

prefix_vrp_count.py — VRP Count Collector (Full Technical Documentation)

1. Purpose & Role

`prefix_vrp_count.py` computes **vrp_count**, the number of distinct **Validated ROA Payloads (VRPs)** that cryptographically authorize a given prefix.

This metric is essential for the platform because it quantifies the **strength of ROA coverage**, not just its presence. It is used for:

- prefix-level vulnerability ML classification
- ROA completeness and consistency analysis
- combined ASN + prefix risk scoring
- structural RPKI security modeling
- global attack-surface estimation

While `has_roa` answers “Is this prefix authorized?”, `vrp_count` answers “How robustly is this prefix protected?”

2. High-Level Behavior

At a high level, the script:

- Loads (`ASN`, `prefix`) pairs from `prefix_data`.
- Ensures required columns for VRP metadata exist.
- Selects all rows missing `vrp_count` (or all rows when `--force` is used).

- Queries **RIPEstat** `/rpki-roas` for each prefix.
 - Retrieves all ROAs/VRPs returned by RIPEstat (using `include=more-specifics`).
 - **Filters VRPs to those whose ROA prefix covers the target prefix** (i.e. ROA prefix is less-specific or equal).
 - Deduplicates VRPs based on (`ASN`, `ROA prefix`, `maxLength`, `Trust Anchor`).
 - Calculates `vrp_count`.
 - Writes results back to SQLite with timestamps and error tracking.
 - Repeats the process continuously every ~60 minutes.
-

3. Metrics Produced

3.1 `vrp_count` (INTEGER)

Number of distinct VRPs **covering** the prefix.

- `0` → no VRP coverage (cryptographically vulnerable)
 - `1` → basic ROA protection
 - `>1` → strong, redundant RPKI coverage
-

3.2 `rpki_checked_at` (TEXT)

ISO8601 timestamp of the **last processing attempt** (successful or failed).

3.3 `rpki_error_last` (TEXT)

Most recent error message if RIPEstat processing failed.

3.4 `updated_at` (TEXT)

Metadata timestamp for auditing and change tracking.

4. Database Contract

4.1 Required input schema

The table must include:

- `asn` INTEGER
- `prefix` TEXT

The collector automatically adds:

- `vrp_count`
- `rpki_checked_at`
- `rpki_error_last`
- `updated_at`

4.2 Update behavior

Updates are performed **in-place**:

```
UPDATE prefix_data
SET vrp_count=?,
    rpki_checked_at=?,
    rpki_error_last=NULL,
    updated_at=?
WHERE asn=? AND prefix=?;
```

No new rows are inserted.

5. Data Flow Overview

5.1 Target selection

`load_targets()` retrieves only uncomputed rows (`vrp_count IS NULL`), unless `--force` is used.

5.2 RIPEstat query

For each prefix:

- **API endpoint:** `/data/rpki-roas/data.json`
- **Parameters:**
 - `resource=<prefix>`
 - `include=more-specifics`

Results are handled by an **AdaptiveRL** rate limiter with robust retry handling for:

- HTTP 429
 - 500 / 502 / 503 / 504
 - timeouts
 - transient client errors
-

5.3 VRP extraction & deduplication

`count_covering_vrps()`:

- Normalizes ROA entries returned by RIPEstat
- Parses ROA prefixes using `ip_network`
- **Selects only ROAs whose prefix covers the target prefix**
- Extracts:
 - ASN
 - ROA prefix
 - `maxLength`
 - Trust Anchor
- Deduplicates VRPs using a Python `set`

This ensures accurate and consistent VRP counting.

6. Performance & Concurrency

Concurrency model

- Up to **800 concurrent workers**
 - `asyncio.Semaphore` controls concurrency
 - `aiohttp.TCPConnector` with connection reuse and keepalives
-

Rate limiting

`AdaptiveRL` ensures:

- Stable request rate
 - Automatic slowdown after API backpressure
 - Gradual recovery when conditions improve
-

Database performance

- **Immediate per-row UPDATE + COMMIT**
 - Minimizes memory usage on the database side
 - SQLite WAL mode is supported externally
 - Designed for **24/7 long-running operation**
-

7. Control Loop Behavior

`one_round()`

Processes all selected prefixes once.

`run_forever()`

Repeats indefinitely with a **3600-second sleep** between rounds.

Advantages:

- Continuous enrichment
 - Self-maintaining state
 - Predictable refresh interval
-

8. CLI Arguments

Argument	Description	Default
<code>--db</code>	Path to SQLite database	asn_data.db
<code>--force</code>	Recompute all rows	False
<code>--concurrency</code>	Max concurrent workers	800
<code>--rps</code>	Target queries per second	16.0
<code>--timeout</code>	HTTP timeout	25 sec
<code>--db-batch</code>	Legacy parameter (unused)	3000

9. Reliability Justification — Why This Data Source Was Selected

RIPEstat `/rpki-roas` is the correct and reliable choice because:

- Aggregates validated VRPs from all 5 RIRs**
RIPE NCC, ARIN, APNIC, LACNIC, AFRINIC — ensuring global completeness.
- Reflects live validator-grade RPKI state**
ROA changes propagate rapidly into the backend validator, providing current data.
- Widely used by operators, IXPs, vendors, and researchers**
Used for:
 - RPKI monitoring
 - BGP correlation
 - large-scale research
 - incident response

4. **Eliminates the need to run a local RPKI validator**, which would require:

- TAL management
- repository synchronization
- RTR sessions
- state consistency handling
- filesystem caching

5. **Stable, high-uptime JSON API**, ideal for:

- High concurrency
- Predictable schema
- Consistent semantics

6. **Independent from BGP visibility**

Even prefixes not currently visible in BGP produce correct VRP data.

10. Usage Examples

Process missing VRP counts:

```
python3 prefix_vrp_count.py --db database/asn_data.db
```

Full recomputation:

```
python3 prefix_vrp_count.py --force
```

Continuous operation:

```
python3 prefix_vrp_count.py --concurrency 800 --rps 16
```

11. Integration with the Platform

`vrp_count` feeds into:

- Prefix Vulnerability ML features
- ROA scope and security completeness scoring
- Attack-surface calculation
- Combined ASN + Prefix ML risk scoring
- Global propagation vulnerability estimation

It must be computed **before any final risk-surface inference**.

12. Limitations

- Depends on RIPEstat API availability
- Does not perform local ROA cryptographic validation
- Does not detect expired or soon-to-expire ROAs
- Ignores malformed prefix entries
- Does not perform origin-AS validity checking (handled by separate components)