

prefix_irr_exact.py — Exact IRR Route Object Counter (Full Technical Documentation)

1. Purpose & Role in the Platform

prefix_irr_exact.py measures how well a prefix is represented in Internet Routing Registry (IRR) databases.

For each (asn, prefix) row in prefix_data, it:

- queries multiple IRR servers in parallel (NTT, RADB, RIPE),
- finds **exact** route / route6 objects that match the prefix,
- counts the number of unique (kind, origin, source) triples, and
- stores the count and diagnostic metadata in the database.

It answers:

“How many exact IRR route/route6 objects exist for this prefix, and from which sources?”

This feeds the **Policy / Routing Hygiene** dimension in the prefix vulnerability model. Poor or missing IRR representation can signal:

- weak filtering hygiene,
- abandoned or mismanaged resources,
- prefixes that are easier to abuse or misconfigure.

2. High-Level Behavior

At a high level, the script:

- Connects to `database/asn_data.db` and targets the `prefix_data` table.
- Ensures all IRR-related columns exist (schema migration on the fly).
- Selects `(asn, prefix)` targets where:
 - `irr_objects_exact IS NULL OR`
 - `irr_query_state IS NULL`,
unless `--force` is used (then **all rows** are processed).

For each target prefix, in parallel:

- queries three IRR servers over WHOIS (TCP/43):
 - `rr.ntt.net` (NTT)
 - `whois.radb.net` (RADB)
 - `whois.ripe.net` (RIPE)
- parses `route / route6` objects,
- extracts **exact matches** (same prefix, same length),
- deduplicates them by `(kind, origin, source)`.

Computes:

- `irr_objects_exact` (count),
- `irr_sources_used` (RPSL `source:` values),
- `irr_query_state` (`VALID_*` / `PARTIAL_FAILED`),
- `irr_servers_ok / irr_servers_fail` (diagnostics).

Updates the DB in-place and logs results.

Runs in an infinite loop, repeating roughly every **60 minutes**.

The script is fully asynchronous and designed for high parallelism with per-server rate limiting.

3. Metrics Produced

For each `(asn, prefix)` in `prefix_data`, the script maintains:

3.1 `irr_objects_exact` (INTEGER)

Number of distinct **exact** IRR route objects found for this prefix, across all queried IRR servers.

Internally, an “exact object” is defined as a unique triple:

`(kind, origin, source)`

where:

- `kind` \in `{"route", "route6"}`
- `origin` is the `origin:` attribute value (uppercased, may be empty string if missing)
- `source` is the `source:` attribute value (uppercased, e.g. RADB, RIPE, NTTCOM)

Only objects whose `route/route6` prefix **exactly equals** the target prefix (same address and prefix length) are counted.

3.2 `irr_sources_used` (TEXT)

A comma-separated list of distinct IRR RPSL sources (from the `source:` attribute) among exact matches.

Example:

`"RADB, RIPE, NTTCOM"`

If no exact objects are found, this may be an empty string.

3.3 `irr_query_state` (TEXT)

High-level classification of the IRR query result for that prefix:

- "VALID_NONZERO"
 - At least one IRR server responded successfully (`servers_ok` non-empty), and
 - `irr_objects_exact > 0`.
- "VALID_ZERO"
 - All responding servers were OK,
 - No errors recorded,
 - But `irr_objects_exact == 0`.
- "PARTIAL_FAILED"
 - Some servers were OK, but at least one server failed,
 - `irr_objects_exact == 0`.
- "ALL_FAILED"
 - All contacted servers failed.
 - **Important (code-accurate behavior):** in this case the script prints a `[null]` log line and **does not write any success row to the database** for that prefix. Therefore, `ALL_FAILED` is effectively a **log-only** state; the row remains with NULL fields and will be retried in future rounds.

The state reflects both data presence and server health.

3.4 irr_servers_ok (TEXT)

Pipe-separated list of IRR server labels that successfully responded.

Example:

`"NTT|RADB|RIPE"`

3.5 irr_servers_fail (TEXT)

Pipe-separated list of IRR server labels that failed (timeouts, connection errors, parse errors, etc.).

Example:

"RADB"

3.6 irr_checked_at (TEXT, ISO8601)

Timestamp (UTC, ISO8601 format) when the last IRR check was performed for this prefix.

Example:

2025-12-08T19:48:32.123456+00:00

3.7 irr_error_last (TEXT)

On errors during processing:

- set to a truncated error string (in practice, the worker prefixes errors with "irr:"),
- cleared (NULL) on successful update.

3.8 updated_at (TEXT, ISO8601)

Timestamp of the last successful write by this script to that row.

4. Database Contract

4.1 Required table

The script targets the `prefix_data` table and expects at least:

- `asn` INTEGER
- `prefix` TEXT

It automatically ensures the presence of the following IRR-related columns:

- `irr_objects_exact` INTEGER
- `irr_checked_at` TEXT

- `irr_error_last` TEXT
- `irr_sources_used` TEXT
- `irr_query_state` TEXT
- `irr_servers_ok` TEXT
- `irr_servers_fail` TEXT
- `updated_at` TEXT

This is done by `ensure_columns()` via `PRAGMA table_info()` and `ALTER TABLE` statements.

4.2 Selection criteria

`load_targets()` selects rows via:

```
SELECT asn, prefix FROM prefix_data
```

with:

- default: `WHERE irr_objects_exact IS NULL OR irr_query_state IS NULL`
- with `--force`: no WHERE clause (recomputes everything)

4.3 Update behavior (success)

On successful IRR lookup, `bulk_update_success()` executes:

```
UPDATE prefix_data
SET irr_objects_exact = ?,
    irr_sources_used = ?,
    irr_query_state = ?,
    irr_servers_ok = ?,
    irr_servers_fail = ?,
    irr_checked_at = ?,
    updated_at = ?,
```

```
    irr_error_last = NULL
WHERE asn = ? AND prefix = ?;
```

Notes:

- `irr_servers_ok` and `irr_servers_fail` are derived from an internal diagnostics string formatted as:
 - "OK1|OK2|...;FAIL1|FAIL2|...".
- `irr_error_last` is explicitly cleared on success.

4.4 Update behavior (error)

On failures in processing (exceptions inside the worker), `bulk_update_error()` writes:

```
UPDATE prefix_data
SET irr_error_last = ?,
    irr_checked_at = ?,
    updated_at = ?
WHERE asn = ? AND prefix = ?;
```

This allows:

- distinguishing between IRR “no data” (`VALID_ZERO`)
 - and genuine technical errors (timeouts, parsing errors, etc.)
-

5. Data Flow Overview

5.1 Prefix loading

`run_once()`:

- connects to DB,

- ensures schema,
- loads (`asn`, `prefix`) targets via `load_targets()`.

5.2 IRR querying (WHOIS over TCP/43)

For each prefix, `count_exact_irr_objects_parallel()`:

- schedules a task per IRR server in `IRR_SERVERS`:
 - (`"rr.ntt.net"`, `43`, `"NTT"`)
 - (`"whois.radb.net"`, `43`, `"RADB"`)
 - (`"whois.ripe.net"`, `43`, `"RIPE"`)

Each task calls `query_one_server()`:

- uses `whois_query()` → `asyncio.open_connection(host, port)`
- sends query: `-T route,route6 -r <prefix>`
- reads the full response with timeouts (`DEF_TIMEOUT_CONNECT`, `DEF_TIMEOUT_READ`)
- `whois_query()` returns raw WHOIS text.

5.3 Parsing IRR objects

The text is parsed by:

`parse_objects(text)`:

- splits by blank lines into blocks (objects),
- ignores comment lines starting with `%` or `#`,
- parses lines with a regex `attr: value`,

- keeps only objects whose **first attribute** is `route` or `route6`,

builds a dict:

```
{
  "route": [...],
  "origin": [...],
  "source": [...],
  ...
}
```

-

`extract_exact_keys(objs, target_prefix):`

- canonicalizes `target_prefix` with `ip_network(..., strict=True)`,
- for each object:
 - canonicalizes each `route/route6` string with `ip_network(..., strict=True)`,
 - keeps only those exactly equal to the target prefix,
 - normalizes:
 - `kind = "route" / "route6"`
 - `origin = each origin value uppercased (or "" if missing)`
 - `rpsl_source = first source value uppercased`
 - adds `(kind, origin, rpsl_source)` to a set.

Result: a set of unique exact route objects per server.

5.4 Aggregation across servers

`count_exact_irr_objects_parallel():`

- builds tasks for each IRR server,
- runs them either:

fast mode (`--fast-positive`):

- uses `asyncio.wait(..., FIRST_COMPLETED)` loop,
- as soon as one server returns any exact object (`exact` non-empty), cancels remaining tasks and returns early,
- faster but diagnostics (`servers_ok/servers_fail`) may be incomplete.

full mode (default):

- waits for all tasks via `asyncio.gather`,
- aggregates results from all servers.

It returns:

- `cnt` = total number of unique (`kind`, `origin`, `source`) triples across all servers
- `srcs` = comma-separated list of distinct RPSL source values
- `servers_ok` = list of labels that successfully answered
- `servers_fail` = list of labels that failed

6. Performance & Concurrency

6.1 Concurrency model

- `DEF_CONCURRENCY = 1000`

- `sem = asyncio.Semaphore(args.concurrency)` limits number of prefixes processed in parallel.
- Each prefix spawns queries to 3 IRR servers concurrently.

6.2 Per-server rate limiting

To avoid overloading IRR servers, the script uses `PerServerRL`:

maintains per-server:

- last request time,
- lock,
- penalty (backoff).

`wait(label):`

- delays until `interval + penalty` has passed since last request for that server.

`ok(label):`

- decreases penalty after successful requests.

`back(label):`

- increases penalty after errors, up to a capped maximum.

`args.rps` is interpreted as **per-server** requests per second.

6.3 Resilience and retries

`query_one_server():`

- on exceptions, calls `rl.back(label)`,
- retries up to `--retry-per-server` times, with small increasing delays,

- if still failing, returns `(label, False, empty_set, error)`.

The design allows the script to:

- maximize throughput,
 - respect external services,
 - continue functioning even if one IRR server is temporarily unstable.
-

7. Control Loop Behavior

7.1 Single run — `run_once(args)`

Steps:

- Connects to DB, ensures schema via `ensure_columns()`.
- Loads targets via `load_targets(db, force=args.force)`.
- If none:
 - prints `[info] Nothing to process (use --force for a full recount)`.
 - closes DB and returns.
- Constructs `PerServerRL` with IRR labels, a concurrency semaphore, and creates a `worker(asn, prefix)` task for each target.
- Installs a SIGINT handler (where supported) to print a graceful-stop message.
- Awaits all worker tasks with `asyncio.wait(..., ALL_COMPLETED)`.
- Closes DB and prints final `processed=` count.

7.2 Continuous mode — `run_forever(args)`

- Logs `[loop] Start <timestamp>`
- Calls `run_once(args)` inside try/except, logging any round-level errors.
- Sleeps `DEF_SLEEP_BETWEEN_ROUNDS` seconds (3600) via `asyncio.sleep`.
- Repeats indefinitely.

This makes the script suitable as a daemon-like service in the collectors layer.

8. CLI Arguments

Argument	Description	Default
<code>--db</code>	SQLite DB path. Code-accurate behavior: if the provided path is not absolute , it is ignored and the script forces <code>database/asn_data.db</code> .	<code>database/asn_data.db</code>
<code>--force</code>	Recompute for all rows, ignoring existing <code>irr_objects_exact / irr_query_state</code>	False
<code>--concurrency</code>	Max concurrent prefixes	1000
<code>--rps</code>	Per-server request rate limit	40.0
<code>--timeout-connect</code>	WHOIS connect timeout (sec)	3
<code>--timeout-read</code>	WHOIS read timeout (sec)	6
<code>--retry-per-server</code>	Retries per IRR server	1
<code>--db-batch</code>	Legacy batch size parameter (present but not used anywhere in logic)	6000
<code>--fast-positive</code>	Stop querying other servers once one server returns at least one exact object	False

9. Reliability Justification — Why These IRR Sources

This script relies on three major public IRR servers:

- rr.ntt.net (NTT)
- whois.radb.net (RADB)
- whois.ripe.net (RIPE)

These were chosen because:

- **Diversity of sources**
NTT, RADB, and RIPE IRR collectively cover a large portion of world-wide IRR route objects. Many networks mirror or register their objects in one or more of these databases.
- **Operational maturity**
They are long-standing IRR operators with stable WHOIS services, widely used by network operators for IRR-based prefix filters and routing policy.
- **Redundancy and cross-validation**
Querying multiple IRRs reduces the risk that:
 - an object exists only in a specific registry, or
 - one IRR has temporary outages.
- By aggregating information from all three and tracking [servers_ok](#) / [servers_fail](#), the platform can distinguish between:
 - “no IRR objects exist” vs.
 - “we could not reach some servers”.
- **Exact-object focus**
The script only counts exact [route/route6](#) objects matching the prefix, which aligns with how many filters are built (prefix- and origin-specific).
- **Transparency for consumers**
The script stores:

- which sources contributed (`irr_sources_used`),
 - which servers succeeded or failed (`irr_servers_ok` / `irr_servers_fail`).
 - This transparency allows downstream systems to incorporate trust decisions or to ignore certain sources if desired.
-

10. Usage Examples

Important (code-accurate): this script runs continuously (daemon-style). There is no built-in “one-shot and exit” mode.

Run (continuous, default DB):

```
python3 prefix_irr_exact.py
```

Force recomputation for all prefixes (continuous loop):

```
python3 prefix_irr_exact.py --force
```

More conservative mode (lower concurrency and RPS):

```
python3 prefix_irr_exact.py --concurrency 300 --rps 10
```

Faster but less detailed diagnostics (`--fast-positive`):

```
python3 prefix_irr_exact.py --fast-positive
```

This mode will stop querying other servers as soon as one server yields at least one exact object.

Note about `--db`: if you pass a relative path, the script will ignore it and still use `database/asn_data.db`. Use an absolute path if you truly want a different DB file.

11. Integration with the Platform

`irr_objects_exact` and its associated metadata integrate into the platform as:

- part of the prefix vulnerability feature set, especially the “Policy / Routing Hygiene” dimension;
- a signal for:
 - how carefully a prefix is maintained in IRR,
 - how likely it is to be covered by operator-built IRR filters,
 - potential exposure due to missing or inconsistent IRR entries.

Downstream, these metrics are used by:

- the prefix ML model (e.g., `prefix_vuln_ml`),
- reports explaining why a prefix is considered weak or well-protected,
- combined ASN+prefix risk scoring.

12. Limitations

The script explicitly acknowledges several limitations:

IRR is not authoritative

IRR data can be stale, duplicated, or inconsistent. Some operators do not maintain IRR objects at all. The metric reflects IRR registration hygiene, not ground truth ownership.

Depends on WHOIS availability

If all three IRR servers fail, the script logs a `[null]` state and **does not set** `irr_objects_exact` / `irr_query_state` for that row (no DB write occurs for that prefix). The prefix will be retried in a future round.

Transient network issues can delay metric completion.

Fast-positive mode trades completeness for speed

With `--fast-positive`, diagnostics about failed servers and IRR coverage may be incomplete. This is acceptable for some use cases but should be understood by operators.

IPv4/IPv6 parsing is strict

If the prefix string cannot be parsed with `ip_network(..., strict=True)`, it is skipped silently for object extraction and will yield 0 exact objects for that server's result.