

MORPHOGENESIS PIR: 2-Party DPF-PIR for Ethereum State

Igor Barinov
Ethereum Foundation
`igor.barinov@ethereum.org`

February 2026

Abstract

We present MORPHOGENESIS PIR, a 2-server Private Information Retrieval (PIR) protocol based on Distributed Point Functions (DPF) for the Ethereum state. The system allows a client to query the balance, nonce, bytecode, or storage of any Ethereum address without revealing the target to the server. A linearized Cuckoo hash table (2^{32} domain, 85% load factor, 2.15 billion rows) is stored in GPU VRAM and scanned via a fused AES \oplus XOR CUDA kernel achieving 2,143 GB/s throughput and 32.1 ms latency on NVIDIA H100. Epoch-Based Delta-PIR provides wait-free snapshot isolation for live updates with $O(1)$ amortized cost. An RPC adapter proxy implements 20+ Ethereum JSON-RPC methods privately, serving as a drop-in replacement for standard providers. Privacy is Information-Theoretic under the 2-server semi-honest model.

1 Overview

MORPHOGENESIS PIR 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:** Each party runs on a single commodity GPU server.

Workspace. The implementation is organized as a Rust workspace with eight crates (Table 1).

2 Mathematical Formulation

2.1 Database Model

Let N denote the number of rows in the database. Each row is an ℓ -bit vector. We model the database as:

$$D : [N] \rightarrow \{0, 1\}^\ell$$

where $[N] = \{0, 1, \dots, N - 1\}$. In the Compact schema, $\ell = 256$ (32 bytes); in the Storage Optimized48 schema, $\ell = 384$ (48 bytes).

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

Table 1: Workspace crate overview.

2.2 DPF Algebra

We use a Distributed Point Function (DPF) [2] for the unit point function $f_\alpha(x) = \mathbf{1}_{x=\alpha}$.

Definition 2.1 (Distributed Point Function). A DPF scheme with domain $[N]$ consists of:

- $\text{Gen}(1^\lambda, \alpha) \rightarrow (k_0, k_1)$: Generate key shares for target index $\alpha \in [N]$
- $\text{Eval}(k_b, x) \rightarrow \{0, 1\}$: Evaluate key share $b \in \{0, 1\}$ at index $x \in [N]$

satisfying:

- **Correctness:** $\forall x \in [N] : \text{Eval}(k_0, x) \oplus \text{Eval}(k_1, x) = \mathbf{1}_{x=\alpha}$
- **Security:** Each k_b is computationally indistinguishable from random, given only that share.

2.3 Server Accumulation

Each server $b \in \{0, 1\}$ computes the XOR-accumulation over all rows, masked by the DPF evaluation:

$$R_b = \bigoplus_{x=0}^{N-1} \left(D(x) \cdot \text{Eval}(k_b, x) \right)$$

where $D(x) \cdot \text{Eval}(k_b, x)$ denotes the ℓ -bit row $D(x)$ if $\text{Eval}(k_b, x) = 1$, and the zero vector otherwise.

The client reconstructs: $D(\alpha) = R_0 \oplus R_1$.

3 Data Model

3.1 Cuckoo Hash Table

The database is a flattened, linearized Cuckoo hash table [3] stored in GPU VRAM (Table 2).

3.2 Row Schemas

3.3 DPF Key Types

4 Query Protocol

4.1 Parallel Cuckoo Addressing

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

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	<code>keccak(address slot)[0..8]</code> for storage lookups

Table 2: Cuckoo hash table parameters.

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

Table 3: Row schemas.

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. **Result:** 78M accounts require only 92M rows (1.18 \times overhead) vs 156M rows (2 \times) with naive Cuckoo.

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

4.2 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 \oplus XOR kernel.
5. **Response:** Servers return 3 encrypted blocks.
6. **Reconstruction:** Client XORs the responses to recover the row.

4.3 Server API

HTTP Endpoints.

WebSocket Endpoints.

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

Table 4: DPF key types.

Method	Path	Description
GET	/health	Status, epoch_id, block_number
GET	/epoch	Epoch metadata (seeds, num_rows, 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)

Table 5: HTTP endpoints.

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

4.4 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).

4.5 Client Configuration

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

5 RPC Adapter

5.1 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.

5.2 Method Classification

Table 8 classifies how each JSON-RPC method is handled.

5.3 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).

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

Table 6: WebSocket endpoints.

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)

Table 7: Client configuration.

2. Batch PIR query all referenced accounts and storage slots.

3. Warm the local cache before EVM execution.

This reduces round-trips from $O(\text{state-accesses})$ to $O(1)$ batched PIR queries.

5.4 Block Cache

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

6 Updates & Consistency (Delta-PIR)

6.1 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. The `scan_consistent` procedure ensures atomicity:

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.

Category	Methods	Mechanism
Private (PIR)	<code>getBalance</code> , <code>getTransactionCount</code> , <code>getCode</code> , <code>getStorageAt</code>	DPF query to PIR servers
Private (EVM)	<code>eth_call</code> , <code>eth_estimateGas</code>	Local revm with <code>PirDatabase</code>
Private (Cache)	<code>eth_getLogs</code> , <code>getTransactionByHash</code> , <code>getTransactionReceipt</code>	Block cache (64 blocks)
Private (Filters)	<code>newFilter</code> , <code>newBlockFilter</code> , <code>newPendingTransactionFilter</code> , <code>uninstallFilter</code> , <code>getFilterChanges</code> , <code>getFilterLogs</code>	Local filter state
Relay	<code>eth_sendRawTransaction</code>	Flashbots Protect
Passthrough	<code>blockNumber</code> , <code>chainId</code> , <code>gasPrice</code> , <code>getBlockByNumber</code> , <code>getBlockByHash</code> , <code>feeHistory</code> , etc.	Forwarded to upstream
Dropped	<code>getProof</code> , <code>sign</code> , <code>signTransaction</code>	Rejected with error

Table 8: RPC method classification. All method names prefixed with `eth_` unless noted.

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

Table 9: Block cache configuration.

6.2 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.

6.3 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.

7 Scan Engines

7.1 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.

7.2 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`).

8 Security Analysis

8.1 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).

8.2 Privacy Theorem

Definition 8.1 (Server View). For a query targeting account α , server b 's view is:

$$\text{View}_b(\alpha) = (k_b^{(1)}, k_b^{(2)}, k_b^{(3)}, e, T)$$

where $k_b^{(j)}$ are the three DPF key shares (for Cuckoo candidates), e is the requested epoch, and T is timing metadata.

Theorem 8.2 (Query Privacy). *Under the DPF security assumption [1], for any two targets $\alpha, \beta \in [N]$:*

$$\text{View}_b(\alpha) \approx_c \text{View}_b(\beta)$$

where \approx_c denotes computational indistinguishability.

Proof. We show each component is indistinguishable:

1. **DPF Keys:** By DPF security, each $k_b^{(j)}$ is computationally indistinguishable from a uniformly random string of equal length.
2. **Timing T :** The scan executes exactly $N + \Delta_{\max}$ iterations regardless of α . Memory access is sequential; no target-dependent branching occurs.
3. **Access Pattern:** The client *always* queries $\{h_1, h_2, h_3\}$, so the access pattern is deterministic.

□

8.3 Leakage Assessment

- **Retry Oracle:** Eliminated. Consistency retries are handled server-side; clients may retry on transport errors, but retries do not depend on the query target.
- **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.
- **Collusion:** If both servers collude, they can XOR their DPF shares to recover the target: $\text{Eval}(k_0, x) \oplus \text{Eval}(k_1, x) = \mathbf{1}_{x=\alpha}$. The 2-server model assumes non-collusion.

8.4 Verifiable PIR (Iceboxed)

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

9 Performance

Hardware	VRAM	Throughput	Latency	Concurrent (< 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

Table 10: GPU scan performance (subtree-optimized kernel, 68.8 GB matrix). H200/B200 extrapolated from synthetic benchmarks.

10 Why “Morphogenesis”

The name honours Alan Turing’s 1952 paper “*The Chemical Basis of Morphogenesis*” [4], in which two interacting chemicals—an activator and an inhibitor—spontaneously create structured patterns from random noise. Our 2-server protocol mirrors this precisely: each server independently sees only pseudorandom noise (one “activator” share, one “inhibitor” share), yet when the client XORs the two responses, the noise cancels everywhere *except* at the target index, producing a single structured “spot” of information from entropy—*morpho-* (form) + *-genesis* (creation).

11 Conclusion

MORPHOGENESIS PIR bridges the gap between theoretical PIR and systems reality. By combining **Parallel Cuckoo Retrieval** (for privacy) with **Epoch-Based Delta-PIR** (for consistency) and a full-featured **RPC Adapter** (for wallet compatibility), we demonstrate private Ethereum state access at 32.1 ms latency with ~18 concurrent clients on a single H100 GPU.

References

- [1] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *EUROCRYPT*, 2015.
- [2] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *EUROCRYPT*, 2014.
- [3] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.
- [4] Alan M. Turing. The chemical basis of morphogenesis. In *Philosophical Transactions of the Royal Society B*, volume 237, pages 37–72, 1952.