

asn_rpki_coverage.py — RPKI Validation Coverage Metric (Full Technical Documentation)

1. Purpose & Role in the Platform

asn_rpki_coverage.py computes RPKI validation coverage for every ASN by validating each of its announced prefixes against **RIPEstat's RPKI validation API**.

The metric quantifies:

- how many prefixes are cryptographically **valid** (**valid**)
- how many are **invalid** (**invalid**)
- how many are **not_found** (no matching ROA, or unknown/unsupported status mapped to **not_found**)
- how many return **errors** (**error**, only when an exception occurs during validation)

It reflects the real-world RPKI hygiene of an ASN and is essential for:

- vulnerability scoring
- hijack susceptibility analysis
- trustworthiness estimation
- filtering-resilience modeling
- ML-based risk classification

ASNs with high RPKI coverage have robust cryptographic validation; those with low coverage are significantly more vulnerable.

In the context of vulnerability, an ASN with low RPKI coverage creates a permissive environment where forged or hijacked announcements are far more likely to originate and propagate successfully, while high-coverage ASNs provide strong resistance against being used as an attack source.

2. High-Level Behavior

During each continuous round of operation, the script:

- Ensures DB schema exists (results table, default: `rpki_results`).
- Fetches all ASNs from `asn_data` in batches (LIMIT/OFFSET paging).
- For each ASN:
 - Retrieves all announced prefixes via RIPEstat (**announced-prefixes**).
 - For each prefix:
 - Performs RPKI validation using:
 - `/data/rpki-validation/data.json`
- For each ASN:
 - Counts results: `valid`, `invalid`, `not_found`, `error`.
 - Computes:
 - RPKI coverage percentage
 - `not_found` ratio
 - Prints real-time status per ASN (via an async stdout printer queue).
 - Upserts results into the results table.
- Repeats after a configurable interval (default: **300 seconds**).

This produces continuous, up-to-date RPKI hygiene metrics for all ASNs.

Important behavioral note (code-accurate):

- If announced-prefix retrieval fails or returns no prefixes, the ASN is treated as having **0 prefixes**, and the script writes `tot=0` with `cov=None` and `nf_ratio=None` (no separate “prefix-fetch error counter” is stored).
-

3. Metrics Produced

For each ASN, the script produces:

3.1 Primary Metrics

`rpki_coverage_pct`

- `100 * valid_prefixes / total_prefixes`
- Rounded to **1 decimal** (only when total > 0)

`rpki_notfound_ratio`

- `not_found_prefixes / total_prefixes`
- Rounded to **6 decimals** (only when total > 0)

3.2 Raw Counts

- `rpki_total_prefixes`
- `rpki_valid_prefixes`
- `rpki_invalid_prefixes`
- `rpki_notfound_prefixes`
- `rpki_error_prefixes`

3.3 Timestamp

- `updated_at` (UNIX timestamp, `int(time.time())`)

These correspond exactly to the columns created in the SQLite table.

4. Database Contract

4.1 Table Schema

The script creates the following table if missing (default name: `rpki_results`, configurable via `--result-table`):

```
rpki_results (  
    asn INTEGER PRIMARY KEY,  
    rpki_coverage_pct      REAL,  
    rpki_notfound_ratio    REAL,  
    rpki_total_prefixes    INTEGER,  
    rpki_valid_prefixes    INTEGER,  
    rpki_invalid_prefixes  INTEGER,  
    rpki_notfound_prefixes INTEGER,  
    rpki_error_prefixes    INTEGER,  
    updated_at             INTEGER  
)
```

4.2 Update Semantics

Each ASN's record is upserted via:

```
INSERT ... ON CONFLICT(asn) DO UPDATE SET ...
```

Each write is committed immediately (one commit per row), meaning each row is flushed to disk individually.

Code-accurate note:

- The CLI argument `--db-commit-every` exists and is passed into the DB writer, but the current script version commits **every row unconditionally** (the `commit_every`

parameter is not used to batch commits).

5. Core Concepts & Data Flow

5.1 ASN Enumeration

ASNs are fetched from `asn_data` using paging SQL with batching:

- `SELECT asn FROM asn_data [WHERE ...] LIMIT ? OFFSET ?`
- Batch size is controlled by `--batch-fetch` (default: `50000`).

5.2 Prefix Retrieval

For each ASN, the script queries RIPEstat:

- `/data/announced-prefixes/data.json?resource=AS<asn>`

Prefixes are cached in a TTL cache (6 hours) to reduce load:

- cache key: `asn` (integer)
- `maxsize=80000`
- `ttl=6 hours`

If prefix retrieval fails (exception), the function returns an empty list (`[]`).

5.3 RPKI Validation

For each prefix, the script queries RIPEstat:

- `/data/rpki-validation/data.json?resource=<asn>&prefix=<prefix>`

Code-accurate details:

- `resource` is sent as the numeric ASN string (e.g., "13335"), **not** "AS13335".

Result classification:

- If API returns `data.status` in ('valid', 'invalid', 'not_found'), it is used as-is.
- If `data.status` is present but not one of those three values, it is mapped to **not_found**.
- If any exception occurs during fetch/parse, status becomes **error**.

Results are cached for 12 hours to minimize identical validation queries:

- cache key: (`asn`, `prefix`)
- `maxsize=400000`
- `ttl=12 hours`

5.4 Counters per ASN

Each ASN maintains counters:

- `total`, `valid`, `invalid`, `not_found`, `error`

When all prefix validations for an ASN complete (tracked by a `pending[asn]` counter reaching 0), results are flushed to the DB queue.

Important concurrency note (code-accurate):

- Validation is spawned via `asyncio.create_task(validate_one(...))` per prefix; there is no strict cap on “in-flight validation tasks” other than practical limits (token bucket, connector connection limit, system resources).

6. Validation Calculations

For each ASN, let:

- `total = number of prefixes`
- `valid = count("valid")`
- `invalid = count("invalid")`
- `not_found = count("not_found")`
- `error = count("error")`

6.1 RPKI Coverage Percentage

`rpki_coverage_pct = 100 * valid / total` (rounded to 1 decimal)

If `total == 0` → `rpki_coverage_pct = None`.

6.2 Not-Found Ratio

`rpki_notfound_ratio = not_found / total` (rounded to 6 decimals)

If `total == 0` → `rpki_notfound_ratio = None`.

6.3 Printed Summary

The script prints per-ASN summaries through the async printer queue.

Code-accurate formatting:

- coverage display:
 - `"None"` if cov is None
 - otherwise `"{cov:.1f}%"` (includes percent sign)
- not_found ratio display:
 - `"None"` if nf_ratio is None

- otherwise formatted with **3 decimals** ("`{nf_ratio:.3f}`"), even though it is computed/stored rounded to 6 decimals

Example output format:

```
✓ AS12345: cov=83.3%, nf_ratio=0.042 (tot=120, v=100, inv=10, nf=5, err=5)
```

Additionally, when an ASN has no prefixes (`pfxs` empty), it prints:

```
✓ AS<asn>: cov=None, nf_ratio=None (tot=0, v=0, inv=0, nf=0, err=0)
```

The printed values correspond to the same internal counters used for DB writes, with the display-format differences noted above.

7. Concurrency Architecture

The script is a high-concurrency asynchronous pipeline, consisting of:

Producer

- Enumerates ASNs from the DB iterator and enqueues them into `q_asn`.
- After enqueueing all ASNs, pushes `None` sentinel values to stop ASN workers.

ASN Workers

- Dequeue ASNs from `q_asn`.
- Fetch announced prefixes from RIPEstat.
- Push `(asn, prefix_list)` into `q_pfx`.

Validation Workers

- Dequeue `(asn, prefix_list)` from `q_pfx`.

- For each prefix, spawn a task that:
 - performs RPKI validation
 - increments counters
 - decrements `pending[asn]`
 - flushes results when `pending[asn] == 0`
- When they receive sentinel `(None, None)`, they flush any remaining counters and exit.

DB Writer

- Dequeues computed per-ASN results from `q_db`.
- Performs UPSERT into the result table.
- Commits every row immediately.

Printer

- Dedicated async logger using a queue to prevent stdout from blocking worker execution.

Concurrency levels (defaults from script constants):

- ASN workers: default **512**
- Validation workers: default **4096**
- Connector global connection limit: default **2000** (set via `aihttp.TCPConnector(limit=...)`)

Code-accurate note:

- The argument is named `--max-conc-per-host`, but it is used as `TCPConnector(limit=...)`, which is a **global** limit, not a per-host limit.
-

8. Token Bucket (RPS Enforcement)

RPS limit is enforced globally with a token bucket:

```
bucket = TokenBucket(rate_per_sec=args.rps, capacity=args.rps * 2)
```

Defaults:

- `rps = 800`
- `capacity = 1600`

Every HTTP request acquisition path calls:

```
await bucket.wait()
```

This prevents overloading RIPEstat by rate-limiting all outbound requests across all workers.

9. TTL Caches

The script uses two TTL caches:

Prefix Cache

- `maxsize: 80,000`
- `ttl: 6 hours`
- `key: asn`
- `value: list of prefixes for that ASN`

Validation Cache

- `maxsize: 400,000`
- `ttl: 12 hours`

- key: (asn, prefix)
- value: validation status string

Caches significantly reduce load and improve performance by reusing results across rounds.

10. CLI Arguments

Supported arguments (exactly as in `parse_args()`):

- `--result-table`
- `--where`
- `--interval`
- `--req-timeout`
- `--max-retries`
- `--backoff-base`
- `--conc-asn-workers`
- `--conc-validation-workers`
- `--queue-max-asn`
- `--queue-max-pfx`
- `--queue-max-db`
- `--db-commit-every`
- `--rps`

- `--max-conc-per-host`
- `--batch-fetch`

The script runs continuously, executing “rounds” separated by a sleep interval.

Code-accurate note:

- `--db-commit-every` exists but does not change commit behavior in the current implementation.

11. Performance & Architecture Notes

- Fully asynchronous (`asyncio` + `aiohttp`)
- Uses `uvloop` when available
- Dedicated printer queue prevents stdout blocking
- WAL-mode SQLite with multiple performance PRAGMAs set in `ensure_db_ready()`
- Very high concurrency:
 - 512 ASN workers
 - 4096 validation workers
 - plus per-prefix spawned validation tasks via `asyncio.create_task`
- Requests use exponential backoff behavior on:
 - `429` (wait up to 60s scale factor)
 - `5xx` (wait up to 30s scale factor)
 - exceptions (wait up to 45s scale factor)

- LIMIT/OFFSET paging avoids loading all ASNs into memory at once
 - Designed for mass-scale RPKI hygiene measurement
-

12. Control Loop Summary

- Setup DB (create results table, set SQLite PRAGMAs)
- Open `aiohttp` session + initialize token bucket
- Start printer
- Start queues
- Producer → ASNs
- ASN workers → announced prefixes
- Validation workers → RPKI validation + counters → DB queue
- DB writer → upserts results (per-row commit)

On round completion:

- Print DB writer summary: total rows upserted (via printer queue)
- Print round timing to stdout: `[ROUND DONE] round_time=...`
- Sleep for `max(1, args.interval)`
- Start next round

The script runs indefinitely.

13. Metric Interpretation

RPKI coverage indicates:

- what percentage of an ASN's routed prefixes have cryptographically valid ROAs (**valid**)
- how many prefixes lack ROAs or are treated as **not_found** (including unknown status mapped to **not_found**)
- how many are **invalid** (origin mismatch / invalid validity state as reported by RIPEstat)
- how many validations fail with exceptions (**error**)

High RPKI coverage:

- stronger authentication posture
- lower vulnerability
- higher trust in routing correctness

Low RPKI coverage:

- insecure origin validation landscape (more prefixes without valid ROAs)
- increased vulnerability to hijacks
- potentially weaker filtering posture in practice

14. Limitations (Strictly Based on Script Behavior)

- Dependent entirely on RIPEstat's RPKI validation API
- Validation may be affected by transient RIPEstat outages
- Prefix retrieval failures are not represented as a separate stored error metric; failures can collapse into **total_prefixes = 0**
- **not_found** includes:

- genuine “no ROA” cases
 - any unexpected/unknown `data.status` mapped to `not_found`
 - `error` is only counted when an exception occurs during validation (not when `data.status` is unknown)
 - No cross-provider validation comparison (only RIPEstat)
 - Invalid prefixes are counted but not analyzed by cause
 - The CLI option `--db-commit-every` does not affect commit batching in the current code
-

15. Why RIPEstat Was Selected

RIPEstat provides a practical source for global RPKI validation at Internet scale:

- provides structured JSON APIs for announced-prefixes and RPKI validation
- returns a standardized status model (`valid`, `invalid`, `not_found`) which the script consumes directly
- works uniformly for IPv4 and IPv6 prefixes (as returned by announced-prefixes)
- integrates well with a high-concurrency async pipeline

This makes RIPEstat a suitable choice for real-time, multi-ASN RPKI coverage analysis in this implementation.