D-FLARE Pipeline Modules and the O(1)-per-row Sliding Window Counting Method

Technical note (for thesis appendix) – English edition

Overview

This note documents the three processing modules in the D-FLARE pipeline and a high-performance O(1)-per-row sliding window counting method used for temporal features. The document is self-contained and intended to be referenced as a methods appendix.

1. log_cleaning.py — Raw Log Cleaning & Standardization

• Ingests raw log files (multi-file, large-file streaming).
• Removes invalid characters/whitespaces and unnecessary fields (e.g., raw_log).
• Normalizes key fields; constructs a canonical column order (idseq in the first column).
• Outputs: cleaned CSV (processed_logs.csv) and a unique-values list (log_unique_values.json).
• Supports pre-sampling/post-sampling/clean-only modes with chunked processing to prevent OOM.
• CLI provides progress bars, colored prompts, and guard rails; a QUIET flag suppresses prompts when called from the pipeline/UI.

2. log_mapping.py — Categorical Mapping & Column Ordering

• Maps categorical fields to integers using stable dictionaries (unknown → sentinel).
• Ensures idseq is preserved and ordered first; removes raw_log to save space.
• Optionally checks coverage against log_unique_values.json before mapping (reporting uncovered values).
• Keeps a stable core column order while preserving any extra columns after it.
• Streaming (chunked) processing with progress bars; QUIET mode supported.

3. feature_engineering.py — Feature Engineering (Network Security Analytics)

• Five feature families (each switchable): (1) traffic statistics, (2) protocol/port features, (3) time-window features, (4) relational features, (5) anomaly indicators.
• Implements low-cost, streaming-friendly transforms for TB-scale data.
• Categorical outputs from engineering (e.g., dstport_bucket, proto_port, sub_action, svc_action) are numeric-encoded: dstport_bucket has a fixed mapping {unknown:0, well_known:1, registered:2, dynamic:3}; string composites are encoded via stable 32-bit hashes (md5 prefix).
• Provides an O(1)-per-row sliding window counter for temporal counts (srcip/dstip/pairs).

Pipeline Flow (ASCII Diagram)

```
Raw logs (txt/csv/gz) | v [ log_cleaning.py ] - Parse K=V lines - Normalize fields -
Remove raw_log - idseq preserved (1st col) - Write processed_logs.csv - Write
log_unique_values.json | v [ log_mapping.py ] - Coverage check (optional, pre-mapping) -
Map categorical fields to ints - Reorder columns (core first) - Write
preprocessed_data.csv | v [ feature_engineering.py ] - Add chosen feature families -
O(1)-per-row window counts - Encode engineered categoricals - Write engineered_data.csv
```

O(1)-per-row Sliding Window Counting

Goal: for each incoming row, derive counts over a recent time window (e.g., last 5 minutes) for keys such as srcip, dstip, and (srcip,dstip). A naïve approach re-scans the window at every row (O(W) per row). Our method maintains rolling aggregates to achieve constant time per row.

Data structures

• A deque of minute buckets: each bucket holds (minute, Counter_src, Counter_dst, Counter_pair).
• Three aggregate Counters: agg_src, agg_dst, agg_pair for the current window.

Algorithm

```
For each row (timestamp ts, src, dst): 1) minute_key = floor_to_minute(ts) 2) While window
is non-empty and (minute_key - window[0].minute) > WINDOW_SIZE: - pop left bucket -
subtract its counters from agg_src / agg_dst / agg_pair - remove zero/negative keys from
aggregates (cleanup) 3) If the last bucket.minute != minute_key, append a new empty bucket
for minute_key 4) Read current counts (before adding this row): cnt_src = agg_src[src]
cnt_dst = agg_dst[dst] cnt_pair = agg_pair[(src,dst)] 5) Update counters with this row:
bucket.Counter_src[src] += 1; agg_src[src] += 1 bucket.Counter_dst[dst] += 1; agg_dst[dst]
+= 1 bucket.Counter_pair[(s,d)] += 1; agg_pair[(s,d)] += 1
```

Complexity & properties

• Time: O(1) per row (amortized), O(N) end-to-end, independent of window size W.
• Memory: proportional to number of minute buckets within the window; each stores only counts observed in that minute.
• Correctness: Equivalent to naïve scanning (counts over rolling window), but using incremental updates.

Pseudocode

```
state: window = deque() // of (minute, c_src, c_dst, c_pair) agg_src, agg_dst, agg_pair =
Counter(), Counter(), Counter() for each row i with timestamp ts, src, dst: m =
floor_to_minute(ts) while window not empty and (m - window.front.minute) > WINDOW_MINUTES:
(old_m, csrc_old, cdst_old, cpair_old) = window.pop_front() agg_src -= csrc_old;
cleanup(agg_src) agg_dst -= cdst_old; cleanup(agg_dst) agg_pair -= cpair_old;
cleanup(agg_pair) if window empty or window.back.minute != m: window.push_back((m,
Counter(), Counter(), Counter())) (_, csrc, cdst, cpair) = window.back emit cnt_5m_srcip =
agg_src[src] emit cnt_5m_dstip = agg_dst[dst] emit cnt_5m_pair = agg_pair[(src,dst)]
csrc[src] += 1; agg_src[src] += 1 cdst[dst] += 1; agg_dst[dst] += 1 cpair[(src,dst)] += 1;
agg_pair[(src,dst)] += 1
```

Comparison with CMS-based Sliding Window (placeholder)

A dedicated section can be added to contrast the above exact-count method with Count-Min Sketch (CMS) over sliding windows (including error bounds, memory footprint, update/query complexity, and decay/aging strategies). Please insert empirical results and citations here.

I/O and Reproducibility Notes

• Output path policy: if an output filename is relative, it is resolved to the same directory as the input file.
• QUIET mode: when the pipeline calls modules programmatically, QUIET=True suppresses interactive prompts/prints.
• Stable categorical encoding: service is mapped via the unique-values list; engineered categorical fields are encoded to integers (fixed bucket mapping) or 32-bit stable hashes (proto_port/sub_action/svc_action).
• Timestamp bucketing uses floor('min') (not the deprecated 'T').