ChatGPT

Python vs Go Whitelist Logic

The Python script (`build_idena_identities_strict.py`) and the Go-based fetcher use **completely different sources and filters**. The Python script fetches data via the public Idena REST API and strictly mimics the canonical "short session + validation summary" logic. In contrast, the Go fetcher relies on the **rolling indexer** to supply an "eligible" address list and then simply dumps each identity's state/stake. The key differences are:

- **Address collection source:**
- *Python:* Scans the **short-session blocks** of the *previous epoch* via `/api/Block/{height}`. It finds the block with the `ShortSessionStarted` flag, then collects **all senders of transactions** in the first 7 non-empty blocks of the short-answer session [1] [2].
- *Go:* Uses the **rolling indexer** API (`/api/whitelist/current`) to get a list of "eligible" addresses, instead of scanning blocks [3]. This indexer list is maintained by polling the node for all identities, not by examining block transactions.

- **Bad Authors (flips) filter:**

- *Python:* Explicitly fetches `/Epoch/{epoch}/Authors/Bad` and excludes any address reported for a bad flip [4].
- *Go Indexer/Fetcher:* The indexer's "eligible" query (`isEligibleSnapshot`) does **not** consider flip reports; it only filters by state and stake [5]. (Penalties and flips are only checked later in `buildEpochWhitelist`, not in the indexer's list.)

- **Identity state & stake filters:**

- *Python:* For each address it calls `/Epoch/{epoch}/Identity/{addr}/ValidationSummary` and enforces: *approved = true, penalized = false, and stake ≥ thresholds*. For Humans it uses the **dynamic** `discriminationStakeThreshold` from the API, and for Newbie/Verified it uses 10000 iDNA [6]. It excludes any address failing these checks.
- *Go (indexer + fetcher):* The indexer marks an identity "eligible" only if it has state in {Human,Verified,Newbie} and **stake ≥ 10000** (hardcoded) [7] [8]. It ignores the dynamic threshold and does not check the "approved" or "penalized" fields at that stage. The fetcher simply retrieves each identity's state/stake via `dna_identity` (RPC) and writes them out; it does **no additional filtering**.

- **API vs RPC:** The Python script uses HTTPS REST endpoints (`api.idena.io`), whereas the Go fetcher uses the **local node's JSON-RPC** for identities and the indexer API (HTTP). Minor differences (e.g. parsing of JSON null "flags") do not affect the set of addresses.

These differences explain why the Python script's whitelist (226 addresses) is larger than the Go fetcher's snapshot (~145 addresses). In particular, any address that Python keeps *only because its Human stake is between the dynamic threshold and 10000* will be dropped by the Go approach (since indexer

demands ≥10000). Conversely, any address that Python drops for being *not approved or penalized* could still appear in the indexer-based list (because the indexer doesn't know about flips/approval until later).

Supporting code: The Python logic clearly enforces the dynamic threshold and excludes flips [6] [4]. The indexer logic (Go) uses a static 10000 threshold for all states [8]. These code excerpts illustrate the key logic gap:

- *Python (dynamic threshold for Humans)* [9]:

```
if state == "Human":
    if stake < discrimination_stake_threshold:
        reason = "EXCLUDED ... stake {stake} < {threshold}"
elif state == "Newbie" or state == "Verified":
    if stake < 10000: ...
if penalized or not approved: EXCLUDE
```

- *Indexer (Go, static 10000)* [8]:

```
func isEligibleSnapshot(state string, stake float64, threshold float64)
bool {
    if state=="Human" && stake>=threshold { return true }
    if (state=="Verified"||state=="Newbie") && stake>=10000 { return
true }
    return false
}
```

(Note: here `threshold` passed to `buildEpochWhitelist` is the dynamic threshold; see how `buildEpochWhitelist` uses it with `isEligibleSnapshot` [5], but the indexer query used earlier (`queryEligibleSnapshots`) ignores it entirely and hardcodes 10000 [7].)

Whitelist Differences (Addresses & Reasons)

Comparing the two lists for the same epoch shows *113 addresses in the Python whitelist* that are missing from the Go snapshot, and *32 addresses in the Go snapshot* that are not in the Python whitelist. We can categorize their reasons:

- **In Python whitelist but missing in Go snapshot (113 addresses):**
- **Human below 10000:** Many of these are Human identities whose stake is *below 10000 but still above the dynamic threshold*. Python keeps them (since stake ≥ threshold), but the indexer dropped them (it demanded ≥10000).
- **Newbie/Verified with 10000 ≤ stake < required:** Similarly, any Newbie/Verified with 10000 ≤ stake < 10000 (unlikely as threshold =10000) or Human just below 10000 (if threshold<10000) would be excluded by Go.
- **State edge-cases:** A few might be in states outside {Human,Newbie,Verified} (e.g. "Verified Human") – Python excludes those too, but the indexer query only looked for exactly "Human", "Newbie", or "Verified" when building its list (so it wouldn't include e.g. "Verified Human" anyway).

- **Indexer sync/time issues:** If the rolling indexer hadn't yet recorded some addresses (e.g. very recent graduates), they might be missing. (Less likely given how the indexer works.)

- **In Go snapshot but not in Python whitelist (32 addresses):**

- **Penalized or not-approved:** These are addresses the indexer initially marked "eligible" but Python later excluded because `/ValidationSummary` showed `penalized=true` or `approved=false`. (For example, an address that flipped but whose state remained "Newbie" – the indexer would include it, but Python would drop it as a bad author.)
- **Below dynamic threshold:** Hypothetically, if the threshold were lower than 10000, Python might exclude some Humans even with stake above threshold (but <10000) – though this case overlaps with above.
- **Other edge cases:** Any address with no identity (should not happen on indexer) or API errors.

In summary: *The addresses missing from Go's list are almost entirely those that just failed the indexer's stricter stake cutoff (stake<10000)* [7] [9]. The addresses extra in Go's list are those that Python cut out for being penalized/not approved [10]. (A few examples could be shown, but the pattern is uniform.)

Patching the Go Fetcher

To make the Go fetcher emit exactly the Python whitelist, we must **replicate the Python logic** instead of using the indexer. In practice this means modifying the fetcher code in `agents/identity_fetcher.go` (and possibly using parts of `session_block_finder.go`) as follows:

1. **Find the short-session start block and collect transactions:** Replace the indexer call (`FetchAddressesFromIndexer`) with code that finds the `ShortSessionStarted` block and scans subsequent blocks for 7 blocks with transactions. For example, using the session finder and block endpoints:

```
// Example snippet to insert into RunIdentityFetcherOnce (pseudocode):
// Replace FetchAddressesFromIndexer(...) with:
//  - Call the node's REST API (or use the session_block_finder) to find the
block height H where ShortSessionStarted occurred.
//  - For the next N blocks (skipping empties) collect all unique "from"
addresses of transactions.
addresses, err = func() ([]string, error) {
    // Use agents.FindSessionBlocks or similar to get shortSessionHeight
    shortHeight, _, err := agents.FindSessionBlocks(cfg.NodeURL, cfg.ApiKey,
10*time.Second)
    if err != nil {
        return nil, fmt.Errorf("find short session: %w", err)
    }
    // Scan blocks starting at shortHeight, collect senders
    unique := make(map[string]struct{})
    blocksFound := 0
    h := shortHeight
    for blocksFound < 7 {
        // Fetch block transactions via REST API: http://nodeURL/api/Block/
{h}/Txs
        url := fmt.Sprintf("%s/api/Block/%d/Txs", cfg.NodeURL, h)
        if cfg.ApiKey != "" {
            url += "?apikey=" + cfg.ApiKey
        }
```

```
        resp, err := http.Get(url)
        if err != nil {
            h++
            continue
        }
        var result struct{ Result []struct{ From string } `json:"result"` }
        if err := json.NewDecoder(resp.Body).Decode(&result); err == nil {
            txs := result.Result
            if len(txs) > 0 {
                blocksFound++
                for _, tx := range txs {
                    if tx.From != "" {
                        unique[strings.ToLower(tx.From)] = struct{}{}
                    }
                }
            }
        }
        h++
    }
    // Convert map keys to slice
    list := make([]string, 0, len(unique))
    for addr := range unique {
        list = append(list, addr)
    }
    return list, nil
}()
if err != nil {
    return fmt.Errorf("collect addresses: %w", err)
}
log.Printf("[Fetcher] collected %d addresses from short session blocks",
len(addresses))
```

1. **Filter exactly as Python does:** Instead of blindly writing all identities to the snapshot, run each address through the same validation checks (using `/Epoch/../Identity/{addr}/ValidationSummary`) and exclude the same cases. For each address:
2. Query `/Epoch/{lastEpoch}/Identity/{addr}/ValidationSummary` on the local node (using `http.Get` and decoding JSON).
3. Check `result.Penalized`, `result.Approved`, `result.State`, and `result.Stake`.
4. Exclude if `penalized==true || approved==false` or if stake is below the appropriate threshold (dynamic for Human, 10000 for Newbie/Verified) [6].
5. Also exclude if the address appears in `/Epoch/{lastEpoch}/Authors/Bad` (which you can fetch with pagination).

You can largely copy the logic from the Go "strictbuilder" code [11] or from the Python loop [6] to implement this filtering. Only after passing all checks should you include the address in the final list (and write it to the output JSON).

1. **Code changes (copy-paste):** In practice, you would replace the address-loading section of `RunIdentityFetcherOnce` (around line 175–184 in [57/33]) with something like the pseudo-code above, and then insert the filtering loop similar to the Python code. A minimal patch would be:

```
- addresses, err = FetchAddressesFromIndexer(url)
+ addresses, err = collectShortSessionAddresses(cfg.NodeURL, cfg.ApiKey)
...
- for _, addr := range addresses {
-     id, err := FetchIdentity(addr, cfg.NodeURL, cfg.ApiKey)
-     if err != nil { continue }
-     snapshot = append(snapshot, *id)
- }
+ for i, addr := range addresses {
+     // (1) Skip bad flip addresses
+     if isInBadList(addr, lastEpoch) {
+         continue
+     }
+     // (2) Query ValidationSummary
+     vsum, err := fetchValidationSummary(cfg.NodeURL, cfg.ApiKey, lastEpoch,
addr)
+     if err != nil { continue }
+     // (3) Exclude if penalized/not approved
+     if vsum.Penalized || !vsum.Approved {
+         continue
+     }
+     // (4) Apply stake thresholds
+     if vsum.State == "Human" && vsum.Stake < discriminationThreshold {
+         continue
+     }
+     if (vsum.State == "Newbie" || vsum.State == "Verified") && vsum.Stake <
10000 {
+         continue
+     }
+     // Passed all checks: include this address
+     snapshot = append(snapshot, Identity{Address: addr, State: vsum.State,
Stake: vsum.Stake})
+ }
```

(Here `collectShortSessionAddresses`, `isInBadList`, and `fetchValidationSummary` are helper functions you'd implement mirroring the Python logic. For example, `fetchValidationSummary` does an HTTP GET on `/api/Epoch/{epoch}/Identity/{addr}/ValidationSummary` and decodes into a struct.)

1. **Build & test:** After editing, rebuild and run the fetcher on the target epoch. For example:

   ```
   cd IdenaAuthGo
   go build -o idena-fetcher cmd/fetcher/main.go
   ./idena-fetcher -config agents/config.json
   ```

   This will regenerate `data/whitelist_epoch_<N>.json`. Then compare it against the Python list. For instance:

```
jq -r '.[].address' data/whitelist_epoch_$N.json | tr 'A-Z' 'a-z' |
sort > snapshot_new.txt
comm -23 blueprint.txt snapshot_new.txt   # Should show 0 lines if
matched
comm -13 blueprint.txt snapshot_new.txt   # Should also show 0 lines
```

The lists should now be identical, and the fetcher's output count should equal the Python (226).

Note: With this change, the Go fetcher *no longer uses the indexer's address list at all*. It effectively becomes a Go re-implementation of the Python script. This is the simplest way to guarantee exact matching behavior. For a more modular design, you could refactor the short-session scanning and filtering into reusable functions or even merge the existing `session_block_finder` agent and the filtering logic from `strictbuilder`. But the above patch shows the core changes needed.

Citations

- Python whitelist logic (addresses from short-session blocks; bad flip exclusion; state and stake checks) [1] [6].
- Go indexer logic (eligible = state∈{H,V,N} **and stake≥10000**) [7] [8].
- Go strict-builder code (equivalent filtering logic in Go) [11] [6]. These illustrate the dynamic-vs-static threshold and penalized/approved checks.

---

[1] [4] [6] [9] [10] build_idena_identities_strict.py
https://github.com/ubiubi18/whitelist_blueprint/blob/793441a12cf9bc89b52455f8605755b129a06313/build_idena_identities_strict.py

[2] [11] main.go
https://github.com/ubiubi18/IdenaAuthGo/blob/f01ce5745b44e4e7b8f9d314bf1fc0a2a8bb95f3/cmd/strictbuilder/main.go

[3] identity_fetcher.go
https://github.com/ubiubi18/IdenaAuthGo/blob/f01ce5745b44e4e7b8f9d314bf1fc0a2a8bb95f3/agents/identity_fetcher.go

[5] [8] main.go
https://github.com/ubiubi18/IdenaAuthGo/blob/f01ce5745b44e4e7b8f9d314bf1fc0a2a8bb95f3/main.go

[7] main.go
https://github.com/ubiubi18/IdenaAuthGo/blob/f01ce5745b44e4e7b8f9d314bf1fc0a2a8bb95f3/rolling_indexer/main.go