

# `prefix_has_roa.py` — Prefix ROA Presence Collector (Full Technical Documentation)

## 1. Purpose and Role in the Platform

`prefix_has_roa.py` determines whether each prefix in the database is covered by at least one valid RPKI ROA.

It answers the fundamental question:

**“Is this prefix cryptographically authorized for routing, independently of any origin ASN?”**

The core output, `has_roa`, is a critical feature for:

- prefix-level vulnerability ML classification,
  - combined ASN+prefix risk scoring,
  - global risk-surface modeling.
- 

## 2. High-Level Behavior

At a high level, the script:

1. Loads prefix data from a SQLite table (`prefix_data` by default).
2. Ensures the table contains required RPKI-related columns.
3. Streams prefixes in large batches (up to 50k at a time).
4. Normalizes each prefix using `ip_network`.

5. Queries **RIPEstat** **rpki-roas** for each prefix.
  6. Evaluates whether the prefix is covered by at least one ROA.
  7. Writes results back to the DB with timestamps and error metadata.
  8. Can run continuously (interval-based) for 24/7 updating.
- 

## 3. Metrics Produced

### 3.1 **has\_roa** (INTEGER)

- **1** → At least one ROA authorizes the prefix (subnet + maxLength check).
- **0** → No ROA authorizes the prefix.
- **NULL** → Unable to determine due to API errors.

### 3.2 **rpki\_status\_text**

Human-readable derived state:

- **"has\_roa"**
- **"no\_roa"**
- **"unknown"**

### 3.3 **rpki\_checked\_at**

UNIX timestamp for the last update.

### 3.4 **rpki\_error\_last**

Error message from the latest failed request (if any).

---

## 4. Database Contract

### 4.1 Requirements

Input table must contain:

`prefix TEXT`

Missing RPKI columns are automatically added:

- `has_roa`
- `rpki_status_text`
- `rpki_checked_at`
- `rpki_error_last`

### 4.2 Update Logic

Rows are updated, not inserted:

```
UPDATE prefix_data
  SET has_roa=?, rpki_status_text=?, rpki_error_last=?,
      rpki_checked_at=?
 WHERE prefix=;
```

---

## 5. Data Flow Overview

### 5.1 Prefix Streaming

Prefixes are read in batches via:

`stream_prefixes(...)`

Each prefix is normalized to canonical CIDR format before querying.

## 5.2 Fetching ROAs from RIPEstat

`ripestat_roas_for_prefix()` calls:

`https://stat.ripe.net/data/rpki-roas/data.json?resource=<prefix>`

Built-in advantages:

- retry/backoff on 429, 5xx,
- temporary caching (6h TTL),
- high-throughput rate limiting (token bucket).

## 5.3 Deciding `has_roa`

The `_authorizes()` function checks:

1. Is the target prefix a subnet of the ROA prefix?
2. Is the prefix length  $\leq$  ROA maxLength?

If **any** ROA qualifies  $\rightarrow$  `has_roa = 1`.

---

# 6. Concurrency & Performance Architecture

## TokenBucket

- Limits RIPEstat queries to `--rps` per second (default 400).
- Allows small bursts (capacity =  $2 \times$  rate).

## aiohttp

High-performance async HTTP with:

- 800+ concurrent connections (configurable),

- connection reuse,
- DNS caching.

## Worker Model

- Producer feeds prefixes into `q_in`.
- Workers process prefixes concurrently.
- Writer flushes results to the DB.

## Printer

Non-blocking logging in a dedicated async task.

---

## 7. Control Loop Behavior

### `one_round(args)`

Processes all prefixes once.

### `main_loop(args)`

Repeats the process every `--interval` seconds.

This allows:

- near real-time updates,
  - fully autonomous operation,
  - continuous enrichment of prefix data.
- 

## 8. CLI Arguments

Argument	Purpose	Default
<code>--db</code>	SQLite file path	<code>asn_data.db</code>
<code>--table</code>	Prefix table	<code>prefix_data</code>
<code>--where</code>	SQL filter	None
<code>--interval</code>	Seconds between rounds	300
<code>--rps</code>	Requests per second	400
<code>--workers</code>	# of async workers	8
<code>--req-timeout</code>	HTTP timeout	12
<code>--max-retries</code>	API retries	5
<code>--max-conc-per-host</code>	Parallel connections	800

---

## 9. Why This Data Source Is Reliable

### Why RIPEstat `rpki-roas` is a highly reliable and appropriate data source

The `rpki-roas` dataset from RIPEstat is one of the most authoritative and widely trusted sources for ROA information because:

#### 1. It aggregates validated RPKI data from all five RIRs

- RIPE NCC
- ARIN
- APNIC
- AFRINIC
- LACNIC

This means the API gives a **global, unified view** of ROAs, not just regional data.

## **2. RIPE's RPKI backend is continuously updated**

ROAs published in the RIR repositories propagate quickly into RIPE's validator infrastructure. For real-time prefix security, freshness is mandatory — and RIPEstat is industry standard.

## **3. It is widely used by operators, researchers, and routing-security tools**

Vendors, IXPs, academic research groups, and routing-security systems use RIPEstat as a canonical source for:

- RPKI validation state,
- BGP/RPKI correlation,
- large-scale analytics.

Using the same source as the global community increases interoperability and credibility.

## **4. It abstracts away validator implementation differences**

Running your own RPKI validator requires:

- sync with all five RIR TALs,
- trust-anchor monitoring,
- RTR sessions,
- periodic sync,
- local cache consistency,
- failure handling.

RIPEstat handles all of this automatically, so the platform can focus on *analytics*, not low-level RPKI infrastructure.

## **5. Provides a stable, well-documented, high-uptime API**

- JSON format,

- versioned endpoints,
- predictable behavior,
- huge capacity (thousands of requests/sec).

Perfect for collector design, which pushes ~400 RPS sustained.

## 6. Consistent ROA semantics across vendors

RIPEstat reliably exposes:

- prefix,
- maxLength,
- ASN,
- validity window,
- multiple ROA variants.

Exactly the structure needed for `_authorizes()` logic.

## 7. Zero dependency on BGP visibility

Unlike RIS, RouteViews, or BMP collectors, ROA information is **independent of routing visibility**.

That means:

- even prefixes not currently visible in BGP will yield correct ROA information.

---

## Conclusion

RIPEstat is the optimal choice for a real-time prefix-ROA collector because it provides:

**global coverage, high reliability, validator-grade accuracy, and standards-aligned semantics.**

This greatly strengthens the correctness of `has_roa`, one of the core ML features.



---

## 10. Usage Examples

**Run once:**

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

**Only update missing ROA states:**

```
python3 prefix_has_roa.py --where "has_roa IS NULL"
```

**Continuous mode (daemon-like):**

```
python3 prefix_has_roa.py --interval 600
```

---

## 11. Interaction With Other Components

`has_roa` feeds into:

- **prefix vulnerability ML**
- **combined ASN-prefix scoring**
- **exposure-chain modeling**

And must be computed **before** model inference or risk recomputation.

---

## 12. Limitations

- Depends on RIPEstat API availability.
- Does not validate ROAs locally.
- Evaluates only prefix-level authorization (not origin-AS-specific validity).
- Ignores malformed prefixes.