

# asn\_prefix\_risk.py — ASN–Prefix Attack Surface Combinator (Technical Documentation)

This document is **strictly aligned with the provided code** and describes exactly how the script combines:

- **ASN attacker risk** (from `asn_ml_risk`)
- **Prefix vulnerability** (from `prefix_ml_vuln`)
- **ROA/ROV context** (from `prefix_data`)

into a ranked **attack-surface table**: `asn_prefix_attack_surface`.

---

## 1. Purpose

The script computes, for each `(prefix, origin_asn)` pair, the **top N attacker ASNs** (from a top-K attacker pool) most likely to produce a meaningful attack surface, and writes one row per triple:

**`(prefix, origin_asn, attacker_asn)`**

Each row includes:

- the ROA scenario classification (`scenario`)
- the chosen dominant attack vector (`attack_type`)
- the RPKI validation state receivers would see (`attacker_rpki_state`)
- feasibility adjustments
- a combined numeric score (`combined_score`)

- confidence fields derived from data completeness
- 

## 2. Inputs (SQLite)

The script requires these tables:

### 2.1 `asn_ml_risk`

Required columns:

- `asn`
- `risk_score`

Optional columns (if present, they are used; otherwise defaulted):

- `data_completeness`
- `score_confidence`

Loaded and sorted by:

- `ORDER BY risk_score DESC LIMIT topk`

### 2.2 `prefix_ml_vuln`

Required columns:

- `prefix`
- `origin_asn`
- `risk_score`

Optional columns:

- `data_completeness`
- `score_confidence`

## 2.3 `prefix_data`

Required columns:

- `prefix`
- `asn`
- `has_roa`
- `roa_maxLength`
- `prefix_length`

The script joins:

- `prefix_ml_vuln pmv`
- `prefix_data pd`  
on:
- `pd.prefix = pmv.prefix AND pd.asn = pmv.origin_asn`

So the combinator only processes prefixes that exist in **both** datasets for the same origin ASN.

---

## 3. CLI Arguments

- `--db` (required): SQLite DB path
- `--topk-asns` (default 500): top-K attacker ASNs selected from `asn_ml_risk`

- `--topn-per-prefix` (default 50): number of attacker ASNs stored per prefix
- `--delta-permissive` (default 2): ROA permissiveness threshold used in scenario classification
- `--permissive-feas` (default 0.7): feasibility factor applied when scenario is `PERMISSIVE_ROA`
- `--strict-nonowner-feas` (default 0.1): feasibility factor when scenario is `STRICT_ROA` and attacker `!=` origin
- `--batch` (default 5000): DB insert batch size
- `--limit-prefixes` (default 0): optional LIMIT on number of joined prefix rows
- `--prefix`: restrict processing to a single exact prefix
- `--prefix-file`: restrict processing to prefixes listed in a file (one per line, lines starting with `#` ignored)
- `--print-summary`: print summary stats at end

Prefix scoping rules (enforced):

- `--prefix` and `--prefix-file` are mutually exclusive.

---

## 4. Output Table

### 4.1 Table name

`asn_prefix_attack_surface`

### 4.2 Primary key

`PRIMARY KEY (prefix, origin_asn, attacker_asn)`

This ensures **idempotent updates** via UPSERT.

### 4.3 Columns (exactly as created)

Identity & scenario:

- `prefix` (TEXT)
- `origin_asn` (INTEGER)
- `attacker_asn` (INTEGER)
- `scenario` (TEXT)
- `attack_type` (TEXT)
- `attacker_rpki_state` (TEXT)
- `roa_gap` (INTEGER)
- `attacker_env_band` (TEXT)
- `success_likelihood_band` (TEXT)

Prefix ML:

- `prefix_vuln_score` (REAL)
- `prefix_data_completeness` (REAL)
- `prefix_score_confidence` (TEXT)

ASN ML:

- `asn_risk` (REAL)
- `asn_data_completeness` (REAL)
- `asn_score_confidence` (TEXT)

Pair confidence:

- `pair_data_completeness` (REAL)
- `pair_score_confidence` (TEXT)

Final scoring:

- `feasibility_factor` (REAL)
- `combined_score` (REAL)
- `updated_at` (TEXT)

Indexes created:

- `prefix`, `origin_asn`, `attacker_asn`, `combined_score`, `pair_data_completeness`, `attack_type`, `attacker_rpk_i_state`, `success_likelihood_band`
- 

## 5. End-to-End Flow (Step-by-step)

### Step 1 — Parse arguments

`parse_args()` loads all CLI options and validates that `--prefix` and `--prefix-file` are not both set.

### Step 2 — Build prefix scope (optional)

- If `--prefix` is set: restrict to that single prefix.
- If `--prefix-file` is set: read prefixes from file, ignore empty lines and lines starting with `#`, restrict to that set.
- Otherwise: process all joinable prefixes.

This scope is applied only in `load_prefix_rows()`.

### Step 3 — Open SQLite connection and set pragmas

- `PRAGMA journal_mode=WAL;`
- `PRAGMA synchronous=NORMAL;`

## Step 4 — Ensure output table exists

`ensure_output_table(conn)` runs `CREATE TABLE IF NOT EXISTS ...` and creates indexes.

## Step 5 — Load attacker ASN candidates (top-K)

`load_attacker_asns(conn, topk):`

- requires `asn_ml_risk` table
- selects `asn`, `risk_score` plus optional `data_completeness` and `score_confidence`
- orders by descending risk score
- parses into a list of tuples: `(asn, risk_score, data_completeness, score_confidence)`
- if optional columns are missing, defaults:
  - `data_completeness = 0.0`
  - `score_confidence = confidence_from_completeness(data_completeness)`

A dict is built:

```
attacker_map[asn] = (risk_score, data_completeness, score_confidence)
```

## Step 6 — Load joinable prefix rows

`load_prefix_rows(conn, limit_prefixes, only_prefixes):`

- requires `prefix_ml_vuln` and `prefix_data`

- requires columns in `prefix_data`: `prefix`, `asn`, `has_roa`, `roa_maxLength`, `prefix_length`
- selects from the JOIN:
  - `prefix`
  - `origin_asn`
  - prefix vulnerability score
  - optional prefix completeness/confidence
  - ROA flags and lengths

Defaults (when optional columns missing):

- `prefix_data_completeness = 0.0`
- `prefix_score_confidence = confidence_from_completeness(prefix_data_completeness)`

## Step 7 — For each (`prefix`, `origin_asn`), compute attack surface vs candidate attackers

For each loaded prefix row:

### 7.1 Scenario classification (`scenario`)

`classify_scenario(has_roa, roa_maxlen, prefix_len, delta)` returns one of:

- `NO_ROA` if `has_roa == 0`
- `STRICT_ROA` if `has_roa != 0` and `roa_maxlen <= prefix_len`
- `PERMISSIVE_ROA` if `has_roa != 0` and `roa_maxlen >= prefix_len + delta`



## 7.2 ROA gap (**roa\_gap**)

**roa\_gap**(has\_roa, roa\_maxlen, prefix\_len):

- returns -1 if **has\_roa** == 0
- else returns **roa\_maxlen** - **prefix\_len** (integer) if conversion succeeds
- else -1

## 7.3 Candidate attacker list

The script always evaluates:

- the **origin\_asn** itself (first)
- plus the top-K attacker ASNs (excluding origin if duplicated)

So candidates size is **1 + (topk - maybe 1)**.

## 7.4 For each **attacker\_asn**, compute derived fields and scores (detailed logic)

For every (**prefix**, **origin\_asn**) row loaded from **prefix\_ml\_vuln JOIN prefix\_data**, the script constructs a candidate list of attacker ASNs (**origin\_asn** + the top-K attacker ASNs from **asn\_ml\_risk**). Then, **for each **attacker\_asn** candidate**, it computes a set of derived fields and scores that together describe:

- how risky the attacker ASN is (from ASN ML),
- how vulnerable the prefix is (from Prefix ML),
- how feasible the attack is under the prefix's ROA state (via a scenario-specific scaling),
- what a validating receiver would label the announcement (RPKI state),
- what attack vector is considered dominant (**attack\_type**),
- and a final explainable “success likelihood band” built from these pieces.

Below is the exact flow and the **reasoning behind each field**.

---

a) **ASN risk & ASN confidence (asn\_risk, asn\_dc, asn\_sc)**

Computed by:

```
asn_risk, asn_dc, asn_sc = attacker_map.get(attacker_asn, (0.0, 0.0, "LOW"))
```

What it means (logic):

- **asn\_risk** is the attacker ASN's ML risk score (0..1). In platform logic, this score is intended to represent "how likely / capable this ASN is to originate malicious announcements that propagate," based on the ASN-side features and training you built.
- **asn\_dc** and **asn\_sc** are the **quality signals** for that ASN score:
  - **asn\_dc** (**data\_completeness**) measures how much of the required underlying feature set was actually present for that ASN.
  - **asn\_sc** (**score\_confidence**) is either taken directly from DB (if present) or derived from **asn\_dc** with the same conservative mapping used elsewhere.

Why default **(0.0, 0.0, "LOW")** exists:

- It is a safe fallback: if an ASN is not found in **attacker\_map**, the script assumes **no attacker evidence** rather than guessing. This prevents accidental inflation of risk due to missing rows.

---

b) **Feasibility factor (feasibility\_factor)**

Computed by:

```
feas = feasibility(scenario, attacker_asn, origin_asn, permissive_feas, strict_nonowner_feas)
```

### What it means (logic):

This is the script's **explicit bridge between** “prefix security state” and “attack practicality.”

Even if:

- the prefix is highly vulnerable (`prefix_risk` high), and
- the attacker ASN is high-risk (`asn_risk` high),

the attack may still be **structurally hard** (e.g., strict ROA with a different origin).

So the script uses `feasibility_factor` as a **scenario-dependent scaling** that models “how much of the theoretical combined risk can actually express itself.”

### Rules and rationale (exactly as code):

- **Scenario: `NO_ROA` → `feasibility` = 1.0**
  - **Why:** if there is no ROA, the script assumes there is no cryptographic authorization constraint in the model, so feasibility is not reduced.
- **Scenario: `PERMISSIVE_ROA` → `feasibility` = `permissive_feas`**
  - **Why:** the script treats permissive ROA as “some friction exists, but not a hard wall,” so it reduces feasibility to a tunable constant (`--permissive-feas`, default 0.7).
  - Important: in this combinator's simplified model, ROA permissiveness affects *feasibility* rather than making the attack “valid.” Validity is handled separately by `attacker_rpki_state`.
- **Scenario: `STRICT_ROA`**
  - If `attacker_asn == origin_asn` → 1.0
    - **Why:** if the origin ASN itself is the attacker (abuse/compromise scenario), strict ROA does not stop it.
  - Else → `strict_nonowner_feas` (default 0.1)
    - **Why:** strict ROA + non-owner origin is modeled as **strongly constraining**, so feasibility drops to a very low constant.

So `feasibility_factor` is not “probability of success.” It is a **structural dampening coefficient**: a prefix can be risky, but strict authorization makes real-world exploitation less feasible unless the attacker is the legitimate origin.

---

c) Pair completeness & pair confidence (`pair_data_completeness`, `pair_score_confidence`)

Computed by:

```
pair_dc = min(prefix_dc, asn_dc)
```

```
pair_sc = confidence_from_completeness(pair_dc)
```

**What it means (logic):**

The script uses a **weakest-link principle** for the combined attacker–prefix pair:

- If the prefix-side signals were computed with poor completeness, the pair quality is poor.
- If the ASN-side signals were computed with poor completeness, the pair quality is poor.
- If either side is weak, the combined pair is weak.

So the pair completeness is:

- `min(prefix_dc, asn_dc)` — the stricter, conservative choice.

Then `pair_sc` maps completeness into a stable label:

- `HIGH` if `>= 0.85`
- `MEDIUM` if `>= 0.50`
- else `LOW`

**Why this matters:**

It gives the consumer of the table a way to filter results:

“Show me only attack surfaces where both the prefix score and ASN score were computed from sufficiently complete data.”

---

#### **d) Combined score (combined\_score)**

**Computed by:**

```
union = 1.0 - (1.0 - prefix_risk) * (1.0 - asn_risk)

out = feas * union

clamp to [0, 1]
```

#### **What it means (logic):**

This is a two-stage model:

#### **Stage 1: Union-style combination of two independent risk channels**

```
union = 1 - (1 - prefix_risk)*(1 - asn_risk)
```

#### **Why this formula:**

- It behaves like a probability-union intuition: if either component is high, the union rises.
- It prevents “double counting” from simply adding scores.
- It is symmetric: prefix risk and ASN risk both contribute meaningfully.

#### **Stage 2: Feasibility scaling**

```
out = feasibility_factor * union
```

#### **Why multiply by feasibility:**

- The union gives a “theoretical exposure” assuming no hard constraints.
- Feasibility then applies scenario constraints (ROA strictness / ownership) as a practical limiter.

Finally, it clamps to `[0, 1]` to keep the metric well-defined.

---

### e) Attack type (attack\_type)

Computed by:

```
atk_type = choose_attack_type(prefix, scenario, prefix_len)
```

#### What it means (logic):

The script chooses a **dominant practical attack vector**, not an exhaustive taxonomy.

It's based on three inputs only:

- IP family inferred from the prefix string (IPv6 if ":" in prefix else IPv4)
- prefix\_len
- scenario (ROA scenario)

It does **not** depend on which ASN is attacking, because it models:

“Given the prefix geometry + ROA situation, what vector is generally the most realistic in this simplified model?”

#### Why **ORIGIN\_MISMATCH** for IPv4 /24+ (and IPv6 /48+)

- In the script's conservative assumptions, more-specifics beyond those thresholds tend to be heavily filtered / less dominant operationally.
- Therefore it labels the dominant vector as “announce the same prefix from another origin” → **ORIGIN\_MISMATCH**.

#### Why **MORE\_SPECIFIC** for shorter prefixes when ROA is not strict

- If the prefix is shorter (IPv4 < /24, IPv6 < /48), the attacker can attempt a subprefix that still falls into a “more plausible” length range under the script's conservative heuristics.
- If ROA is **NO\_ROA** or **PERMISSIVE\_ROA** the script treats this as “not immediately ruled out by strict authorization constraints,” so it picks **MORE\_SPECIFIC**.

#### Why **STRICT\_ROA** tends to push to **ORIGIN\_MISMATCH**

- Under strict ROA, the model treats “more-specific as dominant” as less appropriate (again: simplified heuristic).
  - Even if the attempt would still be **INVALID** for a non-origin attacker, this field is only “vector label.” Enforcement likelihood is handled elsewhere.
- 

#### f) Attacker RPKI state (**attacker\_rpki\_state**)

Computed by:

```
rpki_state = attacker_rpki_state(scenario, attacker_asn, origin_asn)
```

#### What it means (logic):

This models the **validation outcome at a RPKI-validating receiver** for an announcement of that prefix originated by **attacker\_asn**.

- If **scenario == NO\_ROA** → **NOT\_FOUND**
  - **Why:** with no ROA, a validator cannot produce VALID/INVALID, so it yields NOT\_FOUND.
- Else (ROA exists):
  - If **attacker\_asn == origin\_asn** → **VALID**
    - **Why:** the announcement origin matches the “authorized origin” represented by **origin\_asn** in the joined dataset (and also models the “compromised origin” case).
  - Else → **INVALID**
    - **Why:** ROA exists and the origin ASN does not match the authorized origin in the model.

This field is intentionally independent of “will it propagate,” because propagation depends on adoption/enforcement, which is approximated later via **success\_likelihood\_band**.

---

#### g) Attacker environment band (attacker\_env\_band)

Computed by:

```
env_band = attacker_env_band_from_asn_risk(asn_risk)
```

**What it means (logic):**

This is a **proxy classification** of “how permissive the attacker’s surrounding ecosystem is likely to be,” derived only from `asn_risk`.

- `PERMISSIVE` if `asn_risk >= 0.70`
- `STRICT` if `asn_risk <= 0.30`
- else `MIXED`

**Why this is done:**

The script uses this as an interpretable label to slightly modulate `success_likelihood_band`. It does **not** claim to directly measure ROV deployment; it is explicitly a proxy derived from the ASN ML outcome.

---

#### h) Success likelihood band (success\_likelihood\_band)

Computed by:

```
succ_band = success_likelihood_band(combined_score, rpki_state,  
atk_type, env_band)
```

**What it means (logic):**

This field turns the numeric combined score into an **explainable categorical likelihood** by applying small, deterministic adjustments that reflect practical intuition in the model.

It starts from:

- `s = combined_score`

Then adjusts:



## RPKI state adjustment

- If **NOT\_FOUND**  $\rightarrow s += 0.10$ 
  - **Why:** in the model, NOT\_FOUND is treated as easier to pass through networks than INVALID (no explicit cryptographic contradiction).
- If **INVALID**  $\rightarrow s -= 0.15$ 
  - **Why:** invalid routes are assumed to face more policy friction where ROV is enforced.
- If **VALID**  $\rightarrow$  no change
  - **Why:** valid does not add extra “attack success” by itself, it just removes the invalid penalty.

## Attack type adjustment

- If **MORE\_SPECIFIC**  $\rightarrow s += 0.10$ 
  - **Why:** more-specific announcements tend to win best-path selection when accepted, so the model gives them a boost.

## Environment band adjustment

- If **STRICT**  $\rightarrow s -= 0.10$ 
  - **Why:** strict ecosystems reduce propagation likelihood in the model.
- If **PERMISSIVE**  $\rightarrow s += 0.05$ 
  - **Why:** permissive ecosystems slightly increase propagation likelihood in the model.
- **MIXED**  $\rightarrow$  no change

Then it clamps  $s$  to  $[0, 1]$  and maps to bands:

- HIGH if  $s \geq 0.66$
- MEDIUM if  $s \geq 0.33$
- else LOW

**Key point (what this band represents):**

It is not “ground-truth probability.” It is a consistent, explainable **ranking label** derived from:

- ASN ML risk,
- prefix ML risk,
- ROA scenario feasibility scaling,
- and simple propagation heuristics encoded in the function.

## 7.5 Top-N selection per prefix

The script sorts attacker candidates by:

- `combined_score` descending

Then keeps:

- the first `topn_per_prefix` rows

Only these top-N rows are written for that prefix.

## Step 8 — Write results via UPSERT

Rows are inserted into `asn_prefix_attack_surface` in batches using:

- `INSERT ... ON CONFLICT(prefix, origin_asn, attacker_asn) DO UPDATE SET ...`

This updates all fields and refreshes `updated_at`.

## Step 9 — Print summary (optional)

If `--print-summary`:

- prints number of processed prefixes and written rows
  - prints parameters
  - prints that the confidence model is `min(prefix, asn)`
  - prints that extra fields were added
- 

## 6. Deep Explanation of `attack_type`

How it is calculated in the code:

```
attack_type = choose_attack_type(prefix, scenario, prefix_len)
```

What this field represents, conceptually

`attack_type` is not an exhaustive classification of all possible BGP attacks.

It is a **dominant label**, intentionally chosen to answer the question:

**“Given only the prefix geometry and the ROA situation, what is the most realistic and dominant attack vector operationally in a conservative model?”**

The script **does not try to enumerate all possible tactics**, but rather aims to select:

- The **most likely vector to be attempted**
- The **vector with the highest operational impact on average**
- A vector that is **consistent with common Internet filtering**

Therefore:

- `attack_type` **does not depend on the attacking ASN.**
- It **depends only on the properties of the prefix and ROA.**

---

## Model Inputs

The model uses exactly **three factors**:

1. **IP Family:**

- IPv4 if the prefix *does not* contain a `:`.
- IPv6 if the prefix contains a `:`.

2. **Prefix Length:**

- For IPv4, the critical threshold is `/24`.
- For IPv6, the critical threshold is `/48`.

3. **ROA Scenario:**

- `NO_ROA`
- `PERMISSIVE_ROA`
- `STRICT_ROA`

---

## Why these thresholds exist (/24 for IPv4, /48 for IPv6)

### Real-World BGP Rationale

In the real Internet:

- **IPv4 more specific than /24 is:**
  - **frequently filtered**
  - **rarely propagated globally**
  - **unstable as an attack vector**

- **IPv6 more specific than /48** has similar limitations.

Thus, the model states:

*“A rational attacker will not choose a vector that is highly likely to be filtered massively.”*

This is why:

- **/24+ (IPv4)** and **/48+ (IPv6)** are treated differently.
- 

## Exact Logic for IPv4

**Case 1: `prefix_len >= 24`**

→ **ORIGIN\_MISMATCH**

This rule applies regardless of ROA state.

**Why?**

An attack using a more-specific prefix would require announcing:

`/25, /26, ... /32.`

While such announcements **may be RPKI-valid** under permissive ROAs (i.e., `maxLength > 24`), **they are frequently rejected at import** due to widespread operational prefix-length filtering.

In practice:

- `/24` represents the **finest granularity commonly accepted** for IPv4 routing on the global Internet.
- More-specific prefixes beyond `/24` experience sharply reduced propagation, independent of ROA validity.

As a result:

- The **effective propagation probability** of a more-specific attack drops significantly once the legitimate prefix is already at `/24`.

- The more realistic and higher-probability vector becomes **announcing the same prefix from a different ASN**.

Even when ROA exists or is permissive, operational prefix-length policies dominate the decision.

Therefore, for prefixes already at /24, the dominant attack vector shifts to **ORIGIN\_MISMATCH**.

---

### Case 2: **prefix\_len < 24**

Here, there is room for operationally realistic more-specific prefixes.

Subcase: **NO\_ROA, PERMISSIVE\_ROA**

→ **MORE\_SPECIFIC**

**Why? (very important)**

Operationally, when accepted by filtering policies, more-specific prefixes (up to /24 for IPv4 and /48 for IPv6) are preferred in forwarding due to longest-prefix matching, which makes subprefix announcements an effective traffic-steering and hijack vector.

This happens when ROA:

- Either it **does not exist**,
- Or it is **permissive**,

Thus:

- The attacker has a **realistic incentive** to try a subprefix hijack, and this becomes the **dominant vector**.
- 

Subcase: **STRICT\_ROA**

→ **ORIGIN\_MISMATCH**

**Why?**

- **Strict ROA:**
  - Blocks unauthorized subprefixes at the protocol level.

The logic here is:

- **"More-specific no longer makes sense under strict ROA as it signals an explicit intent by the operator to prevent prefix fragmentation."**

Therefore:

- The dominant vector remains **ORIGIN\_MISMATCH**, even if more-specific attacks might technically be attempted.

---

## Exact Logic for IPv6

This is **analogous to IPv4**, with the **/48** threshold.

Why?

- **/48** is the **"natural" operational unit** for IPv6.
- More-specific subprefixes are:
  - **Less accepted,**
  - **More unstable.**

---

## What this field does not do (intentionally):

- ✗ **Does not tell** if the attack will succeed.
- ✗ **Does not tell** if it is **VALID** or **INVALID**.
- ✗ **Does not consider ROV adoption.**

All of these factors are handled **later** in the process, within:

- `attacker_rpki_state`
  - `success_likelihood_band`
- 

## 7. Deep Explanation of `attacker_rpki_state`

How it is calculated in the code:

```
rpki_state = attacker_rpki_state(scenario, attacker_asn, origin_asn)
```

What this field represents, conceptually

`attacker_rpki_state` precisely models:

**“What would a RPKI-validating receiver's verdict be when receiving a BGP announcement for this prefix, originated by `attacker_asn`?”**

This is a **pure cryptographic evaluation model**, independent of:

- Propagation,
  - Local policies,
  - ROV adoption.
- 

**Case 1: `scenario == NO_ROA`**

→ `NOT_FOUND`

**Why?**

- No ROA exists for this prefix.



- The **validator**:
  - Cannot mark the announcement as **VALID**.
  - Cannot mark the announcement as **INVALID**.
- According to **RFC**, the result is **NOT\_FOUND**.

This case is extremely important:

- **NOT\_FOUND** is not the same as **INVALID**.
- Many networks treat **NOT\_FOUND** as more permissive.

---

## Case 2: ROA exists — STRICT ROA

*(maxLength == exact prefix length, or explicitly restrictive)*

### Definition

A ROA is considered **strict** when:

- **maxLength** is equal to the exact prefix length, or
- the operator explicitly signals intent to prevent prefix fragmentation.

### attacker\_rpki\_state logic

**Subcase 2.1: attacker\_asn == origin\_asn**

→ **VALID**

The ROA explicitly authorizes this ASN for the prefix.

**Subcase 2.2: attacker\_asn != origin\_asn**

→ **INVALID**

Why?

- The announcing ASN is not authorized by the ROA.
- Prefix length is irrelevant once the origin ASN does not match.

### Security interpretation

Under strict ROA, more-specific announcements are both:

- cryptographically unauthorized, and
- intentionally discouraged by the resource holder.

This makes **MORE\_SPECIFIC attacks operationally unattractive**, and shifts the realistic attack vector toward **ORIGIN\_MISMATCH** in non-strict networks.

---

## Case 3: ROA exists — PERMISSIVE ROA

*(maxLength allows more-specific prefixes)*

### Definition

A ROA is considered **permissive** when:

- **maxLength** is larger than the announced prefix length,
- allowing operationally realistic subprefix announcements.

### attacker\_rpki\_state logic

**Subcase 3.1: attacker\_asn == origin\_asn**

→ **VALID**

The ASN is explicitly authorized, and the prefix length is within **maxLength**.

**Subcase 3.2: attacker\_asn != origin\_asn**

→ **INVALID**

Why?

- RPKI authorization is **ASN-based**, not prefix-length-based.
- **maxLength** constrains how specific an *authorized* ASN may announce, not which ASNs are allowed.

### Critical clarification

Even when a ROA permits more-specific prefixes via **maxLength**, an announcement originated by an unauthorized ASN remains **cryptographically INVALID**.

However:

- The announcement is **not blocked by maxLength constraints**.
- In non-strict or non-ROV networks, such routes may still propagate.
- Operational filtering, not RPKI, becomes the dominant limiting factor.

### Key takeaway

PERMISSIVE ROA does not make unauthorized origins VALID.

It only removes prefix-length constraints that would otherwise make more-specific announcements immediately implausible.

This distinction is critical:

- RPKI validity answers "Is this cryptographically authorized?"
- Operational reality answers "Will this propagate?"

---

### Why **attacker\_rpk\_i\_state** does not decide propagation

Because in the real world:

- Not all ASes validate.

- Not all ASes reject **INVALID** announcements.
- **Policies differ widely** across networks.

That is why:

- This field is **strictly cryptographic**.
- **Success probability** is estimated later via:
  - `attacker_env_band`
  - `success_likelihood_band`

## 8. How a Final Row is Produced (from raw inputs to final output)

For a single joinable (`prefix`, `origin_asn`):

1. Read prefix vulnerability (`prefix_vuln_score`) and completeness/confidence (or default)
2. Read ROA metadata (`has_roa`, `roa_maxLength`, `prefix_length`)
3. Compute:
  - `scenario`
  - `roa_gap`
  - candidate attackers list
4. For each attacker:
  - load attacker ML risk + completeness/confidence (or default 0/LOW)
  - compute `feasibility_factor`

- compute `pair_data_completeness = min(prefix_dc, asn_dc)`
  - compute `pair_score_confidence`
  - compute `combined_score`
  - compute `attack_type`
  - compute `attacker_rpki_state`
  - compute `attacker_env_band`
  - compute `success_likelihood_band`
5. Sort attackers by `combined_score` descending
  6. Keep top `--topn-per-prefix`
  7. UPSERT rows into `asn_prefix_attack_surface` with a single timestamp (`updated_at`) for the run

---

## 9. Example Output Row (format only; values illustrative)

A single stored record has this conceptual shape:

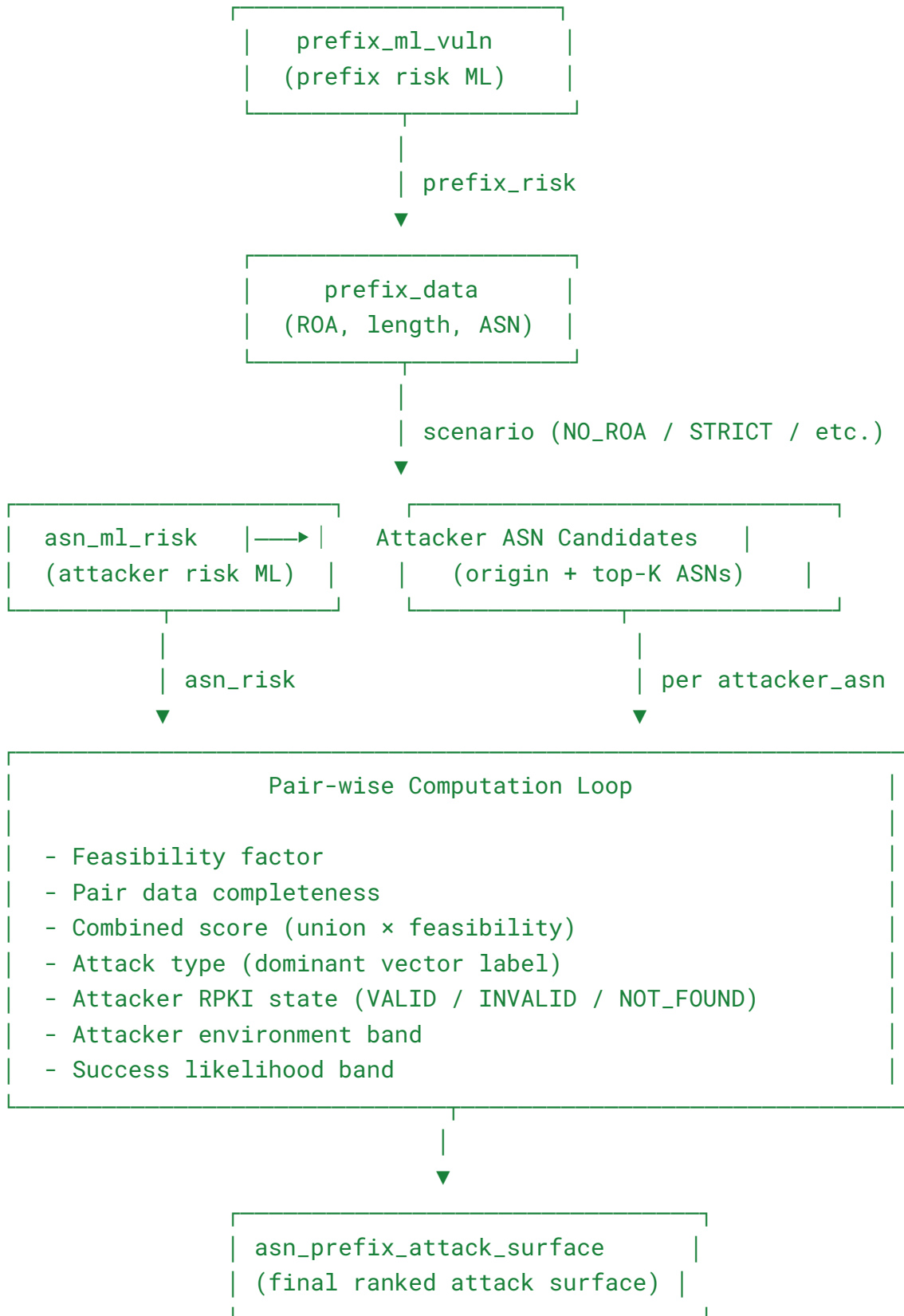
- `prefix="203.0.113.0/23"`
- `origin_asn=64500`
- `attacker_asn=64496`
- `scenario="NO_ROA"`
- `attack_type="MORE_SPECIFIC"`

- `attacker_rpki_state="NOT_FOUND"`
- `roa_gap=-1`
- `attacker_env_band="PERMISSIVE"`
- `success_likelihood_band="HIGH"`
- `prefix_vuln_score=0.82`
- `prefix_data_completeness=0.86`
- `prefix_score_confidence="HIGH"`
- `asn_risk=0.91`
- `asn_data_completeness=0.77`
- `asn_score_confidence="MEDIUM"`
- `pair_data_completeness=0.77`
- `pair_score_confidence="MEDIUM"`
- `feasibility_factor=1.0`
- `combined_score=...`
- `updated_at="2025-12-14T21:07:13Z"`

(Exact numeric results depend on the upstream ML tables and parameters; this example is to show the structure and meaning.)

## 10. End-to-End Data Flow Diagram

The following diagram summarizes the full logical flow of the combinator, from inputs to final outputs.



---

## How to read this diagram

- The model **does not branch probabilistically**.
- Every field is computed deterministically from:
  - prefix properties
  - ROA state
  - attacker ASN risk characteristics
- All attacker–prefix combinations are evaluated independently.
- Final ranking is driven primarily by:
  - `combined_score`
  - contextualized by the derived categorical fields.

---

## Why this design matters

This architecture ensures that the combinator is:

- **Explainable** — every output field has a direct justification
- **Auditable** — no hidden heuristics or black-box steps
- **Composable** — can be extended with time-based or path-based models later
- **Safe for decision support** — without overstating precision