

asn_filters_strict.py — Strict Filters & Vulnerability Metric

1. Purpose & Role in the Platform

`asn_filters_strict.py` computes a **per-ASN strict filtering & vulnerability score**, called `strict_filter_score`, plus a set of supporting counters.

The script responds, for each monitored ASN:

1. **How safe is this ASN as a source for malicious/hijacked BGP announcements?**
– i.e. *Is this ASN “vulnerable” or is it a bad place for an attacker to originate fake routes?*
2. **If an attacker originates a forged announcement “from” this ASN, how likely is it to propagate across the Internet?**
– i.e. *Does the surrounding routing environment and filtering behavior allow bogus routes to leak and spread, or are they typically blocked?*

The logic is based purely on **observable evidence**:

- Live and recent **BGP updates** from RIPE RIS (RIS Live).
- **RPKI ROAs** stored in the local `roas` table.
- **IRR route objects** (from NTT/RADB/RIPE whois).
- Optional **AS relationships** and **IXP route-server lists** from pre-populated local tables.

From these, the script builds for each ASN a multi-dimensional profile:

- How often that ASN appears to be the **most likely culprit** on the AS-PATH of **invalid / suspicious routes** (`leaked_suspect`).
- How often that ASN appears to **block suspicious routes**, only passing **clean** ones even when some vantage points see invalid announcements for the same prefix (`blocked_suspect`).

- How often it sits on paths with **RPKI-invalid origins, wrong origin vs ROA/IRR, over-max-length, valley-free violations, MOAS, no IRR support**, etc.

The result is a **0–1 score per ASN**:

- **Higher score** → ASN looks **harder to abuse** and forged announcements starting from it are **less likely to propagate widely** (strict environment, many “blocking” events).
 - **Lower / moderate score** → ASN is **more vulnerable** and/or sits in an environment where **invalid announcements often propagate**, with little evidence of blocking.
-

2. High-Level Flow

At a high level, `asn_filters_strict.py` operates in two clearly separated phases:

1. **Data collection and accumulation**
2. **Batch score computation (executed only at explicit scoring time)**

A single execution (depending on mode) proceeds as follows:

2.1 Initialize Database Schema

At startup, the script ensures that all required database tables and indexes exist.

This includes (but is not limited to):

- `asn_data` (input ASN universe)
- `bgp_updates` (raw BGP observations)
- `asn_filter_features` (final strict filtering scores)
- `_last_asn_snapshot` (latest per-ASN snapshot)
- auxiliary tables such as `as_rel`, `roas`, `irr_routes`, `ixp_route_servers`

If required, lightweight schema migrations are applied (e.g. adding new columns to `asn_filter_features`).

2.2 Load Target ASNs

The script loads the list of ASNs to monitor from:

`asn_data.asn`

If no ASNs are present, execution stops early.

This ASN list defines:

- which ASNs are subscribed to in RIS Live,
 - which ASNs are eligible for scoring.
-

2.3 Optional Reset of Previous Observations

If the `--truncate-updates` flag is provided, the script clears the `bgp_updates` table before starting a new collection window.

This allows clean re-collection runs when desired.

2.4 Asynchronous Data Collection Phase (Continuous / Time-Bound)

During the **collection phase**, the script runs several collectors in parallel inside a shared `aiohttp` session, protected by rate limiting (`TokenBucket`) and monitored by a background heartbeat task.

The following collectors are involved:

- `collect_bgp_updates()`
Subscribes to RIPE RIS Live via WebSocket and records all relevant BGP UPDATE events involving the target ASNs into `bgp_updates`.

- **collect_as_relationships()**
Offline operation: uses the existing `as_rel` table as-is (no HTTP or CAIDA downloads in this script).
- **collect_ixp_route_servers()**
Offline stub: ensures the `ixp_route_servers` table exists; no PeeringDB calls are performed.
- **collect_irr_routes()**
Queries IRR WHOIS servers (NTT, RADB, RIPE) and stores discovered `route` / `route6` objects per ASN.
- **collect_roas()**
Offline operation: ensures the `roas` table exists and reports its current row count; no live RIPEstat ROA downloads occur here.

Important:

During this phase, **no strict filtering scores are computed or written**.
All observations are accumulated only in `bgp_updates`.

2.5 Extension of AS-Relationship Context

After the initial RIS capture window completes, the script may extend AS-relationship coverage by:

- collecting additional origin ASNs observed in `bgp_updates`,
- merging them with the original ASN set,
- reusing the `as_rel` table to improve attribution context.

This step improves downstream scoring accuracy but does not trigger scoring itself.

2.6 Offline RPKI Re-Evaluation

Once collection is finished:

- an in-memory ROA index (`RoaIndex`) is built from the `roas` table,
- each row in `bgp_updates` is re-evaluated offline to determine:
 - `valid`
 - `invalid`
 - `not_found`

based on prefix coverage, origin ASN, and ROA maxLength.

This step enriches `bgp_updates` but still does not compute per-ASN scores.

2.7 Batch Strict Filtering Score Computation (Delayed)

Strict filtering scores are computed **only when an explicit scoring phase is triggered**, for example:

- at the end of a one-shot run,
- at scheduled intervals (sanity / final windows),
- or during a controlled shutdown.

During this batch phase, `compute_and_store_final_scores()` is executed:

- Reads all `bgp_updates` within the selected time window.
- Deduplicates repeated noisy events.
- Derives legitimate origins and prefix constraints from ROAs and IRR data.
- For each observed anomaly:
 - classifies RPKI violations, wrong-origin cases, over-max-length, missing IRR, valley-free violations, MOAS.

- attributes suspicion to one or two candidate ASNs in the AS-PATH (`attribute_edge_candidates()`).
 - determines whether each ASN is more likely **leaking** or **blocking** suspicious routes.
- Aggregates evidence per ASN into:
 - leaked_suspect
 - blocked_suspect
 - opportunities
 - vantage_point_count
 - anomaly counters
 - evidence_mix
- Computes:
 - `strict_filter_score`
 - `strict_confidence`

All results are written in a **single batch transaction** into:

- `asn_filter_features`
- `_last_asn_snapshot`

Until this batch computation completes and commits, other connections may observe zero rows in these tables — this is expected behavior.

2.8 Optional CSV Export

If `--csv-out` is provided, a CSV summary of all ASNs and their computed metrics is written after scoring completes.

2.9 Heartbeat and Progress Reporting

Throughout execution, a heartbeat task periodically prints:

- total rows in `bgp_updates`
- total rows in `asn_filter_features`

This heartbeat reflects **I/O and accumulation progress only** and should not be interpreted as an indicator of live scoring progress.

3. Database Schema & Inputs

This section describes the database schema used by `asn_filters_strict.py`, distinguishing clearly between:

- **input / raw observation tables**
 - **derived scoring outputs**
 - **auxiliary policy / context tables**
-

3.1. Key Tables

1) `asn_data(asn)`

Purpose

Seed table defining the universe of ASNs to monitor and score.

- Must be populated by an external process.

Contains exactly one column:

```
asn INTEGER PRIMARY KEY
```

-

Usage

- Defines which ASNs are subscribed to in RIS Live.
- Defines the full set of ASNs for which output rows will eventually exist in scoring tables.

If `asn_data` is empty, the script exits early.

2) `bgp_updates`

Raw BGP observation table populated during the **collection phase only**.

Created by `ensure_core_tables()`:

```
CREATE TABLE IF NOT EXISTS bgp_updates(  
    rowid INTEGER PRIMARY KEY AUTOINCREMENT,  
    prefix TEXT NOT NULL,  
    origin_asn INTEGER,  
    as_path TEXT,  
    rpki_status TEXT,  
    vantage_point TEXT,  
    seen_ts REAL  
);
```

Semantics

- One row \approx one observed RIS Live UPDATE or RIB entry relevant to the monitored ASNs.
- `rpki_status` is initially stored as reported or placeholder.

Important

- `rpki_status` is **always recomputed later offline** using the local ROA index (`RoaIndex`).
 - This table grows continuously during collection and is the **sole input** for scoring.
-

3) `asn_filter_features` (Final Scoring Output)

Primary per-ASN output table, populated **only during batch scoring**.

```
CREATE TABLE IF NOT EXISTS asn_filter_features(  
    asn INTEGER PRIMARY KEY,  
    strict_filter_score REAL,  
    strict_confidence REAL,  
    leaked_suspect REAL,  
    blocked_suspect REAL,  
    opportunities REAL,  
    rpki_invalid_cnt REAL,  
    wrong_origin_cnt REAL,  
    over_maxlen_cnt REAL,  
    valley_violation_cnt REAL,  
    moas_cnt REAL,  
    no_irr_support_cnt REAL,  
    evidence_flags TEXT,  
);
```

Population semantics

- Rows are written **only when** `compute_and_store_final_scores()` runs.
- All rows are inserted/updated in a **single batch transaction**.
- During live collection, this table may legitimately contain **zero rows**.

Interpretation (per ASN)

- `strict_filter_score`
Core strict-filtering / vulnerability metric (0–1).
 - `strict_confidence`
Confidence level based on observed opportunities and vantage-point diversity.
 - `leaked_suspect`
Weighted evidence that the ASN leaks or permits suspicious announcements.
 - `blocked_suspect`
Weighted evidence that the ASN blocks suspicious announcements.
 - `opportunities`
`leaked_suspect + blocked_suspect` (total observable “experiments”).
 - `*_cnt` fields
Aggregated anomaly counters:
 - RPKI invalid
 - wrong origin
 - over-max-length
 - valley-free violations
 - MOAS
 - missing IRR support
 - `evidence_flags`
Compact human-readable summary of aggregated evidence (e.g. `obs:5,moas:2`).
-

4) `_last_asn_snapshot`

Denormalized snapshot of the **most recent scoring run**.

- Contains the same per-ASN metrics as `asn_filter_features`.

- Updated in parallel during scoring.
- Optimized for fast “latest state” queries without scanning historical outputs.

5) **asn_filter_features_sanity** (Short-Window Output)

Auxiliary scoring table for **short sanity windows** (e.g. 24h).

- Same schema as **asn_filter_features**.
- Populated independently during sanity scoring runs.
- Used exclusively for:
 - pipeline validation
 - progress monitoring
 - early signal detection

This table does **not** replace or override final scoring.

6) **as_rel** (Optional, Pre-Populated)

```
CREATE TABLE IF NOT EXISTS as_rel(  
    asn INTEGER,  
    neighbor INTEGER,  
    relation TEXT  
);
```

Purpose

Stores CAIDA-style AS relationships:

- provider
- customer

- peer

Behavior

- No HTTP or CAIDA downloads occur in this script.
- Existing data is reused as-is.

Usage

- Edge classification in `valley_free_violated()`
 - Candidate weighting in `attribute_edge_candidates()`
-

7) `ixp_route_servers` (Optional, Pre-Populated)

```
CREATE TABLE IF NOT EXISTS ixp_route_servers(  
    asn INTEGER PRIMARY KEY  
);
```

Purpose

Identifies IXP route servers.

Usage

- Such ASNs are excluded from attribution logic.
 - They are treated as passive reflectors, not culpable entities.
-

8) `irr_routes`

```
CREATE TABLE IF NOT EXISTS irr_routes(  
    prefix TEXT,  
    origin_asn INTEGER,  
    source TEXT  
);
```

Population

- Filled by `collect_irr_routes()` via IRR WHOIS queries.

Usage

- Determines whether `(prefix, origin_asn)` pairs have IRR backing.
 - Absence contributes to `no_irr_support_cnt`.
-

9) `roas`

```
CREATE TABLE IF NOT EXISTS roas(  
    prefix TEXT,  
    asn INTEGER,  
    max_length INTEGER  
);
```

Population

- Filled by an external ROA collector.

Usage in this script

- Ensures table exists.
 - Loads all rows into `RoaIndex`.
 - Recomputes `rpki_status` for all `bgp_updates`.
 - Derives legitimate origins and max-length constraints for scoring.
-
-

4. Data Collection: RIS Live, IRR, ROAs, AS Relationships

This section describes **how raw routing and policy data is collected or loaded**, and **how it is later consumed during batch scoring**.

Importantly, **data collection and score computation are decoupled**: collection populates raw tables, while interpretation happens later.

4.1. BGP Updates via RIPE RIS Live

`collect_bgp_updates()`

Opens a WebSocket connection to:

`wss://ris-live.ripe.net/v1/ws/`

-
- Builds a subscription:
 - Filtered by the target ASN set loaded from `asn_data`
 - Optionally includes the initial RIB snapshot when `--ris-initial-state` is enabled
- Listens continuously during the configured capture window (`--ris-duration`)

Each incoming RIS UPDATE message is parsed and normalized into **one row in `bgp_updates`**, containing:

- `prefix`
Normalized using `ip_network(prefix, strict=False)`
- `origin_asn`
Extracted as the last ASN in the AS_PATH
- `as_path`
String representation of the AS_PATH
- `rpki_status`
Initially stored as reported (or placeholder); later overwritten by offline RPKI

recomputation

- `vantage_point`
RIS monitor identifier
- `seen_ts`
Observation timestamp (float / epoch)

Important behavioral note

During this phase:

- rows are **only appended to `bgp_updates`**
- **no scoring or per-ASN aggregation is performed**

Why this matters for vulnerability & propagation

The system does not simulate attacks or filtering behavior.

Instead, it observes **real BGP announcements as seen from many independent vantage points**.

Each accepted or rejected announcement acts as a natural experiment, revealing:

- where suspicious routes propagate,
- where they are blocked,
- and which AS environments are permissive or strict.

4.2. IRR Route Objects

`collect_irr_routes()`

- Iterates over ASNs in `asn_data`
- For ASNs not yet present in `irr_routes`, queries multiple IRR WHOIS servers:

- `rr.ntt.net`
- `whois.radb.net`
- `whois.ripe.net`
- Parses `route` / `route6`, `origin`, and `source` attributes via `parse_irr_whois()`

Stores deduplicated tuples:

`(prefix, origin_asn, source)`

- in the `irr_routes` table

Timing note

IRR collection:

- runs independently of BGP update ingestion
- **does not directly affect scoring until the batch scoring phase**

Usage during scoring

For each `bgp_updates` row, the scorer checks whether a corresponding IRR route object exists:

- Exact match on `(prefix, origin_asn)`
- Or covering IRR objects (supernet cases)

If no such IRR object exists (and IRR coverage is expected to be non-empty), the event contributes to:

- `no_irr_support_cnt`
-

Interpretation

Frequent involvement in routes without IRR backing suggests that an ASN:

- does not maintain proper IRR hygiene, **or**
- is associated with suspicious or hijacked announcements

Both scenarios increase routing risk and lower trust in that ASN's filtering environment.

4.3. ROAs & RPKI Status (Offline Re-Evaluation)

RoaIndex

- Loads all ROAs from the local `roas(prefix, asn, max_length)` table
- Maintains separate IPv4 / IPv6 indices

Implements:

```
validate(prefix, origin_asn)
```

-

Validation logic:

- **valid**
A ROA covers the prefix with matching origin and acceptable maxLength
 - **invalid**
A ROA covers the prefix but with a different origin or insufficient maxLength
 - **not_found**
No covering ROA exists
-

Offline recomputation step

After the collection phase completes:

- A `RoaIndex` is built in memory
- Every row in `bgp_updates` is re-evaluated
- The `rpki_status` field is overwritten with the locally computed result

This guarantees **consistent RPKI interpretation**, independent of live RIS annotations.

Usage during scoring

RPKI data is used to:

- identify RPKI-invalid announcements
- build per-prefix maps:
 - `legit_origins_per_prefix`
 - `roa_maxlen_per_prefix`
- detect over-max-length and wrong-origin events

Interpretation

ASNs frequently observed on paths carrying RPKI-invalid routes are more likely to:

- operate in permissive filtering environments, or
- participate in regions where invalid routes propagate

Both are strong indicators of reduced routing strictness.

4.4. AS Relationships & Valley-Free Policy

AS relationship data (`as_rel`)

- Loaded from an existing local table
- No CAIDA downloads or HTTP fetches occur in this script

Key helpers:

- `classify_edge(u_as, v_as, as_rels)`
Returns relationship type: provider, customer, peer, or unknown
- `valley_free_violated(path, as_rels)`
Implements the classical valley-free routing model:
 - provider/peer edges = uphill
 - customer edges = downhill
 - a violation occurs if a path goes downhill and later uphill again

Usage during scoring

For each prefix's observed paths:

- valley-free violations are detected
- ASNs appearing on such paths receive increments to `valley_violation_cnt`

Interpretation

Frequent participation in valley-violating paths signals:

- misconfigurations
- route leaks
- non-standard peering behavior

Such environments are typically more exploitable for route hijacking.

4.5. MOAS (Multiple Origin AS) Detection

From accumulated observations:

- `origins_per_prefix[pfx]` collects all observed origin ASNs per prefix

If:

```
len(origins_per_prefix[pfx]) > 1
```

- the prefix is classified as MOAS

All ASNs appearing on paths for MOAS prefixes receive increments to:

- `moas_cnt`

Interpretation

MOAS scenarios are not always malicious, but they:

- blur origin authority
- reduce anomaly clarity
- make hijacks harder to distinguish

ASNs frequently involved in MOAS environments operate in less strict routing ecosystems.

5. Evidence Model per BGP Update

This is the core of the logic.

5.1. Preprocessing & Deduplication

```
compute_and_store_final_scores():
```

1. Read all rows from `bgp_updates` in chunks (to limit memory).
2. For each row:
 - Normalize `prefix` to `ip_network`.
 - Parse `as_path` into `path_list` (list of ints).
 - Normalize `rpki_status` to lowercase; if missing, validate via `RoaIndex`.
 - Keep `(prefix, net, origin_asn, path_list, rpki_status, vp, ts)` as a structured `updates` list.
3. For each prefix:
 - Build `origins_per_prefix[pfx]` – all origins seen.
 - Build `counts_per_prefix_origin[pfx][origin_asn]` – how many times each origin is observed.
4. Build `legit_origin_per_prefix` and `roa_maxlen_per_prefix`:
 - If `roas` table is present:
 - For each (ROA prefix, asn, max_length):
 - Normalize prefix.
 - Treat this `(prefix, asn)` as **legitimate** and store its `max_length`.
 - For prefixes that **have no ROA** but are frequent in BGP:
 - If an origin appears often (≥ 3 times), treat it as a **pseudo-legitimate** origin with weight 0.40.
 - Build super-prefix relations, so that ROAs on super-prefixes also inform legitimacy for sub-prefixes:
 - For each `small` prefix nested in a `big` prefix, `superprefix_origins[small]` inherits origins from `big`.

5. Detect MOAS prefixes:

- `moas_prefixes = { pfx | len(origins_per_prefix[pfx]) > 1 }`.

6. For each prefix `pfx`, group observations: `by_prefix[pfx] = list of updates`.

5.2. Per-Prefix Observation Loop

For each prefix `pfx`:

- Initialize:
 - `saw_suspect = False` — whether any observation is “suspicious”.
 - `edge_susp` — weighted evidence that a given AS **leaked** suspicious routes.
 - `edge_clean` — weighted evidence that a given AS was seen on **only clean** routes when suspicious ones existed.
 - `vp_susp / vp_clean` — vantage points where each edge was seen suspicious / clean.
 - `path_has_valley` — whether some path for this prefix violates valley-free.
- For each observation `o` in `obs`:

Build a **deduplication key**:

- `(prefix, origin_asn, AS_PATH-string, vantage_point)`
- If the same key was seen in the last `DEDUP_TTL` seconds (default 120s), **skip** to avoid counting noisy repetitions.

Determine **candidate edges**:

```
edges = attribute_edge_candidates(o["path_list"], o["origin_asn"],
as_rels, ixp_rs_set)
```

- — This returns up to 2 ASNs with weights that sum to 1. They are the candidates most likely to be:

- the **origin ASN** itself,
 - its neighbours (especially downstream / customers),
 - excluding IXP route-servers. The script excludes IXP route-servers because they are just passive route reflectors, not the networks that make origin/filtering decisions. If we included them, we would wrongly penalize RSs and mask the real culprits (origin/near-origin ASs), ruining the vulnerability metric.
- If no edges → skip (insufficient path info).
 - Determine `rpki_status`:
 - Use `o["rpki_status"]` if present, else consult `RoaIndex`.
 - Initialize `flags = {}` (boolean) and evaluate anomalies:
 - a) RPKI invalid**
 - If `rpki_status == "invalid" → flags["rpki_invalid"] = True.`
 - b) Wrong origin vs ROA / pseudo-legitimate origins**
 - Let `legit_asns = legit_origin_per_prefix[pfx].keys()` when available.
 - If `origin_asn` is not in `legit_asns` (and the set is non-empty) → `flags["wrong_origin"] = True.`
 - c) Over-max-length**
 - Let `roa_ml = roa_maxlen_per_prefix[pfx]`.
 - If there is ROA data and the observed prefix length is **longer than all allowed max_length** → `flags["over_maxlen"] = True.`
 - d) IRR support**
 - Look up (prefix, origin_asn) in `irr_exact`.

- If there is **no IRR backing**, set `has_irr = False`. In suspicious cases, this leads to increments in `no_irr_support_cnt`.
- **e) Valley-free**
 - For each path, check `valley_free_violated(o["path_list"], as_rels)`.
 - If any path is valley-violating, mark `path_has_valley = True`.
- If **any flag is set** → `suspect = True`, else `False`.
- **Attribution step:**
 - If `suspect`:
 - Mark `saw_suspect = True`.
 - For each candidate `e` with weight `w`:
 - `edge_susp[e] += w`.
 - Add vantage point to `vp_susp[e]`.
 - Update anomaly counters for that edge:
 - `rpki_invalid_cnt[e] += w` if `rpki_invalid`.
 - `wrong_origin_cnt[e] += w` if `wrong_origin`.
 - `over_maxlen_cnt[e] += w` if `over_maxlen`.
 - `no_irr_support_cnt[e] += w` if `has_irr == False`.
 - If **not suspect**:
 - For each candidate `e` with weight `w`:

- `edge_clean[e] += w.`
- Add vantage point to `vp_cln[e]`.

After all observations for `pfx`:

- If `path_has_valley`:
 - Every ASN that appeared in `edge_susp` or `edge_clean` gets `valley_violation_cnt[asn] += 1.0`.
- If `pfx` is MOAS:
 - Every ASN in those edges gets `moas_cnt[asn] += 1.0`.
- **Leaked vs Blocked evidence for this prefix**
 - For each ASN `a` in `edge_susp`:
 - `leaked_suspect[a] += edge_susp[a]`.
 - `vp_seen[a].update(vp_susp[a])`.
 - Optionally print a debug log if this is first evidence.
 - If `saw_suspect` is `True`:
 - There exist suspicious announcements for this prefix.
 - For any ASN `a` that appears only in clean edges (`edge_clean`) but not in `edge_susp`:
 - `blocked_suspect[a] += edge_clean[a]`.
 - `vp_seen[a].update(vp_cln[a])`.
- This is the **key mechanism** that differentiates:
 - ASNs that **participate in suspicious routes** (potentially permissive),

- from ASNs that are only seen on **clean routes when suspicious ones exist elsewhere** (likely doing strict filtering, or at least upstream of strict filtering).
 - **Evidence mix**

Independently, the script tracks `evidence_mix[e]`:

 - For each prefix, every ASN `e` that appears in the candidate edges:
 - `evidence_mix[e]["obs"] += 1.0.`
 - If the prefix is MOAS → `evidence_mix[e]["moas"] += 1.0.`
 - `evidence_mix` is later converted to a compact string summarizing how many prefixes / MOAS contexts contributed evidence for an ASN.
-

6. Per-ASN Aggregation & Score Computation

After the per-prefix loop:

Identify ASNs with any evidence

```
asns_all_evidence = set(leaked_suspect) | set(blocked_suspect) |
                    set(valley_violation_cnt) | set(moas_cnt) |
                    set(no_irr_support_cnt)
```

- 1.
2. For each such `asn`:
 - `leaked = leaked_suspect.get(asn, 0.0)`
 - `blocked = blocked_suspect.get(asn, 0.0)`
 - `opp = leaked + blocked` (total “opportunities”)
 - `vps = len(vp_seen.get(asn, set()))`

- `rpki_c = rpki_invalid_cnt.get(asn, 0.0)`
- `wrong_c = wrong_origin_cnt.get(asn, 0.0)`
- `over_c = over_maxlen_cnt.get(asn, 0.0)`
- `valley_c = valley_violation_cnt.get(asn, 0.0)`
- `moas_c = moas_cnt.get(asn, 0.0)`
- `noirr_c = no_irr_support_cnt.get(asn, 0.0)`

3. Minimum evidence filters

Command-line flags:

- `--min-ops (min_ops)` – minimum number of opportunities required.
- `--min-vps (min_vps)` – minimum number of distinct vantage points.

4. If `opp < min_ops` or `vps < min_vps`, the ASN is considered **under-observed**:

- The script calls `upsert_null_row(asn)`:
 - All metrics in `asn_filter_features` set to `NULL` for that ASN.

5. This prevents the UI / ML layer from over-interpreting noise.

Weighted anomaly map

For ASNs with enough evidence:

```
sigmap = {
    "rpki_invalid": rpki_c,
    "wrong_origin": wrong_c,
    "over_maxlen": over_c,
    "valley_violation": valley_c,
    "moas": moas_c,
    "no_irr_support": noirr_c,
}
```

The script uses a set of **hand-tuned weights**:

```
WEIGHTS = {  
    "rpki_invalid":    0.38,  
    "wrong_origin":    0.24,  
    "over_maxlen":     0.14,  
    "valley_violation": 0.14,  
    "moas":            0.04,  
    "no_irr_support":  0.06,  
}
```

6. Intuition:

- **RPKI invalid** and **wrong origin** are the strongest signals of overt hijack.
- Over-max-length and valley violations are strong structural red flags.
- MOAS and no IRR support are weaker contextual signals, but still relevant.

Bayesian blocked vs leaked ratio

```
bayes_score(blocked, leaked):
```

```
alpha = 1.0 + blocked
```

```
beta = 1.0 + leaked
```

```
bayes = alpha / (alpha + beta)
```

7.

- If **blocked** \gg **leaked**, **bayes_score** ≈ 1 \rightarrow ASN typically appears to **block** suspicious announcements.
- If **leaked** \gg **blocked**, **bayes_score** approaches **0** \rightarrow ASN leans towards **leaking** suspicious routes.
- The **+1** smoothing prevents division by zero and encodes a weak prior (0.5 with no data).

8. Nonlinear anomaly aggregation

```
aggregate_weighted_score(sigmoid, blocked, leaked):
```

First apply a **saturating nonlinear function**:

```
def sig(x):  
    return 1.0 - exp(-0.5 * max(0, x))
```

- - For small x , $\text{sig}(x) \approx 0.5 * x$.
 - For large x , $\text{sig}(x) \rightarrow 1$ (saturates).
- Then compute a **weighted average** of these sig values with **WEIGHTS**.

Finally, combine:

```
score = 0.6 * anomaly_agg + 0.4 * bayes_score(blocked, leaked)
```

-
- 9. Interpretation:
 - The **anomaly_agg** term ensures we need **repeated independent evidence** of anomalies to confidently judge an ASN's environment.
 - The **bayes_score** term encodes the **direction** of behavior:
 - Many blocks vs many leaks.

10. This yields **strict_filter_score** in $[0, 1]$:

- **Close to 1:**
 - The ASN is frequently involved in contexts where anomalies are detected *but* the evidence indicates **blocking** behavior or a strongly constrained environment (high blocked, low leaked, consistent flags).
 - This is interpreted as **hard to abuse** for hijacking; forged routes originating from here are **less likely to propagate widely**.
- **Intermediate values (0.4–0.7):**
 - Mixed behavior or low anomaly counts.

- The ASN may be partially strict or partially permissive; results must be combined with other platform metrics.
- **Lower values:**
 - The per-ASN evidence suggests **more leaking than blocking** in the observed opportunities.
 - This points to a **more vulnerable environment**: an attacker has a better chance to make forged announcements propagate.

Confidence metric

```
compute_confidence(opp, vps, OP_REF=50, VP_REF=50):
```

```
ce = min(1.0, opp / OP_REF)
```

```
cv = min(1.0, vps / VP_REF)
```

```
confidence = ce * cv
```

11.

- **opp** → how many **independent opportunities** we saw for this ASN (cases where we could compare suspicious vs clean behavior).
- **vps** → number of distinct vantage points that contributed evidence.

12. Intuition:

- If we observed ~50 or more opportunities and ~50 or more vantage points → confidence saturates near 1.
- With few opportunities / vantage points, confidence is low even if the score is extreme.

13. Evidence string

The script stores a compact **evidence_flags** string, derived from **evidence_mix[asn]**:

- Example: **obs:12,moas:3**.

- This is useful for debugging and for a UI that wants to display *why* the ASN got a certain score.

14. Upsert into DB

```
upsert_real_row(conn, row, print_updates):
```

- Inserts/updates the ASN's metrics in:
 - `asn_filter_features`
 - `_last_asn_snapshot`
- If `--print-asn-updates` is set, prints a detailed one-line summary per ASN.

7. Interpretation: Vulnerability & Propagation

Putting everything together:

- Each BGP update is treated as a **mini-experiment**:
 1. What origin AS was used?
 2. Is it consistent with ROAs / IRR?
 3. Did paths violate valley-free?
 4. Are there MOAS behaviors?
 5. For the same prefix, do some vantage points see invalid announcements while others see clean announcements?
- For each ASN, across many such experiments, we estimate:
 1. **How often it is implicated as the most likely source/culprit of suspicious routes** (leaked_suspect).
→ This reflects how **permissive** it is as an origin or near-origin platform.
 2. **How often it is only seen on clean paths when suspicious versions exist elsewhere** (blocked_suspect).

→ This reflects **strict filtering**, either as an origin ASN doing best practices or as a transit ASN that enforces ROV / IRR / strict policies.

3. **How strong and consistent the anomaly patterns are** (RPKI invalid, wrong origin, over max-len, valley, MOAS, no IRR).

- The final `strict_filter_score` and `strict_confidence` thus capture:

“If someone tried to originate or relay hijacked announcements through this ASN, how likely are those announcements to be accepted and propagated at scale, versus being filtered or never appearing at all?”

In other words:

- **High `strict_filter_score` & high confidence** →
 - **Low vulnerability** (bad choice for attacker).
 - **Low propagation likelihood** for forged routes starting here.
- **Low/moderate `strict_filter_score` with enough confidence** →
 - **High vulnerability** (good candidate for attacker).
 - **Higher propagation likelihood** for hijacked routes originated through this ASN.

This is exactly aligned with the platform’s focus: **classifying ASNs as vulnerable vs non-vulnerable sources and estimating propagation strength** of forged announcements originating from them.

8. Sanity / Short-Window Scoring Mode

This section describes the **short-window (“sanity”) scoring mechanism**, which is intentionally separated from the final 7-day scoring logic.

The purpose of this mode is **validation, monitoring, and early signal detection**, not authoritative risk assessment.

8.1. Motivation and Design Rationale

Strict filtering behavior is often **sparse and event-driven**.

Many ASNs may not exhibit observable opportunities (leaks or blocks) over short periods.

As a result:

- computing a “final” strictness score too early would produce:
 - false zeros,
 - misleading confidence,
 - unstable rankings.
- however, waiting multiple days without feedback introduces operational blind spots.

The **sanity / short-window mode** exists to solve this tension.



It provides:

- early visibility into pipeline correctness,
- confirmation that data collection and attribution work as intended,
- preliminary signals for highly active ASNs,
- without polluting or overriding final results.

8.2. Separation from Final Scoring

Sanity scoring is **fully isolated** from final scoring:

Aspect	Sanity Mode	Final Mode
Time window	Short (e.g. 24h)	Long (e.g. 7 days)

Output tables	<code>asn_filter_features_sanity</code>	<code>asn_filter_features</code>
Snapshot table	<code>_last_asn_snapshot_sanity</code>	<code>_last_asn_snapshot</code>
Intended use	Monitoring / validation	Analysis / ML / reporting
Authoritative	 No	 Yes

At no point does sanity scoring overwrite or modify final scoring outputs.

8.3. Sanity Scoring Tables

8.3.1. `asn_filter_features_sanity`

This table mirrors the schema of `asn_filter_features`, but stores results computed over a **short sliding window**.

Key characteristics:

- populated only when sanity scoring is triggered,
- overwritten on each sanity run,
- may legitimately contain:
 - partial coverage,
 - many NULL scores,
 - low confidence values.

This is expected and correct behavior.

8.3.2. `_last_asn_snapshot_sanity`

A denormalized snapshot of the most recent sanity scoring run.

- optimized for quick inspection and dashboards,
- contains only the latest short-window results,
- never mixed with final snapshots.

8.4. Time Window Semantics (`window_sec`)

Both sanity and final scoring operate on **explicit time windows**, defined by:

```
seen_ts >= (now - window_sec)
```

In sanity mode:

- `window_sec` is typically small (e.g. 86,400 seconds = 24h),
- only recent `bgp_updates` rows are considered,
- older observations are ignored even if still present in the database.

This ensures that sanity results always reflect **current behavior**, not historical accumulation.

8.5. `computed_ts` and Temporal Traceability

Each sanity scoring run records a `computed_ts` timestamp.

This allows downstream consumers to:

- distinguish stale vs fresh sanity results,

- correlate score changes with time,
- audit when each window was computed.

This field is especially important for monitoring systems and debugging.

8.6. Expected Behavior and Interpretation

It is **normal and expected** that sanity scoring produces:

- fewer ASNs with non-NULL scores,
- lower `strict_confidence` values,
- sparse opportunities for most ASNs.

Sanity results should **not** be interpreted as:

- final risk rankings,
- absence of strict filtering,
- proof that an ASN is safe or unsafe.

Instead, they answer questions such as:

- Is the pipeline ingesting data correctly?
 - Are anomalies being detected and attributed?
 - Are highly active ASNs already showing signal?
 - Is the scoring logic stable over time?
-

8.7. Interaction with Daemon Mode

When running in daemon mode:

- data collection runs continuously,
- sanity scoring is triggered periodically (e.g. every 24h),
- final scoring is triggered on a much longer cadence (e.g. every 7 days).

Sanity scoring:

- does **not block** data collection,
- runs independently,
- can be disabled or tuned without affecting final scoring.

8.8. Why Sanity Scoring Is Not Optional

Without a sanity mode:

- pipeline failures might go unnoticed for days,
- schema or logic regressions would surface only at final scoring,
- operators would lack visibility into progress and coverage.

The sanity / short-window mode is therefore a **first-class operational feature**, not a convenience.

9. CLI & Operational Usage (Updated)

This section documents the **command-line interface**, execution modes, and recommended operational workflows for `asn_filters_strict.py`.

The script supports **multiple execution modes**, explicitly separating **data collection** from **batch score computation**.

9.1. Core CLI Arguments

The following arguments are defined by `parse_args()`:

`--db`

Path to the SQLite database file.

- Default: resolved automatically to `PROJECT_ROOT/database/asn_data.db`
 - Must already exist and contain a populated `asn_data` table
 - The script will **not create a new empty database**
-

`--mode`

Execution mode. One of:

- `oneshot`
Run a single bounded collection window, then compute scores once and exit.
- `collect`
Run continuous data collection only (no scoring).
- `score`
Run a single scoring pass on existing data (no collection).
- `daemon`
Run continuous collection with **periodic sanity and final scoring**.

This flag defines the **overall lifecycle** of the script.

`--ris-duration`

Duration (in seconds) of each RIS Live capture window.

- Default: `600`
 - Used in:
 - `oneshot`
 - `collect`
 - `daemon` (per reconnect cycle)
-

`--ris-initial-state`

If set, requests the initial RIB snapshot (`initial_state=true`) at the start of a RIS Live subscription.

This is optional and increases initial data volume.

`--truncate-updates`

If set, deletes all existing rows from `bgp_updates` before starting collection.

Useful for:

- clean re-runs,
 - controlled experiments,
 - short test windows.
-

9.2. Scoring Control Arguments

These arguments influence **when and how scoring occurs**.

`--sanity-every-sec`

Interval (in seconds) at which **sanity / short-window scoring** is triggered in **daemon** mode.

- Typical value: **86400** (24h)
-

--final-every-sec

Interval (in seconds) at which **final / authoritative scoring** is triggered in **daemon** mode.

- Typical value: **604800** (7 days)
-

--window-sec

Time window (in seconds) used for scoring when running in **score** mode.

Controls which **bgp_updates** rows are considered:

```
seen_ts >= now - window_sec
```

--score-kind

Scoring type when running in **--mode score**:

- **sanity**
- **final**

Determines the output tables used.

--min-ops

Minimum number of observed opportunities (leaked + blocked) required for an ASN to receive a non-NULL score.

ASNs below this threshold will have:

- `strict_filter_score = NULL`
 - `strict_confidence = NULL`
-

`--min-vps`

Minimum number of distinct vantage points required to compute a score for an ASN.

Prevents overfitting to single-observer artifacts.

9.3. Output & Debugging Options

`--csv-out`

Path to a CSV file where per-ASN scoring results are exported **after scoring completes**.

CSV export occurs only in:

- `oneshot`
 - `score`
 - `daemon` (during scoring events)
-

`--print-asn-updates`

Print one summary line per ASN when scores are computed.

Useful for:

- validation
- debugging

- small test runs

--verbose

Enable detailed logging of collection, attribution, and scoring steps.

--raw-log

Path to an NDJSON file where raw RIS messages are written verbatim.

Intended strictly for debugging and forensic analysis.

9.4. Execution Semantics & Expected Behavior

Delayed scoring is normal

- During collection:
 - `bgp_updates` grows continuously
 - `asn_filter_features` may remain empty
- Scores are written **only when a scoring phase completes and commits**

This is expected behavior, especially with large ASN sets.

Ctrl+C behavior

In `oneshot` and `daemon` modes:

- A controlled shutdown (`Ctrl+C`) triggers cleanup
- Pending batch scoring transactions are finalized
- Results may appear **immediately after interruption**

This does not indicate a failure.

9.5. Typical Workflows

One-shot experiment (testing / debugging)

```
python3 asn_filters_strict.py \  
  --mode oneshot \  
  --ris-duration 900 \  
  --ris-initial-state \  
  --min-ops 5 \  
  --min-vps 3 \  
  --csv-out strict_scores.csv \  
  --verbose \  
  --print-asn-updates
```

Used for:

- validating logic
 - small experiments
 - controlled snapshots
-

Continuous production run (recommended)

```
python3 asn_filters_strict.py \  
  --mode daemon \  
  --ris-duration 3600 \  
  --sanity-every-sec 86400 \  
  --final-every-sec 604800
```

Behavior:

- continuous BGP ingestion

- daily sanity scoring
 - weekly authoritative scoring
-

9.6. Consuming Results

Downstream systems should read from:

- `asn_filter_features`
(final, authoritative scores)
- `_last_asn_snapshot`
(latest final snapshot)

Optional:

- `asn_filter_features_sanity` for monitoring only

These outputs are designed to feed:

- ML models
 - dashboards
 - decision engines evaluating routing strictness and vulnerability
-

10. Why the Data Sources Are Precise (Short Version)

1. Real BGP Control-Plane Evidence

All behavior is inferred from live RIS UPDATES — real announcements observed across multiple global vantage points, not simulations.

2. Authoritative ROA Validation

RPKI validation uses locally stored official ROAs, providing mathematically verifiable proof of validity/invalidity.

3. **IRR Route Objects from Trusted Registries**

IRR data comes directly from NTT, RADB, and RIPE WHOIS — the same sources used operationally by ISPs worldwide for filtering.

4. **Relationship Data Improves Attribution**

AS-relationship information helps correctly identify origin-side responsibility and prevents false attribution to passive nodes (IXP route-servers).

5. **Per-Prefix Cross-Vantage Comparison**

Comparing multiple vantage points for the same prefix (clean vs suspicious) yields strong, reliable signals for detecting filtering and leak behavior.

11. Limitations

1. **Visibility Bias**

Results depend on RIS Live visibility. If RIS vantage points do not observe certain paths or origins, some leak events may remain unseen.

2. **Heuristics for Valley-Free & MOAS**

Valley detection and MOAS classification are heuristic — they capture real anomalies but may occasionally flag complex multihoming or traffic engineering as suspicious.

3. **Short Collection Windows**

If *ris-duration* is small, few events may be observed. Low evidence leads to NULL scores and reduced confidence.

4. **No Access to Internal Policies**

The score reflects *observed behavior*, not internal router configurations. An ASN may have strict policies configured but rarely exposed during the observation window.