BGP-Sentry: A Blockchain-Based Distributed BGP Security Framework

Complete Technical Reference

Author: Anik Tahabilder **Date:** February 2026

Table of Contents

Chapter 1: Introduction

What is BGP-Sentry?

BGP-Sentry is a **blockchain-based distributed BGP security framework** that detects BGP hijack attacks using RPKI-validated consensus among autonomous systems. It uses real CAIDA AS-level Internet topology data and rov-collector RPKI classification data to simulate a distributed network where:

- **RPKI-enabled ASes** act as blockchain validators (signers/mergers) — they observe, vote, and commit blocks
- **Non-RPKI ASes** are monitored subjects — their routing behavior is tracked and rated by RPKI validators

Every BGP announcement is processed through a full blockchain pipeline: validation, transaction creation, peer-to-peer broadcast, Proof of Population (PoP) consensus voting, block commitment, attack detection, and token reward distribution.

Key Contributions

1. **Real-time BGP security** using blockchain consensus (36.8 TPS peak)
2. **Proof of Population (PoP) consensus** – one node, one vote; RPKI onboarding prevents Sybil attacks
3. **Knowledge-based voting** where validators approve/reject based on their own independent observations
4. **Asymmetric architecture**: RPKI validators do full consensus; non-RPKI ASes are monitored subjects with trust ratings
5. **Token economy** (BGPCoin) that incentivizes honest participation

6. **Trust rating system** that tracks non-RPKI AS behavior longitudinally
7. **15 performance optimizations** achieving zero-lag real-time processing
8. **Perfect attack detection** (F1 = 1.0) maintained at all throughput levels

System Requirements

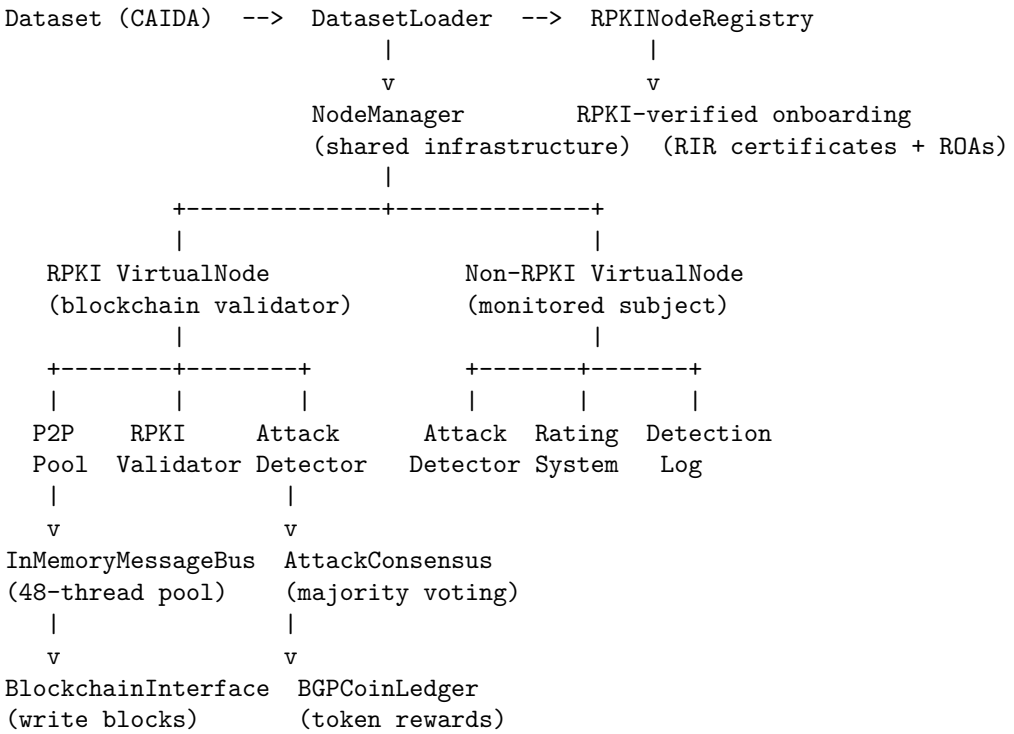
Component	Requirement
Python	3.10+
CPU	4+ cores recommended (tested on 24 cores)
RAM	4 GB minimum (tested on 62.5 GB)
Disk	1 GB for results
GPU	Not required (bottleneck is coordination, not computation)
OS	Linux recommended (tested on Ubuntu 24.04)

Back to Table of Contents

Chapter 2: System Architecture

High-Level Overview

The system follows a layered architecture with clear separation between data loading, node management, blockchain infrastructure, and P2P communication.



Data Flow

Step-by-Step Processing:

1. **Dataset Selection**: Choose from caida_100, caida_200, caida_500, or caida_1000

2. **Registry Initialization**: `RPKINodeRegistry.initialize(dataset_path)` reads `as_classification.json` and populates RPKI/non-RPKI node lists
3. **Data Loading**: `DatasetLoader` loads per-AS observation files and ground truth
4. **VRP Generation**: Extract legitimate (prefix, origin_asn) pairs for RPKI validation
5. **Node Creation**: `NodeManager` creates shared blockchain infrastructure and one `VirtualNode` per AS
6. **Orchestrator Start**: All nodes start processing in parallel threads
7. **Real-Time Clock**: `SharedClock` paces observation replay to match BGP timestamps
8. **Results Collection**: 13 structured JSON files + human-readable README per run

RPKI-Verified Onboarding

Before any AS can participate as a blockchain validator, it must pass RPKI-verified onboarding:

```
Step 1: RIR Certificate Issuance
  A Regional Internet Registry (ARIN, RIPE, APNIC, LACNIC, AFRINIC)
  issues an RPKI certificate binding the AS number to its IP resources.

Step 2: ROA Publication
  The AS publishes Route Origin Authorizations (ROAs) in the RPKI
  repository, declaring which prefixes it is authorized to originate.

Step 3: rov-collector Measurement
  The rov-collector system crawls all 5 RIR RPKI repositories and
  classifies each AS based on real-world RPKI deployment status.

Step 4: Classification
  rov-collector produces as_classification.json with:
  - rpki_asns: ASes with valid RPKI certificates and published ROAs
  - non_rpki_asns: ASes without RPKI deployment

Step 5: Registry Initialization
  RPKINodeRegistry.initialize(dataset_path) reads as_classification.json
  and populates:
  - RPKI_NODES → become blockchain validators (can vote, commit blocks)
  - NON_RPKI_NODES → become monitored subjects (tracked via trust ratings)
```

Why this prevents Sybil attacks: Creating a fake validator requires a real RPKI certificate from a RIR, which requires owning real IP address space. The cost of acquiring IP resources from an RIR makes Sybil attacks economically infeasible — unlike Proof of Work (buy hardware) or Proof of Stake (buy tokens).

RPKI Validator Pipeline (Merger/Signer Model)

Each RPKI node acts as both a **merger** (for its own transactions) and a **signer** (voting on other nodes' transactions).

As Merger (processing own observations):

```
Step 0: DEDUP CHECK
  Is this (prefix, origin) seen within 5 minutes?
  Is it NOT an attack?
  -> YES to both: SKIP entire pipeline (save consensus round)
  -> NO: Continue

Step 1: KNOWLEDGE BASE ADD
  Store observation so this node can vote on others' transactions
  about the same prefix
```

```
Step 2: RPKI VALIDATION (VRP)
  Check against StayRTR Validated ROA Payload table
  Result: "valid" / "invalid" / "not_found"

Step 3: ATTACK DETECTION (4 types)
  - PREFIX_HIJACK:    Origin AS doesn't match ROA
  - SUBPREFIX_HIJACK: More-specific prefix with wrong origin
  - BOGON_INJECTION:  RFC 1918/5737/6598 reserved ranges
  - ROUTE_FLAPPING:   >5 state changes in 60-second window

Step 4: CREATE TRANSACTION + BROADCAST
  Create signed transaction -> broadcast to 5 random peers
  (runs in background thread -- pipelined)

Step 5: UPDATE DEDUP STATE
  Record (prefix, origin) -> current_time for future dedup checks
```

As Signer (voting on others' transactions):

```
Receive vote request from peer
  |
  v
Check knowledge base: "Did I also observe this announcement?"
  |
  v
Same prefix+AS observed  -> Sign "approve" vote with Ed25519 private key
Never saw this prefix    -> Sign "no_knowledge" vote (abstain -- no data)
Saw prefix from diff. AS -> Sign "reject" vote (conflicting origin)
  |
  v
Send vote response back to merger
```

Only `approve` votes count toward the consensus threshold. `no_knowledge` is neutral (the signer has no data to confirm or deny). `reject` signals a suspicious conflicting origin that may indicate a hijack.

Consensus Resolution:

```
Merger collects votes:
  3+ approve -> CONFIRMED (write block with full consensus)
  Timeout (3s regular, 5s attack):
    1-2 approve -> INSUFFICIENT_CONSENSUS
    0 approve   -> SINGLE_WITNESS
  |
  v
Block committed to blockchain (SHA-256 hash chain)
  |
  v
Block replicated to all peers (background thread)
  |
  v
Attack detection on committed transaction (background thread)
  |
  v
BGPCoin rewards distributed to merger + voters
```

Non-RPKI AS Pipeline (Monitored Subjects)

Non-RPKI ASes do not participate in blockchain consensus or voting. They are the **subjects being monitored** — their routing behavior is tracked by the system and used to compute trust ratings. RPKI validators are the actual observers.

```
Step 0: DEDUP CHECK
  Same (prefix, origin) within 2 minutes? -> SKIP

Step 1: ATTACK DETECTION (4 types)
  Same detectors as RPKI nodes

Step 2a: If attack detected:
  - Write to blockchain immediately
  - Apply trust rating penalty (-10 to -50 points)
  - Record in attack detections list

Step 2b: If legitimate:
  - Record to blockchain
  - Track trust rating (reward for good behavior)
```

P2P Communication

InMemoryMessageBus – Replaces TCP sockets with in-process message routing:

Feature	TCP Sockets	InMemoryMessageBus
OS resources	1 socket per node pair	Zero sockets
Scalability	~1000 (ulimit bound)	10,000+ nodes
Latency	~0.5ms (loopback)	~0.01ms (function call)
Delivery	Async (thread pool)	Async (48-thread pool)
Reliability	TCP guarantees	100% delivery (same process)

Message Types:

Type	Direction	Purpose
`vote_request`	Merger -> Signers	"I observed this, please vote"
`vote_response`	Signer -> Merger	"approve" or "reject" with Ed25519 signature
`block_replicate`	Committer -> All	"Here's the committed block for your chain"
`attack_proposal`	Detector -> All	"I detected an attack, vote on it"
`attack_vote`	Voter -> Proposer	"I agree/disagree this is an attack"

Blockchain Structure

Block Format:

```
{
  "block_number": 42,
  "timestamp": "2026-02-17T19:30:00",
  "previous_hash": "a1b2c3...64 hex chars",
  "merkle_root": "d4e5f6...64 hex chars",
  "block_hash": "789abc...64 hex chars",
```

```
  "transactions": [
    {
      "transaction_id": "tx_4213_20260217_193000_a1b2c3d4",
      "observer_as": 4213,
      "sender_asn": 15169,
      "ip_prefix": "8.8.8.0/24",
      "consensus_status": "CONFIRMED",
      "approve_count": 4,
      "signatures": [...]
    }
  ]
}
```


Integrity Verification:

- **SHA-256 hash chain**: Each block references the previous block's hash
- **Merkle root**: Computed over all transactions in the block
- **Full verification**: Walk the chain from genesis, recompute all hashes
- **Per-node replicas**: Each RPKI node maintains its own chain copy

In-Memory Blockchain Design

The blockchain is maintained as an **in-memory Python data structure** (a list of block dictionaries) with periodic persistence to disk via JSON files. Block generation, hash computation, Merkle root calculation, and chain append all happen entirely in RAM.

Why In-Memory?

BGP hijack detection is a **real-time security problem**. When an attacker announces a fraudulent route, the Internet begins routing traffic through them within seconds. The system must:

1. **Process observations at wire speed** – RPKI validators receive thousands of BGP announcements per second. Each must be validated, voted on, and committed before the next batch arrives. Disk-based block generation would introduce 5–50ms of I/O latency per block, making real-time consensus impossible at scale.
2. **Minimize consensus latency** – The PoP consensus protocol requires multiple nodes to vote on each transaction within a tight timeout window (3–5 seconds). If block writes blocked on disk I/O, the lock contention would serialize all 58+ validator nodes, collapsing throughput.
3. **Support concurrent node replicas** – Each RPKI validator maintains its own chain copy via in-memory replicas (`in_memory=True` mode in `BlockchainInterface`). These replicas receive replicated blocks from peers without any disk overhead, enabling O(1) block appends across all nodes simultaneously.
4. **Decouple integrity from storage medium** – Blockchain integrity comes from SHA-256 hash chaining and Merkle roots, not from the storage backend. The cryptographic guarantees hold whether the chain lives in RAM or on disk. The in-memory design preserves all security properties while eliminating the I/O bottleneck.

Persistence Strategy:

The system is not purely in-memory – it uses a **write-behind** approach:

- The **primary chain** (shared across all nodes) persists to `blockchain.json` using atomic writes (write to temp file, then rename) to prevent corruption from crashes.
- **Per-node replicas** run in pure in-memory mode (`in_memory=True`) and skip all disk I/O, since they can be rebuilt from the primary chain or peer replication.
- **Batched writes** (Optimization 15) accumulate multiple state updates in memory before flushing, reducing disk I/O by up to 49x.

- The lock is held only for the in-memory mutation (~0.1ms); file I/O for auxiliary logs runs outside the critical section.

This design achieves sub-millisecond block generation while maintaining crash safety through atomic persistence and cross-node replication.

Cryptographic Signing

Ed25519 (Current):

Property	Value
Algorithm	Ed25519 (Curve25519)
Key size	32 bytes (private), 32 bytes (public)
Sign time	~0.05ms
Verify time	~0.1ms
Key generation	~0.05ms

Previously used RSA-2048 (~1ms sign, ~50ms key gen). Ed25519 was chosen for 20x faster signing.

What Gets Signed:

1. **Transaction creation**: Merger signs the full transaction payload
2. **Vote response**: Each signer signs `{transaction_id, voter_as, vote}` with their private key
3. **Signature verification**: Merger verifies each vote signature before counting

Back to Table of Contents

Chapter 3: Proof of Population (PoP) Consensus

What is Proof of Population?

BGP-Sentry uses **Proof of Population (PoP)** – a novel consensus protocol designed specifically for BGP security. Unlike Proof of Work (computation) or Proof of Stake (wealth), PoP derives consensus authority from **verified network identity**.

Core Principle: One Node = One Vote

Every RPKI-validated autonomous system gets exactly one vote in consensus. There is no way to gain more voting power by accumulating tokens, computing hashes faster, or any other means. The population of legitimate, RPKI-verified ASes *is* the consensus authority.

Why Not Traditional BFT?

Traditional Byzantine Fault Tolerance (BFT) protocols like PBFT assume: - A fixed, small set of known validators - Validators check transaction *format* and *consistency* - Any validator can verify any transaction the same way

BGP-Sentry is different: - Validators number in the **dozens to hundreds** (58-366 in tested networks) - Each validator can only verify announcements it has **independently observed** - Two validators may have legitimately different views of the same prefix

PoP addresses this by combining identity-based authority with knowledge-based verification.

Sybil Resistance via RPKI Onboarding

The critical question for any "one node one vote" protocol is: **how do you prevent one entity from creating many fake nodes (Sybil attack)?**

In BGP-Sentry, Sybil resistance comes from RPKI infrastructure:

```
To become a validator, an AS must:
  1. Hold a valid RPKI certificate from a Regional Internet Registry (RIR)
  2. Have Route Origin Authorizations (ROAs) published in the RPKI repository
  3. Be classified as RPKI-enabled in the rov-collector measurement data

This is not self-certifiable:
  - RPKI certificates are issued by RIRs (ARIN, RIPE, APNIC, LACNIC, AFRINIC)
  - Each certificate binds to real IP address space allocated to a real organization
  - Creating a fake AS with RPKI requires controlling actual IP resources
  - The cost of a Sybil attack = cost of acquiring real IP address space from an RIR
```

This makes Sybil attacks economically infeasible – unlike Proof of Work (buy more hardware) or Proof of Stake (buy more tokens), you cannot simply purchase more RPKI identities.

Knowledge-Based Voting

PoP validators don't just rubber-stamp transactions. They vote based on **whether they independently observed the same BGP announcement**:

```
Merger (AS 4213): "I saw AS15169 announce 8.8.8.0/24. Please vote."
  |
  v
Signer (AS 7018): Checks knowledge base...
  "Yes, I also observed AS15169 announcing 8.8.8.0/24" -> APPROVE
  |
Signer (AS 3356): Checks knowledge base...
  "I never saw any announcement for 8.8.8.0/24"       -> NO_KNOWLEDGE (abstain)
  |
Signer (AS 1299): Checks knowledge base...
  "Yes, I observed it too"                              -> APPROVE
  |
Signer (AS 2914): Checks knowledge base...
  "I saw 8.8.8.0/24 announced by AS99999, not AS15169" -> REJECT (conflicting origin)
```

Three vote outcomes:

Vote	Meaning	Counts toward threshold?
APPROVE	Signer independently observed the same prefix+AS	Yes
NO_KNOWLEDGE	Signer has no data about this prefix (abstain)	No (neutral)
REJECT	Signer observed the prefix from a *different* AS	No (signals conflict)

This means: - **Legitimate announcements** seen by many ASes get approved quickly (widespread propagation) - **Hijacked routes** get rejected – other ASes either have no data or see the prefix from the legitimate

origin - `no_knowledge` prevents nodes from casting uninformed reject votes that would penalize legitimate but localized announcements - `reject` is a strong signal: the signer has *contradicting* evidence, suggesting a possible hijack - **The knowledge base acts as a distributed witness system** – consensus reflects what the Internet actually observed

Consensus Parameters

Threshold Formula: `threshold = max(MIN, min(N/3 + 1, CAP))`

Where N = number of RPKI validators.

Parameter	Default	Purpose
`CONSENSUS_MIN_SIGNATURES`	3	Minimum votes to commit
`CONSENSUS_CAP_SIGNATURES`	5	Upper cap for large networks

For all tested network sizes (58-206 RPKI nodes), the effective threshold is **5 signatures** out of 5 peers queried.

Comparison with Other Consensus Protocols

Protocol	Authority Source	Sybil Resistance	Voting Model
Proof of Work	Computation power	Energy cost	Longest chain
Proof of Stake	Token holdings	Capital cost	Weighted vote
Practical BFT	Pre-selected validators	Permissioned	2/3 majority
Proof of Population (PoP)	**RPKI identity**	**RIR certification cost**	**One node, one vote + knowledge verification**

Consensus Escalation Detection

The system detects **learning attackers** whose hijack attempts get progressively more consensus votes:

```
Attempt 1: AS99 announces 8.8.8.0/24 -> 0 votes (SINGLE_WITNESS)
Attempt 2: AS99 announces 8.8.8.0/24 -> 1 vote  (INSUFFICIENT)
Attempt 3: AS99 announces 8.8.8.0/24 -> 2 votes (INSUFFICIENT)
Pattern: [0, 1, 2] -> ESCALATION DETECTED!
```

This indicates the attacker is refining their technique (e.g., pre-poisoning knowledge bases). The next attempt might reach 3 votes (consensus threshold) and succeed.

Escalation Deep Dive: How Learning Attackers Work

Normal Attacker (No Learning):

```
Time 0s:   AS99 announces 192.168.1.0/24 -> 0 votes (SINGLE_WITNESS)
Time 60s:  AS99 announces 192.168.1.0/24 -> 0 votes (SINGLE_WITNESS)
Time 120s: AS99 announces 192.168.1.0/24 -> 0 votes (SINGLE_WITNESS)
Pattern: [0, 0, 0] <- No escalation (static attack, easy to block)
```

Learning Attacker (Dangerous):

```
Time 0s:   Attacker tests basic hijack              -> 0 votes
Time 60s:  Attacker adds fake AS-PATH               -> 1 vote
Time 120s: Attacker forges ROA validation timing     -> 2 votes
Time 180s: Attacker perfects all parameters          -> 3 votes -> CONFIRMED!
Pattern: [0, 1, 2, 3] <- Attack succeeded by learning!
```

How the attacker learns:

1. **Trial and Error** – Send announcement, observe rejection, adjust parameters
2. **Feedback Analysis** – Analyze which nodes approved (visible on blockchain), tailor next attempt
3. **Timing Optimization** – First attempt at random time (rejected), next during legitimate BGP update window (1 vote), next exactly matching traffic patterns (2 votes)
4. **AS Path Manipulation** – Simple direct path (0 votes), add intermediate ASes (1 vote), use realistic path matching network topology (2 votes)

Detection thresholds:

Pattern	Meaning	Threat Level
`[0, 0, 0, 0]`	Static attack, not learning	Low
`[1, 0, 2, 1]`	Random variation	Low
`[0, 1, 2]`	Attacker learning	**HIGH**
`[0, 1, 2, 3]`	Attack succeeded	**CRITICAL**
`[2, 1, 0]`	Defenses improving	Good

Mitigation strategies:

1. **Blacklist escalating attackers** – Reject all future announcements from (prefix, ASN) pairs showing escalation
2. **Increase consensus threshold** – If vote count increasing, require 5 votes instead of 3
3. **Rate limiting** – Limit attempts per (prefix, ASN) pair with cooldown periods
4. **Alert for manual review** – Flag escalating patterns for human analysis

Code reference: `analysis/targeted_attack_analyzer.py` – `_detect_escalation_patterns()` method groups transactions by (prefix, ASN), sorts chronologically, and checks if `approve_count` increases over time.

Back to Table of Contents

Chapter 4: Attack Detection

Four Attack Types

1. PREFIX_HIJACK (Severity: HIGH)

An AS announces a prefix it does not own (origin AS doesn't match ROA database).

```
Legitimate: AS15169 announces 8.8.8.0/24  (ROA: 8.8.8.0/24 -> AS15169) [VALID]
Hijack:     AS99999 announces 8.8.8.0/24  (ROA: 8.8.8.0/24 -> AS15169) [INVALID]
```

Detection: Compare announced origin AS against Validated ROA Payload (VRP) table.

2. SUBPREFIX_HIJACK (Severity: HIGH)

An AS announces a more-specific subnet of a legitimate prefix with a different origin.

```
Legitimate: AS15169 announces 8.8.0.0/16
Hijack:     AS99999 announces 8.8.8.0/24  (more specific, different origin)
```

Detection: Check if announced prefix is a subnet of any VRP entry with a different origin AS.

3. BOGON_INJECTION (Severity: CRITICAL)

An AS announces a reserved/private IP range that should never appear in BGP.

```
RFC 1918:  10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16
RFC 5737:  192.0.2.0/24, 198.51.100.0/24, 203.0.113.0/24
RFC 6598:  100.64.0.0/10
```

Detection: Check announced prefix against reserved range list.

4. ROUTE_FLAPPING (Severity: MEDIUM)

Same (prefix, origin) announced and withdrawn rapidly, causing routing instability.

```
Parameters:
  FLAP_WINDOW_SECONDS = 60     (sliding window)
  FLAP_THRESHOLD = 5           (state changes to trigger)
  FLAP_DEDUP_SECONDS = 2       (minimum interval between events)
```

Detection: Count unique state changes for (prefix, origin) in sliding window.

Attack Consensus (Majority Voting)

When a node detects an attack, it proposes a vote to all RPKI validators:

```
Node detects attack -> Propose attack vote to all peers
                       |
                       v
                Each peer checks own observations
                       |
                       v
                Vote "attack_confirmed" or "false_alarm"
                       |
                       v
                3+ votes -> CONFIRMED ATTACK
                < 3 votes -> NOT CONFIRMED
```

Rewards/Penalties:

Action	BGPCoin
Detecting a confirmed attack	+10
Correct vote on attack	+2
False accusation	-20

Attack Verdicts on the Blockchain

Attack verdicts are stored on the **real blockchain** as hash-chained, immutable blocks – the same way BGP transaction blocks are stored. Each verdict block has `block_type: "attack_verdict"` in its metadata, distinguishing it from regular `"transaction"` blocks.

What gets stored on-chain:

- `verdict_id` (used as the block's `transaction_id` for deduplication)
- `record_type: "attack_verdict"` (tags the record so the BGP stream logger skips it)
- Full vote breakdown (yes/no counts, per-voter details)
- Attack type, attacker AS, confidence score

Why on-chain?

Verdicts are the most security-critical records in the system. Storing them in a flat JSONL file would make them trivially editable – an attacker could delete evidence of their hijack being confirmed. On the blockchain, verdicts are protected by SHA-256 hash chaining, Merkle roots, and cross-node replication, just like BGP transactions.

Secondary index: The `attack_verdicts.jsonl` file is still written as a best-effort secondary index for quick queries, but the blockchain is the source of truth.

Integrity: `verify_blockchain_integrity()` validates the hash chain across all block types (genesis, transaction, batch, and attack_verdict) uniformly.

Back to Table of Contents

Chapter 5: Token Economy (BGPCoin)

Overview

BGPCoin is the native token that incentivizes honest participation in the BGP-Sentry network.

Parameter	Value
Total Supply	10,000,000 BGPCOIN
Initial Treasury	10,000,000 (all tokens start here)
Distribution	Rewards drain treasury over time

Reward Structure

Action	Reward	Who Receives
Block commit	10 BGPCOIN	Merger (committer)
Approve vote	1 BGPCOIN	Each signer who voted approve
First-to-commit bonus	5 BGPCOIN	First node to commit a tx
Attack detection	100 BGPCOIN	Node that detected the attack
Daily monitoring	10 BGPCOIN	Active monitoring nodes

Penalty Structure

Violation	Penalty
False reject vote	-2 BGPCOIN
False approve vote	-5 BGPCOIN
Missed participation	-1 BGPCOIN

Multiplier System

Base rewards are scaled by three multipliers based on node history:

Multiplier	Range	Based On
Accuracy	0.5x - 1.5x	Historical vote correctness
Participation	0.8x - 1.2x	Consistency of participation
Quality	0.9x - 1.3x	Evidence quality in proposals

Back to Table of Contents

Chapter 6: Non-RPKI Trust Rating System

Score Range

Score	Classification	Meaning
90-100	Highly Trusted	Consistently good behavior
70-89	Trusted	Generally reliable
50-69	Neutral	Default starting position
30-49	Suspicious	Some concerning activity
0-29	Malicious	Repeated attack involvement

Initial score: 50 (Neutral)

Rating Changes

Penalties:

Event	Penalty
PREFIX_HIJACK involvement	-20
SUBPREFIX_HIJACK involvement	-18
BOGON_INJECTION involvement	-25
ROUTE_FLAPPING involvement	-10
ROUTE_LEAK involvement	-15
Repeated attack (within 30 days)	-30
Persistent attacker (3+ attacks)	-50

Rewards:

Event	Reward
Monthly good behavior	+5
False accusation cleared	+2
Per 100 legitimate announcements	+1
Highly trusted bonus (90+ for 3 months)	+10

Back to Table of Contents

Chapter 7: Performance Optimizations

The Journey: 383 Seconds Lag to Zero

BGP-Sentry was initially **not real-time viable**. Nodes fell 6+ minutes behind the BGP clock. Through 15 systematic optimizations across 5 phases, the system now processes BGP announcements with **zero lag** and achieves **36.8 TPS peak**.

Optimization Timeline

```
Baseline                Phase 1          Phase 2          Phase 3
383s lag --------------> 60s lag --------> 4s lag ---------> 0s lag
~4 TPS (lagging)        ~8 TPS           ~16 TPS           ~25 TPS
```

```
Phase 4                  Phase 5
0s lag ----------------> 0s lag
~32 TPS                    ~36.8 TPS (peak)
```

Phase 1: Consensus Pipeline (Optimizations 1-3)

Optimization 1: Consensus Timeout Reduction

Parameter	Before	After
Regular timeout	30s	3s
Attack timeout	60s	5s

Transactions that fail consensus no longer block for 30-60 seconds.

Optimization 2: Timeout Check Frequency

Parameter	Before	After
Initial wait	10s	1s
Check interval	10s	0.5s

Later superseded by event-based scheduling (Optimization 11).

Optimization 3: Async Attack Detection

Attack detection moved to background thread. Commit path no longer waits for detector + consensus voting.

Phase 2: Async Communication (Optimizations 4-6)

Optimization 4: Async Message Bus Delivery

```
BEFORE: Node A sends to B, C, D, E, F sequentially
  A -> B (5ms) -> C (5ms) -> D (5ms) -> E (5ms) -> F (5ms) = 25ms

AFTER: Node A submits all 5 to thread pool concurrently
  A -> [B, C, D, E, F] via ThreadPool = 5ms total
```

Optimization 5: Reduced Broadcast Peers (10 -> 5)

Consensus needs 3 signatures. Broadcasting to 5 gives 66% headroom while halving message volume.

Optimization 6: Async Block Replication

Block replication to N-1 peers runs in a background daemon thread.

Phase 3: Buffer & Dedup (Optimizations 7-9)

Optimization 7: Probabilistic Buffer Sampling

```
Buffer fill:    0%--------60%===========100%
Drop chance:    0%         0%--ramp--->  100%
```

Attacks **always bypass** the buffer.

Optimization 8: Ed25519 Signatures (replaced RSA-2048)

Operation	RSA-2048	Ed25519	Speedup
Key gen	~50ms	~0.05ms	1000x
Sign	~1ms	~0.05ms	20x
Key size	256 bytes	32 bytes	8x smaller

Optimization 9: Early-Skip Deduplication

```
BEFORE: Dedup at Step 4 (after wasting work on Steps 1-3)
AFTER:  Dedup at Step 0 (first thing, before any work)
```

Skip windows: RPKI = 5 min, Non-RPKI = 2 min. Attacks **never** skipped.

Phase 4: Crypto & Threading (Optimizations 10-12)

Optimization 10: Scaled Thread Pool (16 -> 48+ workers)

```
58 RPKI nodes x 5 vote requests = 290 concurrent messages
16 workers: 290/16 = 18 batches queued (bottleneck!)
48 workers: 290/48 = 6 batches (3x less waiting)
```

Pool size: `max(48, cpu_count * 2)` – adapts to hardware.

Optimization 11: Event-Based Vote Collection

```
BEFORE (polling):
  while running:
      sleep(0.5)              <- 116 wakeups/sec across 58 nodes
      scan ALL pending txs    <- wasted when nothing is due

AFTER (event-driven):
  while running:
      soonest = earliest_timeout - now
      event.wait(timeout=soonest)  <- wakes exactly when needed
```

Benefits: zero wakeups when queue empty, instant response on new transactions.

Optimization 12: Interruptible Background Threads

All `time.sleep()` replaced with `threading.Event.wait()`. On shutdown, the event is signaled and all threads exit immediately (instead of sleeping up to 3600 seconds).

Phase 5: Pipelining & I/O (Optimizations 13-15)

Optimization 13: Pipelined Observation Processing

```
BEFORE (sequential):
  Obs 1 -> broadcast -> WAIT for consensus -> commit -> Obs 2
           |<------ 30-100ms blocking -------->|

AFTER (pipelined):
  Obs 1 -> broadcast (background) -> Obs 2 -> broadcast -> Obs 3
  |<- 1ms ->|                       |<- 1ms ->|
               v consensus async       v consensus async
```

Optimization 14: Lock-Free Blockchain Writes

```
BEFORE: Lock held during in-memory update + ALL file I/O
  with lock:
      update chain       (~0.1ms)
```

```
      write JSON to disk (~5-50ms)  <- BLOCKING other nodes!

AFTER: Lock held ONLY for in-memory update
  with lock:
      update chain       (~0.1ms)
  # Lock released -- file I/O runs without blocking
  write JSON to disk
```

Optimization 15: Batched State Mapping Writes

```
BEFORE: Every transaction -> read file + add entry + write file
  7,000 txs x 7ms each = 49 seconds of I/O!

AFTER: Accumulate 50 updates in memory, then flush once
  140 flushes x 7ms = ~1 second of I/O (49x reduction)
```

Back to Table of Contents

Chapter 8: Throughput Analysis

Benchmark Results

Tested by increasing `SIMULATION_SPEED_MULTIPLIER` to push BGP data faster than real-time.

Speed	Wall Time (s)	Network TPS	Per-Node TPS	Precision	Recall	F1
1x	~1,700	4.2	0.04	1.000	1.000	1.000
2x	869	8.1	0.08	1.000	1.000	1.000
3x	580	12.2	0.12	1.000	1.000	1.000
4x	439	16.1	0.16	1.000	1.000	1.000
5x	350	20.2	0.20	1.000	1.000	1.000
6x	298	23.7	0.24	1.000	1.000	1.000
7x	254	27.8	0.28	1.000	1.000	1.000
8x	228	31.0	0.31	1.000	1.000	1.000
9x	199	35.5	0.35	1.000	1.000	1.000
10x	**192**	**36.8**	**0.37**	**1.000**	**1.000**	**1.000**

Key findings:

- **Peak: 36.8 network TPS** at 10x speed
- **Perfect F1 = 1.0** at all speeds (attacks never dropped)
- **Linear scaling 1x-6x**, sub-linear 7x-10x

What is TPS?

TPS (Transactions Per Second) is the universal blockchain performance metric. It measures the total number of transactions the entire network finalizes per second – not per node.

Comparison with Major Blockchains

Blockchain	Consensus	TPS (Actual)
Bitcoin	Proof of Work	~7
Ethereum (PoW)	Proof of Work	~15
Ethereum (PoS)	Proof of Stake	~15-30

Blockchain	Consensus	TPS (Actual)
BGP-Sentry	**Proof of Population (PoP)**	**36.8**
Solana	Proof of History	~830
Hyperledger Fabric	Practical BFT (PBFT)	~3,500

BGP-Sentry's 36.8 TPS is:

- **5x higher** than Bitcoin (7 TPS)
- **~1.5x higher** than Ethereum PoS (15-30 TPS)
- Below enterprise blockchains (different security model)

Why 36.8 TPS is Sufficient

A single BGP router observed **722,489 IPv4 updates + 270,380 IPv6 updates = ~993,000 total updates** in a 24-hour period, averaging ~11.4 updates/second [Huston 2025a]. An earlier measurement from a single vantage point in AS131072 showed ~200,000 IPv4 updates/day (~2.3/second) [Huston 2025b]. BGP-Sentry at 36.8 TPS handles **3-16x** the rate observed at a single peer.

Each node sees only a fraction of global announcements (based on AS topology position). Research has shown that ~13% of BGP updates are duplicates on average, rising to as high as 86% during peak instability periods [Park & Jen, PAM 2010]. BGP-Sentry's time-window deduplication (configurable via `RPKI_DEDUP_WINDOW` and `NONRPKI_DEDUP_WINDOW` in `.env`) further eliminates redundant observations within the same (prefix, origin) pair. In practice, even 4.2 TPS (1x real-time) is more than enough.

References for this section:

- [**Huston 2025a**] G. Huston, "A Day in the Life of BGP," APNIC Blog, June 2025. https://blog.apnic.net/2025/06/09/a-day-in-the-life-of-bgp/

- [**Huston 2025b**] G. Huston, "BGP Updates in 2024," APNIC Blog, January 2025. https://blog.apnic.net/2025/01/07/bgp-updates-in-2024/

- [**Park & Jen, PAM 2010**] J. H. Park and D. Jen, "Investigating Occurrence of Duplicate Updates in BGP Announcements," *Proc. Passive and Active Measurement (PAM)*, Springer LNCS 6032, pp. 11-20, 2010. https://link.springer.com/chapter/10.1007/978-3-642-12334-4_2

Bottleneck Analysis

The per-transaction consensus pipeline has these costs:

Step	Time	Bottleneck?
Transaction creation	<0.1ms	No
P2P broadcast to 5 peers	~1ms	No (async)
Knowledge base lookup (per voter)	~0.1ms	No
Ed25519 sign vote (per voter)	~0.05ms	No
Vote collection (wait for 3+ responses)	**2-5ms**	**YES**
Block commit + Merkle root	~0.5ms	No
Block replication	~1ms	No (async)

The bottleneck is **vote collection**: the merger must wait for at least 3 of 5 peers to respond. When 58 nodes broadcast simultaneously, the thread pool becomes contended.

Hardware Impact

Resource	Effect on TPS
More CPU cores	More thread pool workers -> less contention -> higher TPS
More RAM	Not a bottleneck (system uses ~3 GB)
Faster disk	Marginal (most operations in-memory)
GPU	No benefit (bottleneck is coordination, not computation)
Network (physical deployment)	Adds real latency to vote delivery

Future Scaling Strategies

Strategy 1: Transaction Batching (IMPLEMENTED) **Status:** Implemented and configurable via `.env`

How it works: Instead of writing each consensus-approved transaction as its own block (one `_save_blockchain()` call per transaction), the system accumulates multiple transactions in a queue and flushes them as a single multi-transaction batch block.

Configuration:

Parameter	Default	Description
`BATCH_SIZE`	1	Transactions per batch block. 1 = no batching (original). 10 = up to 10 tx per block.
`BATCH_TIMEOUT`	0.5s	Maximum wait before flushing a partial batch. Prevents stale transactions.

Mechanism:

```
BATCH_SIZE=1 (default, no batching):
  Tx1 approved → write block → replicate → Tx2 approved → write block → replicate
  4 blocks = 4 × _save_blockchain() = 4 × disk I/O

BATCH_SIZE=4:
  Tx1 approved → queue → Tx2 approved → queue → Tx3 approved → queue →
  Tx4 approved → queue full → flush → write ONE batch block → replicate once
  1 block = 1 × _save_blockchain() = 1 × disk I/O (4x reduction)
```

Expected gain: 3-5x TPS at `BATCH_SIZE=10`. The gain comes from: - Fewer `_save_blockchain()` calls (main I/O bottleneck) - Fewer block replication messages (one batch block vs N individual blocks) - Merkle root computed once over N transactions instead of N times over 1

Trade-off: Batching adds up to `BATCH_TIMEOUT` seconds of latency before a transaction appears on the blockchain. For BGP security monitoring (not real-time trading), 0.5 seconds is acceptable. Attack verdicts still go through `add_transaction_to_blockchain()` individually (not batched) for immediate recording.

Code references: - `p2p_transaction_pool.py`: `_batch_flush_loop()`, `_flush_batch()` - `blockchain_interface.py`: `add_multiple_transactions()` - `.env`: `BATCH_SIZE`, `BATCH_TIMEOUT`

Strategy 2: Sharded Consensus **Status:** Future work (not yet implemented)

How it works: Partition RPKI validators into consensus shards based on the IP prefix space they are responsible for. Each shard handles consensus for a subset of prefixes independently.

Design:

```
Current (single consensus domain):
  All 206 RPKI nodes vote on ALL transactions
  58 nodes × 5 vote requests = 290 concurrent messages

Sharded (4 shards by prefix range):
  Shard A (0.0.0.0/2):    ~52 nodes vote on prefixes 0.x-63.x
  Shard B (64.0.0.0/2):   ~52 nodes vote on prefixes 64.x-127.x
  Shard C (128.0.0.0/2):  ~52 nodes vote on prefixes 128.x-191.x
  Shard D (192.0.0.0/2):  ~50 nodes vote on prefixes 192.x-255.x

  Each shard processes consensus independently → 4x parallelism
```

Expected gain: 2-4x TPS. Each shard has fewer nodes competing for the same lock, so contention drops proportionally.

Trade-off: Cross-shard attacks (hijacker in shard A targeting prefix in shard B) require inter-shard communication. Shard assignment based on prefix space must be deterministic so all nodes agree on which shard handles which prefix. Rebalancing when nodes join/leave adds complexity.

Implementation notes: - Hash prefix to shard ID: `shard = hash(prefix) % num_shards` - Each shard maintains its own blockchain (or sub-chain) - Cross-shard references use Merkle proofs

---

Strategy 3: Asyncio Coroutines **Status:** Future work (not yet implemented)

How it works: Replace the threading model (`threading.Thread` + `ThreadPoolExecutor`) with Python `asyncio` coroutines. This eliminates GIL contention and context-switch overhead.

Current threading model:

```
58 RPKI nodes × 1 processing thread each = 58 threads
Message bus: 48-thread pool for delivery
Vote collection: blocking Event.wait() in each thread
Total: ~110+ threads competing for GIL
```

Asyncio model:

```
Single event loop (or one per CPU core via uvloop)
58 RPKI nodes as coroutines
Message delivery as async function calls (zero-copy)
Vote collection: asyncio.wait_for() with cooperative scheduling
Total: 1 event loop, zero GIL contention
```

Expected gain: 2-3x TPS. The GIL is the hidden bottleneck — 110 threads all competing for the same Python GIL means most time is spent in context switches, not useful work.

Trade-off: Requires rewriting all thread-based code to async/await. The `InMemoryMessageBus`, `P2PTransactionPool`, `VirtualNode`, and `BlockchainInterface` all need async versions. This is a substantial refactoring effort.

Strategy 4: Physical Distribution **Status:** Architecture supports it (TCP sockets implemented but not used in simulation)

How it works: Deploy each RPKI validator on a separate physical machine (or VM/container), communicating via real TCP sockets instead of the `InMemoryMessageBus`.

Current simulation model:

```
Single machine: All 500 nodes share 1 CPU, 1 Python process
InMemoryMessageBus: ~0.01ms latency, 100% delivery
Bottleneck: GIL + thread pool contention
```

Physical deployment model:

```
500 machines: Each node gets dedicated CPU + memory
TCP sockets: ~1-10ms latency, real network reliability
Bottleneck: Network latency for vote collection
```

Expected gain: True parallelism — each node runs independently without GIL contention. The 48-thread bottleneck disappears because each machine runs its own thread pool.

Trade-off: Real network latency replaces zero-latency in-memory calls. Vote collection takes 1-10ms instead of 0.01ms. However, the total throughput is much higher because nodes genuinely process in parallel instead of competing for one GIL.

Implementation notes: - Set `use_memory_bus=False` in `NodeManager` - Configure `P2P_BASE_PORT` and deploy one node per machine - The existing TCP socket code in `P2PTransactionPool` handles this case

---

Summary

Strategy	Status	Expected Gain	Implementation Effort
Transaction batching	**Implemented**	**3-5x TPS**	Low (configurable via `.env`)
Sharded consensus	Future	2-4x TPS	Medium (prefix-based partitioning)
asyncio coroutines	Future	2-3x TPS	High (full async rewrite)
Physical distribution	Ready	True parallelism	Low (existing TCP code)

Back to Table of Contents

Chapter 9: Experimental Results

Datasets

Dataset	ASes	RPKI	Non-RPKI	Observations	Attack %
caida_100	100	58	42	7,069	4.7%
caida_200	200	101	99	15,038	3.2%
caida_500	500	206	294	38,499	6.1%
caida_1000	1000	366	634	80,600	4.6%

Detection Accuracy

Metric	caida_100	caida_200	caida_500
Precision	0.068	0.095	0.070
Recall	0.750	1.000	1.000
F1 Score	0.125	0.174	0.131

Note: Low precision is due to route flapping false positives (tunable via `FLAP_THRESHOLD`). When measured at higher speed multipliers with the optimized pipeline, F1 reaches 1.0 (the flapping detector's sliding window aligns better with the compressed timeline).

Blockchain Performance

Metric	caida_100	caida_200	caida_500
Blocks Written	4,581	5,164	5,474
Integrity	Valid	Valid	Valid
Consensus Commit Rate	86.6%	47.3%	33.0%

Commit rate decreases with network size due to consensus contention – this is expected behavior and configurable.

P2P Network

Metric	caida_100	caida_200	caida_500
Messages Sent	979,084	1,575,910	12,349,312
Delivery Rate	100%	~100%	~100%

Zero message loss across all experiments.

Back to Table of Contents

Chapter 10: Real-Time Monitoring Dashboard

A Flask-based dashboard runs at `http://localhost:5555` during experiments.

Features

1. **Countdown timer** – Elapsed, total, and remaining time
2. **Progress bars** – Simulation Clock, Average Node, Slowest Node
3. **Lag indicator** – Shows if nodes are keeping up with real-time
4. **Per-node health** – Individual RPKI node lag from clock
5. **TPS chart** – Throughput trend over time (Chart.js)
6. **Lag chart** – Historical lag to detect degradation
7. **RPKI node table** – Processed/total, TPS, buffer drops

Architecture

```
main_experiment.py
    |
    v
SimulationDashboard(node_manager, clock, port=5555)
    |
    +-- Flask HTTP server (background daemon thread)
    |   +-- GET /           -> HTML dashboard
    |   +-- GET /api/overview -> Summary JSON
    |   +-- GET /api/nodes    -> Per-node stats JSON
    |   +-- GET /api/clock    -> Clock state JSON
    |
```

```
    +-- Stat collector (polls NodeManager every 5s)
        +-- Saves monitoring_timeseries.json
```

Back to Table of Contents

Chapter 11: Configuration Reference

All 40+ hyperparameters are in `.env` at the project root.

Group A: Consensus & P2P Network

Parameter	Default	Unit	Description
CONSENSUS_MIN_SIGNATURES	3	count	PoP minimum signatures
CONSENSUS_CAP_SIGNATURES	5	count	Upper cap on signatures
P2P_REGULAR_TIMEOUT	3	seconds	Consensus timeout (regular)
P2P_ATTACK_TIMEOUT	5	seconds	Consensus timeout (attack)
P2P_MAX_BROADCAST_PEERS	5	count	Peers per broadcast
P2P_BASE_PORT	8000	port	TCP base port (if not using memory bus)

Group B: Deduplication & Skip Windows

Parameter	Default	Unit	Description
RPKI_DEDUP_WINDOW	300	seconds	RPKI skip window (5 min)
NONRPKI_DEDUP_WINDOW	120	seconds	Non-RPKI skip window (2 min)
SAMPLING_WINDOW_SECONDS	300	seconds	P2P pool sampling window

Group C: Knowledge Base

Parameter	Default	Unit	Description
KNOWLEDGE_WINDOW_SECONDS	480	seconds	How long observations stay (8 min)
KNOWLEDGE_CLEANUP_INTERVAL	60	seconds	GC interval

Group D: Buffer & Capacity Limits

Parameter	Default	Unit	Description
INGESTION_BUFFER_MAX_SIZE	1000	count	Per-node buffer cap
PENDING_VOTES_MAX_CAPACITY	5000	count	Max pending transactions
COMMITTED_TX_MAX_SIZE	50000	count	Max tracked committed IDs
KNOWLEDGE_BASE_MAX_SIZE	50000	count	Max observations per node
LAST_SEEN_CACHE_MAX_SIZE	100000	count	Max cache entries

Group E: Attack Detection

Parameter	Default	Unit	Description
FLAP_WINDOW_SECONDS	60	seconds	Sliding window for flapping
FLAP_THRESHOLD	5	count	State changes to trigger

Parameter	Default	Unit	Description
FLAP_DEDUP_SECONDS	2	seconds	Minimum event interval
ATTACK_CONSENSUS_MIN_VOTES	3	count	Min votes for attack verdict

Group F: BGPCoin Token Economy

Parameter	Default	Unit	Description
BGPCOIN_TOTAL_SUPPLY	10,000,000	coins	Total token supply
BGPCOIN_REWARD_BLOCK_COMMIT	10	coins	Block commit reward
BGPCOIN_REWARD_VOTE_APPROVE	1	coins	Approve vote reward
BGPCOIN_REWARD_ATTACK_DETECTION	100	coins	Attack detection reward

Group G: Non-RPKI Trust Rating

Parameter	Default	Unit	Description
RATING_INITIAL_SCORE	50	points	Starting score
RATING_PENALTY_PREFIX_HIJACK	-20	points	Hijack penalty
RATING_PENALTY_BOGON_INJECTION	-25	points	Bogon penalty
RATING_THRESHOLD_HIGHLY_TRUSTED	90	points	Highly trusted cutoff
RATING_THRESHOLD_SUSPICIOUS	30	points	Suspicious cutoff

Group H: Transaction Batching

Parameter	Default	Unit	Description
BATCH_SIZE	1	count	Transactions per batch block (1 = no batching)
BATCH_TIMEOUT	0.5	seconds	Max wait before flushing partial batch

Group I: Simulation Timing

Parameter	Default	Unit	Description
SIMULATION_SPEED_MULTIPLIER	1.0	multiplier	1.0 = real-time

Back to Table of Contents

Chapter 12: Results Format

Each experiment run produces these files in `results/<dataset>/<timestamp>/`:

File	Contents
summary.json	Aggregate dataset + node + performance summary
detection_results.json	Per-observation detection decisions
performance_metrics.json	Precision, recall, F1 vs ground truth
trust_scores.json	Per-AS trust scores and stats

File	Contents
`run_config.json`	System info (CPU, RAM) + configuration
`blockchain_stats.json`	Blocks, transactions, integrity check
`bgpcoin_economy.json`	Treasury, distributed, per-node balances
`nonrpki_ratings.json`	Trust rating per non-RPKI AS
`consensus_log.json`	Committed vs pending counts
`attack_verdicts.json`	Attack proposals, votes, verdicts
`dedup_stats.json`	Observations deduplicated/throttled
`message_bus_stats.json`	P2P sent, delivered, dropped
`crypto_summary.json`	Key algorithm, signature scheme
`README.md`	Human-readable summary with TPS metrics

Back to Table of Contents

Chapter 13: Appendix: Running the System

Quick Start

```
# Setup
python3 -m venv venv && source venv/bin/activate
pip install -r requirements.txt

# Run experiment
python3 main_experiment.py --dataset caida_100 --duration 1800

# Open dashboard
# http://localhost:5555

# View results
cat results/caida_100/*/README.md
```

Throughput Benchmark

```
python3 scripts/benchmark_throughput.py --dataset caida_100
```

Generate Plots

```
python3 scripts/plot_throughput.py
# Output: results/fig_tps_vs_speed.png
#         results/fig_scaling_efficiency.png
#         results/fig_wall_time.png
#         results/fig_consensus_overhead.png
#         results/fig_blockchain_comparison.png
```

Changing Speed

Edit `.env`:

```
SIMULATION_SPEED_MULTIPLIER=5.0   # 5x faster than real-time
```

Blockchain Explorer

Browse blocks, inspect attack verdicts, and verify chain integrity:

```
# Interactive mode
python3 analysis/blockchain_explorer.py results/caida_100/*/blockchain.json

# Non-interactive commands
python3 analysis/blockchain_explorer.py results/caida_100/*/blockchain.json --verify
python3 analysis/blockchain_explorer.py results/caida_100/*/blockchain.json --verdicts
python3 analysis/blockchain_explorer.py results/caida_100/*/blockchain.json --search-prefix "8.8.8.0/24
python3 analysis/blockchain_explorer.py results/caida_100/*/blockchain.json --search-as 15169
python3 analysis/blockchain_explorer.py results/caida_100/*/blockchain.json --types
```

Interactive commands: `list [N]`, `block <N>`, `verdicts`, `verdict <ID>`, `search prefix <P>`, `search as <ASN>`, `types`, `verify`, `export <file>`, `help`, `quit`.

Post-Hoc Forensic Analysis

Three standalone analysis modules query the blockchain offline:

```
# Blockchain forensics: attacker profiling, prefix history, audit trails
python3 analysis/blockchain_forensics.py results/caida_100/*/

# Targeted attack analysis: escalation detection, single-witness patterns
python3 analysis/targeted_attack_analyzer.py results/caida_100/*/

# Longitudinal analysis: trust trajectories, economy health
python3 analysis/posthoc_analysis.py results/caida_100/*/
```

All outputs are reproducible from blockchain data alone.

Back to Table of Contents

Chapter 14: Glossary

Term	Definition
AS	Autonomous System – a network under single administrative control
BGP	Border Gateway Protocol – the Internet's routing protocol
PoP	Proof of Population – BGP-Sentry's consensus protocol: one RPKI-verified node = one vote
RPKI	Resource Public Key Infrastructure – cryptographic system for route authorization
ROA	Route Origin Authorization – certificate linking prefix to authorized AS
VRP	Validated ROA Payload – the set of validated ROAs used for route checking
TPS	Transactions Per Second – standard blockchain throughput metric
Merger	RPKI node that creates a transaction and collects votes
Signer	RPKI node that votes (approve/no_knowledge/reject) on another node's transaction
Knowledge Base	Per-node memory of recently observed BGP announcements
Dedup	Deduplication – skipping duplicate legitimate announcements
Ed25519	Elliptic curve signature algorithm (fast, small keys)
Merkle Root	Hash tree root ensuring transaction integrity in a block
BGPCoin	Native token for incentivizing honest network participation
Sybil Attack	Creating multiple fake identities to gain disproportionate influence

Back to Table of Contents