((i, node_id, "Safe", float(p), None))

t_total = time.time() - t0                    # ~ T_mitigation = T_d + T_b (Eq. 28)
```

```
    chain_ok = client.verify_chain()

    print(f"[pipeline] processed {len(probs)} windows in {t_total:.3f}s")
    print(f"[pipeline] intrusions logged: {sum(1 for r in results if r[2] != 'Safe')}")
    print(f"[pipeline] chain integrity verified: {chain_ok}")
    return results, chain_ok


# ---------------------------------------------------------------------------- #
# CLI (dry-run demo)
# ---------------------------------------------------------------------------- #
def main() -> None:
    ap = argparse.ArgumentParser(description="Detection + blockchain mitigation (Algorithm
4)")
    ap.add_argument("--model", help="path to trained .keras model")
    ap.add_argument("--data", help=".npz with array X")
    ap.add_argument("--threshold", type=float, default=0.5)
    ap.add_argument("--live", action="store_true", help="use a real PoA chain (default: dry-
run)")
    ap.add_argument("--contract", default="", help="deployed contract address (live mode)")
    ap.add_argument("--rpc", default="http://127.0.0.1:8545")
    args = ap.parse_args()

    import tensorflow as tf
    import model as _model_module  # noqa: F401  (registers TemporalAttention)
    model = tf.keras.models.load_model(args.model, compile=False)
    blob = np.load(args.data)
    X = blob["X"]
    n = len(X)
    node_ids = [f"gw-{i % 4}" for i in range(n)]
    patient_ids = [f"pat-{i % 10}" for i in range(n)]

    cfg = ChainConfig(
        rpc_url=args.rpc,
        contract_address=args.contract,
        threshold=args.threshold,
        dry_run=not args.live,
    )
    run_pipeline(model, X, node_ids, patient_ids, cfg)


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