Morphogenesis Protocol v5.0

Status: Production Ready **Date:** Feb 2026 **Target:** Single-Server GPU PIR (NVIDIA H100/H200/B200)

1. Overview

Morphogenesis is a **Private Information Retrieval (PIR)** protocol for Ethereum state. A client queries the balance, nonce, code, or storage of *any* address without revealing the target to the server.

Key Properties

- **Privacy:** Information-Theoretic (IT-PIR) via 2-server DPF.
- **Scale:** 2.15 Billion rows (full Mainnet accounts + storage).
- **Latency:** 32.1 ms query time on H100 (subtree kernel), 27.4 ms on B200.
- **Simplicity:** Runs on a single commodity GPU server.

Workspace

Crate	Role
`morphogen-core`	Core types: DeltaBuffer, EpochSnapshot, GlobalState, Cuckoo hashing
`morphogen-dpf`	DPF key trait and implementations (AES-based, fss-rs)
`morphogen-gpu-dpf`	GPU-accelerated DPF using ChaCha8 PRG (CUDA)
`morphogen-storage`	AlignedMatrix and ChunkedMatrix storage primitives
`morphogen-server`	Scan kernel, HTTP/WebSocket server, benchmarks
`morphogen-client`	PIR client with network layer, caching, and batch aggregation
`morphogen-rpc-adapter`	JSON-RPC proxy: private methods via PIR, passthrough to upstream
`reth-adapter`	Reth integration for mainnet snapshot ETL

2. Data Model

2.1 Cuckoo Hash Table

Parameter	Value	Notes
Domain	2^{32}	Covers 2.15B rows
Hash Functions	3	Keyed SipHash
Load Factor	85%	Random-walk insertion minimizes stash
Tag Key	8 bytes	`keccak(address || slot)[0..8]` for storage lookups

2.2 Row Schemas

Schema	Size	Layout	Use
Account Compact	32 B	`[Balance(16) | Nonce(8) | CodeID(4) | Pad(4)]`	Account state
Storage Optimized48	48 B	`[Value(32) | Tag(8) | Pad(8)]`	Storage slots

Cross-ref: `DATA_STRUCTURES.md`

2.3 DPF Key Types

Key	Size	Level	Use
`AesDpfKey`	25 B	Row	AES seed (16) + target (8) + correction (1)
`PageDpfKey`	~491 B (25-bit domain)	Page	fss-rs based, chunked evaluation
`ChaChaKey`	Variable	GPU	ChaCha8 PRG for CUDA fused kernel

Cross-ref: `CRYPTOGRAPHY.md`

3. Query Protocol

3.1 Parallel Cuckoo Addressing

To mitigate adaptive leakage, we employ **Parallel Retrieval**. For target account A with candidate indices h_1, h_2, h_3:

1. Client generates query batch $Q = \{k^{(1)}, k^{(2)}, k^{(3)}\}$.
2. Server executes all 3 queries in a single linear pass.
3. Client receives 3 payloads. Since the Cuckoo failure probability with stash size $s = 256$ is negligible, we treat lookup failure as operationally zero.

Random-Walk Cuckoo Insertion: 3-way Cuckoo hashing with random-walk insertion achieves 85% load factor (vs 50% with deterministic cycling). Each key hashes to 3 candidate positions using independent keyed hash functions. On collision, a random candidate (excluding the just-evicted position) is selected for displacement.

Stash Handling via Delta-PIR: Items that cannot be placed during construction go to a stash. Build-time: rehash with new seeds until stash is empty. Runtime: new accounts go to the Delta buffer at their h_1 position. Clients only need the epoch's hash seeds and table size.

3.2 Query Flow

1. **Addressing:** Client locally computes 3 indices $[h_1, h_2, h_3]$ using the Cuckoo seeds.
2. **DPF Generation:** Client generates DPF keys for these indices.
3. **Request:** Client sends keys to 2 non-colluding servers.
4. **Scan:** Servers scan the entire matrix using a fused AES+XOR kernel.
5. **Response:** Servers return 3 encrypted blocks.
6. **Reconstruction:** Client XORs the responses to recover the row.

3.3 Server API

HTTP Endpoints:

Method	Path	Description
GET	`/health`	Status, epoch_id, block_number
GET	`/epoch`	Epoch metadata (seeds, num_rows, block_number, state_root)
POST	`/query`	Row-level PIR (3 DPF keys -> 3 payloads)
POST	`/query/batch`	Batch PIR (up to `MAX_BATCH_SIZE` queries)
POST	`/query/page`	Page-level PIR (3 PageDPF keys -> 3 pages)

POST	`/query/page/gpu`	GPU page query (ChaChaKey)

WebSocket Endpoints:

Path	Description
`/ws/epoch`	Real-time epoch update stream
`/ws/query`	Single and batch queries over persistent connection

Constants: `MAX_BATCH_SIZE = 32`, request body limit 64 KB, `MAX_CONCURRENT_SCANS = 32`.

3.4 Code Resolution

The 32-byte PIR row is too small for contract bytecode, so we use a sidecar:

1. PIR query returns `Balance`, `Nonce`, and `CodeID`.
2. Client resolves `CodeID` -> `CodeHash` via a public Dictionary (HTTP Range Request).
3. Client fetches bytecode from a public Content Addressable Storage (CAS/CDN).

Cross-ref: *`design/CODE_SERVING.md`*

3.5 Client

Setting	Value
Cache capacity	4096 entries (LRU, epoch-invalidated)
Retries	2 attempts, 200 ms exponential backoff
Connect timeout	5 s
Request timeout	30 s
Batch size	32 (matches server `MAX_BATCH_SIZE`)

Batch queries are auto-chunked at 32 with cache-aware partitioning: cached entries are served locally, only misses go to the server.

4. RPC Adapter

4.1 Architecture

`morphogen-rpc-adapter` is a JSON-RPC proxy on `:8545`, designed as a drop-in replacement for standard Ethereum RPC providers. Compatible with MetaMask, Rabby, Frame, and any EIP-1193 wallet.

4.2 Method Classification

Category	Methods	Mechanism
Private (PIR)	`eth_getBalance`, `eth_getTransactionCount`, `eth_getCode`, `eth_getStorageAt`	DPF query to PIR servers
Private (EVM)	`eth_call`, `eth_estimateGas`	Local revm execution with `PirDatabase` backend
Private (Cache)	`eth_getLogs`, `eth_getTransactionByHash`, `eth_getTransactionReceipt`	Block cache (64 blocks), upstream fallback
Private (Filters)	`eth_newFilter`, `eth_newBlockFilter`, `eth_newPendingTransactionFilter`, `eth_uninstallFilter`, `eth_getFilterChanges`, `eth_getFilterLogs`	Local filter state from block cache
Relay	`eth_sendRawTransaction`	Flashbots Protect (configurable)
Passthrough	`eth_blockNumber`, `eth_chainId`, `eth_gasPrice`, `eth_getBlockByNumber`, `eth_getBlockByHash`, `eth_feeHistory`, `eth_maxPriorityFeePerGas`, `net_version`, `web3_clientVersion`, `eth_accounts`	Forwarded to upstream RPC
Dropped	`eth_getProof`, `eth_sign`, `eth_signTransaction`	Rejected with descriptive error

4.3 Access-List Prefetch

For `eth_call` and `eth_estimateGas`, the adapter uses EIP-2930 access lists to batch-prefetch state:

1. Parse the transaction's `accessList` (or generate one via a dry-run).
2. Batch PIR query all referenced accounts and storage slots.
3. Warm the local cache before EVM execution.

This reduces round-trips from O(state-accesses) to O(1) batched PIR queries.

4.4 Block Cache

Setting	Value
Cached blocks	64 (FIFO)
Poll interval	2 s
Filter expiry	5 min

The cache stores transactions, receipts, and logs for recent blocks. Reorg detection compares block hashes on each poll and invalidates from the divergence point.

5. Updates & Consistency (Delta-PIR)

5.1 Delta Buffer

The `DeltaBuffer` accumulates live state updates between epoch rotations.

Key API:

- `push(row_idx, diff)` — append an update.
- `snapshot_with_epoch()` — atomic read of epoch + entries (single lock).
- `drain_for_epoch(new_epoch)` — consume entries and advance epoch.

Consistent Scan (`scan_consistent`):

1. Read snapshot epoch e_1 from `GlobalState`.
2. Scan main matrix.
3. Scan delta buffer (returns epoch + entries atomically).
4. Verify delta epoch matches e_1; if not, retry.
5. XOR matrix result with delta result.

Backoff: spin (attempts 0-9) -> yield (10-49) -> sleep (50+), max 1000 retries.

5.2 Epoch Management

`GlobalState` provides atomic epoch transitions via `ArcSwap`:

- `load() / store()` — current `EpochSnapshot` (matrix + metadata).
- `load_pending() / store_pending()` — pending `DeltaBuffer`.
- `try_acquire_manager()` — single-writer lock for epoch advancement.

Merge: A background worker constructs M_{e+1} using **Striped Copy-on-Write**. Only memory stripes affected by updates are duplicated; unmodified stripes are shared by reference (zero-copy). The global epoch pointer advances atomically.

5.3 Major Epochs (Seed Rotation)

- **Frequency:** Daily/Weekly.

- **Pipeline:** `reth-adapter` ETL -> R2 upload -> server hot-swap.
- **Purpose:** Rotate Cuckoo seeds to prevent long-term statistical leakage.

Cross-ref: *design/ROTATION.md*

6. Scan Engines

6.1 GPU (CUDA)

- Fused AES+XOR kernel on H100 VRAM (68.8 GB).
- **Throughput:** 1,300 GB/s (raw), 2,143 GB/s (subtree-optimized).
- **Latency:** 32.1 ms (subtree), 53 ms (raw).
- 4KB-aligned paged storage, 128-bit vector loads.

6.2 CPU (AVX-512)

- 8-row unrolled VAES with rayon parallelism.
- Portable fallback for non-AVX-512 hosts.
- Page DPF: chunked evaluation (`OPTIMAL_DPF_CHUNK_SIZE = 65536`).

Cross-ref: *PERFORMANCE.md, PROFILING_GUIDE.md*

7. Security

7.1 Trust Model

2-Server Semi-Honest: The two servers do not collude and follow the protocol. Privacy is Information-Theoretic (IT-PIR). Integrity is trusted (Privacy-Only mode).

7.2 Privacy Argument

Theorem: *The view of Server S is computationally indistinguishable for any two targets α, β.*

- **Transcript:** By DPF pseudorandomness, each key $k^{(j)}$ is indistinguishable from random.
- **Timing:** The scan executes a fixed number of operations $N_{ops} = |M| + |\Delta_{max}|$ regardless of target indices.
- **Access Pattern:** The client *always* queries $\{h_1, h_2, h_3\}$, so the access pattern is deterministic.

7.3 Leakage Assessment

- **Retry Oracle:** Eliminated. Epoch-based consistency means clients never retry on failures.
- **Metadata:** The server learns only that the client is "live" (tracking the chain tip).

- **RPC Adapter:** Method routing minimizes upstream leakage. Private methods never touch the upstream RPC. Transactions relay through Flashbots Protect.

7.4 Verifiable PIR (Iceboxed)

Trustless mode with sumcheck/binius proofs is designed but not in production.

Cross-ref: `design/VERIFIABLE_PIR.md`

8. Performance

Hardware	VRAM	Throughput	Latency	Concurrent Clients (< 600 ms)
NVIDIA B200	192 GB	2,510 GB/s	27.4 ms	~21
NVIDIA H200	141 GB	2,235 GB/s	30.8 ms	~19
NVIDIA H100	80 GB	2,143 GB/s	32.1 ms	~18

Subtree-optimized kernel. H200/B200 extrapolated from synthetic benchmarks.

Cross-ref: `PERFORMANCE.md`

9. Why "Morphogenesis"

This name is a homage to **Alan Turing**, who is both the father of modern computing and the theoretical biologist who proposed the concept of *morphogenesis*—the biological process by which organisms develop their shape.

The metaphor operates on three levels:

9.1 The Morphogen Signal

In biology, a **morphogen** is a signaling molecule that diffuses from a source cell through tissue. Cells measure morphogen concentration; high concentration triggers differentiation into specific tissue types.

In our protocol, the **DPF key is the morphogen**. It "diffuses" through the entire database during the linear scan. Only the specific row where the DPF evaluates to 1—the "concentration peak"—differentiates (activates) and contributes its data to the response.

9.2 Turing Patterns (Reaction-Diffusion)

Turing's 1952 paper, *"The Chemical Basis of Morphogenesis,"* described how two interacting chemicals (an activator and an inhibitor) could spontaneously create complex patterns—spots, stripes—from random noise.

Our 2-server protocol exhibits the same structure: - **Server A** sees pure noise (the "activator" share) - **Server B** sees pure noise (the "inhibitor" share) - **The Magic:** When these two chaotic "chemical waves" interact via XOR at the client, they cancel perfectly everywhere *except* at the target, creating a stable "spot" of information from entropy.

9.3 Genesis: Creation of Form

Morpho- (shape/form) + *-genesis* (creation).

The protocol takes a formless, high-entropy "soup" of encrypted bits and extracts a single, structured **form**—the user's account—without any party observing the extraction.

Since Turing's contributions span both computation theory and biological pattern formation, naming a privacy-preserving protocol after his biological discovery is poetically fitting.

10. References

Document	Description
`DATA_STRUCTURES.md`	Row schemas, Cuckoo parameters
`CRYPTOGRAPHY.md`	Fused kernel, DPF internals
`CRYPTO_ANALYSIS.md`	ChaCha8 vs AES on GPUs
`PERFORMANCE.md`	Benchmark results and analysis
`PROFILING_GUIDE.md`	How to profile scan engines
`design/CODE_SERVING.md`	Bytecode sidecar architecture
`design/ROTATION.md`	Major epoch seed rotation
`design/VERIFIABLE_PIR.md`	Sumcheck/binius proof design
`archive/morphogenesis_paper.md`	Original academic paper
`archive/morphogenesis_EDD.md`	Original engineering design