

# prefix\_roa\_maxlen.py — ROA maxLength Collector (Full Technical Documentation)

---

## 1. Purpose & Role

`prefix_roa_maxlen.py` computes and maintains the **roa\_maxLength** field for each (`asn`, `prefix`) row in the `prefix_data` table.

In RPKI, each ROA can specify a **maxLength**, which defines how specific a prefix announcement is allowed to be (e.g. a ROA for `203.0.113.0/23` with `maxLength=24` authorizes both `/23` and `/24` announcements).

This script answers the question:

“Given this origin ASN and prefix, what is the effective **ROA maxLength** that protects it, if any?”

The resulting `roa_maxLength` metric is used for:

- modeling **subprefix hijack risk**
- detecting overly permissive ROAs (e.g. `maxLength` much longer than the announced prefix)
- feeding the **prefix vulnerability ML model**
- combining with `prefix_length` and `vrp_count` to estimate how precise or dangerous ROA configuration is

If no covering ROA is found, `roa_maxLength = 0`.

---

## 2. High-Level Behavior

At a high level, the script:

1. Connects to a SQLite database (`database/asn_data.db`).
2. Ensures that `prefix_data` has all required RPKI-related columns.
3. Selects `(asn, prefix)` pairs that do not yet have `roa_maxLength` set (or all rows with `--force`).
4. For each `(asn, prefix)`:
  - Queries **RIPEstat** `/data/rpki-roas` for that prefix (with `include=more-specifics`).
  - Filters ROAs that cover the prefix.
  - Picks the most specific covering ROA set and computes the effective `maxLength`, preferring ROAs that match the origin ASN.
5. Updates the database with:
  - `roa_maxLength`
  - `rpki_checked_at`, `updated_at`
  - `rpki_error_last` on failure
6. Repeats the process in an infinite loop, sleeping ~60 minutes between full rounds.

The logic is fully asynchronous and optimized for large-scale batches.

---

## 3. Metrics Produced

### 3.1 `roa_maxLength` (INTEGER)

The **effective ROA maxLength** associated with `(asn, prefix)`:

- `0` → no covering ROA was found for this prefix
- `>0` → the maximum `maxLength` value from covering ROAs, based on:

- the **most specific** covering ROA prefix length
- a **preference for ROAs whose ASN equals the origin ASN**
- fallback to any covering ROA if no ASN match exists

Formally, `pick_roa_maxlen_for_prefix()`:

- finds all ROAs where:
  - ROA prefix `net` covers `prefix` ( $\text{prefix} \subseteq \text{net}$ )
- identifies the **largest `net.prefixlen`** among those (best match)
- among ROAs with that best prefix length:
  - first considers only those where ROA ASN == `origin_asn`
  - if at least one exists → returns the maximum `maxLength` among them
  - otherwise → returns the maximum `maxLength` among *all* candidates with that best prefix length

### 3.2 `rpki_checked_at` (TEXT)

- ISO8601 timestamp of the last attempt to compute `roa_maxLength` for the row.

### 3.3 `rpki_error_last` (TEXT)

- Contains the last error message when RIPEstat querying or parsing fails.
- Set to `NULL` on successful computation.

### 3.4 `updated_at` (TEXT)

- ISO8601 timestamp of the last update performed by this script on the row.
-

## 4. Database Contract

### 4.1 Required input schema

The script operates on the `prefix_data` table and expects at least:

- `asn` INTEGER
- `prefix` TEXT

It automatically ensures the following columns exist (creating them if missing):

- `roa_maxLength` INTEGER
- `rpki_checked_at` TEXT
- `rpki_error_last` TEXT
- `updated_at` TEXT

This is handled by `ensure_columns()` using `PRAGMA table_info()` and `ALTER TABLE`.

### 4.2 Update behavior

Updates are done in-place:

On **success**, `bulk_update_success()` uses:

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

On **failure**, `bulk_update_error()` uses:

```
UPDATE prefix_data
SET rpki_error_last = ?,
```

```
rpki_checked_at = ?,
updated_at = ?
WHERE asn = ? AND prefix = ?;
```

No new rows are inserted; the script only enriches existing records.

---

## 5. Data Flow Overview

### 5.1 Target selection

`load_targets(db, force):`

- Base query:  
`SELECT asn, prefix FROM prefix_data`
- If `--force` is **not** given, it adds:  
`WHERE roa_maxLength IS NULL`

Result: a list of `(asn, prefix)` pairs that require `roa_maxLength` computation.

### 5.2 RIPEstat query

`query_roa_maxlength()` calls:

- URL: `https://stat.ripe.net/data/rpki-roas/data.json`
- Parameters:
  - `resource=<prefix>`
  - `include=more-specifics`

It uses:

- `HTTP.get_json()` with:

- automatic retries on 429 and 5xx
- exponential backoff controlled by `AdaptiveRL`
- handling for 400 and other error codes
- `extract_roas_from_response()` to normalize the response into a list of ROA-like dictionaries from:
  - `data.roas`, or
  - `data.prefixes / data.results / data.items` (fallback)

### 5.3 Computing `roa_maxLength`

`pick_roa_maxlen_for_prefix(prefix, origin_asn, roas):`

1. Parses the target prefix using `ip_network(prefix, strict=False)`.
2. Iterates over ROAs:
  - extracts ROA prefix `pfx` (`prefix/route/roa_prefix`)
  - parses `net = ip_network(pfx, strict=False)`
  - checks `tgt.subnet_of(net)` → only covering ROAs are kept
  - maintains `best_plen = maximum net.prefixlen` seen so far
  - extracts ASN from (`asn/origin_asn/roa_asn`), stripping leading "AS"
  - extracts `maxLength` from (`maxLength/maxlength/max_length`), defaulting to `net.prefixlen`
  - appends (`asn, net.prefixlen, maxlen`) to `normalized`
3. If no covering ROAs are found (`best_plen < 0`) → returns `0`.

4. Filters `normalized` to only those with prefix length == `best_plen` (most specific coverage).
5. Splits into:
  - `with_asn` → candidates where `asn == origin_asn`
  - if `with_asn` is non-empty: returns `max(maxlen)` among them
  - else: returns `max(maxlen)` among all candidates for that best prefix length

This algorithm approximates the *effective* ROA `maxLength` for the given (`origin_asn`, `prefix`), favoring origin-consistent ROAs.

## Extended Algorithm Explanation

### 1. Why the script chooses the most specific covering ROA (Longest Prefix Match)

ROAs can exist at different prefix lengths (e.g., `/16`, `/17`, `/24`).

If several ROAs cover the target prefix, selecting the **one with the longest prefix length**:

- reflects the operator's most precise authorization intent,
- matches validator semantics (more-specific ROAs dominate broader ones),
- prevents broad ROAs (e.g., RIR block ROAs) from overriding customer-specific ones.

This ensures the extracted `maxLength` is not distorted by high-level allocation ROAs.

---

### 2. Why the script *prefers* ROAs belonging to the same ASN

If ROAs with identical prefix length exist for multiple ASNs, the script keeps only those where:

```
ROA_ASN == origin_asn
```

This prevents misleading results caused by:

- parent-block ROAs belonging to another ASN,
- multi-organization allocations,
- ROAs of historical holders.

This preference ensures that the `maxLength` value truly reflects what **the origin ASN** is authorized to announce — not what some upstream holder is allowed to originate.

---

## 6. Performance & Concurrency

### 6.1 Concurrency model

- Uses `asyncio` for fully asynchronous I/O.
- Creates up to `args.concurrency` concurrent worker tasks (default: 800).
- Guards concurrency with `asyncio.Semaphore(args.concurrency)`.

Each worker:

- acquires a semaphore slot
- calls `query_roa_maxlength()`
- writes results to the DB via `bulk_update_success()` / `bulk_update_error()`
- commits immediately
- logs a human-readable line indicating success or warning

### 6.2 Rate limiting

The script uses `AdaptiveRL` to control **request rate** to RIPEstat:

- `interval = 1 / rps` (with safety lower bound)



- `wait()` enforces a pacing interval, plus dynamic `penalty`
- `ok()` decays penalty when responses are healthy
- `back()` increases penalty when:
  - 429 (too many requests)
  - 5xx server errors
  - timeouts / client errors

Combined with `aiohttp.ClientSession` and a tuned `TCPConnector`, the script supports high-throughput, polite RIPEstat access.

---

## 7. Control Loop Behavior

### 7.1 Single round — `main_async(args)`

- Connects to the DB and ensures schema.
- Loads targets via `load_targets()`.
- If there is nothing to do, prints an informational message and exits.
- Sets up:
  - `aiohttp.ClientSession` with timeout and headers
  - `AdaptiveRL` instance for rate control
  - a semaphore for concurrency
- For each `(asn, prefix)`, spawns a `worker(asn, prefix)` task.
- Installs a SIGINT handler (where supported) for graceful stopping.
- Waits for all tasks to complete.

- Closes DB and prints final stats.

## 7.2 Continuous loop — `run_forever(args)`

- Logs `[loop]` `Starting a round...`
- Calls `main_async(args)` inside a `try/except` block
- Logs any round-wide failure (`[loop-warn]`)
- Sleeps `DEF_SLEEP_BETWEEN_ROUNDS` seconds (3600s)
- Repeats forever

This makes the collector suitable as a long-running service.

---

## 8. CLI Arguments

The script uses `argparse` with the following options:

| Argument                   | Description  | Default   |
|----------------------------|--|---|
| <code>--db</code>          | SQLite DB path   | resolved to <code>database/asn_data.db</code> if relative |
| <code>--force</code>       | Process all rows, ignoring existing <code>roa_maxLength</code> | <code>False</code>  |
| <code>--concurrency</code> | Max concurrency for async workers                              | <code>800</code>  |
| <code>--rps</code>         | Target requests per second to RIPEstat                         | <code>16.0</code>   |
| <code>--timeout</code>     | HTTP timeout in seconds  | <code>25</code>   |
| <code>--db-batch</code>    | Legacy “rows per commit” parameter (currently unused)          | <code>1</code>  |

Behavioral note:

- If `--db` is not absolute, `main()` overwrites it with the standard `DB_FILE` under `database/`.
- 

## 9. Reliability Justification — Why This Data Source Was Selected

The correctness of `roa_maxLength` depends entirely on the correctness of the ROA data source. This script uses **RIPEstat** `/data/rpki-roas`, which is an appropriate choice for enterprise-grade RPKI analytics for several reasons:

### 9.1 Aggregates VRPs from all five RIRs

RIPEstat's `rpki-roas` endpoint aggregates validated RPKI payloads from:

- RIPE NCC
- ARIN
- APNIC
- LACNIC
- AFRINIC

This gives a **global** view of ROAs, which is essential for an Internet-wide security platform.

### 9.2 Validator-grade semantics

RIPEstat's RPKI backend:

- follows standard ROA semantics (prefix, maxLength, ASN)
- maintains data in a way consistent with RFC 6482-style structures
- allows the script to implement origin-aware, prefix-specific logic without re-implementing a full validator

## 9.3 Operational maturity and industry usage

RIPEstat is widely used by:

- network operators
- IXPs
- security vendors
- researchers

for RPKI/BGP correlation. Building on this ecosystem increases:

- trust in the data
- interoperability with existing tools
- credibility of the resulting metrics

## 9.4 Offloading complexity

Running your own global RPKI validator infrastructure would require:

- handling multiple Trust Anchor Locators (TALs)
- syncing ROA repositories
- managing RTR sessions
- ensuring local cache consistency and failure handling

By consuming RIPEstat's `rpki-roas` API instead, the script can:

- focus on **analytics and modeling** (how `maxLength` interacts with actual announcements)
- avoid heavy operational overhead
- keep the design simple, maintainable, and reproducible

## 9.5 Stable, JSON-based, high-uptime API

RIPEstat provides:

- a consistent JSON schema
- predictable behavior for `maxLength`, ASN, and prefix fields
- sufficient performance for continuous, high-volume queries

This aligns directly with the script's high-concurrency, async architecture.

---

## 10. Usage Examples

**Compute `roa_maxLength` for all missing rows:**

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

**Force recomputation for every `(asn, prefix)` row:**

```
python3 prefix_roa_maxlen.py --db database/asn_data.db --force
```

**Tune concurrency and rate limits:**

```
python3 prefix_roa_maxlen.py --concurrency 400 --rps 8 --timeout 20
```

**Run continuously as a background service:**

```
python3 prefix_roa_maxlen.py
```

The script will:

1. Compute `roa_maxLength` for all missing rows.
2. Sleep 60 minutes.
3. Re-run to process any new prefixes that appeared in `prefix_data`.

---

## 11. Integration with the Platform

`roa_maxLength` is a key feature for:

- **Prefix Vulnerability ML model**, where it combines with:
  - `prefix_length`
  - `has_roa`
  - `vrp_count`
  - `roa_coverage_scope`
- Detecting dangerous ROA configurations, such as:
  - very permissive `maxLength` (e.g. ROA for `/16` with `maxLength=24`)
  - configurations that allow fine-grained subprefix hijacks
- Building higher-level metrics:
  - subprefix hijack exposure
  - misconfiguration risk
  - mapping between RPKI intent and observed announcements

It is also used in reporting and visualization to show:

- how strict or lax ROA policies are
- where careless ROA `maxLength` values create systemic risk

---

## 12. Limitations

Even though the script is robust, its limitations are made explicit:

- It depends on **RIPEstat API** availability and correctness.
- It does **not** perform local cryptographic validation of ROAs.
- It trusts RIPEstat to provide:
  - correct maxLength values
  - correct associations between ROAs, prefixes, and ASNs
- It does not itself decide whether announcements are **valid/invalid** under RPKI — it only computes a structural parameter used elsewhere in the pipeline.