Undine: Distributed Extendable Merkle Tree Storage with Private Retrieval and Efficient Reconfiguration

Nicolas Alhaddad, Harry Eldridge, Julia Guskind, Eran Tromer, and Mayank Varia
Boston University
nhaddad@bu.edu, harryme@bu.edu, guskinju@bu.edu,
tromer@bu.edu, varia@bu.edu

***Abstract*—Clients of transparency logs, public-key directories, and shielded blockchain pools require authenticated access to an ever-changing Merkle tree, without storing it in full or revealing which entry they open. We seek to distribute storage and query work across independent servers as the data grows and participants join, leave, change their shares, or misbehave.**

We present UNDINE, which combines a distributed Merkle tree encoding with private retrieval from independently maintained fragments. Maintainers can choose and efficiently adjust the size of their stored fragments, to offer a flexible tradeoff against the number of maintainer replies needed for authenticated data reconstruction by clients. Maintainers can also perform authenticated updates and appends, at a write cost of just twice the number of modified Merkle nodes.

UNDINE uses a linear encoding of the Merkle tree nodes to divide evenly the storage and retrieval load across maintainers. This is combined with a private information retrieval (PIR) protocol that operates directly over the linearly-encoded fragments. UNDINE avoids repeated storage and computation for nodes shared by Merkle paths across records, producing the same PIR replies with work over tree values linear in the filled prefix. Record sizes adapt as the prefix grows to balance query and reply sizes.

Clients verify against the Merkle root, without commitments to individual PIR databases. Computational query privacy holds against any coalition. UNDINE is also secure against selective-failure leakage (i.e., deductions from acceptance), up to the PIR backend's correctness error, given enough timely honest replies, fixed public timing, and no reply-dependent follow-up.

We evaluate UNDINELOG, our append-only implementation of UNDINE combined with YPIR. On a height-50 sparse Merkle tree with a 4 GiB prefix of nonempty 32-byte leaves and a reconstruction threshold of 64, a maintainer replies in 100 ms (single-core); a client can process 192 replies from maintainers and reconstruct a verified opening in 380 ms, even with 64 corrupted replies. Scaled up to 1 TiB and 256 threshold , a maintainer answers in 1.13 s. Existing shielded pools can use this service without consensus changes.

1. Introduction

A Merkle-tree opening lets a client verify an entry under a recent tree root: it contains the entry and the hashes beside its path to that root. Browsers use openings to check Certificate Transparency logs [1], messengers to check contact keys [2]–[4], and blockchain wallets to spend notes in shielded pools [5]. While a public Merkle root establishes a commitment to all the pertinent entries, in order to retrieve and authenticate a specific entry, that entry and its opening must be somehow stored or derived. A naive solution is via monolithic storage nodes (e.g., blockchain archival nodes), but this induces ever-growing centralized storage costs, and creates a grave privacy leak where such nodes learn of every entry access by clients.

Thus, such applications share the need to obtain fresh openings for an evolving dataset via a system that (i) *distributes storage and maintenance* as the dataset grows; (ii) *preserves availability* when servers disappear, refuse requests, or return incorrect answers; and (iii) *protects query privacy*, hiding which entry a client opens.

In this work we construct a service that achieves these goals. We start from the model introduced by BORG [6], in which independent parties, called *maintainers*, keep small fragments of a linear encoding of a public Merkle tree. Maintainers choose and advertise their own fragments, validate updates in an externally supplied common order, and can change their storage allocations at will. New maintainers obtain fragments from existing ones; clients choose enough compatible maintainers to tolerate departures and malicious replies.

UNDINE augments this model with private openings, and provides a simpler and more efficient encoding for the underlying data. This new encoding speeds up tree maintenance and public replies, and allows maintainers to keep their share of the data "balanced" even as they adjust the storage they provide: the size of the fragment they store is always inversely proportional to reconstruction threshold, i.e., the number of similarly-sized maintainers that clients need to contact in order to do retrievals. UNDINE supports appends and authenticated rewrites; our implementation and evaluation focus on its append-only specialization, UNDINELOG.

An encoding for faster updates and balanced storage. UNDINE provides distributed storage of a sparse Merkle tree

with a large fixed capacity. Its active leaves form a growing prefix, followed by empty leaves. We view of the leave values and internal hashes of the full tree as serialized into a virtual *array* of *cells*, one cell per node, ordered by preorder traversal. The active leaves and their ancestors form a prefix of this array, and we maintain only that prefix. Appends extend the stored prefix without changing existing cell indices. Empty subtrees therefore consume no stored cells, regardless of their size. When a Merkle opening requires a node whose index lies beyond the stored prefix, we supply the public default hash for that node's level.

Each maintainer holds a *fragment* of a linear encoding of the virtual array representing the tree. For a rewrite of a leaf, the old value's opening lets a maintainer validate the change and recompute the affected path hashes. Linearity of the encoding then lets it apply the resulting cell changes to its fragment without reconstructing the whole array.

The *reconstruction threshold* κ is the number of correct replies from distinct compatible maintainers (each holding a similarly sized fragment) that are needed to recover an opening. *Balanced storage* means that the fragment size required to support a reconstruction threshold of κ is about a $2/\kappa$ fraction of the the size of the whole tree (in its array representation), and this stays the case as fragment sizes vary.

UNDINE's encoding scheme splits the tree (in its array-of-cells representation) into groups of κ^2 cells, each arranged on a $\kappa \times \kappa$ grid. Each such grid is distributed among the maintainers in a coded form (specified later), such that each maintainer stores one coded column (used to serve clients' retrievals) and one coded row (used to help other maintainers obtain missing fragments). A changed cell alters just one stored entry in each coded column and row.

A client can do a retrieval by querying κ maintainers who store their fragment using grid parameter κ. For a non-private retrieval (i.e., one which names the retrieval cell in plain text), each such maintainer only needs to read and send back a single entry from the encoded column it stores. Notably, a maintainer storing a larger fragment at a supported lower threshold $\kappa' < \kappa$ can instead compute one compatible reply at threshold κ by combining stored entries, without changing its fragment. Thus, maintainers with different storage sizes can contribute to the same public opening, each supplying one of the κ required correct replies.

If a maintainer wishes to decrease amount of storage it allocates to fragments (e.g., as the number of groups grows and local storage runs out), it can halve the size of each fragment by doubling the κ parameter for that fragment. To do so, the maintainer implicitly *stitches* four adjacent groups into one $2\kappa \times 2\kappa$ grid, defined by combining the rows and columns (by a certain formula). The resulting encoded column and row, of size 2κ each, take just half the space of the 4 original encoded columns and rows of size κ (though retrieval will, of course, need correct replies from 2κ such maintainers instead of just κ). Crucially, this involves no communication and preserves balanced storage through repeated doublings at supported thresholds (apart from padding in the final group). The group size remains κ^2 at every threshold, so κ is the only coding parameter that maintainer need to announce when they advertise their fragment's availability.

By comparison, BORG's encoding updates a whole stored column for a changed cell. With the smallest balanced groups (κ^2 cells), doubling the threshold locally does not halve storage (loses balanced storage). Larger initial groups allow more local doublings that roughly halve storage, but make updates and public replies more expensive. Alternatively, a BORG maintainer can raise its threshold while keeping storage balanced by downloading about a fragment's worth of data from other maintainers.

Concretely, our implementation of UNDINELOG is indeed faster than BORGLOG (BORG's append-only specialization for every maintainer operation (Table 1)). Public retrievals retain the same sizes and client reconstruction. Furthermore, BORG did not support retrieval using maintainers with different κ values, as above.

Private Merkle openings from one server or many. Private information retrieval (PIR) retrieves a database record without revealing its index, whereas an opening needs an entry and hashes scattered along a Merkle path. One PIR lookup per tree level gives unequal workloads because the levels contain different numbers of nodes.

We first explore the retrieval problem for one server holding the whole uncoded tree. A logical record contains the values of a small subtree and the sibling hashes above it, so one query retrieves a complete opening. Materializing these records would duplicate shared tree nodes. A linear single-server PIR computes a sum of records weighted by the query. We regroup that sum by shared tree values, combining their query weights before multiplying each stored value. The server gives exactly the same PIR reply from one stored copy of the tree, with a scan linear in its filled size (Proposition 4). Threads can divide this scan evenly and combine their partial sums before the PIR backend forms the reply. This works with any such PIR and needs no separate reorganization of the tree; backend preprocessing remains additional state.

Building on this construction, we extend private retrieval to the multi-server setting, in which several maintainers hold fragments of a linear encoding of the Merkle tree. Work on coded-storage and robust PIR already studies distributed records and faulty replies [7]–[9]. Here private retrieval must also follow the model's verified updates and changes in membership and storage allocation. Answering directly from the maintained fragments divides storage and the linear scan across maintainers. The client sends the same private query to each selected maintainer, decodes the PIR replies, and applies the code's reconstruction. It authenticates the recovered opening against the shared Merkle root, without commitments to individual maintainers' PIR databases. The PIR scheme supplies computational query privacy against any coalition; the code supplies recovery from wrong and missing replies.

In UNDINE, primary columns alone form the PIR database, about a $1/\kappa$ share of the filled tree; auxiliary repair rows are not queried. Scanning only this fragment lets a maintainer serve larger logs at the same database and

scan cost by increasing κ, while requiring more replies for reconstruction. The selected maintainers share a root, stored or retained threshold, and record layout; serving another threshold requires its fragment and PIR preprocessing.

Larger records shorten queries but enlarge replies. We balance these costs as the database grows by choosing the number of leaves per record as a public function of the filled length. For UNDINE's coded replies, this function also accounts for κ, since each maintainer returns only a share of the subtree.

For coded fragments, recovery also prevents *selective failure*, in which corrupted records leak information about the queried position through client acceptance [10]. With UNDINE's code, the protocol contacts $\kappa + 2b + m$ maintainers, chosen independently of the requested position, to tolerate b wrong and m missing replies. The same additional maintainers protect every position; disjoint uncoded shards need separate replication. With enough timely honest replies, every position yields a valid opening, so acceptance provides no distinguishing signal except through the PIR backend's correctness error. This guarantee also requires fixed public timing and no reply-dependent follow-up (Theorem 3).

TreePIR [11] is the closest system for private Merkle openings, but its construction does not meet this maintenance model. It uses a balanced coloring to divide a perfect tree of height h into h databases of nearly equal size, with one node from every opening in each database. Servers must use an agreed coloring, which couples their storage allocations, and an opening needs a correct response from every color class. Our split is instead controlled by κ, independently of tree height. TreePIR lists sparse and growing trees as open problems; its growing-log options include padding or a forest of perfect trees, whose smaller trees have smaller anonymity sets. UNDINE directly serves a growing prefix of a sparse tree, with one anonymity set for all filled positions.

We also benchmark UNDINELOG against TreePIR over the same YPIR [12] backend at a matched operating point. Per-server PIR state, computation, and communication are comparable, while UNDINELOG has faster client query preparation: TreePIR prepares one query per color class, whereas our query is shared (Appendix C).

Application to shielded pools and benchmarks. In privacy-preserving blockchain protocols, such as Zcash [5] and Tornado Cash [13], the implementation of shielded pools uses an append-only commitment trees. UNDINELOG lets a returning wallet obtain a fresh private spending opening for a known note position without replaying missed commitments. Independent maintainers follow the existing public commitment stream; the service requires no changes to consensus rules, transaction formats, or the authenticated tree.

Zcash's Sapling and Orchard pools together held about 3.7 GiB of commitments at the end of September 2026 [14], motivating our 4 GiB workload. Orchard and Ironwood each permit at most 2^{32} commitments (128 GiB of leaf data) [5], [15]. We evaluate height-50 sparse trees with 32-byte leaves and capacity for 2^{50} commitments (32 PiB); the largest evaluated prefix contains 1 TiB (2^{35} commitments), all within one query anonymity set.

We use YPIR for both the single-server construction and UNDINELOG's private openings. Clients keep no database-dependent state; maintainers update PIR preprocessing after appends. On a 4 GiB filled prefix at $\kappa = 64$, measured primary PIR database storage and linear scan time per maintainer are each about $64\times$ smaller than for our single-server construction on the same prefix. Each maintainer also packs its scan result into encrypted replies, a YPIR step whose cost follows reply size and explains the smaller full-answer speedup. The evaluation measures complete per-party processing, including RPC encoding and parsing for our TCP implementation, and serialized message sizes. On this prefix, a maintainer answers in 100 ms; a client processes 192 replies and reconstructs a verified opening in 380 ms, even with 64 corrupted replies. Both timings use one core (Section 7).

Contributions.

1. *An encoding for faster updates and balanced storage.* UNDINE speeds up tree maintenance, public replies and adjustment to the κ coding parameter. Each maintainer can repeatedly double κ locally without communication while preserving balanced storage, and serve public requests at supported higher thresholds from its existing fragment.
2. *Balanced private retrieval over a Merkle tree.* A generic construction over linear single-server PIR retrieves complete openings from one stored tree, with a server-side optimization that avoids redundant storage and computation for shared path nodes. Its PIR work can be divided evenly, and record size adapts as the filled prefix of the sparse tree grows.
3. *Distributed PIR for growing trees and changing maintainers.* A generic extension to linear encodings of Merkle trees divides storage and scan work across maintainers as the tree grows and membership changes. Clients authenticate openings against the shared Merkle root without individual database commitments. The protocol provides computational query privacy against any coalition, with fault recovery supplied by the chosen code and selective-failure resistance under explicit fault and timing conditions.
4. *A practical implementation and evaluation of* UNDINELOG. We instantiate PIR with YPIR, evaluate tree maintenance, private openings, and client reconstruction under malicious replies, and show how the service can support existing shielded pools without consensus changes.

2. BORG Background and Model

UNDINE builds on BORG's maintenance and verification framework [6], replacing the encoding of the maintained array and adding private retrieval functionality.

What is maintained. The data vector is $x_t \in \mathbb{F}^{\ell_t}$ of length $\ell_t \le \mathsf{N} = 2^H$. It changes through update batches that an external sequencer orders and that every maintainer validates before applying. Its public *anchor* is $A_t = (\mathsf{Com}_t, \ell_t)$,

where Com_t is the root of a sparse Merkle tree of height H over x_t. The *maintained state* is an array $M_t \in \mathbb{F}^{L_t}$ of *cells*: the leaves of the *active prefix*, the positions below ℓ_t, and the labels of the internal nodes whose subtrees meet it. The order is the preorder of the tree of height H restricted to those nodes, so a node keeps its place when the prefix grows. Labels of empty subtrees are public defaults, so $L_t = O(\ell_t + \log \mathsf{N})$. An opening under A_t is assembled from the relevant cells of M_t and public default labels.

BORG encodes M_t into *fragments* held by independent *maintainers*. Each fragment is a public linear function of M_t. A maintainer stores one or more fragments and advertises each under a claim identifier (cID), which names the fragment, together with its own identifier, the anchor for which the fragment is current, and the fragment's parameters: in BORG the reconstruction threshold κ and segment size Seg. A *caller*, a client that opens positions or a maintainer that repairs or reconfigures a fragment, selects a collection of advertisements called its local *registry*. The caller sends one request to each selected maintainer; those that answer are its *helpers*. A maintainer answers a read with its share of the requested cells. The caller runs BORG's reconstruction procedure $\mathsf{Combine}$, which recovers the requested cells from κ distinct correct replies with the same anchor and threshold, and can correct errors given additional replies. It uses top-down Merkle checks to invoke error correction only when a check fails, excluding identified faulty helpers from the rest of the query. The caller accepts the resulting opening only if the Merkle verifier Verify accepts it under A_t. After an accepted batch, every maintainer updates its own fragment from the batch's induced change $M_{t+1} - (M_t, 0)$, with the old array zero-padded to the new length. Repair uses helper replies; reconfiguration uses local state or helper replies.

BORG's encoding. BORG cuts M_t into *segments* of Seg consecutive cells. A segment with cell values v_i at distinct public points α_i is encoded by a polynomial $F(X)$ of degree less than Seg satisfying $F(\alpha_i) = v_i$. These points form its *semantic grid*. At fixed segment contents and grid, F is independent of κ. A threshold is a *lens* on F: for $\mathsf{Seg} = \kappa s$, write $F(X) = P(X^s, X)$ with $P(Z, W)$ of degree below κ in Z and below s in W. In this view, the semantic grid consists of the points (α_i^s, α_i), where $P(\alpha_i^s, \alpha_i) = v_i$. The relation $Z = X^s$, $W = X$ is the *pin*. A cID determines a column identifier C and row labels w. For each segment, its maintainer stores the *primary column* $W \mapsto P(C, W)$ as s evaluations and, for repair, $\max\{1, \lceil \mathsf{Seg}/\kappa^2 \rceil\}$ *auxiliary rows* $Z \mapsto P(Z, w)$ of κ values each. These slices across all segments form its fragment. To read cell v_i, each helper evaluates its column at $W = \alpha_i$; the caller interpolates the replies in Z to recover $P(\alpha_i^s, \alpha_i)$. Repair of a primary column interpolates the helpers' auxiliary values in W. The balanced-storage condition is $\mathsf{Seg} \geq \kappa^2$: a fragment then holds $O(\mathsf{Seg}/\kappa)$ values per segment, giving a $2/\kappa$ share of the state apart from final-segment padding.

At fixed Seg, BORG changes the threshold by changing the lens and obtaining any missing target state from helpers. Alternatively, a local *stitch* merges two neighbouring segments and doubles both κ and Seg.

Assumptions. The sequencer gives all honest parties the same order of candidate batches; an honest maintainer advances from A_t only after it has validated the next batch, and every operation names the anchor it uses.

3. Bivariate Encoding for Balanced Storage

We first work with a fixed array: one segment for encoding and reads, and four segments for a stitch and its undo.

3.1. Why a product grid

BORG's stitch takes (Seg, κ) to $(2\mathsf{Seg}, 2\kappa)$ and halves the balance ratio, $2\mathsf{Seg}/(2\kappa)^2 = \frac{1}{2}\mathsf{Seg}/\kappa^2$: the primary column length $s = \mathsf{Seg}/\kappa$ stays fixed while each auxiliary row doubles in length. Starting at $\mathsf{Seg} = \kappa^2$, even one stitch violates $\mathsf{Seg} \geq \kappa^2$; repeating it loses the intended inverse-threshold storage scaling. Preserving $\mathsf{Seg} = \kappa^2$ when doubling κ instead requires merging four segments, taking (κ^2, κ) to $(4\kappa^2, 2\kappa)$.

UNDINE encodes a segment's κ^2 cells directly on a *product semantic grid* $G^Z \times G^W$, with κ distinct points on each axis. Cell (a, j) has value v_{aj} at the public point (z_a, α_j): the segment is the unique polynomial $P(Z, W)$ of degree below κ in each variable satisfying $P(z_a, \alpha_j) = v_{aj}$. The two coordinates are independent, with no pin relating them.

For a cID with public assignments (C, w) outside G^Z and G^W, respectively, the maintainer stores the *primary column* $P(C, W)$ as its κ evaluations on G^W and the *auxiliary row* $P(Z, w)$ as its κ evaluations on G^Z. Each set determines its polynomial. The maintainer thus stores 2κ values for the segment's κ^2 cells.

Cell updates. Let Λ_z^Z and Λ_α^W be the Lagrange basis polynomials of the two axes. Changing the cell at (z_a, α_j) by Δ adds $\Delta\Lambda_{z_a}^Z(Z)\Lambda_{\alpha_j}^W(W)$ to P. Only two stored entries change: the primary column's entry at α_j increases by $\Delta\Lambda_{z_a}^Z(C)$, and the auxiliary row's entry at z_a by $\Delta\Lambda_{\alpha_j}^W(w)$. In BORG, the corresponding univariate correction to F is nonzero at every stored off-grid primary point, so every primary sample of the touched segment must be updated.

3.2. Doubling κ locally: the stitch

Consider an array of $4\kappa^2$ cells split into four segments, labelled $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$. For one cID, a maintainer stores a primary column and an auxiliary row for each, a total of 8κ values (Figure 1(a)). The stitch combines them into one segment at threshold 2κ. It preserves every cell's value and field coordinates, while replacing the four columns and rows with one column and one row of 2κ entries each: 4κ stored values in total (Figure 1(b)).

Grids and stored state. The four grids are chosen compatibly before encoding. On the combined grid, segment

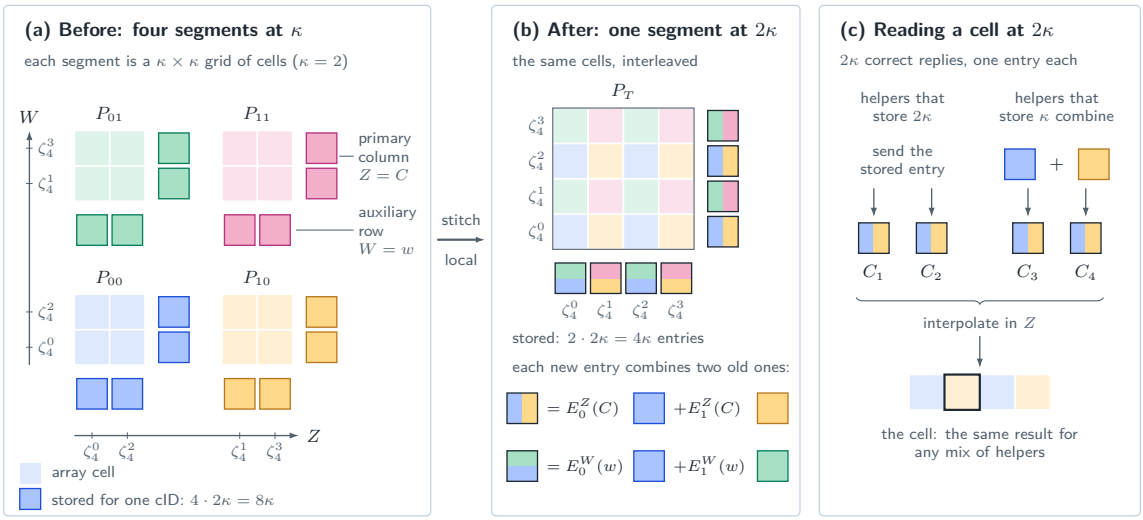

Figure 1. The stitch for $\kappa = 2$. (a) Pale squares are the cells of the four segments and solid squares the entries stored for one cID: each segment's primary column at $Z = C$ and auxiliary row at $W = w$. (b) The stitched segment holds the same cells, interleaved on both axes; each new stored entry combines two old entries with public weights, from the two segments whose colours it shows. (c) A read of one cell at 2κ: a helper storing 2κ sends its stored entry; one storing κ computes that entry from two stored ones. The caller interpolates 2κ correct replies in Z.

(μ, ν) occupies the even or odd positions of the Z-axis according to μ, and of the W-axis according to ν. For a primitive 2κ-th root of unity $\zeta_{2\kappa}$ and public scale factors $\rho_Z, \rho_W \in \mathbb{F}^\times$, the combined axes are $G_T^Z = \{\rho_Z \zeta_{2\kappa}^j : 0 \le j < 2\kappa\}$ and $G_T^W = \{\rho_W \zeta_{2\kappa}^j : 0 \le j < 2\kappa\}$, and each splits into an even half and an odd half,

$$\begin{aligned} G_\mu^Z &= \{\rho_Z \zeta_{2\kappa}^{2i+\mu} : 0 \le i < \kappa\}, \\ G_\nu^W &= \{\rho_W \zeta_{2\kappa}^{2i+\nu} : 0 \le i < \kappa\}, \end{aligned} \tag{1}$$

for $\mu, \nu \in \{0, 1\}$. Segment (μ, ν) occupies $G_\mu^Z \times G_\nu^W$, and its κ^2 cells determine a polynomial $P_{\mu\nu}$ of degree below κ in each variable. The maintainer uses the same assignments (C, w) for all four segments, with $C \notin G_T^Z$ and $w \notin G_T^W$. Its stored columns and rows are

$$L_{\mu\nu,C}(W) = P_{\mu\nu}(C, W), \qquad R_{\mu\nu,w}(Z) = P_{\mu\nu}(Z, w), \tag{2}$$

represented by their κ values on G_ν^W and G_μ^Z, respectively.

The stitch. Each new column entry combines entries from the two segments that share its W-coordinate; each new row entry combines entries from the two segments that share its Z-coordinate. To express the weights, define the public *selectors*

$$\begin{aligned} E_\mu^Z(Z) &= \tfrac{1}{2}\big(1 + (-1)^\mu (Z/\rho_Z)^\kappa\big), \\ E_\nu^W(W) &= \tfrac{1}{2}\big(1 + (-1)^\nu (W/\rho_W)^\kappa\big), \end{aligned} \tag{3}$$

of degree κ. On the combined semantic axes, each is 1 on its chosen half and 0 on the other. At the off-grid assignments C and w, they supply the combination weights. The stitched segment is

$$P_T(Z, W) = \sum_{\mu,\nu \in \{0,1\}} E_\mu^Z(Z) E_\nu^W(W) P_{\mu\nu}(Z, W), \tag{4}$$

with slices $L_{T,C}(W) = P_T(C, W)$ on G_T^W and $R_{T,w}(Z) = P_T(Z, w)$ on G_T^Z. Each summand has degree below 2κ in both variables. Under BORG's pin, the selectors depend on Z alone: applying its stitch twice merges four segments but also quadruples κ. The independent axes let UNDINE merge four while only doubling it.

Enumerate the four segments' grids by $z_{\mu,i} = \rho_Z \zeta_{2\kappa}^{2i+\mu}$ and $\alpha_{\nu,i} = \rho_W \zeta_{2\kappa}^{2i+\nu}$ for $0 \le i < \kappa$, the combined axes in increasing powers of $\zeta_{2\kappa}$, and write $L[k]$ and $R[k]$ for the k-th stored column and row entries.

Theorem 1 (Balanced local four-segment stitch)**.** *P_T is the unique polynomial of degree below 2κ in each variable that agrees with the four segments on their cells. A maintainer computes its column and row of P_T from its four stored columns and rows on the domains of* (1)*, with no communication and $O(\kappa)$ field operations:*

$$L_{T,C}[2i + \nu] = E_0^Z(C)\, L_{0\nu,C}[i] + E_1^Z(C)\, L_{1\nu,C}[i], \tag{5}$$

$$R_{T,w}[2i + \mu] = E_0^W(w)\, R_{\mu 0,w}[i] + E_1^W(w)\, R_{\mu 1,w}[i]. \tag{6}$$

Four segments at $(\mathsf{Seg}, \kappa) = (\kappa^2, \kappa)$ become one at $(4\kappa^2, 2\kappa)$.

Proof in Appendix A.

Updating then stitching equals stitching then updating: both produce the column and row of the unique polynomial encoding the updated cells on the combined grid.

3.3. Halving κ: undoing the stitch

Undoing the stitch returns from one segment at 2κ to the four segments at κ, keeping the same cells and assignments (C, w). The maintainer must recover its four columns and four rows, a total of 8κ entries, from the stitched column and

row of 4κ entries and additional information. Each stitched entry combines two old entries. The maintainer's columns and rows for segments $(0,0)$ and $(1,1)$ supply one entry of every pair, allowing it to solve for the other.

Proposition 1 (Undoing a stitch)**.** *Given a maintainer's stitched column and row and its columns and rows for segments* $(0,0)$ *and* $(1,1)$*, solving the stitch formulas* (5) *and* (6) *for each missing entry recovers its columns and rows for segments* $(1,0)$ *and* $(0,1)$ *using* $O(\kappa)$ *field operations.*

Proof in Appendix A.

Retaining these 4κ values when stitching makes the undo local at the cost of storing them; otherwise the maintainer repairs them (Section 3.4).

3.4. Reads and repair

For one segment, reads and repair use BORG's two interpolation directions [6]. To read a cell at (z_a, α_j), helper i sends its stored column entry $P(C_i, \alpha_j)$, with no arithmetic. From κ correct replies, the caller interpolates $P(Z, \alpha_j)$ and evaluates it at z_a, or at every point of G^Z to read all cells sharing α_j. In BORG, the helper instead interpolates its column to compute the read share. Symmetrically, a read with $Z = z_a$ fixed uses the helpers' stored row entries $P(z_a, w_i)$ to recover $P(z_a, W)$ and all cells sharing z_a.

To repair a primary column $P(C, W)$, each helper evaluates its auxiliary row at C, using $O(\kappa)$ field operations. The caller interpolates the replies in W and evaluates the recovered polynomial on G^W. Repair of an auxiliary row $P(Z, w)$ similarly evaluates helpers' columns at w and interpolates in Z.

Proposition 2 (Reads and repair)**.** *Let the selected cIDs have distinct column identifiers and distinct row labels, and let their number be at least* $\kappa + 2b + m$*. With at most* b *corrupted maintainers and* m *further missing ones, the point read, the fixed-*Z *read, and both repairs above return the correct output.*

Proof in Appendix A.

Repair also supplies the additional columns and rows needed to undo a stitch. Repairing those for segments $(0,0)$ and $(1,1)$ under the maintainer's own cID downloads 4κ field elements from κ correct helpers, compared with 8κ for all four segments. An alternative procedure uses 2κ correct helpers storing the stitched encoding, with the same 4κ download (Appendix A). Fault tolerance uses additional replies as above.

4. UNDINE

UNDINE applies the encoding of Section 3 to the extendable array M_t of active tree nodes from Section 2. It is BORG [6] with this encoding and a simpler model: $\mathsf{Seg} = \kappa^2$ at every threshold, including after a local stitch, so the advertisement's only coding parameter is κ.

Public layout and fragments. Setup fixes supported thresholds $\kappa_j = 2^j \kappa_{\mathsf{base}}$ up to a largest level. At threshold κ, M_t is divided into $\lceil L_t/\kappa^2 \rceil$ consecutive segments. Applying BORG's layout to each grid axis assigns each aligned group of four segments the compatible grids of (1); every cell keeps its field coordinates across levels. Unused offsets are zero, so absent segments have zero columns and rows and the stitch also covers a partially filled group. Appendix A gives the public mapping from array offsets to grid points.

A cID fixes $C = \mathsf{Column}_{\mathsf{cID}} \in \mathcal{C}_{\mathsf{off}}$ and $w \in \mathcal{W}$ across all segments and levels. These public domains are chosen at setup outside every supported semantic Z-axis and W-axis, respectively. Its fragment concatenates the primary columns and auxiliary rows of all active segments, with 2κ stored values per segment.

4.1. Advertisements and registries

An advertisement names the maintainer, its cID, the anchor, and the threshold:

$$\mathsf{advert} = (\mathsf{mID}, \mathsf{cID}, A_t, \kappa).$$

A registry is $\mathcal{R}_t = (A_t, \mathcal{A}_t, \kappa_t, n_t, b_t, m_t)$, with n_t selected advertisements and budgets of b_t corrupted and m_t further missing maintainers. Its supported threshold κ_t is the threshold at which the caller combines replies. An advertisement is compatible when it names A_t and $\kappa_t = 2^j \kappa$ for some $j \geq 0$. Thus the smallest compatible threshold is the largest advertised threshold among the selected maintainers. The other registry conditions are inherited: maintainer identifiers, column identifiers, and row labels are each distinct, and $n_t \geq \kappa_t + 2b_t + m_t$. We call a registry *well formed* when compatibility and distinctness hold, and *threshold-valid* when the inequality holds.

4.2. Serving public requests at a higher threshold

A maintainer that advertises κ answers a public request at $2^j \kappa$ by applying the stitch only to the entries the request touches. Its stored fragment and advertisement stay at κ. The request names the combining threshold, so maintainers that store different levels can participate in the same registry. Figure 1(c) illustrates a read at 2κ using helpers that store either κ or 2κ.

Proposition 3 (Serving a higher threshold)**.** *Fix* $j \geq 1$ *and a segment* T *at threshold* $2^j\kappa$*, containing* 4^j *segments at threshold* κ*. A maintainer with assignments* (C, w) *that stores their columns and rows computes each entry of the combined segment's column* $L_{T,C}(W) = P_T(C, W)$ *and row* $R_{T,w}(Z) = P_T(Z, w)$ *as a combination of* 2^j *stored entries, with public weights (equations* (11) *and* (12)*).*

Proof in Appendix A.

The replies are therefore evaluations of the same polynomial, regardless of the helpers' stored levels, and the caller combines them as usual. Padding has the same public values at every level, so this also holds for partially filled segments. A maintainer may cache and update the computed values; retaining the complete higher-level fragment and dropping

the old one is a stitch. Section 7.2 compares answering directly, computing shares on demand, and stitching first. The private-opening construction of Section 5 instead uses a registry whose maintainers store or retain one common served level.

4.3. Inherited operations and UNDINELOG

UNDINE keeps BORG's batch validation, authenticated state, and opening protocol. An accepted batch determines the maintained-state delta $\Delta_t = M_{t+1} - (M_t, 0)$. Each maintainer applies it with the update rule of Section 3.1, changing one column entry and one row entry per changed cell. Updating commutes with stitching, so maintainers at different levels remain consistent.

Reads and repair apply Section 3.4 using the public mapping from array cells to grid points. A contiguous array range is served by reading whole segments in its interior and the required cells in its boundary segments. Onboarding runs both repairs for every active segment: with κ correct helpers, it downloads 2κ field elements per segment, exactly the number of entries in the new fragment.

UNDINELOG is the append-only specialization, following BORGLOG. It inherits BORGLOG's sparse Merkle layout and frontier: the opening of position $\ell_t - 1$, or the public empty boundary when $\ell_t = 0$. The frontier and the new entries determine the next anchor and Δ_t, so the log publishes ordinary appends and each maintainer updates independently. Onboarding or repair recovers the frontier with an ordinary opening. No change to the log's authenticated layer is needed.

4.4. Security

Security follows from BORG's composition theorem [6]. The encoding supplies the required linear fragments, reads with error correction, local updates, and fragment materialization. Let $\mathcal{T}_{\text{UNDINE}}$ comprise repair and onboarding at the same threshold, the local stitch, and its undo using helpers at either threshold. Threshold changes preserve the cID and its assignments; Appendix A restates the composition theorem and checks its hypotheses for the new encoding and registry compatibility rule. BORG's *extendable distributed vector commitment* (EDVC) notion asks for correctness, integrity, availability, and compactness.

Theorem 2 (UNDINE is an EDVC)**.** *Under the sequencing and validation assumptions of Section 2 and collision resistance of the Merkle label function,* UNDINE *satisfies EDVC correctness, integrity, and availability for* $\mathcal{T}_{\text{UNDINE}}$ *for every well-formed, threshold-valid registry with at most* b_t *corrupted and* m_t *further missing maintainers. A fragment advertised at* κ *holds* $2\kappa\lceil L_t/\kappa^2\rceil \le 2L_t/\kappa + 2\kappa$ *field elements. In* UNDINELOG*, the frontier adds* $O(\log \mathsf{N})$ *labels. For* $\kappa^2 \le L_t$*, the total is* $O(L_t/\kappa + \log \mathsf{N})$*, with* $L_t = O(\ell_t + \log \mathsf{N})$*, giving EDVC compactness.*

Proof in Appendix A.

5. Private Merkle Openings

A client knows a position $x < \ell_t$ and wants an opening under an anchor $A_t = (\mathsf{Com}_t, \ell_t)$ without revealing x. The opening needs a leaf and sibling hashes scattered over the tree, while PIR retrieves one database record. We construct these records from the existing preorder layout, without storing another copy of the tree. We first describe retrieval from one server, then from maintainers holding coded fragments. The construction works with any linear single-server PIR satisfying the interface below.

The PIR interface. For a database D of R equal-length records of symbols modulo p, $\mathsf{Setup}(D)$ produces query-independent server preprocessing st and any client decoding state required by the backend. To retrieve record r, $\mathsf{Query}(r)$ produces a client secret, auxiliary request material k (empty if unused), and a query q from which a public expansion, using k, yields one *weight* q_r per record in a ring $\mathcal{Q}$ that acts on record symbols by multiplication. For SIMPLEPIR-style schemes the weights are the query's integer coefficients modulo Q; for SEALPIR-style schemes they are expanded ciphertexts. The client uses its secret to Decode the reply of $\mathsf{Answer}(\mathsf{st}, k, q)$, with any decoding state implicit in the notation. Correctness means an honest server's reply yields Rec_r except with probability at most δ, the PIR backend's correctness error. Query privacy means (k, q) is computationally indistinguishable for any two records.

We require a *linear* PIR: the server computes $s = \sum_r q_r \mathrm{Rec}_r$ in $\mathcal{Q}$, symbol by symbol, then returns $f(\mathsf{st}, k, s)$, where f accesses the weights only through s.

5.1. One server

First consider a server that stores the tree without coding. Figure 2 follows the opening of leaf 4 from the tree to a PIR reply.

Groups and patterns. The server cuts the preorder array into *groups* of G consecutive cells, the analogue of a maintainer's segments; write c_g for the group at index g. In panel (a), $G = 3$. Leaf 4 needs its own cell and the siblings 5, w_1, and u, shown in orange. Group c_3 contains the subtree of w_0, including leaves 4 and 5. Returning c_3 together with the individual cells w_1 and u therefore gives an opening of either leaf. This collection is a *pattern*, treated as one logical PIR record.

In general, choose a height γ with $2^{\gamma+1} \ge G$. Each pattern contains the whole groups covering a subtree of height γ, followed by its non-default upper siblings as individual cells. The subtree occupies a consecutive range of at most $2^{\gamma+1} - 1$ stored cells; the upper siblings are shared by all leaves below its root. There are $R = \lceil \ell_t / 2^\gamma \rceil$ patterns, numbered from the left with roots u_r. The client opening x requests $r = \lfloor x/2^\gamma \rfloor$.

The record of a pattern. Let S_r be the groups covering the subtree of u_r. There are at most J consecutive groups in S_r, where $J = \lceil (2^{\gamma+1} - 1)/G \rceil + 1$, and at most $U = \max\{\lceil \log_2 \ell_t \rceil - \gamma, 0\}$ non-default upper siblings. The record

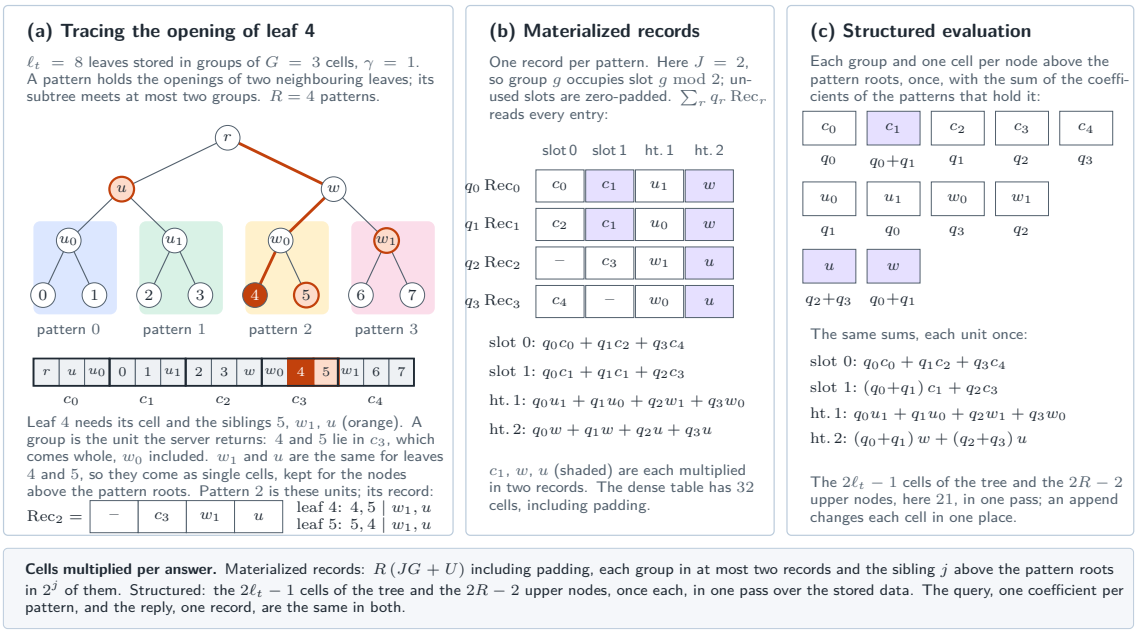

Figure 2. Private openings on eight leaves, stored in preorder in groups of three cells. (a) One pattern contains the openings of leaves 4 and 5: the group covering the subtree of w_0, plus its upper siblings as individual cells. (b) Materializing all records duplicates groups and upper siblings across patterns. (c) Instead, the server adds the query weights of patterns using each group or upper sibling and multiplies that stored unit once. Both computations give the same PIR reply.

has $K = J + U$ *slots*, reserved spaces for its groups and upper siblings: J group slots of G cells each, followed by U one-cell slots for upper siblings. Group c_g occupies slot $g \bmod J$, distinct for the consecutive groups of S_r. Keeping each group in the same slot across records lets the server add the corresponding query weights before multiplying that group. Slot $J+j$ holds $v_{r,j}$, the upper sibling at height $\gamma+j$. The last group and unused slots are padded with zeros; the client supplies default hashes locally. Each leaf's opening is at publicly determined offsets in this record Rec_r. In panel (a), $J = 2$ and $U = 2$, so the rule uses $g \bmod 2$. Thus c_3 occupies slot $3 \bmod 2 = 1$, giving $\mathrm{Rec}_2 = (\mathbf{0}, c_3, w_1, u)$; the unused slot 0 is filled with zeros.

Answering from the stored tree. Figure 2(b) lays out these records as rows. Group c_1 appears in records 0 and 1, and upper sibling u in records 2 and 3. Materializing the records repeats both storage and multiplication. Panel (c) instead multiplies c_1 once by $q_0 + q_1$, and u once by $q_2 + q_3$, giving the same sum. In general, collect the query weights of all patterns using each stored unit before multiplying it. For an upper node v, let P_v be the contiguous range of patterns below its sibling; write v also for its cell value. Let a_k indicate the slot of the server's PIR response at position $0 \le k < K$. Applied symbol by symbol, the PIR sum is

$$a_i = \sum_{g \equiv i \ (J)} Q_g\, c_g, \qquad Q_g = \sum_{r\,:\,c_g \in S_r} q_r,$$
$$a_{J+j} = \sum_{v\,:\,\mathrm{ht}(v)=\gamma+j} Q_v\, v, \qquad Q_v = \sum_{r \in P_v} q_r, \tag{7}$$

for $0 \le i < J$ and $0 \le j < U$, with arithmetic in $\mathcal{Q}$ and $\mathrm{ht}(v)$ the height of v. A group intersects at most two pattern subtrees, so Q_g has at most two terms. The first pass multiplies each relevant group and each of $O(R)$ upper nodes once. For a perfect tree with $R = 2^U$ patterns, the materialized records contain $RU = R \log_2 R$ upper-sibling occurrences but only $2R - 2$ distinct upper nodes. Factoring multiplies each node's symbols once, removing a logarithmic factor from the upper-sibling work.

Proposition 4 (Answer from a once-stored tree)**.** *For every query q, the slots of (7) equal $\sum_r q_r \mathrm{Rec}_r$. The server therefore gives the same reply as PIR over materialized records, including for malformed queries. The backend's correctness and query privacy apply unchanged.*

Proof. In each group slot, regroup $\sum_r q_r \mathrm{Rec}_r$ by group g; in each cell slot, regroup it by node v. These give the two sums in (7). Linearity makes the reply a function of this sum and the same preprocessing and request material. □

The first pass is linear in the filled tree. Appends preserve existing cell positions and, at a fixed γ, pattern numbers. The layout therefore needs no separate preprocessing to reorganize the tree for PIR; any preprocessing required by the PIR backend remains.

Choosing the pattern size. Larger patterns shorten the query and enlarge the reply, while the first pass remains linear in the tree. With β_q request bytes per weight, the bytes of one coefficient for SimplePIR-style schemes and the ciphertext bytes amortized over the weights it expands to

for compact queries, and β_c bytes per returned cell, the main transfer terms are $\ell_t \beta_q / 2^\gamma$ up and $2^{\gamma+1}\beta_c$ down; for compact queries the up term is a step function and the minimum pattern size may bind; group boundaries add $O(G)$ cells and upper siblings add U. Balancing the main terms gives

$$2^{\gamma^\star} = \sqrt{\ell_t \, \beta_q / (2\beta_c)}, \tag{8}$$

The public function $\gamma(\ell_t)$ rounds this choice to an integer height subject to $2^{\gamma+1} \geq G$.

5.2. Maintainers

PIR over coded fragments. The construction extends to any linear code applied independently to each group. Keep the patterns and slot positions of Section 5.1, replacing each group by the maintainer's stored fragment of it. For an upper sibling, its slot can always hold the entire fragment of the group containing that node: reconstructing the group recovers the sibling. Each slot reserves a fixed, publicly known number of coded entries. Unused slots contain zeros, so all pattern records have the same length. Proposition 4 applies to these stored entries in each output slot, so maintainers answer without materializing the records.

Reconstructing an opening. The client selects maintainers for one anchor and a common code configuration, group layout, and PIR parameters. It sends the same private query to each, decodes the PIR replies, then uses the code's ordinary decoder to recover the needed groups and assemble and verify the opening. The number of maintainers needed and the tolerance of wrong and missing fragments are those of the chosen code.

Instantiation in UNDINELOG. Here $G = \kappa^2$, and a maintainer's primary column contains κ entries per segment. These columns suffice for group reconstruction; auxiliary rows are not queried. As a small optimization, UNDINELOG can shorten each upper-sibling slot to one stored column entry using the point-read procedure of Section 3.4. The resulting record has J column slots and U individual entries, totaling $J\kappa + U$ entries, as illustrated in Figure 3 (Appendix D).

Keeping queries and responses balanced. A maintainer returns a $1/\kappa$ share of each subtree, so the query term of Section 5.1 is unchanged and the response term becomes $2^{\gamma+1}\beta_c/\kappa$, with balanced pattern size $2^{\gamma^\star_\kappa} = \sqrt{\ell_t \kappa \beta_q / (2\beta_c)}$, $\sqrt{\kappa}$ times the one-server choice in (8). At a fixed κ, quadrupling the log doubles this size, so both main transfer terms grow as $\sqrt{\ell_t}$. The client and maintainers use the same public function $\gamma(\ell_t, \kappa)$ to round the height and enforce $\gamma \geq \max\{0, 2\log_2 \kappa - 1\}$ when the balanced choice is smaller. For $\kappa \geq 2$, the minimum subtree has $\kappa^2 - 1$ cells and may cross a segment boundary, so its record reserves two column slots plus the upper sibling entries.

Serving in UNDINELOG. Each maintainer answers from $\kappa\lceil L_t/\kappa^2 \rceil$ stored field elements, roughly $2\ell_t/\kappa$, and the first pass is linear in that size. The maintainers compute in parallel; increasing κ reduces each one's stored input and scan while increasing the number of replies needed.

The client selects a threshold-valid registry for one anchor A_t at one stored or retained threshold κ_t, so the maintainers share segment boundaries and pattern layout. Public openings can also use compatible lower thresholds (Section 4.2); private openings require this common layout, so serving another level requires retaining its fragment and PIR preprocessing. During appends, maintainers continue serving recent anchors for a public window covering anchor selection through arrival of the complete requests.

5.3. Privacy against malicious maintainers

A malicious server can corrupt selected records and observe whether the client accepts, learning whether its query touched those records. This is *selective failure*. In UNDINELOG, Combine corrects malicious shares within the fault budget, so every position yields a valid opening and acceptance reveals nothing. The protocol also fixes the client's observable timing and follow-up behavior independently of the replies.

Protocol. To open x privately under A_t, the client

(i) selects advertisements for A_t and a common stored or retained threshold κ_t into a well-formed, threshold-valid registry $\mathcal{R}_t$, by a rule independent of x;

(ii) runs $\mathsf{Query}(r)$ with fresh randomness for the pattern $r = \lfloor x/2^\gamma \rfloor$, where $\gamma = \gamma(\ell_t, \kappa_t)$, and sends the same (ℓ_t, κ_t, k, q) to each selected maintainer in registry order;

(iii) collects replies until a fixed deadline, decodes them, and runs Combine with error correction on all received shares, accepting the opening if Verify accepts under A_t;

(iv) sends no follow-up for this opening, schedules its next public action at a fixed time, and chooses later registries independently of the replies' contents.

Request transmissions follow a public schedule that allows a public upper bound on query-generation time. The reply deadline and next action time are also chosen from public parameters, independently of x. The next action allows time after the deadline for client processing at every position and fault pattern within the budget.

Theorem 3 (Private openings)**.** *In the protocol above, let at most b_t selected maintainers be corrupted and at most m_t further ones miss the deadline, including under load.*

(a) Correctness. *For every strategy of the corrupted maintainers, the client accepts the opening of x under A_t, except with probability at most $|\mathcal{R}_t|\,\delta$, where δ is the PIR backend's correctness error.*

(b) Privacy. *For any two positions $x_0, x_1 < \ell_t$, the joint views of the corrupted maintainers in the openings of x_0 and of x_1 are computationally indistinguishable, under the query privacy of the PIR backend and up to the PIR correctness error $|\mathcal{R}_t|\,\delta$. The views include the requests, the anchor, the registry, the lengths and times of all messages of the client, and whether the client accepts.*

Proof in Appendix A. The bound b_t concerns malicious maintainers only, and only for the acceptance bit in (b):

the request reveals nothing to any coalition, so an honest-but-curious coalition of all maintainers learns nothing. With a registry of $n \geq \kappa_t + 2b_t + m_t + 2d$ maintainers, the client accepts unless more than d honest replies fail, so the term $|\mathcal{R}_t|\,\delta$ in (a) and (b) becomes the probability that $d+1$ honest replies fail.

The guarantee requires enough honest replies before the deadline. Beyond the fault budget, reconstruction can fail and acceptance can depend on the replies. Step (iv) also rules out a reply-dependent retry or dropping a maintainer because of its private reply. A later registry may exclude it based on an independent public audit, such as a failed public opening.

6. Instantiation with YPIR

We instantiate Section 5 with YPIR+SP, the SimplePIR-based variant of YPIR for retrieving larger records [12], [16]. The client holds no database-dependent state, query privacy needs no non-collusion assumption, and the linear first pass supports Proposition 4. Its server preprocessing also admits updates after appends.

Records and preprocessing. We use YPIR's unchanged parameters, estimated at 128 bits of security [12]. The entries of a pattern record are serialized as plaintext symbols. The logical matrix D has one row per pattern and one column per symbol; maintainers answer from their stored columns without materializing D. $\mathsf{Setup}(D)$ produces a server hint $\mathsf{Hint} = \mathbf{A}D$, for a public seeded matrix $\mathbf{A}$, and reply-packing tables derived from it. Both stay at the maintainer. Appendix B specifies packing and padding.

Query, answer, and decoding. $\mathsf{Query}(r)$ produces a fresh secret, query q, and packing key k. The client sends the same (k, q) to each selected maintainer. Answer computes $s = q^\top D$ using (7), then packs s into reply ciphertexts using its preprocessing and k: the function $f(\mathsf{st}, k, s)$ of the interface. Decode recovers the record's symbols and reassembles its entries for ordinary opening reconstruction. For matrices with at most 2^{18} rows and columns, an honest reply of c ciphertexts has YPIR decryption-failure probability $\delta \leq c\,2^{-40}$. Our information-theoretic Combine is exact within its bound on wrong and missing shares and adds no failure probability.

The pattern size. The backend gives $\beta_q = 7$ bytes per query coefficient and $\beta_c \approx 110$ bytes per returned 32-byte entry. The main query and response terms therefore balance at $2^{2\gamma} \approx 0.032\,\ell_t\kappa$, subject to the minimum pattern size of Section 5.2. Appendix B gives the public rounding rule.

Updating the preprocessing after an append. Before advertising A_{t+1}, the maintainer patches the hint by aggregating changed symbols over contiguous ranges of pattern rows, then regenerates the packing tables. This equals fresh preprocessing of the new fragment (Proposition 5). When the pattern size increases, the maintainer prepares the next layout's preprocessing from its columns before the public step, patching both layouts until the transition.

7. Evaluation

We evaluate whether UNDINELOG improves maintenance over BORGLOG, keeps private serving practical as logs grow and thresholds rise, and lets clients reconstruct openings efficiently even with malicious maintainers.

7.1. Experimental setup

Our Rust implementation shares BORGLOG's sparse Merkle layer, append-only application, wire format, and benchmarks, replacing its coded-array layout. The tree has height $H = 50$, entries occupy 32 bytes, and append batches contain 4 MiB. Maintenance comparisons use $\kappa \in \{8, 16, 32, 64\}$ and $\mathsf{Seg} = \kappa^2$ in both layouts, giving equal initial fragment sizes. $\mathsf{Seg} = \kappa^2$ is also BORG's fastest operating point for these operations: a larger segment would let it keep balance through local stitches, at the price of longer columns in every append and point reply. Repair and reconfiguration run on a 12 MiB prefix and are reported per 4 MiB. Private-serving workloads span 1 GiB to 1 TiB, testing large sparse logs of the kind used by shielded pools; the 4 GiB point is about the size of Zcash's shielded pools today, whose Sapling and Orchard trees held 124.5 million note commitments at the end of September 2026 [14], about 3.7 GiB of 32-byte entries.

Both layouts use the BLS12-381 scalar field through arkworks, with field elements serialized in 32 bytes. Merkle labels are domain-separated BLAKE3 digests reduced into this field. Reconstruction uses barycentric interpolation; error correction uses Gao's Reed–Solomon decoder, shared by both implementations.

We report mean operation times on one pinned core of an Intel Core Ultra 7 255U, using the backend's avx2 kernels. Independent repetitions stop when a 95% Student-t confidence interval, corrected for twelve sampling looks, has at most 5% relative half-width; all 440 timing series meet this target. The artifact [17] records the sampling schedule, run identifier, and supplemental measurements.

Private-opening timers cover the complete local processing of each stage: fresh key and query generation with request serialization; server request parsing through reply serialization; and client reply parsing, decryption, share extraction, Combine, and Verify. Helper timings in the maintenance comparison also include response serialization. We report serialized message sizes separately from processing times, excluding network transit and protocol waits. Initial PIR preprocessing times are estimates from measured hint and packing kernels; answering and updates are timed directly. Server and client stages are measured separately, and each recovered opening is checked against the expected entries.

7.2. Maintenance compared to BORGLOG

Table 1 shows faster maintainer operations throughout. Appends change two stored entries per new cell instead of recomputing every stored entry of its segment, giving 1.4–1.6× faster updates. Public point responses read stored

TABLE 1. UNDINELOG (UL) AND BORGLOG (BL) AT $\mathsf{Seg} = \kappa^2$. TIMES ARE PER MAINTAINER, HELPER, OR CLIENT. REPAIR AND RECONFIGURATION ARE NORMALIZED TO 4 MIB. A DASH DENOTES AN UNAVAILABLE OPERATION OR ONE REQUIRING NO HELPER.

	$\kappa = 8$		$\kappa = 16$		$\kappa = 32$		$\kappa = 64$	
	UL	BL	UL	BL	UL	BL	UL	BL
Append one 4 MiB batch (ms)	164	262	162	231	161	224	164	236
Fragment growth per batch (MiB)	2.00	2.00	1.00	1.00	0.500	0.500	0.250	0.250
Point opening								
helper (µs)	8.78	179	8.23	196	8.19	233	7.90	310
helper response (bytes)	1,251	1,251	1,251	1,251	1,251	1,251	1,251	1,251
client, κ responses (ms)	0.224	0.276	0.322	0.371	0.610	0.662	1.63	1.69
client, 3κ responses, κ corrupted (ms)	0.622	0.700	2.94	3.03	14.3	14.4	55.1	55.3
Repair of one fragment, per 4 MiB								
receiver (ms)	36.9	489	21.1	167	15.4	111	14.8	79.2
one helper (ms)	3.79	29.7	1.31	9.31	0.543	3.36	0.261	1.28
download (MiB)	2.00	2.00	1.00	1.00	0.501	0.501	0.251	0.251
$\kappa \to 2\kappa$ by a local stitch, per 4 MiB								
receiver (ms)	1.47	66.9	0.627	36.9	0.309	29.3	0.192	30.2
fragment afterwards (MiB)	1.00	1.50	0.501	0.750	0.251	0.376	0.128	0.189
$\kappa \to 2\kappa$ by BL's helper-assisted lens change, per 4 MiB								
receiver (ms)	–	90.8	–	69.7	–	64.9	–	70.6
one helper (ms)	–	11.7	–	3.68	–	1.93	–	2.83
download (MiB)	–	1.88	–	0.969	–	0.493	–	0.249
fragment afterwards (MiB)	–	2.50	–	1.25	–	0.626	–	0.314
$2\kappa \to \kappa$, per 4 MiB								
receiver, helpers at 2κ (ms)	16.4	130	12.7	24.2	11.1	18.3	11.0	16.8
one helper at 2κ (ms)	0.712	13.3	0.311	4.45	0.150	1.79	0.0806	0.703
download, helpers at 2κ (MiB)	1.00	1.38	0.501	0.719	0.251	0.368	0.128	0.187
receiver, helpers at κ (ms)	2.21	–	0.891	–	0.397	–	0.238	–
download, helpers at κ (MiB)	1.00	–	0.501	–	0.251	–	0.128	–

entries instead of interpolating, making helpers 20–39× faster with unchanged response sizes and client reconstruction algorithms. Repair also downloads the same amount, while its receiver is 5.4–13× faster and each helper 4.9–7.8× faster.

Doubling κ locally requires no helpers or download and halves the fragment while preserving balance. It is 46–157× faster than BORGLOG's local stitch, which loses balance, and 62–367× faster than its helper-assisted lens change, which downloads about one fragment and enlarges it by a quarter. Halving κ downloads 1.4–1.5× less; the receiver and helper are 1.5–7.9× and 8.7–19× faster, respectively. Helpers at the target threshold supply stored entries, avoiding the per-segment interpolation needed with helpers at twice that threshold.

A maintainer can also answer public openings at a higher threshold without changing its stored fragment (Table 7 in Appendix C). Its point response has the same bytes as a response from a fragment stored at that threshold and takes 1.4–3.1× the time; stitching the entire fragment would cost 165–3,180× one such response. Supplemental range measurements show 1.1–22× faster helpers with unchanged response sizes. The artifact [17] includes the full range comparison.

7.3. Private serving and maintenance

We instantiate private openings with YPIR as in Section 6. A single server holding the whole tree runs the same backend, isolating the effect of distributing its work and storage. Table 2 measures private-query processing and the PIR preprocessing updates after 4 MiB appends; Table 1 covers the fragment update itself.

Growing logs and increasing thresholds. Table 2 first holds $\kappa = 64$ fixed: answering remains practical as the log grows from 1 to 256 GiB, taking 49.9 ms to 1.15 s on one core. The database and first pass grow with the fragment. The next panel instead fixes the log at 4 GiB. Raising κ from 8 to 64 reduces the database from about 1 GiB to 128 MiB, the first pass from 104 ms to 9.38 ms, and the full answer from 252 ms to 100 ms.

Against the measured single server on the same log, a maintainer at $\kappa = 64$ has a 64× smaller database and 6.3× smaller preprocessing tables, a 58–64× faster first pass, and a 8.6–12× faster full answer. The table also shows what this reduction permits: a maintainer serving 64 GiB and the single server serving 1 GiB both scan 2 GiB, taking 158 ms and 151 ms, respectively. Thus a maintainer handles a 64-times larger log at approximately the same scan cost. Reply packing explains why full-answer speedups differ from first-pass speedups.

Balanced communication. The pattern rule adjusts to both log size and κ. In its balanced regime, query coefficients and replies both scale as $\sqrt{\ell_t/\kappa}$ per maintainer, subject to the minimum pattern size, group boundaries, and backend padding. The table separates these coefficients from the full request, which also carries about 462 KiB of fresh YPIR key material. Query coefficients and replies remain approximately balanced across both sweeps, with discrete steps as patterns

TABLE 2. PRIVATE SERVING ON ONE CORE, PER MAINTAINER CID OR SINGLE SERVER. *Database* COUNTS MAINTAINER PRIMARY COLUMNS OR SINGLE-SERVER TREE ENTRIES; *preproc. size* COUNTS THE HINT AND PACKING TABLES. *Request total* AND *reply* ARE SERIALIZED SIZES; *query* COUNTS REQUEST COEFFICIENTS ONLY. *Answer* INCLUDES THE FIRST PASS. PREPROCESSING TIME IS ESTIMATED FROM MEASURED KERNELS; *update* TIMES PIR PREPROCESSING AFTER A 4 MIB APPEND, EXCLUDING THE APPEND. THE SINGLE SERVER IS MEASURED UP TO 4 GIB; LARGER WHOLE-TREE DATABASES EXCEED THE MACHINE'S MEMORY.

Log	κ	Database	Preproc. size	Request		Reply	First pass	Answer	Preproc. time	Update
				Total	Query					
Maintainer: growing log at $\kappa = 64$										
1 GiB	64	32.0 MiB	385 MiB	490 KiB	28.0 KiB	36.0 KiB	2.61 ms	49.9 ms	3.54 s	2.70 s
4 GiB	64	128 MiB	769 MiB	518 KiB	56.0 KiB	72.0 KiB	9.38 ms	100 ms	7.64 s	5.32 s
16 GiB	64	512 MiB	1.25 GiB	574 KiB	112 KiB	120 KiB	36.5 ms	184 ms	15.9 s	8.78 s
64 GiB	64	2.00 GiB	2.38 GiB	686 KiB	224 KiB	228 KiB	158 ms	440 ms	40.7 s	16.6 s
256 GiB	64	8.00 GiB	4.76 GiB	910 KiB	448 KiB	456 KiB	603 ms	1.15 s	121 s	33.1 s
Maintainer: increasing threshold on the 4 GiB log										
4 GiB	8	1.00 GiB	1.25 GiB	686 KiB	224 KiB	120 KiB	104 ms	252 ms	20.8 s	9.90 s
4 GiB	16	512 MiB	1.25 GiB	574 KiB	112 KiB	120 KiB	42.8 ms	190 ms	15.6 s	9.07 s
4 GiB	32	256 MiB	641 MiB	574 KiB	112 KiB	60.0 KiB	19.7 ms	95.5 ms	7.93 s	4.51 s
4 GiB	64	128 MiB	769 MiB	518 KiB	56.0 KiB	72.0 KiB	9.38 ms	100 ms	7.64 s	5.32 s
Maintainer: larger log and threshold										
1 TiB	256	8.00 GiB	4.88 GiB	910 KiB	448 KiB	468 KiB	570 ms	1.13 s	126 s	34.4 s
Single server with the same PIR backend										
1 GiB	1	2.00 GiB	2.38 GiB	686 KiB	224 KiB	228 KiB	151 ms	429 ms	40.7 s	–
4 GiB	1	8.00 GiB	4.76 GiB	910 KiB	448 KiB	456 KiB	603 ms	1.15 s	123 s	–

and ciphertext counts change.[1]

Serving a terabyte. At 1 TiB and $\kappa = 256$, a maintainer answers in 1.13 s from an 8.00 GiB database—approximately the same work as at 256 GiB and $\kappa = 64$. The whole-tree server would instead scan a 2.00 TiB database per query. Raising κ therefore keeps the work within a maintainer's capacity as the log grows. At this size, primary columns, equally sized auxiliary rows, and 4.88 GiB of preprocessing account for 20.9 GiB for one anchor, excluding representation overhead and transient allocations. Retaining additional anchors requires additional PIR state.

Keeping the state current. After a 4 MiB append, updating the PIR preprocessing costs 1.3–3.7× less than the estimated fresh preprocessing across the fixed-κ log sweep, and takes 34.4 s at 1 TiB. Each run checks the patched hint and recovered openings against the updated log. At 1 and 4 GiB, materializing a separate record for every pattern uses 1.2–1.3× the primary-column database space and makes the first pass 1.3–1.6× slower, while producing byte-for-byte identical replies. Full answers take 1.0–1.1× as long, since packing is unchanged. Appendix C gives the full threshold and pattern-size sweeps.

The code divides the scan among maintainers. At 4 GiB, the first passes of the κ maintainers needed without faults total 600 ms, close to the single server's 603 ms; 3κ maintainers hold three copies of the tree and total 1.80 s. Each maintainer also packs its own reply, a YPIR step that follows the reply size rather than the database.

1. All measured configurations stay within YPIR's parameter bound. Union bounds on decryption failure in honest replies are $192 \cdot 6 \cdot 2^{-40} \approx 2^{-30}$ per opening at 4 GiB and $\kappa = 64$, and $768 \cdot 39 \cdot 2^{-40} \approx 2^{-25}$ at 1 TiB and $\kappa = 256$. With $d = 1$ these parameters give a per-opening term below 2^{-50} if failures of distinct ciphertexts are independent. The measured 3κ registries use $d = 0$, so 192 maintainers tolerate 64 wrong replies under the theorem's bound and 63 under $d = 1$.

TABLE 3. CLIENT PROCESSING PER OPENING. *Req.* GENERATES AND SERIALIZES THE FRESH REQUEST. *Priv.* PARSES AND DECRYPTS REPLIES, EXTRACTS SHARES, AND RUNS Combine AND Verify; *pub.* RUNS ONLY Combine AND Verify ON DECODED RESPONSES. BOTH 3κ CASES RUN ERROR CORRECTION; THE κ-HONEST COLUMNS ARE A LOWER BOUND USING DIRECT COMBINATION.

			κ honest		3κ honest		3κ, κ lie	
κ	Log	req. (ms)	priv. (ms)	pub. (ms)	priv. (ms)	pub. (ms)	priv. (ms)	pub. (ms)
Increasing threshold on the 4 GiB log								
8	4 GiB	41.3	22.2	0.338	66.1	0.451	66.6	0.737
16	4 GiB	37.4	44.3	0.456	132	1.32	135	3.02
32	4 GiB	38.0	44.5	0.863	136	4.56	149	16.8
64	4 GiB	35.8	107	2.24	333	17.5	380	64.2
256	4 GiB	34.0	379	29.3	1,320	274	2,069	1,021
Growing log at $\kappa = 64$								
64	4 GiB	35.8	107	2.24	333	17.5	380	64.2
64	8 GiB	38.0	106	2.31	330	18.2	379	66.6
64	16 GiB	37.6	176	2.39	535	18.7	585	70.0
64	32 GiB	41.4	176	2.45	537	19.3	588	72.8

7.4. Client reconstruction and communication

The client opens the middle position using pre-generated serialized encrypted replies for distinct cIDs of an allocated, unwritten log, isolating its processing from server setup and answering. In the corrupted case, each of κ maintainers supplies one incorrect path entry; the timings measure this fault pattern.

Increasing the threshold. Table 3 first fixes the log at 4 GiB. Raising κ from 8 to 256 increases the replies from 24

to 768, but processing remains practical: at $\kappa = 256$, it takes 1.32 s with all replies honest and 2.07 s with 256 corrupted.

Growing the log. At fixed $\kappa = 64$, growing the log eightfold, from 4 to 32 GiB, increases private reply processing from 333 ms to 537 ms with honest replies, and from 380 ms to 588 ms with 64 corrupted. More patterns do not require reconstructing more openings: the client recovers one selected record per maintainer and reconstructs the requested path. The ordinary reconstruction and verification comparator grows only from 64.2 ms to 72.8 ms with corrupted replies, as the path gains non-default siblings. The extra private work processes the PIR replies, whose sizes grow in steps with the patterns.

Request preparation and communication. Preparing and serializing a fresh request takes 34.0 ms–41.4 ms. Summing serialized messages over 3κ maintainers gives 97.1 MiB up (86.6 MiB packing key, 10.5 MiB query coefficients) and 13.5 MiB down at 4 GiB and $\kappa = 64$, and 683 MiB up (347 MiB key) and 351 MiB down at 1 TiB and $\kappa = 256$. These totals are calculated from message sizes; client processing is measured at the sizes in Table 3. Across 3κ maintainers, the balanced query and reply terms scale as $\sqrt{\ell_t \kappa}$, while repeated transmission of the fixed-size key grows with κ.

8. Related Work

Distributed maintained logs. UNDINE inherits BORG's maintenance and security framework [6]. Elastic Reed–Solomon codes [18] and convertible codes [19] change thresholds by downloading missing parts of the new encoding. StripeMerge [20] merges two encodings at twice the threshold, avoiding communication where the source fragments are colocated. Our product grid is a two-dimensional Reed–Solomon code, as in data-availability encodings [21], [22]. The contribution is the stored state: one column and one row at off-grid points shared across segments and levels. Every maintainer can then double its threshold locally and repeatedly while preserving balanced storage, undo the stitch from half a fragment, and serve reads from stored entries. Regenerating codes [23] reduce repair download; here, as in BORG, repair downloads the fragment it rebuilds.

Private retrieval of Merkle openings. TreePIR [11] partitions a perfect binary tree of height h into h sub-databases using a *balanced ancestral coloring*. An opening retrieves one proof node per class with one PIR query, served at one server or one server per class in parallel. The coloring construction supports heterogeneous class sizes and is reusable at fixed height, including for padded trees. TreePIR also considers growing forests, with smaller anonymity sets in smaller trees. Its servers share a coloring, coupling their storage allocations; a UNDINELOG maintainer can locally double its threshold to reduce its stored share while other maintainers keep their fragments.

TreePIR's client computes all h database offsets in $O(h^3)$ worst-case time; a UNDINELOG client selects its pattern record with one shift (Section 5.1), and UNDINELOG keeps stored positions as the active prefix grows (Table 4). Over the same YPIR backend on a 128 GiB log at $h = \kappa = 32$, both systems transfer the same bytes, hold about the same PIR state per server, and their servers prepare and answer in similar time; the only client step that differs is request preparation, one query for all maintainers against 32 queries with independent secrets in our TreePIR configuration, at a cost independent of the position in both (Table 5).

TABLE 4. PRIVATE MERKLE OPENINGS WITH ONE TREEPIR SERVER PER COLOR CLASS (h: HEIGHT; n: NODE COUNT). LAYOUT SETUP EXCLUDES MAINTENANCE, ENCODING, AND PIR PREPROCESSING. INDEX SELECTION ASSUMES PUBLIC PARAMETERS AND WORD-SIZED INDICES, BEFORE QUERY GENERATION. UNDINELOG USES A COMMON STORED OR RETAINED THRESHOLD; RECOVERY AND PRIVACY INCLUDING ACCEPTANCE REQUIRE THEOREM 3'S PROTOCOL AND FAULT BUDGET.

Comparison	TreePIR	UNDINELOG
Additional tree-layout setup	Coloring, $O(n \log \log n)$	Stored preorder reused
Client PIR index selection	$O(h^3)$, all h offsets	$O(1)$
PIR queries prepared per opening	h, one query per color	1, reused
Replies for a fault-free opening	h, one per class	Any κ maintainers
Record returned per PIR	One proof node	Coded pattern share
Tree representation	Fixed-height perfect tree	Growing active prefix
Reducing stored share	Coordinated color allocation	Local stitch
Recovery from wrong or missing replies	None built in	Within budget

For Certificate Transparency, Kales et al. [24] use tiers of perfect subtrees, embed the bottom proof in the certificate, and retrieve the upper proof with multi-server batch PIR [25]. Batch PIR [26] retrieves several records with one query; our patterns exploit the shared structure of nodes in Merkle openings. Oblivious message retrieval [27] privately finds a wallet's notes; we privately open a position the wallet already knows.

Linear and single-server PIR. SimplePIR and DoublePIR [16] use a client-held linear database hint; incremental PIR [28] updates that hint. HintlessPIR [29] and YPIR [12] compute its contribution at the server, leaving clients without database-dependent state. Section 5 uses the linear first pass of these schemes. SealPIR [30] and Spiral [31] satisfy the interface in their one-dimensional form, or in the first dimension of their recursion, with ciphertext weights expanded from a compact query under the client's keys; they need no server-side hint or packing tables and pay instead a homomorphic product per record in the first pass.

Authenticated PIR [10], [32] and VeriSimplePIR [33] detect deviations and abort; authenticated PIR includes acceptance in the adversary's view to capture selective failure. Our guarantee corrects replies within a fault budget, following robust PIR [8], [9], [34]. Coded-storage PIR [7], [35] obtains information-theoretic privacy against bounded collusion from its code; here the single-server PIR hides the query from any colluding set of maintainers, and the code only distributes storage and computation and corrects wrong and missing replies.

9. Conclusion

UNDINE offers a flexible and efficient system for distributed storage of data, with private and authenticated retrievals. Its new encoding scheme, replacing that of BORG, allows maintainers to double the reconstruction threshold κ and reduce storage locally (without communication), while its stored share stays balanced; to serve public requests at supported higher thresholds from the fragment it holds; and halves κ with less download. It is thus faster than BORGLOG on every shared maintainer operation at equal initial storage.

UNDINE also gives a generic way to distribute the storage and query computation of linear PIR. Private openings use the stored encoded fragments directly, support growing, sparse trees without a separate tree-layout step, and tolerate missing and malicious replies within the chosen fault budget. Over the same YPIR backend, UNDINELOG matches TreePIR's per-server cost and communication, while its client generates one PIR query and reuses it across maintainers instead of generating a separate query for each TreePIR database. Serving stays practical as logs grow, since maintainer can adjust their PIR work by changing κ; and clients reconstruct even under corrupted replies. Thus, applications such as shielded pools can gain distributed private openings without changing their transactions or authenticated trees.

References

[1] B. Laurie, A. Langley, and E. Kasper, "Certificate transparency," Internet Engineering Task Force, RFC 6962, Jun. 2013.

[2] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "CONIKS: Bringing key transparency to end users," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 383–398.

[3] M. Chase, A. Deshpande, E. Ghosh, and H. Malvai, "SEEMless: Secure end-to-end encrypted messaging with less trust," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS 19)*, 2019, pp. 1639–1656.

[4] H. Malvai, L. Kokoris-Kogias, A. Sonnino, E. Ghosh, E. Öztürk, K. Lewi, and S. Lawlor, "Parakeet: Practical key transparency for end-to-end encrypted messaging," in *Proceedings of the 2023 Network and Distributed System Security Symposium (NDSS 23)*, 2023.

[5] D.-E. Hopwood, S. Bowe, T. Hornby, and N. Wilcox, "Zcash protocol specification, version 2026.7.0 [NU6.2]," Electric Coin Company, 2026. [Online]. Available: https://zips.z.cash/protocol/protocol.pdf

[6] N. Alhaddad, E. Tromer, and M. Varia, "BORG: Extendable distributed vector commitments from reconfigurable erasure codes," Cryptology ePrint Archive, Paper 2026/1552, 2026. [Online]. Available: https://eprint.iacr.org/2026/1552

[7] R. Tajeddine, O. W. Gnilke, D. Karpuk, R. Freij-Hollanti, and C. Hollanti, "Private information retrieval from coded storage systems with colluding, Byzantine, and unresponsive servers," *IEEE Transactions on Information Theory*, vol. 65, no. 6, pp. 3898–3906, 2019.

[8] I. Goldberg, "Improving the robustness of private information retrieval," in *IEEE Symposium on Security and Privacy*, 2007.

[9] C. Devet, I. Goldberg, and N. Heninger, "Optimally robust private information retrieval," in *21st USENIX Security Symposium*, 2012.

[10] S. Colombo, K. Nikitin, H. Corrigan-Gibbs, D. J. Wu, and B. Ford, "Authenticated private information retrieval," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[11] Q. Cao, S. H. Dau, R. Gagiano, D. Huynh, X. Yi, P. L. Le, Q. H. Luu, E. Viterbo, Y. C. Huang, J. Zhu, M. M. Jalalzai, and C. Feng, "TreePIR: Efficient private retrieval of merkle proofs via tree colorings with fast indexing and zero storage overhead," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 4476–4494. [Online]. Available: https://doi.org/10.1109/SP61157.2025.00032

[12] S. J. Menon and D. J. Wu, "YPIR: High-throughput single-server PIR with silent preprocessing," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5985–6002. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/menon

[13] Tornado Cash contributors, "Tornado Cash: Core contracts," GitHub repository, accessed October 2026. [Online]. Available: https://github.com/tornadocash/tornado-core

[14] Blockworks Advisory, "Zcash tokenholder report, Q3 2026," 2026, accessed October 2026. [Online]. Available: https://blockworks.com/api/investor-report/zcash-token-holder-report-q3-2026/pdf

[15] Zcash Improvement Proposals, "ZIP 258: Deployment of the NU6.3 network upgrade," https://zips.z.cash/zip-0258, 2026, accessed October 2026.

[16] A. Henzinger, M. M. Hong, H. Corrigan-Gibbs, S. Meiklejohn, and V. Vaikuntanathan, "One server for the price of two: Simple and fast single-server private information retrieval," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[17] N. Alhaddad, H. Eldridge, J. Guskind, E. Tromer, and M. Varia, "UndineLog evaluation artifact: implementations, benchmarks, reproduction pipeline, and supplementary tables," GitHub repository, 2026. [Online]. Available: https://github.com/nicolas3355/undinelog-publication-artifact

[18] S. Wu, Z. Shen, P. P. C. Lee, Z. Bai, and Y. Xu, "Elastic reed–solomon codes for efficient redundancy transitioning in distributed key–value stores," *IEEE/ACM Transactions on Networking*, vol. 32, no. 1, pp. 670–685, 2024.

[19] F. Maturana and K. V. Rashmi, "Convertible codes: Enabling efficient conversion of coded data in distributed storage," *IEEE Transactions on Information Theory*, vol. 68, no. 7, pp. 4392–4407, 2022.

[20] Q. Yao, Y. Hu, L. Cheng, P. P. C. Lee, D. Feng, W. Wang, and W. Chen, "StripeMerge: Efficient wide-stripe generation for large-scale erasure-coded storage," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, 2021, pp. 483–493.

[21] M. Al-Bassam, A. Sonnino, V. Buterin, and I. Khoffi, "Fraud and data availability proofs: Detecting invalid blocks in light clients," in *Financial Cryptography and Data Security (FC 2021)*, ser. LNCS, vol. 12675. Springer, 2021, pp. 279–298.

[22] M. Yu, S. Sahraei, S. Li, S. Avestimehr, S. Kannan, and P. Viswanath, "Coded Merkle tree: Solving data availability attacks in blockchains," in *Financial Cryptography and Data Security (FC 2020)*, ser. LNCS, vol. 12059. Springer, 2020, pp. 114–134.

[23] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE Transactions on Information Theory*, vol. 56, no. 9, pp. 4539–4551, 2010.

[24] D. Kales, O. Omolola, and S. Ramacher, "Revisiting user privacy for certificate transparency," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 432–447.

[25] W. Lueks and I. Goldberg, "Sublinear scaling for multi-client private information retrieval," in *Financial Cryptography and Data Security (FC 2015)*, ser. LNCS, vol. 8975. Springer, 2015, pp. 168–186.

[26] M. H. Mughees and L. Ren, "Vectorized batch private information retrieval," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[27] Z. Liu and E. Tromer, "Oblivious message retrieval," in *Advances in Cryptology – CRYPTO 2022*, 2022.

[28] Y. Ma, K. Zhong, T. Rabin, and S. Angel, "Incremental offline/online PIR," in *31st USENIX Security Symposium*, 2022.

[29] B. Li, D. Micciancio, M. Raykova, and M. Schultz-Wu, "Hintless single-server private information retrieval," in *Advances in Cryptology – CRYPTO 2024*, 2024.

[30] S. Angel, H. Chen, K. Laine, and S. Setty, "PIR with compressed queries and amortized query processing," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[31] S. J. Menon and D. J. Wu, "Spiral: Fast, high-rate single-server PIR via FHE composition," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.

[32] M. Dietz and S. Tessaro, "Fully malicious authenticated PIR," in *Advances in Cryptology – CRYPTO 2024, Part IX*, ser. LNCS, vol. 14928, 2024, pp. 113–147.

[33] L. de Castro and K. Lee, "VeriSimplePIR: Verifiability in SimplePIR at no online cost for honest servers," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[34] A. Beimel and Y. Stahl, "Robust information-theoretic private information retrieval," in *Security in Communication Networks (SCN 2002)*, ser. LNCS, vol. 2576, 2003, pp. 326–341.

[35] N. B. Shah, K. V. Rashmi, and K. Ramchandran, "One extra bit of download ensures perfectly private information retrieval," in *2014 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2014, pp. 856–860.

Appendix A. Proofs and the Public Layout

The public layout. The layout is BORG's, applied to each axis. Setup fixes a base threshold κ_{base}, a largest supported level, and roots of unity ζ_M for the powers of two up to the largest axis size, with $\zeta_{2M}^2 = \zeta_M$; the array is cut into blocks of the largest supported segment size and the segment index below restarts in each block. For a grid size S and an index $\tau = \sum_t \tau_t 2^t$ let

$$\rho_{S,\tau} = \prod_{t:\,\tau_t=1} \zeta_{2^{t+1}S}, \qquad G_{S,\tau} = \{\rho_{S,\tau}\zeta_S^i : 0 \le i < S\}; \tag{9}$$

then $G_{2S,\tau} = G_{S,2\tau} \sqcup G_{S,2\tau+1}$, the even and the odd half. UNDINE has thresholds $\kappa_j = 2^j\kappa_{\mathsf{base}}$ at *levels* $j \ge 0$ and segments of $\mathsf{Seg}_j = \kappa_j^2 = 4^j\mathsf{Seg}_{\mathsf{base}}$ cells. At level j, cell y of the array lies in segment $\tau = \lfloor y/\mathsf{Seg}_j \rfloor$ at offset $u = y \bmod \mathsf{Seg}_j$. Both indices split into a Z part and a W part: writing $\tau = \sum_t (2\tau_t^Z + \tau_t^W)4^t$, the odd bits of τ form τ^Z and the even bits τ^W; writing $u = q\mathsf{Seg}_{\mathsf{base}} + r$ with $r < \mathsf{Seg}_{\mathsf{base}}$, q splits into q^Z, q^W as τ does, $r = r^Z\kappa_{\mathsf{base}} + r^W$, and $u^Z = q^Z\kappa_{\mathsf{base}} + r^Z$, $u^W = q^W\kappa_{\mathsf{base}} + r^W$, both below κ_j. On each axis the order is $\pi_j(u^Z) = r^Z 2^j + \mathrm{rev}_j(q^Z)$, where rev_j reverses the j bits of its argument, BORG's bit-reversal order with κ_{base} in the place of its segment size, and likewise for u^W. Segment τ occupies $G_{\kappa_j,\tau^Z} \times G_{\kappa_j,\tau^W}$ and the cell at u sits at the point

$$\big(\rho_{\kappa_j,\tau^Z}\,\zeta_{\kappa_j}^{\pi_j(u^Z)},\ \rho_{\kappa_j,\tau^W}\,\zeta_{\kappa_j}^{\pi_j(u^W)}\big). \tag{10}$$

Segment τ at level $j+1$ contains the four level-j segments $4\tau+2\mu+\nu$, whose grids are the even and odd halves of its two axes, the four domains of (1) with $\rho_Z = \rho_{\kappa_{j+1},\tau^Z}$ and $\rho_W = \rho_{\kappa_{j+1},\tau^W}$; and because $\pi_{j+1}(\mu\kappa_j + u^Z) = 2\pi_j(u^Z) + \mu$ and $\rho_{\kappa_j,2\tau^Z+\mu} = \rho_{\kappa_{j+1},\tau^Z}\zeta_{\kappa_{j+1}}^{\mu}$, and likewise on W, the cell at offset u of child $2\mu+\nu$ sits at the same point of the field in the child and in the parent. Four consecutive segments therefore stitch into one without moving a cell, at every level, and Theorem 1 applies repeatedly. Unused offsets hold zero: a segment beyond the filled prefix is the zero polynomial, with zero slices that a cID supplies itself, so the same formulas hold at a partially filled boundary.

The EDVC properties. BORG defines an EDVC by four properties, which we state for the objects of Section 2. A *registry replacement* at anchor A_t is a pair of registries $(\mathcal{R}_t, \mathcal{R}'_t)$ in which a maintainer's fragment is materialized under a new advertisement from the replies of $\mathcal{R}_t$ and its own fragment; a scheme fixes the family $\mathcal{T}$ of replacements it supports. *Correctness*: for every valid history, an honest maintainer's fragment equals the public linear function of M_t its advertisement names, and an opening that Combine returns from correct replies passes Verify under A_t. *Availability* for $\mathcal{T}$: from a well-formed, threshold-valid registry with at most b_t corrupted and m_t further missing maintainers, Combine returns the opening of every supported query, a fragment is repaired or onboarded, every replacement in $\mathcal{T}$ is materialized, and M_t is reconstructed, except with negligible probability. *Integrity*: two verified answers to one query under one anchor agree, and a batch that fails validation does not advance the anchor. *Compactness*: a fragment advertised at κ holds $O(\lceil \ell_t/\kappa \rceil + \mathrm{polylog}(\mathsf{N}, \lambda))$ field elements.

Proof of Theorem 1.

Proof. Each summand of (4) has degree at most $2\kappa - 1$ in each variable, and at $(z, \alpha) \in G_\mu^Z \times G_\nu^W$ the selectors leave the one term $P_{\mu\nu}(z, \alpha)$, so P_T agrees with the sources on all $4\kappa^2$ cells. For uniqueness let D be the difference of two such polynomials: for every $\alpha \in G_T^W$ the polynomial $D(Z, \alpha)$ has 2κ roots and degree below 2κ, so it is zero, and then each coefficient of Z in D vanishes on all of G_T^W with degree below 2κ, so it is zero. For the column, set $Z = C$ in (4) at $W = \alpha_{\nu,i}$: the factor E_ν^W is 1 and $E_{1-\nu}^W$ is 0, which leaves the two terms of (5), whose stored values are the sources' at the same α; the parent enumerates $\alpha_{\nu,i}$ as entry $2i + \nu$. The row is symmetric. The two powers $(C/\rho_Z)^\kappa$ and $(w/\rho_W)^\kappa$ behind the selector constants cost $O(\log \kappa)$ operations and the 4κ output entries two each. Figure 1(a,b) shows the entries for $\kappa = 2$. □

Proof of Proposition 2.

Proof. In each operation the replies are evaluations of one polynomial of degree less than κ at distinct points: $P(\cdot, \alpha_j)$ at the column identifiers C_i, $P(z_a, \cdot)$ at the row labels w_i, $P(C_\star, \cdot)$ at the row labels, and $P(\cdot, w_\star)$ at the column identifiers. If e replies are incorrect, at least $\kappa + 2e$ replies arrive under the stated fault budgets, so Reed–Solomon decoding returns that polynomial. Evaluating it at the requested points gives the output. □

Proof of Proposition 3. Fix $j \ge 1$ and a segment T at threshold $2^j\kappa$. It contains 4^j segments at threshold κ, indexed by pairs $(\boldsymbol{\mu}, \boldsymbol{\nu}) \in \{0,1\}^j \times \{0,1\}^j$, the positions taken at the j stitches that lead from the source segment

to T, with source polynomial $P_{\boldsymbol{\mu\nu}}$ on the axes $G^Z_{\boldsymbol{\mu}} \times G^W_{\boldsymbol{\nu}}$. Let $E^Z_{\boldsymbol{\mu}}$ and $E^W_{\boldsymbol{\nu}}$ be the products of the j selectors of (3) along the paths of $\boldsymbol{\mu}$ and $\boldsymbol{\nu}$, each with the scale factor and the threshold of its own stitch: on the axes of T they are the indicators of $G^Z_{\boldsymbol{\mu}}$ and $G^W_{\boldsymbol{\nu}}$. The stored evaluations of the cID for T are, for $\alpha \in G^W_{\boldsymbol{\nu}}$ and $z \in G^Z_{\boldsymbol{\mu}}$,

$$L_{T,C}(\alpha) = \sum_{\boldsymbol{\mu}\in\{0,1\}^j} E^Z_{\boldsymbol{\mu}}(C)\, L_{\boldsymbol{\mu\nu},C}(\alpha), \tag{11}$$

$$R_{T,w}(z) = \sum_{\boldsymbol{\nu}\in\{0,1\}^j} E^W_{\boldsymbol{\nu}}(w)\, R_{\boldsymbol{\mu\nu},w}(z). \tag{12}$$

Proof. Applying (4) j times gives

$$P_T(Z,W) = \sum_{\boldsymbol{\mu},\boldsymbol{\nu}} E^Z_{\boldsymbol{\mu}}(Z)\, E^W_{\boldsymbol{\nu}}(W)\, P_{\boldsymbol{\mu\nu}}(Z,W). \tag{13}$$

For $\alpha \in G^W_{\boldsymbol{\nu}}$, the indicator property leaves only the terms with $\boldsymbol{\nu}' = \boldsymbol{\nu}$ in (13), and $P_{\boldsymbol{\mu\nu}}(C,\alpha)$ is the stored evaluation $L_{\boldsymbol{\mu\nu},C}(\alpha)$. The row identity is symmetric. By Theorem 1, applied j times, P_T is the unique polynomial of degree less than $2^j\kappa$ in each variable that carries the semantic values of all source segments, and the cID keeps the same (C,w) at every level. Its level-$2^j\kappa$ state for T is therefore $(P_T(C,\alpha))_\alpha$ and $(P_T(z,w))_z$. □

Proof of Proposition 1.

Proof. The four identities are (5) with $\nu = 0$ and $\nu = 1$, and (6) with $\mu = 0$ and $\mu = 1$, each solved for its unknown source entry. The divisions are defined: $E^Z_0(C) = 0$ only if $(C/\rho_Z)^\kappa = -1$, that is, $C \in G^Z_1$, and $E^Z_1(C) = 0$ only if $C \in G^Z_0$. Since $C \notin G^Z_T$, neither occurs. The same argument applies to $E^W_0(w)$ and $E^W_1(w)$, because $w \notin G^W_T$. □

BORG's composition theorem. The following theorem is inherited from BORG [6]; we include its statement to make explicit the hypotheses used by UNDINE.

Theorem 4 (BORG's composition theorem [6])**.** *Let every honest maintainer validate the candidate batches in the sequencer's order, rejecting invalid ones, and compute the change Δ_t of an accepted batch from public data, and let a coded array over M_t with replacement family $\mathcal{T}$ provide: (i)* fragments*, a public linear map $L_{\mathsf{cID},t}$ with every honest maintainer's advertised fragment equal to $L_{\mathsf{cID},t}(M_t)$; (ii)* reads*, a helper reply and a caller reconstruction that return every cell an opening needs, and all of M_t, from the replies of a registry within its fault budgets; (iii)* updates*, by which an honest maintainer computes $L_{\mathsf{cID},t+1}(M_{t+1})$ from its fragment and Δ_t without reconstructing M_t; and (iv)* materialization *of a fragment for initialization, repair, onboarding, and every replacement in $\mathcal{T}$. Then the composition is a correct EDVC, available for $\mathcal{T}$ under the array's registry and fault conditions, with integrity when the Merkle vector commitment is binding, and compact when every fragment advertised at threshold κ holds $O(\lceil \ell_t/\kappa \rceil + \mathrm{polylog}(\mathsf{N},\lambda))$ elements for the security parameter λ and $L_t = O(\ell_t + \mathrm{polylog}(\mathsf{N},\lambda))$.*

We apply this theorem to $\mathcal{T}_{\text{UNDINE}}$ of Section 4.4. Each replacement preserves the cID and its assignments and materializes its target fragment at one anchor under the source registry's fault condition.

Proof of Theorem 2.

Proof. The array of Section 4 provides the four operations. (i) Every entry stored by an honest maintainer is a value of a polynomial whose coefficients are linear in the cells, by (2) and the layout. (ii) Every read and repair is the Reed–Solomon decoding of one polynomial of degree below κ_t from one value per helper, in Z at distinct column identifiers for a point read and the repair of a row, in W at distinct row labels for a fixed-Z read and the repair of a column (Proposition 2); a threshold-valid registry leaves at least $\kappa_t + 2e$ replies when e are incorrect, which determines the polynomial; reading every segment reconstructs M_t; and helpers at a lower level contribute values of the same polynomials (Proposition 3). (iii) An update is the entrywise patch of Section 4.3, computed from the fragment and Δ_t, and it commutes with the stitch: the update of each source changes $P_{\mu\nu}$ by the interpolant of Δ_t on its grid, the stitch (4) is linear in the sources, and the result agrees with the updated cells on the parent grid, so it is the updated parent's interpolant by uniqueness; honest maintainers at every level thus hold fragments of the same M_{t+1}. (iv) Initialization is from padding; repair and onboarding are the procedures of Proposition 2 from a registry threshold-valid at the maintainer's level, with UNDINELOG additionally recovering its frontier as the opening of position $\ell_t - 1$ (or using the public empty boundary when $\ell_t = 0$); the stitch materializes the fragment at 2κ from the maintainer's own (Theorem 1); and the undo materializes the fragment at κ from the stitched fragment and additional information recovered from a registry at κ (Proposition 1) or at 2κ (Appendix A); each is the fragment of the same M_t by uniqueness of the interpolant. Collision resistance makes the sparse Merkle tree binding, so two verified answers under one anchor agree, and an honest maintainer validates every batch before advancing: integrity. Each segment contributes 2κ stored values, giving $2\kappa\lceil L_t/\kappa^2 \rceil$ in total. When $\kappa^2 \le L_t$, the segment tail is absorbed by $O(L_t/\kappa)$; UNDINELOG's frontier adds $O(\log \mathsf{N})$ labels. Together with $L_t = O(\ell_t + \log \mathsf{N})$, this gives compactness. □

Proof of Theorem 3.

Proof. (a) A reply of an honest maintainer decodes to its record, by the correctness of the backend and Proposition 4, except with probability δ, and so yields its true entry for every cell of the opening; a union bound over the honest maintainers gives the exception. A reply of a corrupted maintainer yields arbitrary field elements, or is malformed and discarded. For each opening cell (z,α) with segment polynomial P, the client thus holds values of $Q_\alpha(Z) = P(Z,\alpha)$, of degree less than κ_t, at distinct column identifiers, at least $\kappa_t + 2e$ of them when e are incorrect, because the registry is threshold-valid. Reed–Solomon decoding returns Q_α and hence the cell, exactly as in Theorem 2. The decoded

cells are the leaf and the labels of the path of x in M_t, so Verify accepts them under A_t. This is the only step that depends on the code: for another linear code applied per group, replace it by that code's decoder, with the registry size at which the decoder recovers every group from at most b_t wrong and m_t missing replies; parts (a) and (b) then hold verbatim.

(b) The corrupted maintainers see the request of step (ii) and the public behavior of the client. The registry does not depend on the position by step (i). The query q has the same length for x_0 and x_1: its length is a public function of ℓ_t, κ_t, and the backend's parameters. The fresh call to Query makes (k, q) one sample of the backend's query distribution for the pattern of x, and by query privacy the samples for the patterns of x_0 and x_1 are indistinguishable, including the auxiliary material and its observable length; every corrupted maintainer receives the same sample. Request transmission times follow the public schedule. By step (iii) the client stops collecting at a time fixed in advance, and by part (a) it then accepts, whatever the corrupted maintainers returned. By step (iv) nothing the client does afterwards depends on the replies. Its remaining public behavior is thus the same function of public values in both openings, and so is whether it accepts: by part (a) it accepts in both, except with probability $|\mathcal{R}_t|\,\delta$. A distinguisher between the two views therefore distinguishes the two query samples. □

Undoing a stitch: the procedure. The maintainer obtains its columns and rows for segments $(0,0)$ and $(1,1)$, that is, $L_{00,C}, R_{00,w}$ and $L_{11,C}, R_{11,w}$, under its own cID at threshold κ. They are either retained locally or recovered with the repair procedure of Section 3.4 from maintainers that advertise κ. The columns and rows for the other two segments follow entry by entry, for $0 \le i < \kappa$:

$$L_{10,C}[i] = \frac{L_{T,C}[2i] - E_0^Z(C)\, L_{00,C}[i]}{E_1^Z(C)},$$
$$L_{01,C}[i] = \frac{L_{T,C}[2i+1] - E_1^Z(C)\, L_{11,C}[i]}{E_0^Z(C)}, \qquad (14)$$
$$R_{01,w}[i] = \frac{R_{T,w}[2i] - E_0^W(w)\, R_{00,w}[i]}{E_1^W(w)},$$
$$R_{10,w}[i] = \frac{R_{T,w}[2i+1] - E_1^W(w)\, R_{11,w}[i]}{E_0^W(w)}. \qquad (15)$$

Undoing a stitch with helpers at threshold 2κ. An alternative undo procedure uses helpers that advertise 2κ when no registry at threshold κ is available. Define the partial stitches $P_{\mu *} = \sum_\nu E_\nu^W P_{\mu\nu}$ and $P_{*\nu} = \sum_\mu E_\mu^Z P_{\mu\nu}$, of degree below κ in one variable and below 2κ in the other. By the indicator property of the selectors, for $z \in G_\mu^Z$ and $\alpha \in G_\nu^W$, $P_T(z, W) = P_{\mu *}(z, W)$, $P_T(Z, \alpha) = P_{*\nu}(Z, \alpha)$, $P_{\mu *}(Z, \alpha) = P_{\mu\nu}(Z, \alpha)$, and $P_{*\nu}(z, W) = P_{\mu\nu}(z, W)$. Helper i with assignments (C_i, w_i) interpolates its stored row R_{T,w_i} on G_0^Z, the values $P_{0*}(z, w_i)$, and evaluates it at the requester's C, and interpolates its stored column L_{T,C_i} on G_0^W, the values $P_{*0}(C_i, \alpha)$, and evaluates it at the requester's w, returning $P_{0*}(C, w_i)$ and $P_{*0}(C_i, w)$ at $O(\kappa)$ operations. The requester decodes $P_{0*}(C, W)$, of degree below 2κ, from the first values at the distinct row labels w_i, and $P_{*0}(Z, w)$ from the second at the distinct column identifiers C_i, under the registry condition at threshold 2κ; evaluating them on G_0^W, G_1^W and G_0^Z, G_1^Z gives $L_{00,C}$, $L_{01,C}$, $R_{00,w}$, and $R_{10,w}$, and the remaining columns and rows follow from (5) and (6) as in (14) and (15), with $E_1^Z(C)$ and $E_1^W(w)$ nonzero as there. Each helper sends two field elements per stitched segment, a download of 4κ with 2κ correct replies, the same as from a registry at κ.

Appendix B. YPIR Parameters and Patched Preprocessing

Parameters and packing. We use LWE dimension 2^{11}, a 56-bit query modulus Q, plaintext modulus $p = 2^{14}$, and 12 KiB RLWE reply ciphertexts, with YPIR's estimated 128-bit security [12]. A column slot of κ entries is packed into $\lceil 256\kappa/14 \rceil$ symbols; an individual-entry slot uses 19. The matrix D is padded to a power-of-two row count of at least 2^{11} and a column count divisible by 2^{11}. Each reply ciphertext packs 2^{11} symbols. All measured matrices fit within the 2^{18} by 2^{18} dimensions for which the decryption-failure bound is 2^{-40} per honest ciphertext.

The query has one 7-byte coefficient per padded row. Its fresh packing key depends on the secret and fixed cryptographic parameters, independently of the anchor and record count. The seeded matrix $\mathbf{A}$ has one column a_r of 2^{11} values per row r of D. Each reply ciphertext requires 32 MiB of hint and 96 MiB of packing tables, derived from those hint columns.

Public pattern size. An entry contributes approximately $(256/14)(12\,\mathrm{KiB}/2^{11}) \approx 110$ bytes to the reply before rounding. With 7 bytes per query coefficient, the balance in Section 5.2 gives $2^{2\gamma} \approx 0.032\,\ell_t\kappa$. For $\ell_t > 0$ we use

$$\gamma(\ell_t, \kappa) = \max\Big\{0,\ 2\log_2\kappa - 1,\ \left\lfloor \frac{\lceil \log_2 \ell_t \rceil + \log_2 \kappa - 5}{2} \right\rfloor\Big\}.$$

The fixed packing key, group boundaries, and ciphertext padding account for differences between this balance and the full serialized sizes.

Patching the hint. Fix a pattern height γ. An append preserves existing group positions and pattern numbers. A group occurs in at most two adjacent patterns; a node above the pattern roots occurs in the contiguous range below its sibling. Compare each slot before and after the append and partition its changed rows into maximal ranges on which its old symbol vector c and new vector c' are each constant, with missing slots treated as zero. For such a range I, patch symbol column (u, k) of the hint by

$$\mathsf{Hint}_{:,\,(u,k)} \leftarrow \mathsf{Hint}_{:,\,(u,k)} + (c'_k - c_k) \sum_{r \in I} a_r.$$

The sum is computed once per range. After patching all ranges, the maintainer derives the packing tables from the updated hint.

Proposition 5 (Patched preprocessing)**.** *At fixed γ, the patched hint and packing tables equal a fresh preprocessing of the updated fragment, including when padding grows or a new upper-sibling slot is added.*

Proof. Let D, D' be the old and new logical matrices, padding D with zero rows and columns to the shape of D'. Existing slots retain their symbol offsets, and the seed gives the same a_r for each old row. On every range I of slot u, column (u, k) of $D' - D$ is the constant $c'_k - c_k$, and it is zero outside the changed ranges. The patches therefore sum to $\mathbf{A}(D' - D)$, giving $\mathbf{A}D' = \mathbf{A}D + \mathbf{A}(D' - D)$. The packing tables are a function of this hint and the fixed backend parameters. □

A change in γ changes the pattern layout and requires new preprocessing. At fixed κ, the rule increases γ at most once per quadrupling of the log; maintainers can prepare and patch that next layout before the public transition. A stitch or undo similarly requires preprocessing the fragment at its new threshold.

Appendix C. Additional Comparisons

TreePIR with the same PIR backend. We implement TreePIR's coloring over our YPIR backend and measure both systems on a 128 GiB log (2^{32} leaves) at $h = \kappa = 32$, where each opens a position with 32 replies (Table 5). A TreePIR server colors its class and writes the labels as its database; a UNDINELOG maintainer gathers its database from the primary columns of its fragment, with no auxiliary rows needed. The databases have equal hint, packing, request, and reply sizes, and the servers run the same backend steps once the database is arranged. TreePIR prepares one query per color class for the index of the node needed from that class. UNDINELOG prepares one query for the same pattern index across all maintainers and reuses its serialized bytes. In our YPIR configuration, the different TreePIR queries also use independent secrets because they share a public query matrix. At $h = 32$, the client decodes replies generated from the stored records under its keys, which decode in the same time as real replies to within 0.2%.

Threshold and pattern-size sweeps. Table 6 gives the full threshold sweep on the 4 GiB log and pattern-size sweep on the 64 GiB log. At fixed log size and threshold, larger patterns shorten the query and enlarge the reply, increasing reply-packing work. The public choice $\gamma = 16$ balances query coefficients and reply bytes; smaller patterns offer faster answers at the cost of longer queries. Requests also include the fixed packing key.

On the same pattern layout, BORG needs a column of s values per upper sibling and UNDINE one stored entry, giving 1,408 versus 589 record entries at $\kappa = 64$ on the 4 GiB log.

Answering above the stored threshold. Table 7 shows that serving at $\kappa = 512$ from a fragment stored at $\kappa = 64$ takes 19.9 µs per point response and 137 µs per 4 MiB range response, while stitching the entire fragment costs 63.2 ms.

TABLE 5. PRIVATE OPENING OF ONE POSITION ON A 128 GIB LOG (2^{32} LEAVES) AT $h = \kappa = 32$, WITH BOTH SYSTEMS OVER THE SAME YPIR BACKEND. THE CLIENT SENDS 32 SERIALIZED REQUESTS OF 910 KIB (448 KIB QUERY, 462 KIB PACKING KEY), RECEIVES 32 SERIALIZED REPLIES OF 444 KIB, AND VERIFIES THE OPENING AGAINST THE ROOT. SERVER ROWS GIVE ONE TREEPIR COLOR-CLASS SERVER AND ONE UNDINELOG MAINTAINER; PREPARATION STARTS FROM THE STORED DATA. TIMES ARE MEANS ON ONE CORE.

	UNDINELOG	TreePIR
Client, one opening		
PIR queries prepared	1	32
prepare the requests	48.3 ms	1.54 s
decode and verify the replies	321 ms	320 ms
total	370 ms	1.86 s
upload, all requests	28.4 MiB	28.4 MiB
download, all replies	13.9 MiB	13.9 MiB
One server		
PIR database	9.20 GiB	9.25 GiB
hint and packing tables	4.63 GiB	4.63 GiB
color the class	–	6.19 s
fill the database	12.1 s	13.2 s
build hint and packing tables	122 s	118 s
preparation, total	134 s	137 s
answer one query	1.19 s	1.31 s

TABLE 6. PRIVATE OPENING OF ONE POSITION BY ONE CID, BY THRESHOLD AND PATTERN SIZE. THE FIRST FOUR ROWS USE THE PUBLIC PATTERN SIZE ON A 4 GIB LOG; THE LAST FOUR VARY IT ON A 64 GIB LOG AT $\kappa = 64$, WHOSE PUBLIC CHOICE IS $\gamma = 16$. COL. COUNTS COLUMN SLOTS IN A RECORD. REQUESTS AND REPLIES ARE SERIALIZED SIZES; ANSWER TIME INCLUDES PARSING AND SERIALIZATION ON ONE CORE. PREPROCESSING TIME IS ESTIMATED FROM MEASURED KERNELS; UPDATE TIMES PIR PREPROCESSING AFTER A 4 MIB APPEND, EXCLUDING THE APPEND.

							Preprocessing		
κ	γ	Col.	Datab. (MiB)	Req. (KiB)	Reply (KiB)	Answer (ms)	size (MiB)	time (s)	Update (s)
8	12	129	1,020	686	120	252	1,280	20.8	9.90
16	13	65	512	574	120	190	1,280	15.6	9.07
32	13	17	256	574	60.0	95.5	641	7.93	4.51
64	14	9	128	518	72.0	100	769	7.64	5.32
64	14	9	2,050	1,360	72.0	241	769	30.1	5.58
64	15	17	2,050	910	120	299	1,280	32.3	8.92
64	16	33	2,050	686	228	440	2,440	40.7	16.6
64	17	65	2,050	574	456	699	4,870	60.1	33.0

Appendix D. Private Openings Across Maintainers

Figure 3 extends the single-server illustration of Figure 2 to the UNDINELOG records of Section 5.2, including the upper-sibling optimization.

TABLE 7. PUBLIC RESPONSES FROM ONE MAINTAINER ON A 12 MIB LOG, INCLUDING SERIALIZATION. FLY COMPUTES SHARES AT THE OPENED THRESHOLD FROM THE STORED FRAGMENT; STORED USES A FRAGMENT ALREADY AT THAT THRESHOLD. SIZE IS THE RANGE RESPONSE IN KIB, IDENTICAL FOR BOTH. STITCH CONVERTS THE WHOLE FRAGMENT FROM THE STORED TO THE OPENED THRESHOLD.

κ		Point (µs)		4 MiB range (µs, KiB)			Stitch
stored	opened	fly	stored	fly	stored	size	(ms)
8	16	11.5	8.08	1,320	484	520	4.52
8	32	16.1	7.97	1,310	193	259	6.62
8	64	24.0	7.66	1,270	89.7	130	8.24
16	32	11.3	7.87	527	195	259	2.20
16	64	15.4	7.60	554	90.0	130	3.87
16	128	22.6	7.29	573	43.1	65.6	7.15
32	64	11.3	7.69	234	90.1	130	1.85
32	128	14.9	7.29	256	43.1	65.6	5.20
32	256	20.9	7.01	273	23.4	33.5	16.4
64	128	10.9	7.30	112	43.8	65.6	3.73
64	256	14.2	7.15	127	24.5	33.5	15.2
64	512	19.9	6.77	137	14.9	17.5	63.2

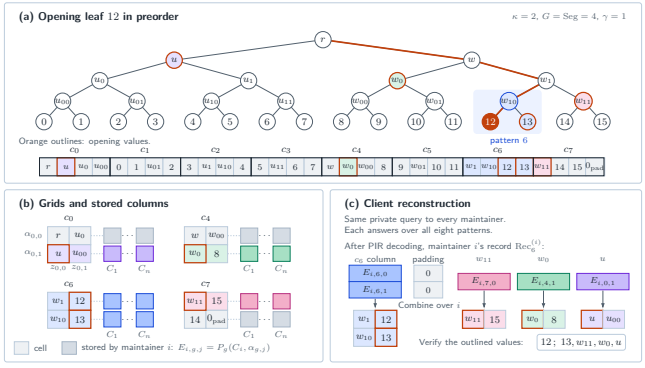

Figure 3. A private opening across UNDINELOG maintainers, with $\kappa = 2$ and $G = \mathsf{Seg} = 4$. (a) Pattern 6 covers leaves 12 and 13. (b) Only the four relevant segments are expanded; auxiliary rows are omitted. Polynomial P_g encodes segment c_g on its local grid (Z horizontal, W vertical; pale cells). Maintainer i stores $E_{i,g,j} = P_g(C_i, \alpha_{g,j})$: grid row j's polynomial in Z, evaluated at the maintainer's column identifier C_i outside the grid. Solid squares right of each row show these entries for maintainers 1 and n; 0_{pad} pads the array. (c) The decoded record contains seven entries: the c_6 column, two zeros for an unused column slot, and one share per upper sibling. Combining the records across maintainers recovers the whole c_6 segment and, for each upper sibling, only its grid row, one of the segment's κ rows: for u, the client interpolates $P_0(Z, \alpha_{0,1})$ from $E_{i,0,1}$ and evaluates it on the row. The opening takes the outlined values and is verified against the root.