BORG: Extendable Distributed Vector Commitments from Reconfigurable Erasure Codes

Nicolas Alhaddad, Eran Tromer, and Mayank Varia
Boston University

***Abstract*—Updatable vector commitments let users store and authenticate values contained within an evolving data vector. Existing work on updatable vector commitments studies how clients can store only the values and authentication proofs relevant to them, and can refresh stale opening proofs, with the help of an online maintainer that keeps the current vector and proof state. This work studies the complementary problem: how to decentralize the maintainer, in order to distribute the cost and availability requirements.**

We formalize this problem as extendable distributed vector commitments (EDVC). A public control plane, such as consensus or a trusted party, determines batches of updates and extensions of stored vector. Maintainers accept and apply a batch only after validating it against the current authenticated state. The resulting state is stored across many maintainer nodes, each of which holds a publicly assigned coded data fragment and updates it non-interactively. A stale client can obtain the current state, and a fresh authenticated opening, from the current maintainers (without replaying the updates history).

We construct the first EDVC protocol, called Borg, from two components: a sparse vector commitment, and a coded storage layer that can be updated by linear operations. This construction applies when all active positions lie within a known growing prefix of the vector. We then construct a coded storage layer with the operations needed for this setting, including single-position openings, aligned range openings, public updates, adding new nodes, repairing failed nodes, and moving storage responsibilities between nodes. This is then used to flexibly distribute the data alongside a sparse Merkle tree for authentication.

Our construction is particularly simple when the maintained data is an append-only public log (e.g., a blockchain's block archive, or certificate transparency logs). In this setting, distributed updates to the data and its authentication structure can be made just by broadcasting the new log entries, with no coordination across maintainer nodes. We empirically evaluate maintainers' cost for various workflows, showing that nodes can process hundreds of thousands of updates per second. We also show that node addition, repair of lost local state, and changes to the proof-serving threshold can all be supported efficiently.

1. Introduction

Many systems include long-term data archives that should remain accessible to future clients. Long after some data (e.g., block, blob, log entry or issued certificate) was recorded, a client may need to retrieve it, and furthermore, verify its correctness with respect to a commitment to the most recent version of the archive, which reflects additions or updates done in the interim. This paper asks how to decentralize the requisite storage and retrieval services among numerous maintainer nodes that may be unreliable and weak, while ensuring integrity, availability and freshness under updates.

Fixed-payload systems under the data-availability (DA) umbrella make available a single payload, whereas the archive clients in this work ask for something more: the value that the current authenticated history assigns to a position (e.g., a blockchain block or a certificate transparency log entry). Examples of fixed-payload systems include data availability sampling (DAS) [1], Ethereum's EIP-4844 [2], Peer Data Availability Sampling (PeerDAS) [3], Verifiable Information Dispersal (VID) its variants [4]–[6], DAS with repair [7], and decentralized blob storage systems such as Walrus [8]. In all of these, the client starts from one payload reference—a data root, commitment, blob ID, or dispersal certificate—and checks, samples, or recovers fragments of that referenced payload. Those checks prove that bytes match the referenced payload; they do not authenticate the map from history positions to payload references.[1]

One could apply the above mechanisms separately to each block or blob, but this is problematic. If every storage node stores fragments of every old payload, each node's storage grows linearly with the whole history. If different payloads are assigned to different node subsets, then an old payload remains available only while its subset remains available, and an attacker can target that subset. This is the tradeoff identified by incremental decentralized data archival (iDDA) [9]. Furthermore, the problem is dynamic: maintainers may later leave, join, or change how much storage they are willing to contribute, but a per-payload assignment fixes who serves each old object. Fixed-payload DA, VID, and AVID-style systems also make fragments of one encoded payload publicly checkable against the payload reference [4]–[8], [10]. This is necessary for sampling or recovering fragments of that payload, but it fixes how that

1. See Section 11 for further discussion of these related works.

object is encoded, checked, and recovered. Changing the reconstruction rule, encoded payload size, or authenticated fragment set changes the verification material and therefore creates a new reference.

To maintain the flexibility to redistribute and re-encode archive data, as the set and capabilities of maintainer nodes evolve, we thus wish to publish commitments to the archive data, but not to the specific way it is currently distributed. Maintainer nodes each keep some coded fragments of the archive, and clients can query (a sufficiently large subset of the) maintainer nodes in order to recover a desired portion of the archive. This raises a difficulty: maintainer nodes' responses (and the underlying local state) are not directly committed to, and thus malicious nodes may send inconsistent or incorrect results. Nonetheless, the client must be able to combine responses, discard incorrect ones, and recover the correct archived value alongside the requisite authentication data. Furthermore, this should be done without reconstructing the whole archive's history, and with low overhead when retrieving large chunks of the archive.

Extendable and distributed commitments. A Vector Commitment (VC) is the natural authenticated object for the archive values themselves. A VC commits to an indexed vector and lets a client verify an *opening*: a value at one position, or here a contiguous range, together with authentication data checked against the public commitment. An extendable VC lets new archive entries be appended into the next positions of the same committed vector. We call the current commitment together with the current vector length the *anchor*. With the current anchor, a client can verify an old position or range under one current commitment rather than tracking a separate reference for every old payload.

A standard VC still assumes a manager that stores the current vector and the auxiliary proof data used to answer future queries. We call this manager-side information the *maintained state*. Existing updatable-VC work refreshes stale client openings when such a manager already keeps the live state [11]–[13]; recent distributed VC work lets parties that already hold fixed vector pieces jointly compute commitments and position proofs [14]. Neither makes the maintained state repairable, movable, and reconfigurable as history grows.

We thus formalize that maintainer problem as an *Extendable Distributed Vector Commitment (EDVC)*: a cryptographic primitive for an extendable VC where fresh openings are produced from data that is encoded and distributed across many maintainer nodes (rather than a centralized manager). Each maintainer node publishes an *advertisement* for the coded *data fragment* it locally stores, which may vary in size according to that node's resources. Clients collect such advertisements in a *local registry*, and can retrieve any value from the stored data by making queries to nodes in their registry, and then combining the results. The number of nodes that need to be queried depends on the advertisement parameters, and the success probability depends on what fraction of the advertising nodes are honest and responsive. Furthermore, EDVC supports updates (given some trusted/consensus source of updates), restoring/repairing maintainer nodes' data fragments, onboarding new nodes, and reconfiguring parameters — all of which can be done by similarly querying a threshold of maintainers (or, in some cases, wholly locally).

Our schemes. BORG is our EDVC construction for vectors whose populated positions form a growing prefix. For the vector commitment, BORG uses a sparse Merkle tree, so the public anchor binds the vector values themselves and clients verify ordinary hash openings for positions or ranges.

To make those openings distributed and reconfigurable, BORG introduces a reconfigurable erasure-coded storage layer for the maintained state, called the *coded array*. It is an array of field elements that are distributed across maintainers. Importantly, the encoding itself does not directly depend on the public anchor commitment. This allows the data to be re-dispersed (e.g., if more maintainer nodes join the network, or if nodes' storage capacities change) without needing a new commitment anchor—and therefore, without the need for a central party who knows the entire dataset in order to produce another commitment anchor.

As an EDVC, BORG also supports decentralized updates. A maintainer does not need the whole maintained state to update its own fragment. Each candidate batch carries enough authenticated old-state material for the maintainer to check the affected old values against the current sparse-Merkle anchor. After this check succeeds, the batch is accepted, and the maintainer computes the changed Merkle labels and hence the public field-element differences in the maintained state. Because each advertised fragment is a linear projection of that maintained state, the maintainer can apply the corresponding projected differences to its own old fragment, obtaining the new fragment without reconstructing the whole vector or trusting the batch producer to compute the state correctly.

The flagship scheme in this paper is BORGLOG, which is a specialization of BORG to append-only public logs such as blockchain block archives and transparency logs. Such logs already publish entries in a public order. BORGLOG can run as an overlay on that ordinary stream: maintainers independently track entries, update their advertised fragments from the new entries, and answer fresh openings for old positions or ranges under the current anchor. The log does not need to publish BORG-specific proofs or coordinate with maintainers. A new or repaired maintainer can obtain only the fragment it advertises from current maintainers, rather than downloading and replaying the whole archive.

Construction overview. To support maintainers with different storage and availability needs, BORG needs the data-fragment layout itself to be reconfigurable. The code represents each segment by a univariate polynomial whose evaluations are the segment contents. Maintainers do not store this polynomial directly. Instead, for a chosen reconstruction threshold, the same univariate polynomial is decomposed into a bivariate representation with different row and column degree bounds. Each maintainer stores one primary column of this bivariate representation and one or more auxiliary rows used for repair, onboarding, and range retrieval.

Reconfiguring system parameters (e.g., threshold number of nodes needed for retrievals) changes the decomposition, and therefore changes the advertised projections stored by maintainers—while leaving the segment contents unchanged.

The code also supports increasing the segment size when the current segment size is no longer the right operating point for the chosen threshold. We choose the segment layout and evaluation domains so that adjacent segments can be stitched locally: each maintainer combines its stored primary columns and auxiliary rows from the two source segments into the corresponding primary column and auxiliary rows for the larger segment using public algebraic selector formulas. The segment-size change is therefore local. Together, these operations let the system repair lost fragments, onboard new maintainers, and change storage or reconstruction parameters with minimal overhead.

Ordinary fixed-block erasure codes recover one fixed encoded payload; BORG's array instead supports single-position reads, aligned range reads, accepted appends or rewrites, repair, onboarding, and later changes to how much maintainers store or how many replies are needed, all under the same vector commitment and without decoding the whole vector or changing the public commitment. Clients combine maintainer responses, then accept only if the resulting ordinary sparse Merkle tree opening verifies under the anchor.

Implementation. We implement BORG and provide experimental results for end-to-end workflows. For example, in the BORGLOG measurements, each maintainer independently processes a 4 MiB append in 295 ms, matching Bitcoin's maximum 4 MiB block scale. A client can ask 192 maintainers for any 4 MiB contiguous range, tolerate 64 corrupted maintainers, and still reconstruct and verify the range in 6.039 s. The same measured storage growth extrapolates to about 64 GiB of local maintainer state for a 1 TiB Bitcoin-scale archive, or about 32 GiB if the maintainer keeps only the part needed to answer openings and not the extra repair/onboarding state.

Contributions. We make the following contributions.

- We define EDVC, a cryptographic primitive for extendable vector commitments whose openings can be produced by many maintainers rather than one manager. The syntax separates public anchors and accepted updates from maintainer advertisements and local registries.
- We formalize the EDVC guarantees. Honest execution matches the centralized VC; verified openings are bound to the public anchor; and a batch that fails validation against the current anchor cannot advance it. Availability is conditional on the selected local registry: with enough correct replies, the requested opening, advertised fragment, or full vector can be reconstructed; otherwise the request may fail but cannot change what verifies under the anchor. The state associated with each advertisement must also scale with the current vector length and its advertised parameters, rather than with maximum vector capacity.
- We show how to construct EDVC from a VC and an unauthenticated coded array when old values and their opening proofs, together with new values, publicly determine each change to the VC's maintained state without reconstructing it.
- We construct BORG for vectors whose active positions lie in a growing prefix, using a sparse Merkle tree for authentication and a reconfigurable coded array. The coded array supports single-position and aligned range reads, local updates, repair, onboarding, and supported changes to its reconstruction threshold and segment size without changing the committed vector. We prove that BORG realizes EDVC under the stated registry and fault conditions.
- We specialize BORG to BORGLOG, a non-invasive overlay for append-only public logs. We implement the coded array, BORG, and BORGLOG and evaluate fixed-array operations, arbitrary rewrites, appends, verified openings with corrupted responses, repair, onboarding, and reconfiguration.

Organization. Section 2 states the system model and registry assumptions, and Section 3 defines the EDVC syntax and guarantees. Section 4 gives a generic EDVC construction from a vector commitment and an unauthenticated coded array. Sections 5 and 6 construct the fixed-size and extendable coded arrays, including repair, reconfiguration, and optional retrievability audits. Section 7 combines the coded array with a sparse Merkle tree to obtain BORG, Section 8 specializes BORG to append-only archives, and Section 9 establishes the security analysis. Section 10 evaluates the coded array, BORG, and BORGLOG. Finally, Section 11 reviews related work, Section 12 discusses applications and open problems, and Section 13 concludes.

2. System Model

This section describes who takes part in an EDVC, what public information they use, and what failures the protocol must handle. It also explains how the vector changes and how each party chooses the maintainers it will use.

Participants. An EDVC deployment has three principal roles. The *update sequencer* is an external control plane, such as a centralized public log or consensus protocol, that places candidate update batches in a common public order. It is trusted only to give honest parties the same ordered sequence, not to validate the batches. A *maintainer* publishes an advertisement for its own state (a data fragment) it claims to store, validates each ordered candidate batch, updates its own state when validation succeeds, and answers retrieval or maintenance requests. A *caller* selects advertisements to form a local registry and invokes an EDVC operation. A caller requesting an authenticated opening is a *client*; a maintainer acts as a caller when repairing, onboarding, or reconfiguring an advertised responsibility. For a particular invocation, the selected maintainers are called *helpers*, and the caller that chooses the registry and its fault budgets is its *registry owner*. Different callers may select different registries; EDVC does not require agreement on one global registry.

Logical vector and public authenticated state. Fix a security parameter λ, a value space $\mathcal{X}$, and a public maximum vector length $\mathsf{N} \in \mathbb{N}_{>0}$. Time is indexed by $t = 0, 1, 2, \ldots$. At time t, the logical vector is

$$x_t \in \mathcal{X}^{\ell_t},$$

where $0 \le \ell_t \le \mathsf{N}$, and the public anchor is

$$A_t = (\mathsf{Com}_t, \ell_t),$$

where Com_t authenticates x_t. The public anchor is not a registry and does not specify which maintainers hold local state.

Update sequencer and batches. A candidate batch for the transition from time t to time $t+1$ has the form

$$B_t = (\ell_{t+1}, \mathsf{updates}_t, \mathsf{oldAuth}_t).$$

Here $\ell_t \le \ell_{t+1} \le \mathsf{N}$, and $\mathsf{updates}_t = ((i_1, v_1), \ldots, (i_m, v_m))$ assigns distinct positions $i_j \in [0, \ell_{t+1})$ to values $v_j \in \mathcal{X}$. Every newly exposed position must be assigned:

$$[\ell_t, \ell_{t+1}) \subseteq \{i_1, \ldots, i_m\}.$$

The batch determines the next vector by

$$x_{t+1}[i] = \begin{cases} v_j & \text{if } i = i_j \text{ for some } j \in \{1, \ldots, m\}, \\ x_t[i] & \text{otherwise.} \end{cases}$$

The second case occurs only for $i < \ell_t$. Assignments below ℓ_t rewrite old positions, while assignments in $[\ell_t, \ell_{t+1})$ initialize newly exposed positions.

Fix a public default value $0_{\mathcal{X}} \in \mathcal{X}$. For purposes of validating an extension, positions outside the active prefix have this default value. Thus, the old value of every newly exposed position is $0_{\mathcal{X}}$, even though that position is not yet part of x_t.

The component $\mathsf{oldAuth}_t$ contains the authenticated old-state material needed to validate the batch. For every rewritten position $i < \ell_t$, it identifies the old value and supplies evidence that this value is authenticated by A_t. Any authenticated material needed for the default values of newly exposed positions must be included in, or derivable from, $\mathsf{oldAuth}_t$. The representation of this material is left unspecified.

We write $\mathsf{ValidBatch}(A_t, B_t) = 1$ when $\ell_t \le \ell_{t+1} \le \mathsf{N}$, the indices are in range and distinct, every newly exposed position is assigned, the old authenticated material verifies against A_t, and applying the batch deterministically yields one next anchor A_{t+1} for x_{t+1}. The update sequencer is trusted only to put candidate batches in one public order; it is not trusted to make a batch valid, validate $\mathsf{oldAuth}_t$, compute A_{t+1}, or certify that an anchor is correct. Honest maintainers advance only after this local validation succeeds. We call a candidate batch *accepted* when it passes this validation.

Active-prefix scope. We focus on the case where the authenticated data is an extensible history, represented as an updatable vector with an active prefix. The public length ℓ_t identifies the positions in the active prefix. We do not consider arbitrary sparse vectors without a public active prefix.

Maintainer advertisements. A maintainer claims that it holds local state by publishing

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

Here $\mathsf{mID} \in \{0,1\}^\lambda$ is the maintainer identifier used by the surrounding deployment to identify or contact the maintainer. The identifier $\mathsf{cID} \in \{0,1\}^\lambda$ is the claim identifier for the local state named by the advertisement. The value A is the public anchor, $\kappa \in \mathbb{N}_{>0}$ is the reconstruction threshold, and $\mathsf{Seg} \in \mathbb{N}_{>0}$ is the segment size chosen by the maintainer. The reconstruction threshold is the minimum number of correct compatible replies needed to complete an operation. The mechanism for choosing mID is deployment-specific; a maintainer may choose cID at random. The mechanisms for publishing, authenticating, discovering, and selecting advertisements are outside the EDVC model.

Local registries. A caller may be a client, an existing maintainer, or a recovering maintainer. It selects a compatible finite set of advertisements according to its own policy. Different callers may select different sets. At time t, a local registry is

$$\mathcal{R}_t = (A_t, \mathcal{A}_t, \kappa_t, \mathsf{Seg}_t, \mathsf{numHelpers}_t, \mathsf{errTol}_t, \mathsf{missTol}_t)\ .$$

Here A_t is the selected public anchor, $\mathcal{A}_t$ is the selected set of advertisements, $\kappa_t \in \mathbb{N}_{>0}$ is the selected reconstruction threshold, $\mathsf{Seg}_t \in \mathbb{N}_{>0}$ is the selected segment size, $\mathsf{numHelpers}_t \in \mathbb{N}_{\ge 0}$ is the number of selected advertisements, and $\mathsf{errTol}_t, \mathsf{missTol}_t \in \mathbb{N}_{\ge 0}$ are the Byzantine and fail-stop fault budgets chosen by the registry owner. We write

$$\mathsf{Claims}(\mathcal{R}_t) = \{\mathsf{cID} : (\mathsf{mID}, \mathsf{cID}, A, \kappa, \mathsf{Seg}) \in \mathcal{A}_t\}.$$

The registry is *well formed*, written $\mathsf{ValidReg}(\mathcal{R}_t) = 1$, if

- every advertisement in $\mathcal{A}_t$ has anchor A_t, threshold κ_t, and segment size Seg_t;
- the maintainer identifiers are distinct;
- the claim identifiers are distinct; and
- $|\mathcal{A}_t| = \mathsf{numHelpers}_t$.

Consequently, each $\mathsf{cID} \in \mathsf{Claims}(\mathcal{R}_t)$ identifies a unique selected advertisement and hence a unique corresponding mID within the registry.

Registry threshold validity. A Byzantine maintainer may deviate arbitrarily and counts against errTol_t. A fail-stop maintainer stops sending replies and counts against $\mathsf{missTol}_t$. A well-formed registry is *threshold-valid* for these fault budgets when

$$\mathsf{ThresholdValid}(\mathcal{R}_t) = 1 \iff \mathsf{numHelpers}_t \ge \kappa_t + 2\mathsf{errTol}_t + \mathsf{missTol}_t.$$

If enough helpers answer correctly, the requested operation can be completed. When $\mathsf{missTol}_t = 0$, threshold validity reduces to $\mathsf{numHelpers}_t \ge \kappa_t + 2\mathsf{errTol}_t$. When $\mathsf{errTol}_t = \mathsf{missTol}_t = 0$, the registry need only contain the κ_t maintainers required for reconstruction. These fault budgets are local choices made by the registry owner.

The usual $3f+1$ committee is the special case $\kappa_t = f+1$, $\mathsf{errTol}_t = f$, and $\mathsf{missTol}_t = 0$. Selected maintainer helpers do not need to coordinate: each request is one round from the caller to the selected helpers, followed by local reconstruction and verification by the caller.

Reconfiguration. A caller may replace its local registry by another registry with different maintainers, κ, Seg, or fault budgets. For a client, this changes how it fetches elements of the committed vector, but a successful fetch has the same result. For a maintainer, this may change the size or shape of its data fragment. A registry change does not change the committed vector or public anchor. Those change only when an accepted batch produces a new anchor.

For an old registry $\mathcal{R}$ and a replacement registry $\mathcal{R}'$ at the same public anchor, the registry owner may record which new claims continue old advertised responsibilities using a partial map

$$\mathsf{succ}_{\mathcal{R}\to\mathcal{R}'} : \mathsf{Claims}(\mathcal{R}) \rightharpoonup \mathsf{Claims}(\mathcal{R}').$$

If $\mathsf{succ}_{\mathcal{R}\to\mathcal{R}'}(\mathsf{cID}) = \mathsf{cID}'$, then cID' is the replacement registry's identifier for the responsibility previously identified by cID. When the identifier is unchanged, we write

$$\mathsf{Preserved}(\mathcal{R}, \mathcal{R}', \mathsf{cID}) = 1 \iff \mathsf{succ}_{\mathcal{R}\to\mathcal{R}'}(\mathsf{cID}) = \mathsf{cID}.$$

A fresh target claim that is not named by this successor map is an onboarded responsibility rather than the continuation of an old responsibility.

Threat model. The only assumption about the update sequencer is consistent ordering: honest parties receive the same candidate batches in the same order. Candidate batch contents may be adversarial, including malformed or invalid updates. The update sequencer is not trusted for batch validity, old-state authentication, or public anchor correctness.

The adversary can corrupt maintainer nodes, publish misleading advertisements, create and advertise many maintainer identities, and control the network. A corrupted maintainer may fail to answer, return malformed or incorrect data, equivocate across callers, corrupt its local state, or advertise local state that it does not hold. The fault budgets apply to the selected maintainers identified by the mID fields of the advertisements in $\mathcal{A}_t$ for one request, not to all advertised maintainers.

The caller sends at most one request to each selected maintainer and uses at most one response from each selected maintainer identity. Selected maintainers do not coordinate as part of an EDVC request. Replies are attributable to maintainer identities. The adversary may delay, reorder, duplicate, inject, modify, or drop messages. A message not attributable to a selected maintainer identity is not a response from that maintainer. Multiple inconsistent messages from one identity count as the behavior of that identity.

In a synchronous setting, the caller may stop waiting after a deadline based on the assumed bound on message delay. In an asynchronous setting, there is no fixed deadline, and the caller chooses whether to keep waiting. In either case, a selected honest maintainer whose reply is not used counts as non-responding for that request. Correctness and availability require enough honest selected maintainers and sufficiently few corrupted or non-responding selected maintainers, as specified by the registry threshold condition. EDVC does not provide maintainer selection or Sybil resistance; these are supplied outside the EDVC protocol.

3. EDVC Syntax and Security

This section defines the EDVC abstraction. Definition 1 gives its formal syntax. We then relate this syntax to dynamic vector commitments before stating the correctness, availability, integrity, and compactness requirements for an EDVC.

3.1. Syntax

The syntax uses the public anchors, candidate batches, advertisements, and local registries defined in Section 2. For $\mathsf{cID} \in \mathsf{Claims}(\mathcal{R}_t)$, the symbol $\mathsf{dataFragment}_{\mathsf{cID},t}$ denotes the data fragment held at time t by the maintainer advertising cID.

Definition 1 (EDVC syntax)**.** *An* extendable distributed vector commitment *scheme is a tuple*

$$\begin{aligned}\mathsf{EDVC} = (&\mathsf{Setup}, \mathsf{Commit},\\ &\mathsf{UpdateCommitment}, \mathsf{Open}, \mathsf{Verify},\\ &\mathsf{InitDataFragment}, \mathsf{UpdateDataFragment},\\ &\mathsf{RepairDataFragment}, \mathsf{ReconfigDataFragment},\\ &\mathsf{Decode}, \mathsf{OpenShare}, \mathsf{Combine}).\end{aligned}$$

The algorithms have the following interfaces.

- $\mathsf{Setup}(1^\lambda, \mathsf{N}) \to \mathsf{pp}$.
  This randomized setup algorithm outputs public parameters pp *for vectors of maximum length* N.
- $\mathsf{Commit}(\mathsf{pp}, x) \to (\mathsf{Com}, \mathsf{data})$.
  Here $x \in \mathcal{X}^\ell$ is a vector of length ℓ where $\ell \le \mathsf{N}$. The output Com *is the vector commitment to x, and* data *is the centralized auxiliary opening state sufficient to produce openings. The corresponding EDVC public anchor is*
  $$A = (\mathsf{Com}, \ell).$$
- $\mathsf{UpdateCommitment}(\mathsf{pp}, A_t, B_t) \to A_{t+1}$ *or* $\perp$.
  Here $A_t = (\mathsf{Com}_t, \ell_t)$ is the current public anchor, and
  $$B_t = (\ell_{t+1}, \mathsf{updates}_t, \mathsf{oldAuth}_t)$$
  is the candidate batch ordered by the external control plane. The map $\mathsf{updates}_t$ gives the assigned positions and new values, and ℓ_{t+1} gives the size of the new vector. For every rewritten old position $i < \ell_t$, $\mathsf{oldAuth}_t$ contains the old value and an opening proof for that old value under A_t.
  The output is the next public anchor A_{t+1}, or $\perp$ if the batch is not valid for A_t. A batch for which the output is not $\perp$ is accepted.
- $\mathsf{Open}(\mathsf{pp}, A_t, \mathsf{data}_t, q) \to (a, \pi_q)$ *or* $\perp$.

This is the ordinary centralized VC opening algorithm. The input q is a supported query, $A_t = (\mathsf{Com}_t, \ell_t)$ is the public anchor where Com_t is the commitment to the vector and ℓ_t is the length of the vector, and data_t is the centralized auxiliary opening state for the vector. The output is an answer a and an opening proof π_q.
In a deployed EDVC system, clients do not expect one online maintainer to run this algorithm. It is the ideal centralized reference that the distributed OpenShare *and* Combine *path must reproduce.*

- $\mathsf{Verify}(\mathsf{pp}, A_t, q, a, \pi_q) \in \{0, 1\}$.
This is the ordinary VC verification algorithm. It checks whether (a, π_q) is a valid authenticated answer to query q under commitment Com_t and active length ℓ_t where $A_t = (\mathsf{Com}_t, \ell_t)$. It does not depend on any registry.
- InitDataFragment *initializes the state of a maintainer using the interface*

$$\mathsf{InitDataFragment}(\mathsf{pp}, \mathsf{advert}_0) \\ \to \mathsf{dataFragment}_{\mathsf{cID},0} \text{ or } \bot.$$

This algorithm is run by the maintainer that publishes the advertisement

$$\mathsf{advert}_0 = (\mathsf{mID}, \mathsf{cID}, A_0, \kappa, \mathsf{Seg}).$$

It initializes that maintainer's data fragment for the advertised responsibility cID *at the initial anchor A_0. In the online setting, A_0 is the anchor of the empty vector. A nonempty initial vector can be modeled by an offline bootstrap or by applying an initial sequence of accepted batches.*

- UpdateDataFragment *updates the state of a maintainer using the interface*

$$\mathsf{UpdateDataFragment}(\mathsf{pp}, \mathsf{advert}_t, \mathsf{advert}_{t+1}, \\ \mathsf{dataFragment}_{\mathsf{cID},t}, B_t) \\ \to \mathsf{dataFragment}_{\mathsf{cID},t+1} \text{ or } \bot.$$

This algorithm is run by an active maintainer to update its own data fragment after an accepted batch. The old advertisement is

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

and the updated advertisement is

$$\mathsf{advert}_{t+1} = (\mathsf{mID}, \mathsf{cID}, A_{t+1}, \kappa, \mathsf{Seg}).$$

The maintainer identifier mID*, claim identifier* cID*, reconstruction threshold κ, and segment size* Seg *are unchanged. Only the public anchor and the data fragment change. The input $\mathsf{dataFragment}_{\mathsf{cID},t}$ is the old data fragment for* cID*. The output is the updated data fragment for the same advertised responsibility after applying B_t. Changes to the reconstruction threshold κ or segment size* Seg *are handled by* ReconfigDataFragment*, not by ordinary* UpdateDataFragment*.*

- RepairDataFragment *repairs the state of a maintainer using the interface*

$$\mathsf{RepairDataFragment}(\mathsf{pp}, A_t, \mathcal{R}_t, \mathsf{cID}, \\ \{(\mathsf{cID}_h, \eta_{\mathsf{cID}_h})\}_{\mathsf{cID}_h \in I}) \\ \to \mathsf{dataFragment}_{\mathsf{cID},t} \text{ or } \bot.$$

This algorithm repairs the data fragment for $\mathsf{cID} \in \mathsf{Claims}(\mathcal{R}_t)$. The repairing maintainer uses the local registry $\mathcal{R}_t$ to choose other maintainers to contact. The set

$$I \subseteq \mathsf{Claims}(\mathcal{R}_t)$$

contains the claim identifiers of maintainers that replied. For each $\mathsf{cID}_h \in I$, the packet η_{cID_h} is the repair response returned by the maintainer advertising claim cID_h. The algorithm may use error correction according to the registry parameters $(\kappa_t, \mathsf{errTol}_t, \mathsf{missTol}_t)$.

- ReconfigDataFragment *updates the state of a maintainer to be compatible with a new advertisement using the interface*

$$\mathsf{ReconfigDataFragment}(\mathsf{pp}, A_t, \mathcal{R}_t, \mathsf{advert}^\star, \\ \{(\mathsf{cID}_h, \eta_{\mathsf{cID}_h})\}_{\mathsf{cID}_h \in I}) \\ \to \mathsf{dataFragment}_{\mathsf{cID}^\star,t} \text{ or } \bot.$$

This algorithm materializes a data fragment for the target advertisement

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

It is run by the maintainer that wants to serve the advertised responsibility $\mathsf{cID}^\star$ under the target parameters. This covers onboarding a new claim, repairing a claim under a new registry, or changing the advertised threshold or segment size. When the call accompanies a registry replacement $\mathcal{R}_t \to \mathcal{R}'_t$ at the same public anchor, the target advertisement is selected in $\mathcal{R}'_t$. If the call reconfigures an existing responsibility $\mathsf{cID} \in \mathsf{Claims}(\mathcal{R}_t)$, the registries record $\mathsf{succ}_{\mathcal{R}_t \to \mathcal{R}'_t}(\mathsf{cID}) = \mathsf{cID}^\star$. The special case $\mathsf{Preserved}(\mathcal{R}_t, \mathcal{R}'_t, \mathsf{cID}) = 1$ has $\mathsf{cID}^\star = \mathsf{cID}$. For onboarding, $\mathsf{cID}^\star$ is fresh and is not named by the successor map.
The old registry $\mathcal{R}_t$ supplies the helper maintainers:

$$I \subseteq \mathsf{Claims}(\mathcal{R}_t).$$

The output is the data fragment for the target advertised responsibility $\mathsf{cID}^\star$ at the same public anchor A_t.

- Decode *reconstructs the full vector using the interface*

$$\mathsf{Decode}(\mathsf{pp}, A_t, \mathcal{R}_t, \{(\mathsf{cID}_h, \eta_{\mathsf{cID}_h})\}_{\mathsf{cID}_h \in I}) \\ \to x_t \text{ or } \bot.$$

This algorithm reconstructs the full vector committed by A_t from maintainer packets. It may be run by any party that wants to reconstruct the entire vector from a threshold-valid registry satisfying the compatibility predicate defined below. The set $I \subseteq \mathsf{Claims}(\mathcal{R}_t)$ contains the claim identifiers indexing the maintainer packets used for reconstruction. This operation is included for completeness and recovery; the normal read path uses

OpenShare *and* Combine *rather than decoding the full vector.*

- OpenShare *produces an opening share using the interface*

$$\mathsf{OpenShare}(\mathsf{pp}, \mathsf{advert}_t, \mathsf{dataFragment}_{\mathsf{cID},t}, q) \to \sigma_{\mathsf{cID},q} \text{ or } \perp.$$

This algorithm is run by a maintainer answering query q. The advertisement

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

identifies the public anchor and the advertised responsibility. The input $\mathsf{dataFragment}_{\mathsf{cID},t}$ *is the maintainer's data fragment for* cID*. The output $\sigma_{\mathsf{cID},q}$ is one maintainer reply for query q.*

- Combine *constructs an authenticated answer using the interface*

$$\mathsf{Combine}(\mathsf{pp}, A_t, \mathcal{R}_t, q, \{(\mathsf{cID}_h, \sigma_{\mathsf{cID}_h,q})\}_{\mathsf{cID}_h \in I}) \to (a, \pi_q) \text{ or } \perp.$$

This algorithm is run by a client that wants an authenticated answer to query q. The client uses its local registry $\mathcal{R}_t$ to choose maintainers to contact. The set

$$I \subseteq \mathsf{Claims}(\mathcal{R}_t)$$

contains the claim identifiers of maintainers that replied. The algorithm combines their opening shares into the same kind of answer/proof pair that centralized Open *would output. It may discard malformed replies and may use error correction according to the registry parameters $(\kappa_t, \mathsf{errTol}_t, \mathsf{missTol}_t)$. The client accepts the output only if*

$$\mathsf{Verify}(\mathsf{pp}, A_t, q, a, \pi_q) = 1, \quad \text{where } A_t = (\mathsf{Com}_t, \ell_t).$$

Supported parameters and compatible registries. Along with its algorithms, an EDVC scheme specifies two deterministic public predicates

$$\mathsf{SupportedParams}(\mathsf{pp}, \kappa, \mathsf{Seg}), \quad \mathsf{CompatibleReg}(\mathsf{pp}, \mathcal{R}).$$

The first identifies the advertised parameter pairs supported by the scheme. The second identifies registries whose selected advertisements can be used together by the registry-based algorithms. Whenever $\mathsf{CompatibleReg}(\mathsf{pp}, \mathcal{R}_t) = 1$, both

$$\mathsf{ValidReg}(\mathcal{R}_t) = 1, \quad \mathsf{SupportedParams}(\mathsf{pp}, \kappa_t, \mathsf{Seg}_t) = 1$$

must hold. A caller can evaluate both predicates from public information and rejects an incompatible registry before invoking a registry-based algorithm. Correctness, availability, and compactness are required only for supported parameters and compatible registries.

The base syntax intentionally has no VerifyShare algorithm. A single maintainer reply is not, by itself, a proof that the maintainer has updated its entire data fragment correctly. Such a verifier would require an additional certificate system. In the base interface, robustness comes from collecting enough replies from the caller's registry, using error correction when the registry has enough redundancy, and accepting a client-visible answer only after the final Verify check against the public anchor.

Relation to dynamic vector commitments. In standard dynamic and updatable vector commitments [11], [12], a manager holds the complete auxiliary opening state data_t and runs an update of the schematic form

$$\mathsf{Update}_{\mathsf{pp}}\big(\mathsf{Com}_t, \{(i, v_i, v_i')\}_{i \in S_t}, \mathsf{data}_t\big) \to (\mathsf{Com}_{t+1}, U_t, \mathsf{data}_{t+1}).$$

The old values v_i are ordinary manager inputs and need no accompanying opening proofs because the manager holds the current opening state; transition validity is therefore manager-local. The object U_t is the additional public information used to refresh stale opening proofs, while the updated positions and values are separate public update data.

EDVC instead distributes the role of data_t across advertised maintainer data fragments. The external control plane orders each candidate batch $B_t = (\ell_{t+1}, \mathsf{updates}_t, \mathsf{oldAuth}_t)$, and every honest party validates that batch against the current anchor before advancing. For a rewritten position, $\mathsf{oldAuth}_t$ supplies the old value and the evidence needed to authenticate it without access to the complete opening state. Without this evidence, a maintainer cannot determine from its own data fragment whether the claimed old value is correct, and applying a difference derived from an incorrect old value would produce an incorrect updated data fragment. Thus B_t is distinct from U_t: B_t contains the proposed updates and authenticated old-state material, whereas U_t is the stale-proof refresh object from the dynamic-VC interface. The algorithm UpdateCommitment validates B_t and computes the next public anchor; after acceptance, UpdateDataFragment applies the same batch to one preserved data fragment. The three roles are therefore separate: the control plane orders candidate batches, UpdateCommitment validates a batch and computes the next anchor, and UpdateDataFragment refreshes one maintainer's data fragment.

For retrieval, Open retains its standard VC meaning: it is the centralized reference algorithm for a party holding data_t. In EDVC, maintainers produce OpenShare replies, and a client runs Combine to obtain an answer/proof pair and checks it with Verify against the public anchor. The base syntax therefore omits the stale-proof ProofUpdate interface: a stale client requests a fresh distributed opening under the current anchor.

Robustness is provided at the registry level. A caller collects enough replies, uses error correction when its registry contains the required redundancy, and accepts an answer only after the final Verify check. One maintainer reply does not certify that the maintainer has updated its entire data fragment; such a claim would require an additional certificate system. Accordingly, the base syntax has no VerifyShare algorithm.

3.2. Correctness, Availability, and Integrity

In this subsection, we define the security guarantees provided by an EVDC. This section follows the model of Section 2 and the syntax of Definition 1. In particular, $A_t = (\mathsf{Com}_t, \ell_t)$ is the public anchor after the first t accepted batches, x_t is the logical vector authenticated by A_t, and B_t is the accepted batch for the transition from time t to time $t+1$. The notions of well-formed local registry, threshold-valid local registry, advertisement, and claim identifier are those of Section 2; supported parameters and compatible registries are defined above.

Valid histories. A *valid history* to time t is generated as follows. First,

$$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{N}).$$

Then an initial active-prefix vector with $|x_0| \le \mathsf{N}$ is committed:

$$\mathsf{Commit}(\mathsf{pp}, x_0) \to (\mathsf{Com}_0, \mathsf{data}_0),$$
$$A_0 = (\mathsf{Com}_0, |x_0|).$$

For each $j < t$, the external control plane orders a candidate batch B_j. The batch satisfies $\mathsf{ValidBatch}(A_j, B_j) = 1$, and

$$\mathsf{UpdateCommitment}(\mathsf{pp}, A_j, B_j) \to A_{j+1} \neq \bot.$$

Thus, B_j is accepted and advances the public anchor. The history determines the logical vectors

$$x_0, x_1, \ldots, x_t$$

by the batch semantics of Section 2. We write data_t for the centralized reference state that one full maintainer would hold in order to run centralized Open for x_t. This reference state is used only to state correctness; EDVC does not assume that one online maintainer holds it.

Honest advertised data fragments. Let $\mathcal{R}_t$ be a compatible local registry for the current anchor A_t, so $\mathsf{CompatibleReg}(\mathsf{pp}, \mathcal{R}_t) = 1$. For $\mathsf{cID} \in \mathsf{Claims}(\mathcal{R}_t)$, let

$$\mathsf{advert}_t = (\mathsf{mID}, \mathsf{cID}, A_t, \kappa_t, \mathsf{Seg}_t)$$

be the unique advertisement in $\mathcal{R}_t$ with claim identifier cID. The honest data fragment for that advertised responsibility in the current valid history is written

$$\mathsf{HonState}(\mathsf{pp}, \mathsf{advert}_t).$$

3.2.1. Correctness. Correctness states that honest local computations preserve the intended public anchor and the intended advertised data fragment states.

Definition 2 (Correctness)**.** *An EDVC scheme is* correct *if the following properties hold, except with negligible probability over* Setup *and the internal randomness of the algorithms, for every valid history, every advertised parameter pair* (κ, Seg) *satisfying* $\mathsf{SupportedParams}(\mathsf{pp}, \kappa, \mathsf{Seg}) = 1$*, every compatible local registry mentioned below, every supported query, and every received reply collection satisfying the stated Byzantine bound.*

(i) ***Centralized opening.*** *If*

$$\mathsf{Open}(\mathsf{pp}, A_t, \mathsf{data}_t, q) \to (a, \pi_q), \qquad A_t = (\mathsf{Com}_t, \ell_t),$$

then

$$\mathsf{Verify}(\mathsf{pp}, A_t, q, a, \pi_q) = 1.$$

(ii) ***Public update.*** *Let B_t satisfy $\mathsf{ValidBatch}(A_t, B_t) = 1$. Then*

$$\mathsf{UpdateCommitment}(\mathsf{pp}, A_t, B_t) \to A_{t+1} \neq \bot,$$

where A_{t+1} is the public anchor for the vector obtained from x_t by applying B_t.

(iii) ***Initial data fragment state.*** *Let*

$$\mathsf{advert}_0 = (\mathsf{mID}, \mathsf{cID}, A_0, \kappa, \mathsf{Seg})$$

be an advertisement selected in a compatible local registry for A_0. If

$$\mathsf{InitDataFragment}(\mathsf{pp}, \mathsf{advert}_0) \to \mathsf{dataFragment}_{\mathsf{cID},0},$$

then

$$\mathsf{dataFragment}_{\mathsf{cID},0} = \mathsf{HonState}(\mathsf{pp}, \mathsf{advert}_0).$$

(iv) ***Data fragment state update.*** *Let*

$$\mathsf{advert}_t = (\mathsf{mID}, \mathsf{cID}, A_t, \kappa, \mathsf{Seg}),$$
$$\mathsf{advert}_{t+1} = (\mathsf{mID}, \mathsf{cID}, A_{t+1}, \kappa, \mathsf{Seg}).$$

Suppose $\mathsf{SupportedParams}(\mathsf{pp}, \kappa, \mathsf{Seg}) = 1$. *Suppose*

$$\mathsf{dataFragment}_{\mathsf{cID},t} = \mathsf{HonState}(\mathsf{pp}, \mathsf{advert}_t),$$

and suppose $\mathsf{ValidBatch}(A_t, B_t) = 1$ *and*

$$\mathsf{UpdateCommitment}(\mathsf{pp}, A_t, B_t) \to A_{t+1}.$$

Then

$$\mathsf{UpdateDataFragment}(\mathsf{pp}, \mathsf{advert}_t, \mathsf{advert}_{t+1}, \mathsf{dataFragment}_{\mathsf{cID},t}, B_t) \to \mathsf{HonState}(\mathsf{pp}, \mathsf{advert}_{t+1}).$$

Data fragment state update preserves mID, cID, κ, *and* Seg*; only the public anchor and the local fragment state itself change.*

3.2.2. Availability. Availability is local to the registry chosen by the caller. EDVC does not claim availability for arbitrary registries. It says that if the caller's selected registry contains enough compatible maintainers, and if the actual reply pattern is within the registry's fault budgets, then the operation succeeds.

Throughout the four availability definitions below, $I \subseteq \mathsf{Claims}(\mathcal{R}_t)$ denotes the selected claim identifiers indexing the replies the caller uses. We consider invocations in which at most errTol_t selected maintainers are Byzantine and at most $\mathsf{missTol}_t$ additional selected honest maintainers have no reply used by the caller. Every reply indexed by I from an honest maintainer is the prescribed honest reply. A Byzantine maintainer may return an arbitrary reply or no reply.

Definition 3 (Opening availability)**.** *An EDVC scheme has* opening availability *if the following holds, except with negligible probability. Fix a valid history and a compatible, threshold-valid local registry $\mathcal{R}_t$ for A_t. Suppose centralized opening is defined:*

$$\mathsf{Open}(\mathsf{pp}, A_t, \mathsf{data}_t, q) \to (a^\star, \pi_q^\star).$$

Let a client use opening shares indexed by $I \subseteq \mathsf{Claims}(\mathcal{R}_t)$ under the fault condition above. Then

$$\mathsf{Combine}(\mathsf{pp}, A_t, \mathcal{R}_t, q, \{(\mathsf{cID}_h, \sigma_{\mathsf{cID}_h,q})\}_{\mathsf{cID}_h \in I}) \to (a, \pi_q)$$

for some output satisfying

$$a = a^\star \quad \textit{and} \quad \mathsf{Verify}(\mathsf{pp}, A_t, q, a, \pi_q) = 1.$$

Definition 4 (Repair availability)**.** *An EDVC scheme has* repair availability *if the following holds, except with negligible probability. Fix a valid history and a compatible, threshold-valid local registry $\mathcal{R}_t$ for A_t. Let $\mathsf{cID} \in \mathsf{Claims}(\mathcal{R}_t)$, and let*

$$\mathsf{advert}_t = (\mathsf{mID}, \mathsf{cID}, A_t, \kappa_t, \mathsf{Seg}_t)$$

be the advertisement in $\mathcal{R}_t$ with claim identifier cID*. Suppose the repairing maintainer uses repair packets indexed by $I \subseteq \mathsf{Claims}(\mathcal{R}_t)$ under the fault condition above. Then*

$$\begin{aligned}&\mathsf{RepairDataFragment}(\mathsf{pp}, A_t, \mathcal{R}_t, \mathsf{cID},\\ &\quad \{(\mathsf{cID}_h, \eta_{\mathsf{cID}_h})\}_{\mathsf{cID}_h \in I})\\ &\qquad \to \mathsf{HonState}(\mathsf{pp}, \mathsf{advert}_t).\end{aligned}$$

For public parameters pp, a *reconfiguration family* $\mathcal{T}_{\mathsf{pp}}$ is a stated family of same-anchor registry replacements. It specifies the replacements for which reconfiguration availability is claimed; it need not include every replacement that an implementation can perform.

Definition 5 (Reconfiguration availability)**.** *An EDVC scheme has* reconfiguration availability *for $\mathcal{T}_{\mathsf{pp}}$ if the following holds, except with negligible probability. Fix a valid history and a compatible, threshold-valid local registry $\mathcal{R}_t$ for A_t. Fix also a compatible replacement registry $\mathcal{R}'_t$ for the same public anchor with $(\mathcal{R}_t, \mathcal{R}'_t) \in \mathcal{T}_{\mathsf{pp}}$, and write $\mathcal{A}'_t$ for its set of advertisements. Let*

$$\mathsf{advert}^\star = (\mathsf{mID}^\star, \mathsf{cID}^\star, A_t, \kappa^\star, \mathsf{Seg}^\star)$$

be a target advertisement in $\mathcal{A}'_t$. If the call reconfigures an existing responsibility $\mathsf{cID} \in \mathsf{Claims}(\mathcal{R}_t)$, require

$$\mathsf{succ}_{\mathcal{R}_t \to \mathcal{R}'_t}(\mathsf{cID}) = \mathsf{cID}^\star.$$

In particular, $\mathsf{Preserved}(\mathcal{R}_t, \mathcal{R}'_t, \mathsf{cID}) = 1$ means that $\mathsf{cID}^\star = \mathsf{cID}$. For onboarding, $\mathsf{cID}^\star$ is fresh and is not named by the successor map. Suppose the target maintainer uses reconfiguration packets indexed by $I \subseteq \mathsf{Claims}(\mathcal{R}_t)$ under the fault condition above. Then

$$\begin{aligned}&\mathsf{ReconfigDataFragment}(\mathsf{pp}, A_t, \mathcal{R}_t, \mathsf{advert}^\star,\\ &\quad \{(\mathsf{cID}_h, \eta_{\mathsf{cID}_h})\}_{\mathsf{cID}_h \in I})\\ &\qquad \to \mathsf{HonState}(\mathsf{pp}, \mathsf{advert}^\star).\end{aligned}$$

Definition 6 (Decode availability)**.** *An EDVC scheme has* decode availability *if the following holds, except with negligible probability. Fix a valid history and a compatible, threshold-valid local registry $\mathcal{R}_t$ for A_t. Suppose a caller uses decoding packets indexed by $I \subseteq \mathsf{Claims}(\mathcal{R}_t)$ under the fault condition above. Then*

$$\mathsf{Decode}(\mathsf{pp}, A_t, \mathcal{R}_t, \{(\mathsf{cID}_h, \eta_{\mathsf{cID}_h})\}_{\mathsf{cID}_h \in I}) \to x_t.$$

Definition 7 (Available EDVC)**.** *An EDVC scheme is* available *for $\mathcal{T}_{\mathsf{pp}}$ if it has opening availability, repair availability, reconfiguration availability for $\mathcal{T}_{\mathsf{pp}}$, and decode availability.*

3.2.3. Integrity. EDVC has no agreement property among maintainers: the external control plane fixes the order of candidate batches, and deterministic local validation determines which batches are accepted. The relevant integrity properties are that successful answers under the same public anchor cannot conflict, and that an invalid authenticated batch cannot advance the public anchor.

Definition 8 (Binding)**.** *An EDVC scheme is* binding *for its supported queries if, for every PPT adversary and every $\mathsf{N} = \mathsf{poly}(\lambda)$, the probability that the adversary outputs*

$$A,\ q,\ a, \pi,\ a', \pi'$$

such that

$$\begin{aligned}&\mathsf{Verify}(\mathsf{pp}, A, q, a, \pi) = 1,\\ &\mathsf{Verify}(\mathsf{pp}, A, q, a', \pi') = 1,\\ &a \neq a'\end{aligned}$$

is negligible over

$$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{N}).$$

The anchor A is adversarially chosen; it need not arise from a valid history.

Definition 9 (Update soundness)**.** *An EDVC scheme has* update soundness *if the following holds for every PPT adversary and every valid history with current public anchor A_t. The probability that the adversary outputs a batch B_t for which both*

$$\begin{aligned}\mathsf{ValidBatch}(A_t, B_t) &= 0,\\ \mathsf{UpdateCommitment}(\mathsf{pp}, A_t, B_t) &\neq \bot\end{aligned}$$

hold is negligible.

Together with public-update correctness, update soundness ensures that, except with negligible probability, every accepted batch produces the correct next public anchor.

Definition 10 (Integrity)**.** *An EDVC scheme satisfies* integrity *if it is binding and has update soundness.*

Binding is the EDVC analogue of read consistency: for a fixed public anchor and query, two different verified answers cannot both be accepted. Thus Byzantine maintainers may cause an opening attempt to fail, but they cannot make an honest client accept a different answer unless binding is broken.

Definition 11 (Compactness)**.** *An EDVC scheme is* compact *if, for every valid history and every compatible local registry $\mathcal{R}_t$ for $A_t = (\mathsf{Com}_t, \ell_t)$, every advertisement*

$$\mathsf{advert}_t = (\mathsf{mID}, \mathsf{cID}, A_t, \kappa_t, \mathsf{Seg}_t)$$

selected in $\mathcal{R}_t$ satisfies

$$|\mathsf{HonState}(\mathsf{pp}, \mathsf{advert}_t)| = O\Bigg(\left\lceil \frac{\ell_t}{\kappa_t} \right\rceil + \mathrm{polylog}(\mathsf{N}, \lambda, \mathsf{Seg}_t) \Bigg),$$

Definition 12 (EDVC realization)**.** *An EDVC construction realizes the abstraction targeted in this paper if it is correct, is available for a stated reconfiguration family $\mathcal{T}_{\mathsf{pp}}$, satisfies integrity, and is compact.*

4. EDVC from a Vector Commitment and an Unauthenticated Coded Array

This section describes how our construction combines a vector commitment with an unauthenticated coded array to obtain an EDVC. We represent the vector commitment's centralized prover state data_t at anchor A_t as an array of field elements

$$M_t \in \mathbb{F}^{L_t}.$$

The array M_t is not a new authenticated object: the vector commitment still defines the public anchor and verifier.

An unauthenticated coded array distributes M_t among maintainers. To answer a query, maintainers reconstruct the cells that centralized Open needs, and the client verifies the resulting opening under A_t. When an accepted batch changes the committed vector, it also changes the prover state from M_t to M_{t+1}; each preserved maintainer updates its coded data fragment using the corresponding delta. Repair, onboarding, reconfiguration, and full decode are recovery operations over the same current state M_t.

This composition requires a vector commitment whose chosen prover-state representation has publicly computable deltas: every accepted vector update must determine the corresponding change from M_t to M_{t+1} without access to the full prover state.

Public prover-state delta. The representation uses non-decreasing lengths $L_t \le L_{t+1}$. Let

$$\widetilde{M}_t = (M_t, 0^{L_{t+1}-L_t}) \in \mathbb{F}^{L_{t+1}}$$

be M_t extended with zeros. For every batch B_t accepted under A_t, the old values and opening proofs in $\mathsf{oldAuth}_t$, together with the new values in $\mathsf{updates}_t$, must allow any party to compute

$$\Delta_t = M_{t+1} - \widetilde{M}_t \in \mathbb{F}^{L_{t+1}}$$

from pp, A_t, and B_t, without reconstructing M_t.

Remark 1 (Examples)**.** *This requirement is relative to the chosen prover-state representation. For a Merkle-tree commitment, the authenticated old values and paths, together with the new values, determine all changed leaf and internal labels [15]. For a KZG-based vector commitment represented in evaluation or coefficient form, the old and new values determine the corresponding polynomial delta through the public Lagrange basis. The same requirement is satisfied by Verkle-tree and homomorphic-tree representations whenever the authenticated old material contains the path information needed to determine every changed maintained cell [12].*

4.1. Unauthenticated Coded Array

To describe the distributed storage layer independently of a particular code, we model it through the following interface. The interface captures exactly what the composition needs: compact coded parts of M_t, reconstruction of the cells required by Open, local updates from the public delta Δ_t, and recovery or creation of coded parts when maintainer responsibilities change.

Definition 13 (Unauthenticated coded-array interface)**.** *Fix a state array $M_t \in \mathbb{F}^{L_t}$ and a reconfiguration family $\mathcal{T}_{\mathsf{pp}}$. An* unauthenticated coded array *provides the following operations.*

(i) ***Data fragments.*** *For every advertised responsibility with identifier* cID*, there is a public linear map such that*

$$\mathsf{dataFragment}_{\mathsf{cID},t} = L_{\mathsf{cID},t}(M_t).$$

(ii) ***Reads and decode.*** *The array provides a helper-side response procedure and a caller-side reconstruction procedure for the state cells needed by centralized* Open *on every supported query. It can also reconstruct all of M_t, from which x_t can be recovered.*

(iii) ***Updates.*** *Given the delta Δ_t determined by an accepted batch, every honest maintainer whose responsibility remains assigned can apply it to compute*

$$\mathsf{dataFragment}_{\mathsf{cID},t+1} = L_{\mathsf{cID},t+1}(M_{t+1})$$

from its old data fragment and Δ_t, without reconstructing M_t.

(iv) ***Materialization.*** *The array provides procedures to initialize an advertised data fragment, repair a lost data fragment, onboard a new responsibility, and materialize the target data fragment for every replacement in $\mathcal{T}_{\mathsf{pp}}$.*

For a compact EDVC, every advertisement selected in a compatible registry must satisfy

$$|\mathsf{dataFragment}_{\mathsf{cID},t}| = O\left(\left\lceil \frac{L_t}{\kappa_t} \right\rceil + p_t \right),$$

where L_t is the current filled length of the array and p_t is polylogarithmic in $(\mathsf{N}, \lambda, \mathsf{Seg}_t)$. Thus unused maximum capacity does not contribute to a maintainer's data-fragment storage.

The read, materialization, and decode guarantees hold for every compatible, threshold-valid local registry under the fault condition of Section 3.2. Reconfiguration is guaranteed only for replacements in $\mathcal{T}_{\mathsf{pp}}$. The array does not validate batches or authenticate individual maintainer responses; those checks remain at the vector-commitment layer.

4.2. Composition

Given a vector commitment with publicly computable prover-state deltas and an unauthenticated coded array, the EDVC algorithms of Definition 1 are obtained as follows.

(i) Setup combines the public parameters of both components, while Commit, Open, and Verify are unchanged from the vector commitment.

(ii) UpdateCommitment validates B_t as specified in Section 2. It checks the old values and opening proofs for rewritten positions under A_t and the required default values for newly exposed positions. If validation succeeds, it returns the correct A_{t+1}; otherwise it returns $\perp$.
To run UpdateDataFragment, a maintainer first checks that its old and new advertisements preserve mID, cID, κ, and Seg. It repeats the same validation, checks that the resulting anchor equals the A_{t+1} in its new advertisement, and derives Δ_t from B_t. It then invokes the coded-array update procedure on $\mathsf{dataFragment}_{\mathsf{cID},t}$ and returns the resulting $\mathsf{dataFragment}_{\mathsf{cID},t+1}$, or $\perp$ if any check fails.

(iii) OpenShare returns a coded-array response for the state cells needed by centralized Open on query q. Combine reconstructs those cells and performs the same opening computation. The client accepts only after Verify accepts the result.

(iv) InitDataFragment, RepairDataFragment, and ReconfigDataFragment use the corresponding array materialization procedures; reconfiguration rejects replacements outside $\mathcal{T}_{\mathsf{pp}}$. Finally, Decode reconstructs M_t and recovers x_t.

Theorem 1 (Generic EDVC composition)**.** *Assume the centralized vector-commitment algorithms are correct, the chosen field-array representation has publicly computable prover-state deltas as specified above,* UpdateCommitment *rejects invalid batches except with negligible probability, and the coded array satisfies Definition 13. Then the composition is a correct EDVC and is available for $\mathcal{T}_{\mathsf{pp}}$ under the coded array's registry and fault conditions. If the vector commitment is binding, the composition satisfies integrity.*

If every selected data fragment satisfies the displayed storage bound and

$$L_t = O(\ell_t + \mathrm{polylog}(\mathsf{N}, \lambda)),$$

then the composed EDVC is compact. Under these storage conditions it realizes the EDVC abstraction of Definition 12 when the vector commitment is binding.

Proof. Array initialization establishes

$$\mathsf{dataFragment}_{\mathsf{cID},0} = L_{\mathsf{cID},0}(M_0).$$

The local update repeats the same deterministic batch validation, so it applies Δ_t only for the accepted batch and anchor. The coded-array update therefore preserves

$$\mathsf{dataFragment}_{\mathsf{cID},t+1} = L_{\mathsf{cID},t+1}(M_{t+1}),$$

for every responsibility that remains assigned. Materialization establishes the same equality for initialized, repaired, onboarded, and reconfigured responsibilities.

For a query q, the array reconstructs the state cells used by centralized Open, so Combine returns the same answer/proof pair. Reconstructing all of M_t gives x_t. The array properties give the stated availability. Public-update correctness and soundness follow from UpdateCommitment, while integrity follows from the unchanged vector-commitment verifier. Finally, substituting the bound on L_t into the per-maintainer storage bound gives $O(\lceil \ell_t/\kappa_t \rceil + p_t)$, which is the compactness requirement. □

To obtain an EDVC from this composition, it remains to construct the coded array and to choose the vector commitment. Section 5 constructs the code for a small fixed-size array, including the transformations needed when its parameters change. Section 6 combines these fixed-size arrays into one prefix-growing coded array whose storage tracks its filled prefix. Section 7 then instantiates BORG by combining that coded array with a concrete vector commitment.

5. A Small Fixed-Size Coded Array

This section implements a small fixed-size coded array of exactly Seg field elements; this resolves the gap from Section 4 in which EDVC maintenance was reduced to the coded-array interface. Looking ahead, Section 6 will show how to make this coded array extendable.

The code we construct is a variant of Reed–Solomon [16] that supports reconfiguration as a first-class operation. Standard one-dimensional Reed–Solomon recovers a fixed codeword, but repair or parameter changes require reconstructing and reencoding the data. Two-dimensional Reed–Solomon exposes another dimension, enabling local repair instead of full reconstruction. It does this by arranging the data as coefficients of a bivariate polynomial with different degree bounds in the two variables. The fragment at index i stores both the row at i and the column at i of the bivariate evaluation table. To repair or create the fragment at i, other fragments supply evaluations along these slices, allowing the row and column at i to be interpolated locally. However, those dimensions are fixed: changing the reconstruction threshold or segment size still requires rebuilding the encoded representation. Updates are also expensive in this fixed two-dimensional layout, because each change applies a two-dimensional Lagrange correction.

The primitive preserves the row/column repair path while allowing the reconstruction threshold to change without rebuilding encoded segments. It also supports fast updates, applying small closed-form patches to stored fragments instead of recomputing a two-dimensional correction. Finally, it introduces stitching: adjacent encoded segments can be combined into a larger segment, allowing the array to grow its segment size without decoding and reencoding the data.

5.1. Parameters and Responsibilities

An advertised responsibility is one maintainer's local data fragment. For one fixed-size array, the claim identifier cID

determines one primary column and a prefix of auxiliary row labels. A local registry selects compatible responsibilities with distinct assignments, so selected maintainers hold different coded parts of the same represented array.

The two advertised code parameters are the reconstruction threshold κ and the segment size Seg, which is both the number of semantic field elements represented by one segment and the roots-of-unity grid size. We require $1 \le \kappa \le \mathsf{Seg}$, Seg to be a power of two, $\kappa \mid \mathsf{Seg}$, and $\mathbb{F}$ to contain the required roots of unity. Define $s = \mathsf{Seg}/\kappa$ and

$$\mathsf{RowsPerClaim} = \max\{1, \lceil s/\kappa \rceil\} = \max\{1, \lceil \mathsf{Seg}/\kappa^2 \rceil\}.$$

These dimensions determine the two stored components. A primary column stores s evaluations used for point reads. Auxiliary row labels are extra coded state for repair/onboarding and segment-aligned reads. Those operations need s distinct auxiliary evaluations, so each advertised responsibility includes $\mathsf{RowsPerClaim}$ row labels and $\kappa\mathsf{RowsPerClaim} \ge s$ once κ good advertised responsibilities are selected. These parameters expose the segment-size tradeoff: larger Seg reduces the number of active segments and amortizes bulk retrieval, but at fixed κ increases per-segment local state and row-direction work; smaller Seg has the opposite cost profile.

Setup fixes large public domains $\mathcal{U}_{\mathsf{off}} \subseteq \mathbb{F}^\times, \qquad \mathcal{W} \subseteq \mathbb{F}$. Let $H_s = \{h \in \mathbb{F}^\times : h^s = 1\}$. For every public layout and every $s = \mathsf{Seg}/\kappa$ supported by pp, $\mathcal{U}_{\mathsf{off}}$ is chosen as a union of full H_s-cosets such that aH_s avoids every semantic segment grid for every $a \in \mathcal{U}_{\mathsf{off}}$. Hence $\mathcal{C}^{(s)}_{\mathsf{off}} = \{a^s : a \in \mathcal{U}_{\mathsf{off}}\}$ is disjoint from the semantic column identifiers for the supported layout. Auxiliary row labels are assigned from $\mathcal{W}$. We assume

$$|\mathcal{U}_{\mathsf{off}}|, \min_s |\mathcal{C}^{(s)}_{\mathsf{off}}|, |\mathcal{W}| \ge 2^\lambda/\mathsf{poly}(\lambda).$$

For Borg, $\mathsf{SupportedParams}(\mathsf{pp}, \kappa, \mathsf{Seg}) = 1$ exactly when κ, Seg, the field, and the public domains in pp satisfy the arithmetic, root-of-unity, domain-separation, and domain-size requirements above.

For a claim identifier cID, define

$$a_{\mathsf{cID}} = H_{\mathsf{rep}}(\mathsf{pp}, \mathsf{cID}), \qquad r_{\mathsf{cID},j} = H_{\mathsf{row}}(\mathsf{pp}, \mathsf{cID}, j).$$

At advertised parameters (κ, Seg), the public primary column is $\mathsf{Column}_{\mathsf{cID}} = a_{\mathsf{cID}}^{\mathsf{Seg}/\kappa}$, and the advertised row labels are the first $\mathsf{RowsPerClaim}$ entries of the stable stream $(r_{\mathsf{cID},0}, r_{\mathsf{cID},1}, \ldots)$.

For Borg, $\mathsf{CompatibleReg}(\mathsf{pp}, \mathcal{R}_t) = 1$ exactly when

- $\mathsf{ValidReg}(\mathcal{R}_t) = 1$;
- $\mathsf{SupportedParams}(\mathsf{pp}, \kappa_t, \mathsf{Seg}_t) = 1$;
- the primary columns derived from the selected claim identifiers are pairwise distinct; and
- all row labels in the advertised prefixes derived from the selected claim identifiers are pairwise distinct.

These conditions are publicly checkable. A caller recomputes the assignments from the registry before using it and separately checks any public successor relation used by a supported reconfiguration.

Each advertised responsibility uses an independently sampled $\mathsf{cID} \leftarrow \{0,1\}^\lambda$, fresh for every separately advertised local state. We model H_{rep} and H_{row} as independent, domain-separated random-oracle-derived maps whose outputs are uniform in $\mathcal{U}_{\mathsf{off}}$ and $\mathcal{W}$, respectively. They are also domain-separated from every Merkle, update, transcript, and Fiat–Shamir hash used by the surrounding protocol.

Lemma 1 (Claim-assignment collisions)**.** *Fix (κ, Seg) such that $\mathsf{SupportedParams}(\mathsf{pp}, \kappa, \mathsf{Seg}) = 1$, and let $s = \mathsf{Seg}/\kappa$, and suppose Q selected fresh identifiers are sampled independently from $\{0,1\}^\lambda$. The probability that their advertisements contain a duplicate cID, a duplicate derived primary column, or a duplicate derived row label in the advertised prefixes is at most*

$$O(Q^2/2^\lambda) + O(Q^2/|\mathcal{C}^{(s)}_{\mathsf{off}}|) + O((Q\mathsf{RowsPerClaim})^2/|\mathcal{W}|).$$

When the assignment domains are exponential in λ, these terms are negligible for polynomially many selected advertisements. Registry construction rejects every detected collision.

Proof. The first term is the birthday bound for independently sampled claim identifiers. Conditioned on distinct identifiers, the domain-separated random-oracle outputs give uniform representatives in $\mathcal{U}_{\mathsf{off}}$. Since this domain is a union of full H_s-cosets, their s-th powers are uniform over $\mathcal{C}^{(s)}_{\mathsf{off}}$, giving the primary-column birthday bound. The row-label term follows by the same argument for the independent row map. Detected collisions are public registry-selection failures rather than decoding errors. □

The one-segment algebra in this section is written in normalized local coordinates. For a public coset segment $G_{S,b} = \rho_{S,b}\langle\zeta_S\rangle$, the same formulas are applied after the change of variables $Y = X/\rho_{S,b}$. The anchored formulas are stated explicitly below.

5.2. Encoded State

For one fixed-size array, enumerate the grid as $G = \{x_0, \ldots, x_{\mathsf{Seg}-1}\}$. A segment value vector $M = (M_0, \ldots, M_{\mathsf{Seg}-1})$ is represented by the unique polynomial $F \in \mathbb{F}[X]$ with $\deg F < \mathsf{Seg}$ satisfying

$$F(x_j) = M_j \qquad \text{for } 0 \le j < \mathsf{Seg}.$$

All Seg grid values are semantic array cells; there is no encoding-reserved grid point. These Seg constraints uniquely determine F.

For the advertised κ, write the X^s-decomposition

$$F(X) = \sum_{r=0}^{s-1} X^r f_r(X^s),$$

and define the bivariate view

$$P(Z, W) = \sum_{r=0}^{s-1} f_r(Z) W^r.$$

For every $\alpha \in \mathbb{F}$, $F(\alpha) = P(\alpha^s, \alpha)$. Because $s = \mathsf{Seg}/\kappa$, the degree in Z is at most $\kappa - 1$, and the degree in W is at most $s - 1$. This decomposition is a view of the already pinned polynomial F. If κ changes at fixed Seg, then s changes and F is decomposed differently, but the represented segment does not change. Section 5.8 gives a finite-field example of this decomposition and the read, repair, update, and segment-double calculations below.

Off-grid cosets are what let a claim identifier name a stable primary column. An off-grid representative $a \in \mathcal{U}_{\mathsf{off}}$ names the maintained coset aH_s. Every point $\beta \in aH_s$ has the same column coordinate $\beta^s = a^s$, so the whole coset is one primary column of the bivariate view while still consisting of ordinary evaluations $F(\beta)$. Two representatives a, b define the same storage column exactly when $aH_s = bH_s$, equivalently $a^s = b^s$. The public column identifier is therefore the coset coordinate $C = a^s$, not the arbitrary representative a. Applying an accepted update changes the values stored on that column, but it does not change the column named by cID.

For identifier cID, the representative a_{cID} enumerates the stored primary coset $a_{\mathsf{cID}}H_s$, and the public column identifier is $C = \mathsf{Column}_{\mathsf{cID}} = a^s_{\mathsf{cID}}$. The primary state is the univariate slice $L_C(W) = P(C, W)$, stored as the s evaluations $\{(W, L_C(W)) : W \in a_{\mathsf{cID}}H_s\}$. Since $L_C(W) = F(W)$ on this coset, primary storage is direct evaluation storage of the pinned segment polynomial, not a separately committed codeword. The auxiliary state for row label w is the row slice $R_w(Z) = P(Z, w)$, stored in a fixed public representation of a polynomial of degree at most $\kappa - 1$, for example by its evaluations at κ distinct public field points. This representation lets the maintainer evaluate $R_w(C_\star) = P(C_\star, w)$ for a target column $C_\star$. For this fixed-size array, a maintainer's data fragment consists of the primary column for $\mathsf{Column}_{\mathsf{cID}}$ and the row values for labels in $\mathsf{Rows}_{\mathsf{cID}}$. Section 6 applies this same assignment to every active segment of the full array. Figure 1 gives a visual roadmap for the representation, its two repair directions, and the supported threshold changes at fixed Seg. The following subsections formalize these operations.

5.3. Column and Row Algebra

This subsection records the algebraic facts used by the implementation's column, row, repair, and update paths.

Lemma 2 (Primary-column repair from row labels)**.** *Fix a target column identifier $C_\star$. If a caller obtains the values*

$$P(C_\star, w_1), \ldots, P(C_\star, w_s)$$

at s distinct row labels $w_1, \ldots, w_s$, then the whole primary-column slice $W \mapsto P(C_\star, W)$ is uniquely determined. In particular, the caller can recover the on-curve samples $F(\alpha) = P(C_\star, \alpha)$ for every α with $\alpha^s = C_\star$.

Proof. For fixed $C_\star$, the slice $L_\star(W) = P(C_\star, W)$ is a univariate polynomial in W of degree at most $s - 1$. The s distinct samples $(w_j, L_\star(w_j))$ determine $L_\star$ by interpolation. Evaluating the interpolated polynomial at the coset points α with $\alpha^s = C_\star$ gives the corresponding semantic samples because $F(\alpha) = P(\alpha^s, \alpha) = P(C_\star, \alpha)$. □

Lemma 3 (Closed-form point update in the bivariate lens)**.** *Let $G_\rho = \{\rho\zeta^j : 0 \le j < \mathsf{Seg}\}$, and let $x_i \in G_\rho$. Let L_i be the Lagrange basis polynomial for x_i on G_ρ. For every β,*

$$L_i(\beta) = \frac{x_i}{\mathsf{Seg}\rho^{\mathsf{Seg}}} \cdot \frac{\beta^{\mathsf{Seg}} - \rho^{\mathsf{Seg}}}{\beta - x_i},$$

with the usual limiting value $L_i(x_i) = 1$. Let

$$C_\rho = \rho^s, \qquad C_i = x_i^s,$$

and let $\Lambda^{(\rho)}_{C_i}$ be the Lagrange basis polynomial for C_i on the Z-grid G^s_ρ, which has size κ. Then

$$\Lambda^{(\rho)}_{C_i}(Z) = \frac{C_i}{\kappa C_\rho^\kappa} \cdot \frac{Z^\kappa - C_\rho^\kappa}{Z - C_i},$$

with the usual limiting value $\Lambda^{(\rho)}_{C_i}(C_i) = 1$. Under a point update $F'(X) = F(X) + \Delta L_i(X)$, the induced row update is

$$R'_w(Z) = R_w(Z) + \Delta\, \gamma_i(w) \Lambda^{(\rho)}_{C_i}(Z),$$
$$\gamma_i(w) = \frac{1}{s} \sum_{r=0}^{s-1} \left(\frac{w}{x_i}\right)^r.$$

Proof. For the coset G_ρ, the vanishing polynomial is $X^{\mathsf{Seg}} - \rho^{\mathsf{Seg}}$. Its derivative at $x \in G_\rho$ is $\mathsf{Seg}x^{\mathsf{Seg}-1}$, so the usual Lagrange formula gives

$$L_i(\beta) = \frac{\beta^{\mathsf{Seg}} - \rho^{\mathsf{Seg}}}{(\beta - x_i)\mathsf{Seg}x_i^{\mathsf{Seg}-1}} = \frac{x_i}{\mathsf{Seg}\rho^{\mathsf{Seg}}} \cdot \frac{\beta^{\mathsf{Seg}} - \rho^{\mathsf{Seg}}}{\beta - x_i},$$

because $x_i^{\mathsf{Seg}} = \rho^{\mathsf{Seg}}$. The same calculation on the quotient grid

$$G^s_\rho = \{C_\rho(\zeta^s)^j : 0 \le j < \kappa\}$$

gives the displayed formula for $\Lambda^{(\rho)}_{C_i}$.

It remains to express L_i in the X^s-decomposition. Let H be the subgroup of s-th roots of unity. Since

$$X^s - x_i^s = \prod_{h^s=1} (X - x_i h),$$

we have

$$\frac{1}{X - x_i} = \frac{\sum_{r=0}^{s-1} x_i^{s-1-r} X^r}{X^s - x_i^s}.$$

Substituting this identity into the closed form for L_i, and writing $Z = X^s$, yields

$$L_i(X) = \sum_{r=0}^{s-1} X^r \left(\frac{1}{s} x_i^{-r} \Lambda^{(\rho)}_{C_i}(Z)\right).$$

Therefore the update changes the r-th coefficient polynomial by

$$f'_r(Z) = f_r(Z) + \Delta \frac{1}{s} x_i^{-r} \Lambda^{(\rho)}_{C_i}(Z).$$

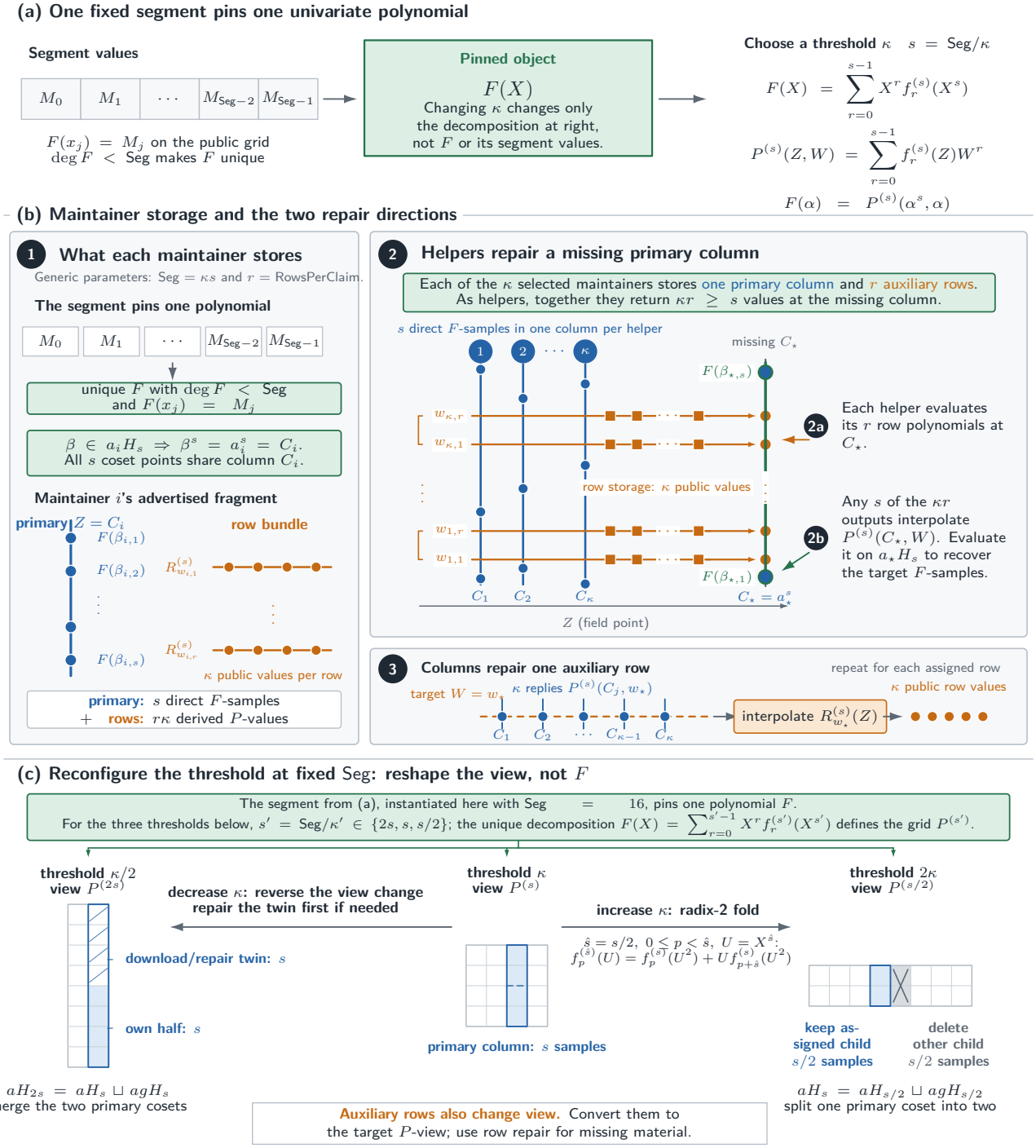

Figure 1. Fixed-size coded-array representation, repair, and threshold reconfiguration. **(a)** The Seg segment values determine a unique polynomial F of degree less than Seg. Choosing κ sets $s = \mathsf{Seg}/\kappa$ and hence the bivariate view $P^{(s)}$, without changing F. **(b)** Each maintainer stores one primary column containing s direct evaluations of F and $r = \mathsf{RowsPerClaim}$ auxiliary rows, each represented at κ public points. To repair a primary column, helpers evaluate their rows at its column identifier; any s distinct outputs interpolate $W \mapsto P^{(s)}(C_\star, W)$. Conversely, κ primary-column replies interpolate one missing auxiliary row. **(c)** At fixed Seg, doubling κ splits a primary coset, whereas halving κ joins it with its other half, repairing that half when it is not held locally. Both operations change the view of F, not F itself. Auxiliary rows are converted to the target view, using row repair for missing target material. In each displayed identity $H_{2t} = H_t \sqcup gH_t$, g denotes an element of $H_{2t} \setminus H_t$; the two identities may use different choices of g.

Evaluating the updated bivariate polynomial at $W = w$ gives

$$R'_w(Z) - R_w(Z) = \Delta \left(\frac{1}{s} \sum_{r=0}^{s-1} (w/x_i)^r \right) \Lambda_{C_i}^{(\rho)}(Z),$$

which is the claimed row update. □

5.4. Reads and Decoding

For a point $\alpha \in G$, define $Q_\alpha(Z) = P(Z, \alpha)$. A helper holding primary column $C = \mathsf{Column}_{\mathsf{cID}}$ evaluates its stored slice $L_C(W) = P(C, W)$ at $W = \alpha$ and returns $Q_\alpha(C) = P(C, \alpha)$. The polynomial Q_α has degree at most $\kappa - 1$, so κ good helpers with distinct derived primary columns determine Q_α, and the combiner outputs $F(\alpha) = Q_\alpha(\alpha^s)$. Thus a point read is a Reed–Solomon decode in the column direction.

The threshold κ, not the segment size Seg, controls the client-side interpolation degree for one array element. Larger segments change the stored slice held by each helper and the row-direction operations below, but a point read still reconstructs a polynomial of degree at most $\kappa - 1$.

A segment-aligned read asks for the semantic values whose grid points share one column identifier. For a representative $x \in G$, this target column is $\{\alpha \in G : \alpha^s = x^s\}$. If $C_x = x^s$, the relevant polynomial is

$$W \mapsto P(C_x, W),$$

which has degree at most $s - 1$. Interpolation therefore needs s distinct auxiliary evaluations. A helper response for this operation is a bundle: for every $w \in \mathsf{Rows}_{\mathsf{cID}}$, the helper returns $P(C_x, w) = L_{C_x}(w)$. Since $\mathsf{RowsPerClaim} = \max\{1, \lceil s/\kappa \rceil\}$, any κ good helpers whose cID-derived row labels are distinct provide at least $\kappa \cdot \mathsf{RowsPerClaim} \geq s$ auxiliary evaluations, enough to interpolate $W \mapsto P(C_x, W)$ and evaluate it on the coset points.

Full decoding is the larger version of the point-read argument. Each helper returns its whole primary-column bundle. With κ good helpers, the combiner has κ primary columns. For each $r < s$, the coefficient of W^r across those columns gives evaluations of the polynomial $f_r(Z)$, which has degree at most $\kappa - 1$, so all f_r, then P, then F are determined.

For the authenticated EDVC instantiation, the final vector-commitment verifier is still run on the reconstructed answer; a reconstruction that leads to a wrong Merkle opening is rejected by Verify. The unauthenticated layer by itself provides no cryptographic authenticity for individual helper responses.

5.5. Updates from Point Deltas

Let one accepted update change semantic position i by field delta Δ. Since the segment is pinned by F, the updated segment polynomial is obtained by adding the closed-form correction for the touched grid point $x_i \in G$:

$$F'(X) = F(X) + \Delta L_i(X), \qquad L_i(X) = \frac{x_i}{\mathsf{Seg}} \cdot \frac{X^{\mathsf{Seg}} - 1}{X - x_i}.$$

This basis has the closed form above because G is a roots-of-unity grid. A primary sample is just an off-grid evaluation of F, so it updates by direct evaluation of the same correction:

$$F'(\beta) = F(\beta) + \Delta \cdot \frac{x_i}{\mathsf{Seg}} \cdot \frac{\beta^{\mathsf{Seg}} - 1}{\beta - x_i}$$

for every stored coded point $\beta \notin G$. No per-assignment Lagrange table is needed for primary samples: the primary coset stores ordinary points of the same pinned F, and the public grid gives the correction formula.

For rows, the same correction is expressed in the (Z, W) view. Let $C_i = x_i^s$. Since the semantic column identifiers also form a roots-of-unity grid of size κ, their basis polynomial is again closed form:

$$\Lambda_{C_i}(Z) = \frac{C_i}{\kappa} \cdot \frac{Z^\kappa - 1}{Z - C_i}.$$

Define

$$\gamma_i(w) = \frac{\kappa}{\mathsf{Seg}} \sum_{r=0}^{s-1} \left(\frac{w}{x_i} \right)^r.$$

This is evaluated as a geometric series: for $\rho = w/x_i \neq 1$, the sum is $(\rho^s - 1)/(\rho - 1)$, and for $\rho = 1$ it is s. Then every maintained row updates as

$$R'_w(Z) = R_w(Z) + \Delta\, \gamma_i(w) \Lambda_{C_i}(Z).$$

Thus the row update is also closed form: the coset decomposition factors the Seg-point correction into a κ-point column-grid correction and the finite sum $\gamma_i(w)$. Batches are handled by linear superposition. Therefore a maintainer can update all stored field elements in its advertised data fragment using only the accepted vector update batch and its old local state. The important locality statement is per stored field element: the arithmetic needed to update one stored primary sample or one stored row sample is evaluation of the closed-form public correction above. It does not require decoding the segment or using a cached table of segment-wide Lagrange values. A maintainer's total batch work is still proportional to the number of stored cells it actually holds and the number of point deltas in the accepted batch; the construction avoids decoding or rewriting an entire encoded segment just to apply one semantic update.

5.6. Repair, Onboarding, and Reconfiguration

Figure 1(b), steps 2 and 3, shows the two complementary repair directions: auxiliary rows repair a primary column, whereas primary columns repair an auxiliary row.

To repair or materialize a primary column $C_\star$, the target is the slice $W \mapsto P(C_\star, W)$, which has degree at most $s - 1$. Each helper contributes its auxiliary row-label bundle evaluated at $C_\star$:

$$\{(w, P(C_\star, w)) : w \in \mathsf{Rows}_{\mathsf{cID}}\}.$$

Any κ good helper bundles whose derived row labels are distinct provide at least s evaluations and reconstruct $L_{C_\star}$.

To repair or materialize a row label w, helpers provide primary-column evaluations $P(\mathsf{Column}_{\mathsf{cID}}, w)$. The polynomial $Z \mapsto P(Z, w)$ has degree at most $\kappa - 1$, so κ good helpers suffice.

Onboarding a new advertised responsibility derives $\mathsf{Column}_{\mathsf{cID}}$ and $\mathsf{Rows}_{\mathsf{cID}}$ from the new cID. The new primary column is materialized by the primary-column repair procedure. The new auxiliary row-label bundle is materialized by running row repair for every $w \in \mathsf{Rows}_{\mathsf{cID}}$.

Reconfiguration changes the data fragment a maintainer advertises under the same anchor. The construction supports onboarding or repair at the current parameters and three factor-two target changes: κ-double, κ-halve, and segment-double.

The κ-double and κ-halve operations are possible because they keep the represented segment polynomial F fixed. Only the decomposition parameter $s = \mathsf{Seg}/\kappa$, the cosets, and the advertised row-label prefix change. When the column length halves, a current primary coset splits into two child cosets and a registry may retain one or more children. When the column length doubles, a target primary coset is assembled from the current coset and its twin. If the twin is already local, the primary conversion is local; otherwise the missing target primary material is obtained by primary repair. For these fixed-Seg changes, target row labels that are not already local in the target advertised responsibility are materialized by row repair.

Segment-double is different: it changes the pinned polynomial by combining two compatible source segment polynomials into one target segment polynomial. The target segment contains the source semantic evaluations exactly: the left source semantic evaluations on the even half of the target grid and the right source semantic evaluations on the odd half. The stitch itself is completely local: each target primary or row evaluation is a public linear combination of the corresponding source evaluations. It uses no helper download, no interpolation, and no repair.

5.7. Factor-Two Reconfiguration

Figure 1(c) summarizes the two threshold changes at fixed Seg. We now derive their algebraic identities and the separate identity used by segment-double.

At fixed Seg, κ-double and κ-halve change the reconstruction threshold while preserving the segment contents. Segment-double combines two adjacent segments and changes (Seg, κ) to $(2\mathsf{Seg}, 2\kappa)$. Section 6 applies these identities to the full extendable array.

The fixed-segment κ-double operation requires s even, equivalently $2\kappa \mid \mathsf{Seg}$, so the target column length $\hat{s} = s/2$ is integral. The fixed-segment κ-halve operation requires κ even, so the target threshold $\kappa/2$ and target column length $2s$ are integral. Segment-double requires the next dyadic root of unity and parent anchor from the supported public layout horizon.

First consider the κ-double and κ-halve operations at fixed Seg. Fix one segment polynomial F and make the column length explicit:

$$F(X) = \sum_{r=0}^{s-1} X^r f_r^{(s)}(X^s),$$
$$P^{(s)}(Z, W) = \sum_{r=0}^{s-1} f_r^{(s)}(Z) W^r.$$

Changing the advertised data-fragment parameters changes this decomposition of the same F; it does not change the represented segment polynomial.

Lemma 4 (Radix-2 fold for κ-double)**.** *Suppose the source configuration has column length s, and the target configuration has column length $\hat{s} = s/2$, corresponding to $\hat{\kappa} = 2\kappa$ at fixed* Seg*. Let $U = X^{\hat{s}}$. Then, for each $p = 0, \ldots, \hat{s} - 1$,*

$$f_p^{(\hat{s})}(U) = f_p^{(s)}(U^2) + U f_{p+\hat{s}}^{(s)}(U^2).$$

Proof. Since $s = 2\hat{s}$, split the X^s-decomposition into paired residues p and $p + \hat{s}$:

$$F(X) = \sum_{p=0}^{\hat{s}-1} X^p \Big(f_p^{(s)}(X^s) + X^{\hat{s}} f_{p+\hat{s}}^{(s)}(X^s) \Big).$$

With $U = X^{\hat{s}}$, we have $X^s = U^2$. Hence

$$F(X) = \sum_{p=0}^{\hat{s}-1} X^p \Big(f_p^{(s)}(U^2) + U f_{p+\hat{s}}^{(s)}(U^2) \Big).$$

Uniqueness of the $X^{\hat{s}}$-decomposition gives the claimed identity. □

The fold is the algebra behind splitting a primary column. Let $H_{\hat{s}} = \{h \in \mathbb{F}^\times : h^{\hat{s}} = 1\}$. Since $H_{\hat{s}} \subset H_s$, any $g \in H_s \setminus H_{\hat{s}}$ gives

$$aH_s = aH_{\hat{s}} \sqcup agH_{\hat{s}}.$$

Thus a source primary column splits into two target primary columns, and its stored samples restrict locally to the two children.

Lemma 5 (Row conversion for κ-double at a split column pair)**.** *Assume $\mathrm{char}(\mathbb{F}) \neq 2$, let $\hat{s} = s/2$, and fix a row label $w \in \mathbb{F}$. Define*

$$A_w(U) = \sum_{p=0}^{\hat{s}-1} f_p^{(s)}(U^2) w^p,$$
$$B_w(U) = \sum_{p=0}^{\hat{s}-1} f_{p+\hat{s}}^{(s)}(U^2) w^p.$$

Then

$$R_w^{(s)}(U^2) = A_w(U) + w^{\hat{s}} B_w(U),$$
$$R_w^{(\hat{s})}(U) = A_w(U) + U B_w(U).$$

For a split child pair $u, -u$ with $u \in \mathbb{F}^\times$, set

$$\Delta_u(w) = \frac{R_w^{(\hat{s})}(u) - R_w^{(\hat{s})}(-u)}{2u}.$$

Then $\Delta_u(w) = B_w(u)$*, and*

$$R_w^{(\hat{s})}(u) = R_w^{(s)}(u^2) + (u - w^{\hat{s}})\Delta_u(w),$$
$$R_w^{(\hat{s})}(-u) = R_w^{(s)}(u^2) + (-u - w^{\hat{s}})\Delta_u(w).$$

Proof. The first identity is the definition of $R_w^{(s)}$ after splitting the terms into residues below and above $\hat{s}$. The second follows from Lemma 4:

$$\begin{aligned} R_w^{(\hat{s})}(U) &= \sum_{p=0}^{\hat{s}-1} \big(f_p^{(s)}(U^2) + U f_{p+\hat{s}}^{(s)}(U^2)\big) w^p \\ &= A_w(U) + U B_w(U). \end{aligned}$$

Evaluating this formula at u and $-u$, subtracting, and dividing by $2u$ gives $\Delta_u(w) = B_w(u)$. Substituting $U = \pm u$ into

$$R_w^{(\hat{s})}(U) = R_w^{(s)}(U^2) + (U - w^{\hat{s}})B_w(U)$$

gives the displayed child-row formulas. □

Corollary 1 (Fresh data needed for the κ-double odd part)**.** *For fixed* $u \in \mathbb{F}^\times$*, define*

$$\Delta_u(W) = \frac{P^{(\hat{s})}(u, W) - P^{(\hat{s})}(-u, W)}{2u}.$$

Then $\deg_W \Delta_u < \hat{s}$*. Hence* Δ_u *is determined by any* $\hat{s}$ *distinct values.*

Writing $L_C^{(\hat{s})}(W) = P^{(\hat{s})}(C, W)$*, if* $\xi^{\hat{s}} = u$*, then*

$$\Delta_u(\xi) = \frac{R_\xi^{(s)}(u^2) - L_{-u}^{(\hat{s})}(\xi)}{2u}.$$

If $\eta^{\hat{s}} = -u$*, then*

$$\Delta_u(\eta) = \frac{L_u^{(\hat{s})}(\eta) - R_\eta^{(s)}(u^2)}{2u}.$$

Thus, if k *old row labels across the two child fibers give such values of* Δ_u*, only* $\hat{s} - k$ *additional opposite-child evaluations are needed.*

Proof. By the proof of Lemma 5,

$$\Delta_u(W) = \sum_{p=0}^{\hat{s}-1} f_{p+\hat{s}}^{(s)}(u^2) W^p,$$

so its degree is less than $\hat{s}$. A univariate polynomial of degree less than $\hat{s}$ is determined by $\hat{s}$ distinct evaluations.

If $\xi^{\hat{s}} = u$, then the old row value at the parent coordinate is the child-column value $R_\xi^{(s)}(u^2) = P^{(\hat{s})}(u, \xi)$, so the displayed formula for $\Delta_u(\xi)$ follows by subtracting the opposite-child value. The case $\eta^{\hat{s}} = -u$ is symmetric: $R_\eta^{(s)}(u^2) = P^{(\hat{s})}(-u, \eta)$. Each such value reduces the remaining interpolation budget by one. □

Lemma 6 (Subgroup doubling pairs columns)**.** *Assume* $\mathrm{char}(\mathbb{F}) \neq 2$*. Let* $H \leq \mathbb{F}^\times$ *have size* s*, let* $\hat{H} \leq \mathbb{F}^\times$ *have size* $2s$*, and assume* $H \subset \hat{H}$*. Pick* $g \in \hat{H} \setminus H$*, so* $\hat{H} = H \sqcup gH$*. For* $a \in \mathbb{F}^\times$*, write* $A = a^s$*. Then*

$$a\hat{H} = aH \sqcup agH, \qquad (ah)^s = A, \qquad (agh)^s = -A$$

for every $h \in H$*.*

Proof. Since $g \in \hat{H}$, $g^{2s} = 1$, so $(g^s)^2 = 1$. If $g^s = 1$, then the order of g divides s, which puts g in the unique subgroup H of $\hat{H}$ of size s, a contradiction. Thus $g^s = -1$. The identities follow from $h^s = 1$:

$$(ah)^s = a^s h^s = A, \qquad (agh)^s = a^s g^s h^s = -A.$$

□

Merging primary columns. By Lemma 6, when the target column length is $2s$, a target primary column is the union of the source primary column and its twin at $-A$. Thus the target primary column is local when both halves are already local. If the twin primary material is not local, the missing half is materialized by primary repair.

Rows under factor-two changes. Lemma 5 describes the local row conversion when a source column splits. For the reverse direction, this construction does not assert that target auxiliary rows can be derived locally from a single target share. Target row-label material for κ-halve is materialized by row repair, using primary-column evaluations as in the rest of the coded array.

Segment-double. Segment-double has a different shape. It changes

$$(\mathsf{Seg}, \kappa, s) \longmapsto (2\mathsf{Seg}, 2\kappa, s),$$

so the column length $s = \mathsf{Seg}/\kappa$ stays fixed.

Lemma 7 (Dyadic segment stitching)**.** *Assume* $\mathrm{char}(\mathbb{F}) \neq 2$*, let* ζ *be a primitive* $2\mathsf{Seg}$*-th root of unity, and let* $\rho \in \mathbb{F}^\times$ *be the public parent segment anchor. Define the parent grid*

$$G_T = \{\rho\zeta^j : 0 \leq j < 2\mathsf{Seg}\},$$

and the child grids

$$G_0 = \{\rho\zeta^{2i} : 0 \leq i < \mathsf{Seg}\},$$
$$G_1 = \{\rho\zeta^{2i+1} : 0 \leq i < \mathsf{Seg}\}.$$

Let $F_0, F_1 \in \mathbb{F}[X]$ *have degree less than* Seg*. Define*

$$E_0(X) = \frac{1 + (X/\rho)^{\mathsf{Seg}}}{2}, \qquad E_1(X) = \frac{1 - (X/\rho)^{\mathsf{Seg}}}{2},$$

and

$$T(X) = E_0(X)F_0(X) + E_1(X)F_1(X).$$

Then $\deg T < 2\mathsf{Seg}$*, and*

$$T(X) = F_0(X) \quad \text{for } X \in G_0,$$
$$T(X) = F_1(X) \quad \text{for } X \in G_1.$$

Moreover, for every maintained point β*,*

$$T(\beta) = \frac{1 + (\beta/\rho)^{\mathsf{Seg}}}{2} F_0(\beta) + \frac{1 - (\beta/\rho)^{\mathsf{Seg}}}{2} F_1(\beta).$$

Thus a maintainer holding the two source values $F_0(\beta)$ *and* $F_1(\beta)$ *computes the target value locally.*

Proof. If $X = \rho\zeta^{2i} \in G_0$, then

$$(X/\rho)^{\mathsf{Seg}} = (\zeta^{2i})^{\mathsf{Seg}} = 1,$$

so $E_0(X) = 1$ and $E_1(X) = 0$. If $X = \rho\zeta^{2i+1} \in G_1$, then

$$(X/\rho)^{\mathsf{Seg}} = (\zeta^{2i+1})^{\mathsf{Seg}} = -1,$$

because ζ has order $2\mathsf{Seg}$. Hence $E_0(X) = 0$ and $E_1(X) = 1$ on G_1. Since each selector has degree Seg and $\deg F_0, \deg F_1 < \mathsf{Seg}$, the stitched polynomial has degree less than $2\mathsf{Seg}$. The displayed local formula follows by evaluating the definition of T at β. □

Lemma 8 (Dyadic stitching in the bivariate lens)**.** *Keep the assumptions of Lemma 7, and write* $\mathsf{Seg} = \kappa s$*. Let*

$$F_j(X) = P_j(X^s, X) \qquad (j \in \{0, 1\})$$

be the X^s-lenses of the two source segments. Set

$$\begin{aligned} C_\rho &= \rho^s, \\ \epsilon_0(Z) &= \frac{1 + (Z/C_\rho)^\kappa}{2}, \\ \epsilon_1(Z) &= \frac{1 - (Z/C_\rho)^\kappa}{2}. \end{aligned}$$

Then

$$P_T(Z, W) = \epsilon_0(Z)P_0(Z, W) + \epsilon_1(Z)P_1(Z, W)$$

satisfies $T(X) = P_T(X^s, X)$, $\deg_Z P_T < 2\kappa$, and $\deg_W P_T < s$.

For a primary column identifier $C = a^s$,

$$P_T(C, W) = \epsilon_0(C)P_0(C, W) + \epsilon_1(C)P_1(C, W).$$

Thus the target primary column is obtained by one public scalar combination of the two source primary columns. For a row label w,

$$P_T(Z, w) = \epsilon_0(Z)P_0(Z, w) + \epsilon_1(Z)P_1(Z, w),$$

so the target auxiliary row is obtained by the same local polynomial operation on the two source rows.

Proof. Since $\mathsf{Seg} = \kappa s$,

$$(X/\rho)^{\mathsf{Seg}} = \left(\frac{X^s}{\rho^s}\right)^\kappa = \left(\frac{X^s}{C_\rho}\right)^\kappa.$$

Therefore $E_j(X) = \epsilon_j(X^s)$, and

$$\begin{aligned} P_T(X^s, X) &= \epsilon_0(X^s)P_0(X^s, X) \\ &\quad + \epsilon_1(X^s)P_1(X^s, X) \\ &= E_0(X)F_0(X) + E_1(X)F_1(X) \\ &= T(X). \end{aligned}$$

The source lenses have $\deg_Z < \kappa$ and $\deg_W < s$. Multiplication by ϵ_0 or ϵ_1 raises only the Z-degree, and only by κ, so $\deg_Z P_T < 2\kappa$ while $\deg_W P_T < s$. The primary-column and row identities are evaluations of the displayed formula for P_T at $Z = C$ and $W = w$, respectively. □

For row states stored as evaluations, the compatible dyadic domains also stitch without communication. Let $\eta = \zeta^s$, so η has order 2κ. The parent Z-domain anchored at C_ρ is

$$\mathcal{Z}_T = \{C_\rho\eta^j : 0 \le j < 2\kappa\},$$

with child domains

$$\begin{aligned} \mathcal{Z}_0 &= \{C_\rho\eta^{2i} : 0 \le i < \kappa\}, \\ \mathcal{Z}_1 &= \{C_\rho\eta^{2i+1} : 0 \le i < \kappa\}. \end{aligned}$$

On $\mathcal{Z}_0$, $\epsilon_0 = 1$ and $\epsilon_1 = 0$; on $\mathcal{Z}_1$, $\epsilon_0 = 0$ and $\epsilon_1 = 1$. Thus the target row evaluation vector on $\mathcal{Z}_T$ is the even/odd stitch of the two source row evaluation vectors on $\mathcal{Z}_0$ and $\mathcal{Z}_1$. If an implementation stores rows on another evaluation domain, the same formula for $P_T(Z, w)$ still gives a local interpolation/evaluation operation, but not a direct reuse of the source evaluation vector.

The stable row-label convention in Section 5.1 is what makes the row stitch apply to advertised row bundles. Segment-double changes κ to 2κ while leaving s fixed, so

$$\begin{aligned} \mathsf{RowsPerClaim}' &= \max\{1, \lceil s/(2\kappa)\rceil\} \\ &\le \max\{1, \lceil s/\kappa\rceil\} \\ &= \mathsf{RowsPerClaim}. \end{aligned}$$

Since row labels are derived from a stable ordered stream independent of Seg, κ, and the anchor, the target row-label bundle is a prefix of the source bundle. No new row labels are needed solely for dyadic stitching.

Figure 2 summarizes the construction at the segment, primary-column, and auxiliary-row levels.

Looking ahead, Section 6 assembles these fixed-size arrays into one extendable array. To preserve the local stitch above, it places the values from the left source segment at alternating positions of the combined segment and the values from the right source segment at the remaining positions. If instead the two source segments occupied consecutive halves of the larger evaluation grid, the formulas above would not apply, and a maintainer holding only $F_0(\beta)$ and $F_1(\beta)$ could not in general compute $T(\beta)$ locally.

This works only when combining the two segments: $(F_0, F_1) \mapsto T$ is locally computable, but a maintainer holding only $T(\beta)$ cannot in general recover both source values without retaining them or obtaining them from helpers.

5.8. Toy Coded-Array Example

The following toy example illustrates the two directions of the coded array. It uses simple semantic points only to make the interpolation visible; the next section specifies the public evaluation points used by the full extendable construction. All arithmetic is modulo 17.

Take $\mathsf{Seg} = 4$, $\kappa = 2$, $s = 2$, and

$$F(X) = 3 + 2X + X^2 + 4X^3.$$

On the toy semantic points $G = \{0, 1, 2, 3\}$, the values are $(3, 10, 9, 7)$. The X^s-decomposition is

$$\begin{aligned} F(X) &= (3 + X^2) + X(2 + 4X^2), \\ P(Z, W) &= 3 + Z + W(2 + 4Z), \end{aligned}$$

so $F(\alpha) = P(\alpha^2, \alpha)$. Consider two helpers: their primary-column identifiers are 16 and 8, and their auxiliary row labels are 6 and 7. For a point read at $\alpha = 2$, they return

$$P(16, 2) = 15, \qquad P(8, 2) = 11.$$

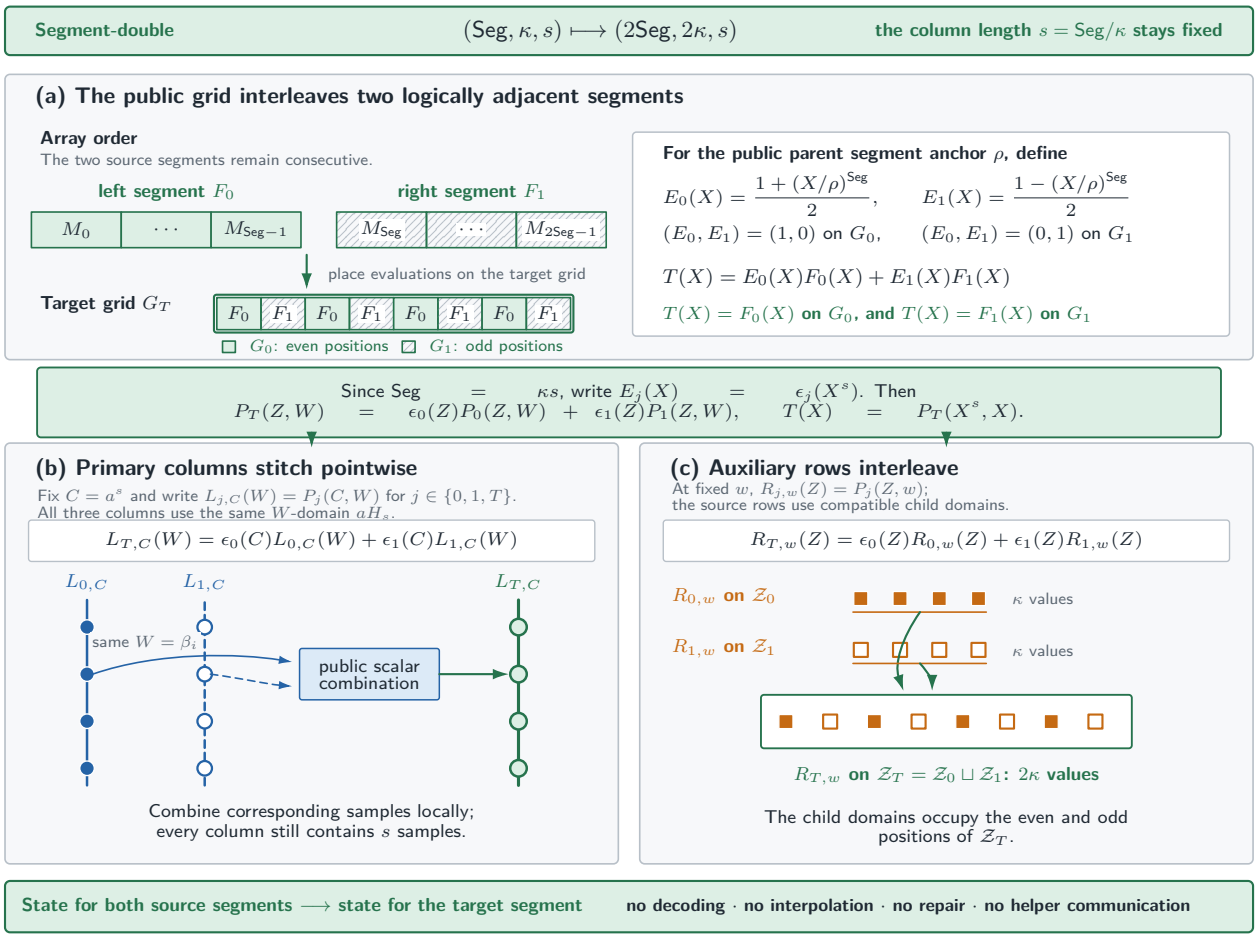

Figure 2. Local segment-double stitching. The transition $(\mathsf{Seg}, \kappa, s) \mapsto (2\mathsf{Seg}, 2\kappa, s)$ keeps s fixed. **(a)** Public selectors combine the source polynomials into T, which agrees with each source on its child grid. **(b)** Because the selectors depend on X through $Z = X^s$, corresponding primary-column samples combine pointwise. **(c)** On compatible child domains, the target auxiliary row is the even–odd interleaving of the source rows. Thus a maintainer holding both source-segment fragments derives the target fragment locally, without decoding, interpolation, repair, or helper communication. The reverse transition is not generally local.

These are two evaluations of the degree-1 polynomial

$$Z \mapsto P(Z, 2) = 7 + 9Z,$$

so interpolation recovers that polynomial and the client outputs

$$F(2) = P(2^2, 2) = P(4, 2) = 9.$$

Repair uses the same polynomial in the other direction. If a new maintainer needs primary column $C_\star = 2$, the same helpers use their row labels and return

$$R_6(2) = 14, \qquad R_7(2) = 7.$$

These are evaluations of $W \mapsto P(2, W) = 5 + 10W$, so interpolation recovers the target primary column. Point reads therefore decode in the Z-direction, while primary-column repair decodes in the W-direction; both reuse the same pinned polynomial P.

Point updates also stay local. If the value at $x = 1$ increases by $\Delta = 5$, the Lagrange basis on G is

$$L_1(X) = 9X(X - 2)(X - 3),$$

so every stored sample at point β adds $5L_1(\beta)$. For the stored primary point $\beta = 4$, $L_1(4) = 4$, so the stored value 11 becomes $11 + 5 \cdot 4 = 14 \bmod 17$. The maintainer updates its own stored samples by evaluating this public correction; it does not decode the segment. In the two-variable view, this correction is represented by

$$Q(Z, W) = 13Z + W(15 + 11Z),$$

with $Q(X^2, X) = 5L_1(X)$, so auxiliary row samples are patched by evaluating this expression at their stored (Z, W) points.

6. An Extendable Unauthenticated Coded Array

Section 5 codes one small fixed-size array, and Section 5.7 adds support for reconfigurable parameter changes. This section extends that construction to an unauthenticated coded array of arbitrary current length L. Its live cells are the prefix $[0, L)$, so it is large enough to hold the complete maintained field array required by Section 4.

The construction has two layout requirements. First, semantic cells must be placed into segments so old global indices remain stable as the prefix grows. Second, adjacent segments must be aligned so segment-double can stitch their encoded states into one larger segment state locally.

The array layer is still unauthenticated: it stores data fragments of the maintained field array, and the EDVC layer checks answer/proof pairs assembled from reconstructed cells against the public anchor.

6.1. Public Layout

Why the construction requires an active prefix. The construction stores a dense representation of a prefix of length L of the data vector (implicitly leaving subsequent elements as 0), where L can be increased over time. Supporting an arbitrary sparse set of occupied positions in a large mostly-0 vector would also require the distributed state to identify those positions. The following counting bound makes this storage pressure explicit for field-valued encodings.

Theorem 2 (Storage pressure for any-κ sparse availability)**.** *Let $q = |\mathbb{F}|$, let $N \geq 2r$, and let $E_P : \mathbb{F}^N \to \mathbb{F}^{L_P}$ be the local encoding map for maintainer P. Suppose that every κ-subset of maintainers can jointly decode every vector in $\mathbb{F}^N$ with exactly r nonzero entries. Then for every κ-subset I,*

$$\sum_{P \in I} L_P \geq r \log_q(N/r).$$

Proof. For a fixed I, the concatenated encoding must be injective on the $\binom{N}{r}(q-1)^r$ vectors with exactly r nonzero entries, while its image has size at most $q^{\sum_{P \in I} L_P}$. Since $\binom{N}{r} \geq (N/r)^r$, comparison of these two quantities gives the stated bound. □

For a materialized prefix, L itself identifies the occupied coordinates.

The public parameters fix a base segment size $\mathsf{Seg}_{\mathsf{base}}$. Larger supported segment sizes are obtained by repeated doubling: $\mathsf{Seg}_\ell = 2^\ell \mathsf{Seg}_{\mathsf{base}}$. This base is public and shared by all registries. For a fixed segment size, the segment intervals, placement permutation, and segment evaluation domains are public.

The placement permutation is chosen recursively. If π_ℓ is the permutation for size Seg_ℓ, then

$$\pi_{\ell+1}(u) = 2\pi_\ell(u), \qquad \pi_{\ell+1}(\mathsf{Seg}_\ell + u) = 2\pi_\ell(u) + 1.$$

Thus, under segment-double, the left source semantic evaluations become the even grid positions of the target segment and the right source semantic evaluations become the odd grid positions.

Let $G_{S,b}$ be the evaluation domain for segment b at segment size S. The domains are chosen so that $G_{2S,b} = G_{S,2b} \cup G_{S,2b+1}$. The left child domain is the even half of the parent domain and the right child domain is the odd half. The closed-form permutation and scale factors are given below.

At size Seg_ℓ, global array cell j is stored in segment

$$b = \lfloor j/\mathsf{Seg}_\ell \rfloor, \qquad u = j \bmod \mathsf{Seg}_\ell,$$

and local semantic offset u is constrained at the public grid position determined by $\pi_\ell(u)$ on $G_{\mathsf{Seg}_\ell,b}$. The coded array is prefix-materialized: its semantic cells are exactly $[0, L)$. At size Seg_ℓ, the active segments are precisely those intersecting this prefix, so there are $\lceil L/\mathsf{Seg}_\ell \rceil$ of them. Unused offsets in the last active segment are fixed public padding, as defined below. When L grows, the same public layout keeps old cells at their global indices, and newly reached offsets become ordinary semantic cells through the accepted update.

The assignment rule from Section 5.1 then applies to each active segment in this public layout. Each advertised cID derives a primary column and a stable ordered row-label list. The advertised responsibility uses the first

$$\mathsf{RowsPerClaim} = \max\{1, \lceil s/\kappa \rceil\}$$

labels from that list. Segment-double keeps s fixed and replaces κ by 2κ, so the target row-label prefix is never longer than the source prefix. This is why the row part of segment-double is local under this layout. By contrast, the κ-double and κ-halve paths at fixed Seg change the advertised responsibilities for one segment polynomial; missing target row-label material is obtained by the repair procedure from Section 5.

Because the layout is public and registry-independent, registries using the same maintained vector and advertised (κ, Seg) derive the same segment intervals, grid positions, segment domains, primary columns, and row-label prefixes. A caller therefore does not trust a maintainer to describe its layout, and the same maintainer state can be selected by another compatible registry without translating stored segment state.

Closed-form layout. We now give a closed-form public layout satisfying the recursive rules above. For a local semantic offset $0 \leq u < \mathsf{Seg}_\ell$, write

$$u = a\mathsf{Seg}_{\mathsf{base}} + r, \qquad 0 \leq r < \mathsf{Seg}_{\mathsf{base}}, \quad 0 \leq a < 2^\ell.$$

Let $\mathrm{rev}_\ell(a)$ be the bit reversal of the ℓ-bit integer a, and define $\pi_\ell(u) = r2^\ell + \mathrm{rev}_\ell(a)$. Then

$$\pi_{\ell+1}(u) = 2\pi_\ell(u), \qquad \pi_{\ell+1}(\mathsf{Seg}_\ell + u) = 2\pi_\ell(u) + 1,$$

so segment-double maps the left source segment to even target grid positions and the right source segment to odd target grid positions.

The formulas below apply to the segment indices supported by the chosen field. For every supported segment size S and every bit position t that may occur with $b_t = 1$ in a

supported segment index, the field must contain a primitive $2^{t+1}S$-th root of unity. The chosen roots determine how many segment indices one recursive layout supports. Larger logical vectors use the block construction in the following remark.

Remark 2 (Segment-double over a fixed field)**.** *The higher-order roots above are needed only for the recursive segment-double layout. If the field cannot support one stitching hierarchy for the entire prefix, the public layout partitions the prefix into blocks aligned with every supported segment size and restarts the segment index within each block. Segment-double remains local within each block; all other operations and bounds are unchanged.*

It remains to choose segment domains compatibly across sizes. Fix roots of unity ζ_M, for the power-of-two orders used by the construction, with $\zeta_{2M}^2 = \zeta_M$. For a segment size S and segment index $b = \sum_{t\ge 0} b_t 2^t$, define

$$\rho_{S,b} = \prod_{t:\, b_t=1} \zeta_{2^{t+1}S}, \qquad G_{S,b} = \{\rho_{S,b}\zeta_S^i : 0 \le i < S\}.$$

The scale factors satisfy $\rho_{S,2b} = \rho_{2S,b}$ and $\rho_{S,2b+1} = \rho_{2S,b}\zeta_{2S}$. Therefore $G_{2S,b} = G_{S,2b} \cup G_{S,2b+1}$, with $G_{S,2b}$ the even half of $G_{2S,b}$ and $G_{S,2b+1}$ the odd half. At level ℓ, cell j lies in $b = \lfloor j/\mathsf{Seg}_\ell \rfloor$ for $u = j \bmod \mathsf{Seg}_\ell$, and the semantic value at offset u is constrained at the point $\rho_{\mathsf{Seg}_\ell,b}\zeta_{\mathsf{Seg}_\ell}^{\pi_\ell(u)}$.

Prefix materialization. The layout above is applied only to the materialized prefix. For a semantic array $M_0, \dots, M_{L-1}$, segment b at size Seg_ℓ is active iff $b\mathsf{Seg}_\ell < L$. Its segment value vector is the full Seg_ℓ-tuple defined, for $0 \le u < \mathsf{Seg}_\ell$, by

$$M_u^{(\ell,b)} = \begin{cases} M_{b\mathsf{Seg}_\ell+u}, & b\mathsf{Seg}_\ell + u < L, \\ 0, & b\mathsf{Seg}_\ell + u \ge L, \end{cases}$$

The first case assigns the semantic value at a position in the materialized prefix. The second case fills each remaining position of the last active segment with a public zero, making that segment polynomial well-defined. The semantic array and the valid targets of Decode are exactly the global positions in $[0, L)$, and the array length is L. Segments with $b\mathsf{Seg}_\ell \ge L$ are inactive and contribute no advertised maintainer state. When the prefix later reaches a padded position, the accepted update assigns its semantic value.

These formulas are public and independent of any registry. Their only role is to make the array layout agree with the factor-two code identities: when two source segments are stitched into a doubled segment, the left source values land on the even half of the target domain and the right source values land on the odd half. Thus a segment-double reconfiguration changes which public segment contains a maintained value, but it does not change the represented values and does not require reencoding the old archive.

6.2. Storage and Operations

We now describe the storage and operation costs for one advertised operating point. Fix an enumeration

$$G = \{x_0, \dots, x_{\mathsf{Seg}-1}\}.$$

Each segment polynomial has the bivariate view P_b obtained by the decomposition in Section 5.

For one advertised responsibility, the local state per active segment is

$$s + \kappa \cdot \mathsf{RowsPerClaim}$$

field elements: s primary samples and RowsPerClaim auxiliary row-label slices of length κ. Equivalently, this is

$$\frac{\mathsf{Seg}}{\kappa} + \kappa \max\left\{1, \left\lceil \frac{\mathsf{Seg}}{\kappa^2} \right\rceil \right\}.$$

Thus storage is this per-segment quantity times the number of active segments in the advertised responsibility, up to the short advertisement itself.

This formula has an important threshold. We call

$$\mathsf{Seg} \ge \kappa^2 \tag{BS}$$

the *balanced-storage condition*. Under this condition, the auxiliary part has the same order as the primary slice, so the maintainer stores $O(\mathsf{Seg}/\kappa)$ field elements per active segment and $O(L/\kappa)$ field elements over an L-cell array, up to the last-segment tail. When $\mathsf{Seg} < \kappa^2$, $\mathsf{RowsPerClaim} = 1$ can force every active segment to carry a full auxiliary slice of length κ. Small segments may still be useful for isolated reads and update locality, but for large κ they can lose the intended L/κ-style storage scaling. For example, with $\mathsf{Seg} = 128$ and $\kappa = 64$, the primary slice has only $\mathsf{Seg}/\kappa = 2$ field elements, but $\mathsf{RowsPerClaim} = 1$, so the auxiliary part contributes 64 field elements per active segment. The advertised data fragment is therefore 66 field elements per active segment, not roughly Seg/κ.

At balanced operating points, the supported divisibility conditions give $\mathsf{RowsPerClaim} = s/\kappa$. Table 1 uses exactly κ correct helpers. Local work counts field-element operations; $\widetilde{O}$ uses standard fast interpolation and multipoint evaluation and suppresses logarithmic factors.

All operations in Table 1 admit optimal download. A segment-aligned read and primary repair each recover s independent field elements and download exactly s. Row repair determines a polynomial of degree at most $\kappa - 1$ and downloads exactly κ evaluations, while full decode recovers Seg independent array elements and downloads exactly Seg. A point read also interpolates a polynomial of degree at most $\kappa - 1$, so it downloads one field element from each of the required κ helpers. Point updates require no helper communication.

Remark 3 (Onboarding download)**.** *Table 1 reports the symmetric onboarding procedure, in which every helper contributes to both primary and row materialization, for a total download of $2s$ field elements. The new primary column intersects each new auxiliary row once, so one*

	Maintainer		Caller	
Operation	Local work	Response	Local work	Download
Data-fragment size per active segment: $2s = 2\mathsf{Seg}/\kappa$ field elements.				
Reads and updates				
Point read	$\Theta(s)$	1	$\widetilde{O}(\kappa)$	κ
Segment-aligned read	$\Theta(s)$	s/κ	$\widetilde{O}(s)$	s
One point update	$\Theta(s)$	0	–	0
Repair, onboarding, and decoding				
Primary repair	$\Theta(s)$	s/κ	$\widetilde{O}(s)$	s
Row repair (one label)	$\Theta(s)$	1	$\widetilde{O}(\kappa)$	κ
Onboarding	$\widetilde{O}(s)$	$2s/\kappa$	$\widetilde{O}(s)$	$2s$
Full decode	$\Theta(s)$	s	$\widetilde{O}(\mathsf{Seg})$	Seg

TABLE 1. PER-ACTIVE-SEGMENT COSTS AT BALANCED OPERATING POINTS, USING EXACTLY κ CORRECT HELPERS. A RESPONSE IS THE NUMBER OF FIELD ELEMENTS RETURNED BY ONE MAINTAINER; DOWNLOAD IS THE TOTAL RECEIVED BY THE CALLER. THE CALLER IS A CLIENT FOR READS AND THE TARGET MAINTAINER FOR REPAIR OR ONBOARDING. THE POINT-UPDATE ROW IS FOR ONE DELTA AND EXCLUDES DELIVERY OF THE ACCEPTED UPDATE BATCH.

helper evaluation per row label may be omitted, reducing the exact download to $2s - \mathsf{RowsPerClaim}$ *without another round. In particular, this saves one field element when* $\mathsf{RowsPerClaim} = 1$.

Point read. A point read for cell j touches only segment $b = \lfloor j/\mathsf{Seg} \rfloor$ and the physical point determined by the layout for local semantic offset $j \bmod \mathsf{Seg}$. The caller downloads one field element for that segment from each of κ usable helpers and interpolates a polynomial of degree at most $\kappa - 1$. Handling corrupted helpers or non-responses adds the extra helper responses specified by the registry. For the helper, the response is its primary slice of length $s = \mathsf{Seg}/\kappa$ for that segment, evaluated at the requested point; without batching, this is linear in the slice length.

Increasing Seg does not change the number of helpers needed for a point read, but it increases s at fixed κ. Isolated point reads therefore tend to prefer smaller segments. The tradeoff changes when many requests hit the same segment: the same primary slice can be evaluated at many requested points, and multipoint or FFT-style evaluation can amortize this work in batched serving paths. Larger segments are useful for such same-segment or bulk access patterns even though a single isolated point read touches a larger local polynomial.

Segment-aligned read. A segment-aligned read touches one segment and reconstructs one polynomial $W \mapsto P_b(C, W)$ of degree at most $s - 1$. Each helper returns a bundle of $\mathsf{RowsPerClaim}$ auxiliary evaluations for that segment. Any κ good helpers with distinct derived row labels provide at least $\kappa \cdot \mathsf{RowsPerClaim} \geq s$ evaluations, enough for the caller to interpolate the slice and evaluate the requested aligned cells. Thus the base download is $\kappa \cdot \mathsf{RowsPerClaim}$ field elements for the touched segment. This is the same order as Seg/κ in the balanced-storage condition, but may be larger when $\mathsf{Seg} < \kappa^2$. Each helper evaluates $\mathsf{RowsPerClaim}$ row-label slices of degree at most $\kappa - 1$ at the target column.

At fixed κ, increasing Seg increases s, so the caller reconstructs a larger polynomial and each helper's primary slice for that segment is longer. Increasing κ decreases $s = \mathsf{Seg}/\kappa$, but it increases helper fanout and the number of responses needed for error correction.

Point updates. An accepted update batch is first grouped by touched segment. For one point delta inside one segment, each stored primary or auxiliary field element receives the closed-form patch from Section 5.5; the maintainer does not decode the segment and does not rewrite a full uncoded segment. No array query or segment reconstruction is needed beyond processing the accepted vector update.

The patch is constant-size for each stored field element once the public parameters are fixed, and applies per point delta. If a maintainer whose advertised responsibility remains assigned stores $s + \kappa \cdot \mathsf{RowsPerClaim}$ cells for a touched segment and the batch has u_b deltas in that segment, its local algebraic work for that segment is proportional to

$$u_b \cdot \big(s + \kappa \cdot \mathsf{RowsPerClaim}\big),$$

with a small constant field-arithmetic patch per product term. Contiguous updates are efficient because they can be grouped by segment; the cost is not a rewrite of the whole ever-growing array.

Repair and onboarding. Primary repair for one segment is the same row-direction interpolation as a segment-aligned read: the repairing caller obtains helper bundles and reconstructs the lost or newly advertised primary slice, a polynomial of degree at most $s - 1$. Each helper evaluates its advertised row-label bundle at the target column for that segment. Materializing one auxiliary row label is the column-direction case: helpers provide primary-column evaluations of a polynomial of degree at most $\kappa - 1$, so the caller downloads κ field elements for that label and segment.

Onboarding a new advertised responsibility materializes its primary slice and then each label in $\mathsf{Rows}_{\mathsf{clD}}$. Per active segment, onboarding downloads $\kappa \cdot \mathsf{RowsPerClaim}$ field

elements for primary repair and another $\kappa \cdot \mathsf{RowsPerClaim}$ field elements to materialize the row-label bundle. This is $2\kappa \cdot \mathsf{RowsPerClaim}$ field elements in total, which is $O(\mathsf{Seg}/\kappa)$ only under the balanced-storage condition. Over a growing array, repair and onboarding repeat this per-segment work for every active segment in the target responsibility. Their total cost is therefore tied to the current array length, while the normalized per-segment work is exactly the read and materialization work above.

Reconfiguration. The κ-double and κ-halve reconfiguration paths keep the represented segment polynomials fixed and change the advertised responsibilities. For each active segment, target material already held by the reconfiguring maintainer is reused. Missing target primary material or row-label material is obtained by the repair and materialization procedures above, and helpers serve the same kinds of values they serve during repair.

Segment-double follows the fixed layout above. Two level-ℓ source segments become one level-$\ell + 1$ target segment, with the left semantic evaluations on the even target grid positions and the right semantic evaluations on the odd target grid positions. The primary and row stitches are the local linear combinations from Section 5.7; they have no helper work and no helper download. The cost is therefore per active segment for the κ-rescale paths, while the segment-double stitch is local work on already maintained source state. These supported paths do not imply arbitrary one-step parameter changes.

Decode. Full decode reconstructs the entire array segment by segment from primary-column bundles. For every active segment, each helper sends its primary slice of length s, and the caller reconstructs the segment polynomial from the returned columns. With the minimum usable helper set, the caller receives $\kappa s = \mathsf{Seg}$ field elements per decoded segment before reconstructing the Seg semantic cells. Larger κ increases helper fanout. Larger Seg increases the work for one decoded segment but reduces the number of active segments for a fixed array length.

Choosing (κ, Seg) balances access costs, the number of active segments, maintainer storage, and helper fanout. Smaller segments make isolated point reads, segment-aligned reads, and per-segment repair more local, but they create more active segments to store, update, repair, onboard, and decode. Larger segments reduce the active-segment count and are better for bulk or many same-segment requests, but each touched segment is a larger polynomial problem. For storage efficiency at threshold κ, the segment should satisfy the balanced-storage condition; otherwise the auxiliary row-label floor can make large κ much less space-efficient than the L/κ intuition suggests. Larger κ increases helper fanout and error-correction cost while shortening primary slices because $s = \mathsf{Seg}/\kappa$. The row-label bundle size $\mathsf{RowsPerClaim}$ is chosen so that κ good helpers still provide enough auxiliary evaluations for row-direction operations.

6.3. Interface Guarantees

The following theorem collects the construction's interface, storage, correctness, and availability guarantees.

Theorem 3 (Prefix-growing coded-array interface)**.** *For every $M \in \mathbb{F}^L$, the construction above satisfies the unauthenticated coded-array interface of Definition 13. Its supported reads are point reads, segment-aligned reads, and compositions of those reads. Its supported local writes are accepted point-delta updates, including updates to newly exposed prefix cells. It supports repair, onboarding, full decode, and the same-parameter and factor-two reconfigurations described above. One advertised data fragment contains*

$$\left\lceil \frac{L}{\mathsf{Seg}} \right\rceil \left(\frac{\mathsf{Seg}}{\kappa} + \kappa \mathsf{RowsPerClaim} \right)$$

field elements, in addition to its short advertisement.

Let $\mathcal{R}_t$ be a local registry with $\mathsf{CompatibleReg}(\mathsf{pp}, \mathcal{R}_t) = 1$ and $\mathsf{ThresholdValid}(\mathcal{R}_t) = 1$. Suppose at most errTol_t selected helpers are Byzantine and at most $\mathsf{missTol}_t$ additional selected honest helpers do not provide a reply used by the caller.

Then accepted point updates, expressed as field deltas, refresh assigned data fragments by the closed-form patches of Section 5.5. The remaining read and recovery operations return the same segment values or advertised data fragment as the all-honest execution. Concretely, they reduce to the following per-segment Reed–Solomon tasks:

1) *For point reads, row repair, and target-row materialization at the current κ, each helper contributes one column-direction evaluation of a polynomial of degree at most $\kappa - 1$. The registry threshold from Section 2 supplies enough returned evaluations for Reed–Solomon correction under the stated fault condition.*
2) *For segment-aligned reads and primary-column repair, each helper contributes a bundle of $\mathsf{RowsPerClaim}$ auxiliary evaluations of a polynomial $W \mapsto P(C_\star, W)$ of degree at most $s - 1$. The returned row-label bundles supply enough distinct auxiliary evaluations for Reed–Solomon correction under the same fault condition.*
3) *For full decode, each helper contributes one full primary-column bundle per active segment. Applying error correction to each coefficient polynomial f_r of degree at most $\kappa - 1$ recovers P and hence F.*
4) *A supported factor-two reconfiguration either materializes target primary columns or rows for the same segment polynomial, or locally stitches two compatible source segment polynomials into one doubled segment. Missing target primary columns or rows are materialized by the procedures above, plus any child or sibling material already held locally.*

When the array is used in the authenticated EDVC instantiation, the final vector-commitment verifier rejects any answer/proof pair assembled from reconstructed cells that is not bound to the public anchor. Error-correction interpolation also identifies the selected evaluation labels whose values are inconsistent with the recovered polynomial; the caller

can map those labels back to selected advertisements and exclude them from later attempts under its registry policy.

Proof. The public layout deterministically assigns each maintained array cell to an active segment and applies the same cID-derived primary and auxiliary responsibilities to every active segment. Every stored evaluation is a public linear function of the semantic cells, so their concatenation gives the map $L_{\mathsf{cID},t}$ for one advertised responsibility. There are $\lceil L/\mathsf{Seg} \rceil$ active segments, and each contributes $\mathsf{Seg}/\kappa + \kappa\mathsf{RowsPerClaim}$ field elements, which gives the displayed storage bound. The empty array has zero data fragments, and a nonempty initial array can be encoded directly during bootstrap.

Point reads, aligned reads, updates, repair, onboarding, and decode are obtained by running the relevant fixed-size operation from Section 5 on each touched active segment. Supported reconfiguration either materializes target primary columns or rows for each preserved segment, or uses the public segment-double layout to stitch compatible sibling segments.

For local maintenance within one touched segment, consider one semantic delta (i, Δ). The primary update adds the value of the univariate correction $\Delta L_i(X)$ at each stored primary sample. The row update is the same correction expressed in the (Z, W) coordinates by Lemma 3:

$$\Delta L_i(X) \quad \longleftrightarrow \quad \Delta\, \gamma_i(W)\Lambda_{C_i}^{(\rho)}(Z).$$

Thus every stored primary or row cell after the update is exactly the corresponding evaluation of the updated polynomial F', and a batch follows by linearity and by grouping deltas by segment. If prefix growth activates a previously inactive segment, its pre-update padded vector and corresponding data fragment are both public zero; applying the same point-delta patches materializes its newly exposed cells.

For a read or recovery invocation, let $I \subseteq \mathsf{Claims}(\mathcal{R}_t)$ index the replies used by the caller, let $b \le \mathsf{errTol}_t$ be the number of selected Byzantine helpers, let $m \le \mathsf{missTol}_t$ be the number of additional selected honest helpers whose replies are not used, and let $e \le b$ be the number of Byzantine helper replies used by the caller. Since the caller uses at most one reply from each selected identity,

$$\begin{aligned} |I| &= \mathsf{numHelpers}_t - b - m + e \\ &\ge \kappa_t + 2\mathsf{errTol}_t + \mathsf{missTol}_t - b - m + e \\ &\ge \kappa_t + 2e. \end{aligned}$$

The last inequality uses $e \le b \le \mathsf{errTol}_t$ and $m \le \mathsf{missTol}_t$. At most e of these replies can be wrong.

For point read at α, the polynomial $Z \mapsto P(Z, \alpha)$ has degree at most $\kappa - 1$. Values from κ distinct primary columns therefore determine it uniquely. The reply bound above supplies at least $\kappa + 2e$ distinct evaluations with at most e wrong values, so Reed–Solomon correction recovers this polynomial; evaluating at α^s gives $F(\alpha)$. For a segment-aligned read at column C_x, the polynomial $W \mapsto P(C_x, W)$ has degree at most $s-1$. The reply bound above gives at least $s + 2e\mathsf{RowsPerClaim}$ distinct auxiliary evaluations, with at most $e\mathsf{RowsPerClaim}$ wrong values, after unusable responses are discarded. The standard Reed–Solomon unique-decoding fact for a univariate polynomial of degree at most $d-1$ says that $d + 2E$ distinct evaluation values determine the polynomial in the presence of E wrong values. Taking $d = s$ and $E = e\mathsf{RowsPerClaim}$ recovers the same slice as in the all-honest case, and evaluating it on the requested coset gives the aligned semantic values.

For full decode of one segment, interpolate each primary column $L_C(W)$ from its s stored samples. The coefficient of W^r in L_C is $f_r(C)$. For every $r < s$, the κ values $\{f_r(C)\}$ at distinct column identifiers determine the polynomial $f_r(Z)$, which has degree at most $\kappa - 1$. With e Byzantine primary-column bundles, each fixed coefficient interpolation has at most e wrong values, so Reed–Solomon correction for degree at most $\kappa - 1$ applies independently for every r. Hence the combiner recovers all f_r, then P, then F. Array decode repeats this argument over all active segments.

Primary-column repair is the segment-aligned read interpolation just described, with the target column equal to the lost primary column. Row repair is the point-read interpolation in the other direction: for a fixed row label w, the row $Z \mapsto P(Z, w)$ has degree at most $\kappa - 1$, so $\kappa + 2e$ distinct primary-column evaluations correct e wrong helper values after unusable responses are discarded. Onboarding runs primary repair for the derived column and row repair for every label in the new $\mathsf{Rows}_{\mathsf{cID}}$, repeating this work for each active segment in the advertised responsibility.

Finally, a κ-double or κ-halve step at fixed Seg changes the X^s-decomposition, the bivariate lens $P^{(s)}$, and the advertised coset columns and row labels as described in Section 5.7. Also, a segment-double step replaces compatible source segment polynomials F_0, F_1 by the target segment polynomial from that section. Whenever target primary material or target row-label material is not already local, it is exactly a primary repair or row repair problem covered above. □

6.4. Proofs of Retrievability

The primary part of an advertised data fragment can be audited without decoding the array. This optional $\mathsf{AuditClaim}$ procedure does not change the EDVC syntax. Fix the current array $M \in \mathbb{F}^L$ and a local registry $\mathcal{R}_t$ with

$$\begin{aligned} \mathsf{CompatibleReg}(\mathsf{pp}, \mathcal{R}_t) &= 1, \\ \mathsf{ThresholdValid}(\mathcal{R}_t) &= 1. \end{aligned}$$

Its selected advertisements share A_t, κ_t, and Seg_t. Let $s = \mathsf{Seg}_t/\kappa_t$. For each active segment $b \in [0, \lceil L/\mathsf{Seg}_t \rceil)$, write P_b for its bivariate view. A maintainer advertising $\mathsf{cID} \in \mathsf{Claims}(\mathcal{R}_t)$, with $C = \mathsf{Column}_{\mathsf{cID}}$, stores the primary polynomial

$$L_{b,C}(W) = P_b(C, W)$$

as s field evaluations.

The auditor sends the same unpredictable seed to every maintainer selected in $\mathcal{R}_t$. Bound to A_t and $\mathcal{R}_t$, the seed

derives a challenge point $\alpha \in \mathbb{F}$ and coefficients $\omega_b \in \mathbb{F}$ for all active segments. The maintainer advertising cID returns the single field element

$$Y_{\mathsf{cID}} = \sum_{b=0}^{\lceil L/\mathsf{Seg}_t \rceil - 1} \omega_b L_{b,C}(\alpha) = \sum_{b=0}^{\lceil L/\mathsf{Seg}_t \rceil - 1} \omega_b P_b(C, \alpha).$$

For this challenge, define

$$Q_{\alpha,\omega}(Z) = \sum_{b=0}^{\lceil L/\mathsf{Seg}_t \rceil - 1} \omega_b P_b(Z, \alpha).$$

Because every P_b has degree at most $\kappa_t - 1$ in Z, so does $Q_{\alpha,\omega}$, and every honest response satisfies

$$Y_{\mathsf{cID}} = Q_{\alpha,\omega}(\mathsf{Column}_{\mathsf{cID}}).$$

The responses therefore form a Reed–Solomon word over the distinct primary columns selected by $\mathcal{R}_t$. Suppose at most errTol_t selected maintainers are Byzantine and at most $\mathsf{missTol}_t$ additional selected honest maintainers do not provide a response used by the auditor. Let $I \subseteq \mathsf{Claims}(\mathcal{R}_t)$ index the responses used by the auditor, and let e be the number of Byzantine responses in I. The reply-count argument from Theorem 3 gives $|I| \geq \kappa_t + 2e$, with at most e incorrect values. Reed–Solomon correction therefore recovers $Q_{\alpha,\omega}$. Its error locations identify the inconsistent claim identifiers and hence the corresponding advertisements; missing responses are erasures.

For a fixed cID, every audit response is a public linear function of its $s\lceil L/\mathsf{Seg}_t \rceil$ primary field elements. Once sufficiently many independent challenges give a linear system of full rank, an extractor solves that system to recover the primary part of $\mathsf{dataFragment}_{\mathsf{cID},t}$. The same construction applies to the auxiliary data: each maintainer returns one response for every $w \in \mathsf{Rows}_{\mathsf{cID}}$, obtained by evaluating the corresponding row-polynomial combination at a common challenge point in the Z-direction.

In a smart-contract deployment, A_t and $\mathcal{R}_t$ are fixed before a random beacon produces the seed. Each selected maintainer first commits to its response, bound to A_t, $\mathcal{R}_t$, the seed, and its cID. Responses are revealed only after the commitment phase closes, after which the contract runs the same error-correction check. This ordering prevents a maintainer from waiting for the honest responses, reconstructing $Q_{\alpha,\omega}$, and then deriving the response for a primary column it did not retain. More generally, a coalition retaining only $r < \kappa_t$ distinct primary fragments can compute only r evaluations of the degree-$(\kappa_t - 1)$ challenge polynomial. Those evaluations are insufficient to interpolate the polynomial or derive valid responses for additional advertised columns. The unpredictable seed prevents challenge-specific precomputation, while committing before honest responses are revealed prevents the coalition from obtaining the missing evaluations during the audit.

7. Borg: EDVC Instantiation

This section completes the composition of Section 4 for BORG. Borg instantiates EDVC with $\mathcal{X} = \mathbb{F}$; applications with other value spaces first apply a fixed public injective encoding into field elements. The prefix-growing array of Section 6 supplies the unauthenticated coded-array interface. It remains to choose the vector commitment and show that its prover-state delta is publicly computable as required there. The authenticated layer must commit to a vector that can be very large and can grow over time, while preserving short openings and logarithmic work per touched position. We therefore instantiate the vector commitment with a sparse Merkle tree of capacity $\mathsf{N} = 2^h$. The public anchor is

$$A_t = (\mathsf{Com}_t, \ell_t),$$

where Com_t is the current root and ℓ_t is the active-prefix length. Positions $i \geq \ell_t$ have the default value and are outside the supported query range until a later accepted transition extends the prefix. Sparse Merkle tree binding gives the cryptographic integrity statement: two different verified answers under the same root yield a collision in the domain-separated Merkle label function. Merkle labels used as maintained-array cells are represented as field cells by a fixed digest-to-$\mathbb{F}$ map; collision resistance is assumed for this composed label function.

The full sparse Merkle tree is not the maintained state. Storing all $\Theta(\mathsf{N})$ leaves and internal labels would make every maintainer pay for empty future capacity, which is exactly what EDVC is meant to avoid. Instead the maintained state M_t contains only the active-prefix part of the tree: the leaf values for indices $i < \ell_t$ and the internal labels whose tree intervals intersect $[0, \ell_t)$. Labels for subtrees disjoint from the active prefix are the public default labels and are not stored in M_t. The root Com_t is part of the public anchor, not part of the maintained array. At each tree level, only the intervals intersecting the active prefix are kept, so the number of maintained cells is $L_t = |M_t| = O(\ell_t + \log \mathsf{N})$.

This is why reconfiguration does not change the anchor. The anchor commits to logical field elements up to the maximum vector length, not to one segment or one advertised data fragment. Supported changes to Seg, κ, or maintainer responsibilities change how the current M_t is encoded and assigned; they do not change Com_t.

The active Merkle cells are linearized by a deterministic preorder inherited from the full sparse Merkle tree, after deleting subtrees disjoint from the active prefix. This order is public and depends only on ℓ_t and N, not on maintainer choices. When the active prefix grows, the same rule determines the positions of old and new cells in the next state array. Thus the sparse Merkle tree supplies the prover-state representation required by Section 4: centralized Open reads the active leaf and non-default authentication labels needed for the query and builds the ordinary answer and proof from those cells and the public default labels. The full state contains the active leaves returned by Decode.

The unauthenticated coded array of Section 6 stores data fragments of this M_t. Here its prefix length is $L = L_t = |M_t|$. The sparse Merkle tree has capacity N, but only the active-prefix maintained state is semantic input to the coded array. In the generic EDVC update path, a candidate batch contains the next active length, the assigned positions and

new values, and the old authenticated material needed to validate any rewritten active positions. UpdateCommitment checks the batch structure and verifies every supplied old value and opening against A_t. If these checks succeed, the vector changes and verified old paths determine the new root and every changed active label. Their public indices and old/new differences form Δ_t. UpdateCommitment returns the resulting next anchor, while UpdateDataFragment derives Δ_t and applies the array's local update rules from Section 5.5 to every data fragment whose advertised responsibility remains assigned.

All EDVC operations are computed from this representation. For a point opening, or for a contiguous range opening, selected helpers return coded-array responses for the planned cells of M_t. The client runs Combine to reconstruct those cells, build the ordinary Merkle opening, and then call Verify. Decode, when invoked, first reconstructs M_t and then extracts the active leaves $x_t[0], \ldots, x_t[\ell_t - 1]$. RepairDataFragment, onboarding, and supported ReconfigDataFragment materialize the advertised data fragment for the same current M_t; they do not change the Merkle root or the active prefix. Availability for these operations is exactly the guarantee for that registry from Theorem 3.

For a contiguous leaf-range query $q = [a, b)$, we use a deterministic public cell set J_q as required by Section 4. The set contains the active leaves in the range, the non-default authentication labels needed on the boundary paths, and any active internal labels in a fixed Merkle cover of subtrees wholly contained in $[a, b)$. These internal labels are decoded cells of M_t; they are used only as intermediate consistency checks during Combine, while the output remains the ordinary query answer and Merkle proof verified by Verify.

The range query is decomposed into coded-array reads: full segment reads when the requested maintained cells cover an encoded segment, and point reads for boundary leaves or authentication labels not covered by such a segment. A full segment read is decoded once and all needed cells inside that segment are retained, including internal Merkle labels that later serve as checks. Default siblings outside the active prefix are not requested, but deduced from public parameters.

Preorder linearization makes full segment reads effective for large contiguous ranges. A large leaf range occupies a long run of maintained cells that also contains most internal labels for the covered subtrees; up to boundary effects, the touched segments contain both returned leaves and the Merkle labels needed to certify them. A helper does not reconstruct the returned range, hash Merkle paths, or run error correction: for each full segment read it returns the stored coded slice for that segment, while boundary cells use the point-response path. Across the threshold set of κ helpers, this is one segment's worth of field elements, not κ copies of the segment: each helper contributes its $1/\kappa$ coded slice. Thus, up to boundary point reads, a large range covered by full segments has bandwidth close to the returned leaves plus the Merkle range material needed to certify them. Raising κ splits that payload across more helpers rather than multiplying the threshold-set download. The client decodes those responses into leaves and Merkle labels and verifies the ordinary Merkle range proof.

Algorithm 1 EDVC Combine for a sparse Merkle tree query

Input. Anchor $A_t = (\mathsf{Com}_t, \ell_t)$, opening query q, public Merkle cover $\mathcal{C}_q$, public query decomposition, and helper responses from the selected registry.
Output. The answer and canonical Merkle proof, or $\perp$.
State. B is the set of excluded helpers, i.e., selected helpers whose later responses are treated as erasures. Cert is the set of labels certified from the root.

1) Initialize $B \leftarrow \emptyset$ and $\mathsf{Cert} \leftarrow \{\mathsf{Com}_t\}$.
2) For each parent/children hash check in $\mathcal{C}_q$, taken from the root downward:
   a) Tentatively decode any needed child labels not already in Cert from their array reads, ignoring responses from helpers in B. If too few usable responses remain even for erasure interpolation, return $\perp$. Retain the tentative labels; inconsistencies caused by Byzantine values are handled by the hash check and correction path below.
   b) If the children hash to their certified parent, add the needed child labels to Cert and continue to the next check.
   c) Otherwise, run error-correction decode on the same array read(s). If correction fails or identifies no new corrupted helper, return $\perp$.
   d) Add the newly identified corrupted helpers to B, retain the corrected labels, and require the same hash check to pass. If it still fails, return $\perp$; otherwise add the needed child labels to Cert.
3) If the requested leaves and boundary authentication labels are in Cert, output the answer and canonical Merkle proof; otherwise return $\perp$.

EDVC Combine **for sparse Merkle tree queries.** For an opening query, the client uses the trusted root in A_t as the top certified label and processes the decoded Merkle cover top-down. Error correction is invoked only after a failed Merkle check. On success, it identifies the selected helpers whose responses disagree with the recovered polynomial, and the client treats all later responses from those helpers as erasures in the same query. This erasure step is a client-side optimization under the same availability condition; helpers still send ordinary coded-array responses and do not participate in error classification. Algorithm 1 gives the sparse Merkle tree instantiation of the EDVC Combine procedure. The procedure maintains the invariant that every label in Cert is either the trusted root or has been checked by a Merkle hash relation against a certified parent. Thus accepting an incorrect certified label implies a collision in the Merkle label function. Under the helper-set condition with at most errTol_t Byzantine helpers, every successful correction round that repairs a failed hash check removes at least one previously unidentified corrupted helper from later tentative decodes; otherwise Combine returns $\perp$. Hence one query triggers at most errTol_t such rounds.

Compactness is the only part of the instantiation that depends on the storage parameters. Section 6 shows that, for a maintained array of L_t cells, an advertised data fragment contains the per-segment local state for that responsibility in each

active segment. Under the balanced-storage condition (BS), this gives an $O(L_t/\kappa_t)$ leading term plus the one-segment tail from the last active segment. Correctness, integrity, availability, repair, onboarding, and supported reconfiguration do not require this storage condition; compactness does.

For the theorem below, let $\mathcal{T}_{\mathsf{Borg,pp}}$ denote the family consisting of the same-parameter registry replacements used for repair and onboarding and the factor-two replacements whose recovery is established by Theorem 3. This is the reconfiguration family covered by the theorem, not an exhaustive characterization of the replacements that Borg can perform.

Theorem 4 (Borg EDVC instantiation)**.** *Assume collision resistance of the domain-separated Merkle label function, an external source that fixes the order of candidate batches, and the helper-set conditions of Theorem 3. Honest parties advance from A_t only when* UpdateCommitment *accepts the batch. Then the maintained state of the sparse Merkle tree over the unauthenticated coded array satisfies EDVC correctness, integrity, and availability for $\mathcal{T}_{\mathsf{Borg,pp}}$ and selected registries over active-prefix vectors of capacity* N*, for the query families whose Merkle cells are included in the public query plan.*

Moreover, let $\mathsf{advert}_t = (\mathsf{mID}, \mathsf{cID}, A_t, \kappa_t, \mathsf{Seg}_t)$ *be a selected advertisement. If this advertisement satisfies the balanced-storage condition* (BS)*, then its honest local state has size, for polylogarithmic p_t,*

$$|\mathsf{HonState}(\mathsf{pp}, \mathsf{advert}_t)| = O\big(\lceil L_t/\kappa_t \rceil + \lceil \mathsf{Seg}_t/\kappa_t \rceil + p_t\big),$$
$$L_t = O(\ell_t + \log \mathsf{N}).$$

Consequently, whenever every selected advertisement satisfies (BS) *and the one-segment term $\lceil \mathsf{Seg}_t/\kappa_t \rceil$ is absorbed by $\left\lceil \frac{\ell_t}{\kappa_t} \right\rceil + p_t$, the construction is compact in the sense of Definition 11. Accepted openings are bound to the public anchor* (Com_t, ℓ_t).

Proof. The state requirements follow from the active-prefix linearization. Centralized Merkle opening reads only the queried active leaves and the non-default labels on the relevant authentication paths; default labels outside the active prefix are public. The active-label preorder gives a public cell index for every maintained label and publicly determines the next indexing when the prefix grows. The number of active-prefix cells is $O(\ell_t + \log \mathsf{N})$, since the intervals intersecting a prefix contribute a geometric series over tree levels plus the boundary path.

For a candidate batch that UpdateCommitment accepts, the procedure verifies the old authenticated material against A_t and computes the next Merkle root. The changed leaf and internal labels, their public indices, and their old/new differences determine Δ_t and hence the coded-array update. The structural checks are deterministic; accepting incorrect old authenticated material would contradict sparse Merkle tree binding. Thus this procedure supplies the publicly computable prover-state delta property and update soundness required by Theorem 1.

Theorem 3 gives cell recovery, decode, repair, onboarding, and reconfiguration in $\mathcal{T}_{\mathsf{Borg,pp}}$ for selected registries satisfying its helper-set conditions. The generic composition theorem then gives the EDVC algorithms and preserves the honest-state invariant for maintained data fragments. Binding of accepted openings is the ordinary sparse Merkle tree binding argument: two different verified answers under one root expose a collision in the Merkle label function. The external source is used only to order candidate batches; it is not trusted for Merkle validity.

It remains to account for storage. Under the balanced-storage condition (BS), Section 6 gives

$$s_t + \kappa_t \mathsf{RowsPerClaim}_t = O\left(\frac{\mathsf{Seg}_t}{\kappa_t}\right)$$

field elements per active segment. There are $\lceil L_t/\mathsf{Seg}_t \rceil$ active segments, so an advertised data fragment stores

$$O\left(\left\lceil \frac{L_t}{\mathsf{Seg}_t} \right\rceil \frac{\mathsf{Seg}_t}{\kappa_t} + p_t\right) = O\big(\lceil L_t/\kappa_t \rceil + \lceil \mathsf{Seg}_t/\kappa_t \rceil + p_t\big).$$

Substituting $L_t = O(\ell_t + \log \mathsf{N})$ and absorbing the logarithmic term into the polylogarithmic overhead gives the compactness bound of Definition 11 whenever the one-segment term $\lceil \mathsf{Seg}_t/\kappa_t \rceil$ is also absorbed by the displayed compactness bound. If (BS) fails, or if this one-segment term is too large for the chosen advertisements, the construction still has the correctness, integrity, and availability properties proved above, but this theorem does not claim EDVC compactness for that advertisement. □

This theorem proves cryptographic integrity of accepted openings. It does not prove that a maintainer set will answer promptly, that an individual coded response is binding before reconstruction, or that an untrusted registry contains enough distinct advertised responsibilities. Those remain deployment assumptions exposed by the EDVC predicates.

8. BORGLOG: Block Archives as Evolving Data Availability

The EDVC update interface is deliberately general: an accepted batch may rewrite any active vector position, not only append new ones. This generality has a cost. To validate a rewrite, maintainers need authenticated material for the previous value under the parent anchor; after validation, they compute the changed maintained-state cells and apply the resulting delta to their advertised data fragment. That is the right interface for a general EDVC, but it is unnecessarily heavy for an existing append-only log. A blockchain archive, certificate-transparency log, or other public log should not have to change its public format to carry EDVC authentication material for previous values.

BORGLOG is the specialization that removes this cost for append streams. The source log continues to publish ordinary entries. A BORG maintainer independently follows the stream, encodes the new entries as vector elements, and uses a compact frontier to derive the EDVC update locally.

Given a parent anchor and the newly appended range, BORG derives the next authenticated vector state and updates the maintainer's advertised data fragment used to serve fresh openings.

The sparse Merkle tree frontier is what lets maintainers do this without extra source-log material. The source append is represented as

$$B_t^{\mathsf{src}} = (\ell_{t+1}, \mathsf{append}_t),$$

where append_t fills exactly the fresh suffix $[\ell_t, \ell_{t+1})$. The corresponding EDVC update list is

$$\mathsf{updates}_t = \{(\ell_t + j, \mathsf{append}_t[j])\}_{j=0}^{\ell_{t+1}-\ell_t-1}.$$

For byte-oriented logs, the appended bytes are first encoded by a fixed public injective application encoding into field elements; $\mathsf{append}_t[j]$ denotes a field element, not a raw byte. For the BLS12-381 scalar field used in the evaluation, one simple encoding packs at most 31 payload bytes into each field element, with the byte length carried by the log format or by an additional field cell.

Every written position is newly exposed, so its previous value is the public default value. The only previous-state authentication material needed to validate the append is the right boundary of the current active prefix. A maintainer stores this boundary as its append frontier. If $\ell_t > 0$, the frontier is the Merkle opening of position $\ell_t - 1$: it contains the leaf label for that position and the sibling labels on its path to the old root Com_t. If $\ell_t = 0$, the frontier is the all-default boundary. The maintainer checks the frontier by recomputing Com_t from the boundary path and comparing it to the old anchor $A_t = (\mathsf{Com}_t, \ell_t)$.

After this check, the maintainer locally constructs the input expected by the generic EDVC update path. The checked frontier gives the old boundary labels next to the fresh suffix, and the public default labels give the old labels inside the previously unallocated part of the tree. Combining these labels with the appended values determines the new authentication labels, the new root, and the maintained-state delta. One boundary path is enough because old subtrees inside the fresh suffix are default, and adjacent non-default old subtrees are summarized on that path. The maintainer then calls the same UpdateDataFragment procedure as in the generic instantiation. If a maintainer has lost its frontier, it recovers the same boundary material from current helpers and checks it against A_t, which is ordinary repair of missing local transition state.

The frontier is small and deterministic. It contains one authentication path at the right boundary of the active prefix, so it has $O(\log \mathsf{N})$ labels. Honest maintainers refresh it after every append. When $\ell_t > 0$, repair or onboarding recovers the frontier from the ordinary opening for $\ell_t - 1$; when $\ell_t = 0$, the frontier is public. This logarithmic state is absorbed by the polylogarithmic term in the compactness bound (Definition 11).

A block is therefore not served as a separate DA instance inside BORG. It is represented by field elements in one growing authenticated vector. The underlying log need not change: it publishes ordinary entries, and its operator need not coordinate with BORG maintainers.

9. Security Analysis

The security argument composes three facts established above. First, the sparse Merkle tree commitment binds accepted openings to the public anchor under collision resistance. Second, the unauthenticated coded array reconstructs the same maintained cells a centralized maintainer would read whenever the selected registry satisfies the threshold condition for the operation. Third, accepted batches determine a unique next anchor and maintained-state delta, so every data fragment whose responsibility remains assigned is updated locally from that same delta.

Consequently, Theorem 4 gives the EDVC guarantee for active-prefix histories: every accepted client output is an ordinary sparse Merkle tree opening under the current anchor; repair and onboarding materialize data fragments of the same maintained state; and supported reconfigurations change advertised responsibilities, not the committed vector. The vector commitment changes only through an accepted update to the vector.

The optional audit of Section 6.4 provides proofs of retrievability for the primary part of each advertised data fragment. Under the same registry threshold and fault conditions, error correction identifies inconsistent responses, and sufficiently many independent successful challenges recover the complete primary state. This audit is separate from the base EDVC guarantee; the same construction also applies to the auxiliary row-label state.

The availability guarantee is deliberately conditional on the registry selected by the caller. A registry with too few correct compatible helpers can fail to answer, and raw coded helper responses are not individually authenticated objects. Integrity is enforced at the end of the client-visible path by sparse Merkle tree verification under A_t.

10. Evaluation

We evaluate the construction in three stages, moving from the coded-array kernel to the generic EDVC instantiation and finally to the append-only application. First, Table 2 isolates one fixed-size coded array and compares our construction with striped, single-polynomial, and two-dimensional Reed–Solomon baselines. We measure local state and write-cache requirements, point writes, point and aligned reads, repair and onboarding, and the supported factor-two reconfigurations. We do not separately benchmark the standalone extendable array: prefix extension and operations over the current prefix are exercised by the authenticated applications below.

Second, we instantiate BORG as an EDVC by placing sparse Merkle authentication over the extendable coded array. Table 3 fixes the coding parameters and reports general rewrites of dispersed positions in the active prefix and fresh point openings alongside sparse-Merkle-only measurements. The latter isolate the underlying authenticated-tree work that

each BORG maintainer repeats before updating or reading its coded data fragment. Repair, onboarding, and reconfiguration are deferred to the end-to-end BORGLOG evaluation. Table 4 then varies Seg and κ for two dispersed-update workloads, measuring how the coding configuration affects both one maintainer's stored state and its authenticated-update time.

Finally, we evaluate BORGLOG, the append-only specialization of BORG. Table 5 varies the append size and coding configuration, reporting one maintainer's append-processing time and newly materialized coded state. Table 6 measures a fresh point opening, separating per-helper work and download from the client's Combine + Verify work, both without faults and with κ corrupted responses among 3κ helpers. Table 7 fixes the balanced configuration $(\mathsf{Seg}, \kappa) = (4096, 64)$ and varies the size of a contiguous range opening. Table 8 measures repair and onboarding, κ-doubling, κ-halving, and the coupled transition $(\mathsf{Seg}, \kappa) \rightarrow (2\mathsf{Seg}, 2\kappa)$, including receiver work, helper work, and downloaded data. Table 9 extends this study to larger segment sizes at $\kappa = 64$.

All measurements are local single-threaded CPU timings. The source code and benchmark artifact are available at https://github.com/nicolas3355/EDVC.

The Rust implementation uses the BLS12-381 scalar field with 32-byte field encodings and BLAKE3 for domain-separated Merkle labels. The measurement host is an Intel Core Ultra 7 255U laptop with 30 GiB RAM, running Linux with `rustc` and `cargo` 1.92.0. Unless stated otherwise, timings are serial local CPU measurements: helpers can be contacted in parallel, but the tables report the maintainer work to process updates or construct one helper response, the helper response bytes, and the client work to Combine + Verify after enough responses arrive. Error-correction decodes use Berlekamp–Welch Reed–Solomon decoding.

Publication timings use independent repetitions, checked after $10, 20, 40, 80,$ and 160 samples. At each check, we compute a Student-t interval with a Bonferroni correction across the five checks. A benchmark stops when every timing series has relative half-width at most 5% at familywise 95% confidence, or after 160 repetitions; all 369 series met this target. Direct-operation benchmarks discard three warmup repetitions; append sampling and warmup are described below. Tables report means of the retained samples, and deterministic values must agree across repetitions.

For the BORG and BORGLOG stages, the external control plane that supplies ordered update batches is outside EDVC and is not evaluated; the benchmarks represent its output by generating each batch deterministically from a seed. Once a batch is available, each maintainer processes it locally and no protocol response is required. All other cross-party operations are implemented through the RPC interface and use one request and one response per contacted maintainer, with no further interaction. Reported costs include endpoint computation and serialization, and wire sizes are reported separately; only update delivery by the external control plane and deployment-specific network latency are omitted.

10.1. Fixed-Size Coded-Array Costs

Table 2 compares the unauthenticated-array encoding with three benchmark-only baselines at the same one-segment operating point. The table states who pays for each operation. These are one-segment microbenchmarks: they measure one maintained responsibility or one client read inside a segment, not the cost of processing an entire unauthenticated array. Table 2 reports these costs for $\mathsf{Seg} = 128$ and source $\kappa = 8$ except where the reconfiguration row names a different target κ. The striped Reed–Solomon baseline [16] splits the Seg payload cells into $\lceil \mathsf{Seg}/\kappa \rceil$ independent codewords of degree at most $\kappa - 1$, padding the last one if needed; each helper stores one evaluation from every codeword. A write at offset i updates only the touched codeword value held by that helper, by adding $\Delta L_{i \bmod \kappa}(\beta)$ at the helper's assigned evaluation point β. Its write cache stores the precomputed constants needed to evaluate these κ Lagrange basis polynomials at that helper's assigned evaluation point, so a write does not recompute the basis polynomial from scratch. This is a point-storage lower bound and does not maintain the auxiliary state used by array repair, onboarding, or reconfiguration.

The single-polynomial baseline is the opposite extreme: the whole segment is one Reed–Solomon codeword. The Seg payload cells pin one polynomial of degree at most $\mathsf{Seg} - 1$, and each helper stores $s = \mathsf{Seg}/\kappa$ evaluations of that polynomial. A write adds $\Delta L_i(\beta)$ to every stored evaluation β of that helper. Its cache stores the precomputed constants needed to evaluate any of the Seg Lagrange basis polynomials at the helper's s stored points. This avoids a dense Seg-by-s table, but the write still applies a separate Lagrange value to each stored cell, and the layout lacks the auxiliary rows used by array repair, onboarding, and reconfiguration. The two-dimensional Reed–Solomon baseline gives each advertised responsibility the same number of primary and auxiliary stored evaluations as the unauthenticated array, but treats the segment as an ordinary two-dimensional Reed–Solomon code. A write at source position (z_i, w_i) applies the generic product patch $\Delta L_{z_i}(Z) L_{w_i}(W)$ to the stored column and auxiliary evaluations. Its cache stores the precomputed constants needed for the two one-dimensional Lagrange evaluations in the column and auxiliary directions, again avoiding a dense product table. These baselines give the comparison two readings. For isolated point reads or writes, striped storage is the lower bound. For the array API, the two-dimensional code is the closest comparator: one maintained responsibility must support point reads, aligned reads, repair, onboarding, and three reconfiguration modes—changing κ upward, changing κ downward, and doubling the segment while also doubling κ. The relevant question is which construction keeps this whole maintainer workflow cheap without a separate write cache or a generic decode-and-reencode step.

The unauthenticated array has no per-responsibility write cache in this benchmark. Its stored primary and auxiliary cells lie on the cosets and decomposition from Section 5, so a public point write uses the closed-form coset update and row-update patches rather than a generic Lagrange correction. In Table 2, this makes the unauthenticated-array write row

slower than plain striped storage for point-only writes, but about $17\times$ faster than the single-polynomial baseline and about $36\times$ faster than the cached two-dimensional Reed–Solomon baseline while maintaining the auxiliary state needed by the array operations. Against the two-dimensional Reed–Solomon baseline, which is the closest comparator, Table 2 shows that the unauthenticated array is faster in every measured operation row and uses the same or less helper payload: equal payload for point reads, aligned reads, and repair/onboarding; less payload for same-Seg κ-rescale; and no helper payload for segment-double.

The reconfiguration rows measure extra structure supplied by the unauthenticated-array encoding. As shown in Table 2, the baselines can materialize a target responsibility, but they do not have local reconfiguration operators: in each baseline cell, the maintainer downloads enough helper material to decode or interpolate the source representation, then materializes the target responsibility under the new assignment. The unauthenticated array instead uses the supported local/specialized reconfiguration paths: $8 \to 16$ downloads the second half of the split primary material and the auxiliary row values for the target row labels, which is one target row label at $\kappa = 16$ for this operating point. The $16 \to 8$ direction keeps the primary conversion local and downloads the auxiliary row values for the target row labels, which are two target row labels at $\kappa = 8$. The segment-double row is the coupled $(\mathsf{Seg}, \kappa) = (128, 8) \to (256, 16)$ transition with fixed $s = 16$: it is a maintainer state-materialization benchmark, not a client append or write benchmark. For that row, each baseline starts from two adjacent maintained source responsibilities, downloads enough helper material to decode both source segments, concatenates the decoded grid values, and re-encodes the one target responsibility. This accounts for the $2\kappa s$ downloaded field elements (8 KiB at $\kappa = 8, s = 16$). The unauthenticated-array column instead starts from a maintainer that already holds the two adjacent source responsibilities and applies the local segment-double stitch, so it downloads no helper material.

10.2. Borg Maintained-State Costs

The sparse Merkle tree layer adds authentication work above the coded array. These measurements focus on the EDVC operation that a centralized sparse Merkle tree maintainer also supports: an accepted transition that rewrites already-active field elements. We do not repeat repair, onboarding, and reconfiguration in this subsection: their data-fragment materialization costs are measured directly in Table 2, and their current-prefix application costs are measured in Section 10.3. Field elements are 32-byte scalar encodings. The measured EDVC transition request is the general active-rewrite path: it rewrites distinct, non-adjacent positions that are already inside the active prefix, and it carries the old values, new values, and a Merkle multiproof for those positions. The measurement records the Merkle tree height, the number of changed Merkle nodes, and the number of supplied proof nodes because path sharing determines the authenticated work. The request is treated as a candidate transition ordered by the external control plane; the table measures maintainer-side processing of that request, including validation. The timed maintainer handler has three steps. First, it verifies the transition data: its local root must equal the proof root, the indices must be in bounds and non-duplicated, proof keys must be well formed, proof nodes must not conflict, and recomputing from the supplied old values and multiproof must recover the old root. Second, it computes the new Merkle labels and new root and derives the authenticated-cell deltas. Third, it applies the resulting field deltas to its advertised data fragment and updates its local root. The centralized sparse-Merkle reference stores the full active tree and reports an ordinary tree update or Merkle-path construction. Its transition row has no external proof to validate because it owns the full state locally. Table 3 uses this reference to isolate the authenticated-tree computation that every BORG maintainer performs before applying the resulting deltas to its coded data fragment; it is not a competing deployment whose cost is paid only once. For the first three update workloads, this computation is about 6–7% of the complete BORG maintainer time. For the densest batch of $131{,}072$ updates, it accounts for 418.313 ms of the 1.469 s total, or about 28%. The larger share shows that dense batches amortize BORG's per-segment coded-update work, making authenticated-tree computation a larger fraction of the total.

The point-opening row gives the analogous component costs. Ordinary sparse-Merkle path construction takes 0.650 μs and verification takes 13.928 μs, while a BORG helper's complete OpenShare work takes 188.729 μs and distributed Combine + Verify takes 271.077 μs. Thus coded-share evaluation and reconstruction dominate the sparse-Merkle-only work, although both complete distributed operations remain below one millisecond. These numbers isolate local CPU work and do not include network transport. The update rows are not monotone in the number of rewritten leaves because denser batches give each touched segment more deltas to process together; Table 4 separates this parameter effect.

Table 4 has a different purpose: it holds the authenticated update workload fixed and varies only the coded-array parameters. The active prefix is $268{,}435{,}456 = 2^{28}$ field elements, i.e., 8 GiB of 32-byte authenticated values. The medium column uses $16{,}384 = 2^{14}$ general updates, and the large column uses $131{,}072 = 2^{17}$ general updates, i.e., 4 MiB of updated 32-byte authenticated values. In each case the updated positions are maximally spaced inside the active prefix. This is an intentionally dispersed stress workload: it suppresses update locality and thereby limits within-segment batching and Merkle-path sharing. It should not be read as a localized update workload, which could benefit from both forms of sharing. The benchmark initializes the active prefix to default values, then times the ordinary UpdateDataFragment path for already-active positions; this makes the 8 GiB setup feasible while preserving the Merkle proof shape and data-fragment patch pattern of a sparse active update. For a fixed update count, the Merkle work is identical across rows: height 50, the same active prefix, and the same changed-label and proof-node counts.

Caller	Operation	Striped RS	Single-polynomial RS	2D RS	Unauthenticated array
Maintainer	State / write cache	512 B / 288 B	512 B / 4.50 KiB	1.00 KiB / 2.34 KiB	1.00 KiB / 0 B
Maintainer	32 distinct point writes	0.045 ms	1.97 ms	4.08 ms	0.114 ms
Client	Point read	0.004 ms, 256 B	0.307 ms, 4.00 KiB	0.127 ms, 256 B	0.085 ms, 256 B
Client	Aligned read, 16 values	0.612 ms, 4.00 KiB	0.569 ms, 4.00 KiB	0.336 ms, 512 B	0.108 ms, 512 B
Maintainer	Repair/onboard current responsibility	0.044 ms, 4.00 KiB	0.514 ms, 4.00 KiB	0.562 ms, 1.00 KiB	0.340 ms, 1.00 KiB
Maintainer	Reconfigure target $\kappa : 8 \to 16$	2.53 ms, 4.00 KiB	0.399 ms, 4.00 KiB	4.19 ms, 4.00 KiB	0.124 ms, 768 B
Maintainer	Reconfigure target $\kappa : 16 \to 8$	3.80 ms, 4.00 KiB	0.398 ms, 4.00 KiB	3.54 ms, 4.00 KiB	0.268 ms, 512 B
Maintainer	Reconfigure target $\mathsf{Seg} \to 2\mathsf{Seg},\ \kappa \to 2\kappa$	4.98 ms, 8.00 KiB	23.4 ms, 8.00 KiB	7.88 ms, 8.00 KiB	0.537 ms, 0 B

TABLE 2. ONE-SEGMENT UNAUTHENTICATED-ARRAY BASELINE COMPARISON AT $\mathsf{Seg} = 128$. EXCEPT FOR THE EXPLICIT $16 \to 8$ RECONFIGURATION ROW, MEASUREMENTS USE SOURCE $\kappa = 8$ AND $s = 16$. THE STATE/CACHE ROW REPORTS ALGEBRAIC FIELD STATE PER MAINTAINED RESPONSIBILITY OR STORED EVALUATION VECTOR, AND EXPLICIT WRITE CACHE STATE, EXCLUDING IMPLEMENTATION-LOCAL BOOKKEEPING. OTHER CELLS REPORT LOCAL WORK FOR THE CALLER SHOWN IN THE FIRST COLUMN AND MODELED HELPER PAYLOAD BYTES FOR ONE SEGMENT-LEVEL OPERATION WHEN THE OPERATION DOWNLOADS HELPER MATERIAL; ASSIGNMENT LABELS DERIVED FROM THE REGISTRY ARE NOT COUNTED AS TRANSMITTED PAYLOAD. THE RECONFIGURATION ROWS MATERIALIZE TARGET RESPONSIBILITIES AT THE SAME SEGMENT SIZE, EXCEPT FOR THE LAST ROW, WHICH MATERIALIZES THE COUPLED SEGMENT-DOUBLE TARGET $(\mathsf{Seg}, \kappa) = (256, 16)$. THE UNAUTHENTICATED-ARRAY CELLS USE THE CONSTRUCTION-SPECIFIC PRIMARY-CONVERSION PLUS ALL TARGET AUXILIARY-ROW REPAIRS FOR SAME-Seg κ-RESCALE, AND LOCAL SEGMENT-DOUBLE STITCHING FOR THE LAST ROW. THE STRIPED REED–SOLOMON REPAIR AND RECONFIGURATION ENTRIES ARE THE CLOSEST STATE-MATERIALIZATION ANALOGUES, NOT A FULL ARRAY AUXILIARY-STATE IMPLEMENTATION.

Operation	Changed nodes	Proof nodes	SMT-only reference	Complete BORG
UpdateDataFragment: 1,024 updates	10,271	8,224	12.498 ms	212.418 ms/helper
UpdateDataFragment: 4,096 updates	32,799	24,608	39.552 ms	654.314 ms/helper
UpdateDataFragment: 32,768 updates	163,871	98,336	195.174 ms	2898.672 ms/helper
UpdateDataFragment: 131,072 updates	393,247	131,104	418.313 ms	1468.624 ms/helper
OpenShare, Combine, Verify: point opening 262,144 active field elements	–	50	0.650 μs server; 13.928 μs client	188.729 μs/helper; 271.077 μs client

TABLE 3. SPARSE-MERKLE-ONLY REFERENCE COSTS AND COMPLETE BORG COSTS FOR AUTHENTICATED REWRITES AND POINT OPENINGS. ALL ROWS USE MERKLE HEIGHT 50. ALL ROWS USE AN ACTIVE PREFIX OF 262,144 FIELD ELEMENTS. BORG ROWS USE $\mathsf{Seg} = 128$ AND $\kappa = 8$. UPDATE ROWS ARE GENERAL UPDATES TO DISTINCT, NON-ADJACENT ALREADY-ACTIVE POSITIONS. THEY REPORT ONE MAINTAINER'S LOCAL HANDLER TIME, INCLUDING TRANSITION-PROOF VALIDATION, NEW-LABEL COMPUTATION, DELTA DERIVATION, AND DATA-FRAGMENT UPDATE. "CHANGED NODES" IS THE NUMBER OF MERKLE LABELS CHANGED BY THE UPDATE BATCH. "PROOF NODES" IS THE SUPPLIED MERKLE MULTIPROOF SIZE FOR UPDATE ROWS AND THE MERKLE PATH LENGTH FOR THE POINT OPENING. THE POINT-OPENING ROW REPORTS SMT-ONLY SERVER PATH CONSTRUCTION, SMT-ONLY CLIENT VERIFICATION, THE LOCAL CORE HANDLER TIME FOR ONE OpenShare RESPONSE, AND THE THRESHOLD CLIENT Combine + Verify TIME.

The variation in Table 4 therefore comes from how many coded field elements one helper maintains and patches. The closed-form update from Section 5.5 is constant-size for one stored coded field element and one authenticated-cell delta once the public parameters are fixed; the total handler time still scales with how many such coded field elements the helper owns in each touched segment. This is why the table reports stored elements per active segment. At fixed Seg, increasing κ shortens the primary slice $s = \mathsf{Seg}/\kappa$, but it also changes the auxiliary responsibility through $\mathsf{RowsPerClaim}$. At fixed κ, larger segments increase the per-active-segment state patched by the helper. This gives a simple rule of thumb for maintainer-side update cost. Ignoring ceilings, one helper stores about

$$\frac{\mathsf{Seg}}{\kappa} + \kappa \cdot \frac{\mathsf{Seg}}{\kappa^2} = \frac{2\mathsf{Seg}}{\kappa}$$

coded elements per active segment while $\kappa < \sqrt{\mathsf{Seg}}$. Increasing κ therefore reduces the coded-update work in this range. Once $\mathsf{RowsPerClaim} = 1$, the stored responsibility is

$$\frac{\mathsf{Seg}}{\kappa} + \kappa,$$

which is minimized near $\kappa \approx \sqrt{\mathsf{Seg}}$; beyond that point, the auxiliary part grows with κ. The exact table entries also include ceilings, fixed Merkle-transition work, and batching inside each touched segment, so storage and runtime are not the same curve.

The state column confirms the balanced-storage condition from Section 6, formalized by (BS). Across the balanced rows, one maintainer stores about 4 GiB for $\kappa = 8$, 2 GiB for $\kappa = 16$, 1 GiB for $\kappa = 32$, and 0.5 GiB for $\kappa = 64$. When $\mathsf{Seg} < \kappa^2$, the auxiliary one-label floor dominates: for example, $(\mathsf{Seg}, \kappa) = (128, 64)$ stores 8.25 GiB, while the balanced configuration $(4096, 64)$ stores 512.00 MiB.

Maintainer update time depends on how this state is distributed across touched segments, not only on the total data-fragment size. At $\kappa = 8$, the $\mathsf{Seg} = 128$ and $\mathsf{Seg} = 4096$ configurations both store about 4 GiB, but the time to process 131,072 updates rises from 40.7 s to 164.7 s because each touched segment contains 32 rather than 1024 stored elements. At $\mathsf{Seg} = 4096$, increasing κ to 32 reduces this count to 256 elements and the update time to 66.8 s. Increasing κ to 64 reduces the count further to 128 elements but takes 69.1 s, illustrating that fixed Merkle work and within-segment batching prevent runtime from exactly following storage. Thus Seg and κ jointly determine per-helper storage and per-touched-segment update work, while κ also determines how many helpers a client must use.

Seg	κ	$s = \mathsf{Seg}/\kappa$	RowsPerClaim	Stored elems./seg.	Active segs.	State	16,384 updates	131,072 updates
128	8	16	2	32	4194305	4.00 GiB	5.58 s	40.7 s
128	16	8	1	24	4194305	3.00 GiB	5.81 s	42.0 s
128	32	4	1	36	4194305	4.50 GiB	6.82 s	48.6 s
128	64	2	1	66	4194305	8.25 GiB	9.00 s	62.5 s
256	8	32	4	64	2097153	4.00 GiB	6.15 s	44.6 s
256	16	16	1	32	2097153	2.00 GiB	6.01 s	43.1 s
256	32	8	1	40	2097153	2.50 GiB	6.94 s	49.5 s
256	64	4	1	68	2097153	4.25 GiB	9.10 s	63.3 s
512	8	64	8	128	1048577	4.00 GiB	7.46 s	52.9 s
512	16	32	2	64	1048577	2.00 GiB	6.61 s	46.7 s
512	32	16	1	48	1048577	1.50 GiB	7.11 s	50.9 s
512	64	8	1	72	1048577	2.25 GiB	9.39 s	64.8 s
1024	8	128	16	256	524289	4.00 GiB	10.00 s	68.6 s
1024	16	64	4	128	524289	2.00 GiB	7.52 s	53.2 s
1024	32	32	1	64	524289	1.00 GiB	7.46 s	52.4 s
1024	64	16	1	80	524289	1.25 GiB	9.48 s	65.8 s
2048	8	256	32	512	262145	4.00 GiB	15.1 s	101.8 s
2048	16	128	8	256	262145	2.00 GiB	9.47 s	65.4 s
2048	32	64	2	128	262145	1.00 GiB	8.33 s	58.0 s
2048	64	32	1	96	262145	768.00 MiB	9.68 s	67.3 s
4096	8	512	64	1024	131073	4.00 GiB	25.3 s	164.7 s
4096	16	256	16	512	131073	2.00 GiB	13.4 s	88.4 s
4096	32	128	4	256	131073	1.00 GiB	10.0 s	66.8 s
4096	64	64	1	128	131073	512.00 MiB	10.3 s	69.1 s

TABLE 4. AUTHENTICATED-UPDATE SENSITIVITY FOR ONE EDVC MAINTAINER. EVERY ROW USES GENERAL UPDATES TO MAXIMALLY SPACED ACTIVE POSITIONS, MERKLE HEIGHT 50, AND ACTIVE PREFIX 268,435,456 FIELD ELEMENTS (8 GIB OF 32-BYTE AUTHENTICATED VALUES). THE BENCHMARK INITIALIZES THIS PREFIX WITH DEFAULT VALUES AND TIMES THE ORDINARY UpdateDataFragment PATH FOR EXISTING ACTIVE POSITIONS. THE 16,384-UPDATE WORKLOAD HAS 262,165 CHANGED MERKLE LABELS AND 229,398 PROOF NODES; THE 131,072-UPDATE WORKLOAD HAS 1,703,957 CHANGED MERKLE LABELS AND 1,441,814 PROOF NODES. THE ACTIVE PREFIX INDUCES 536,870,933 MAINTAINED MERKLE CELLS UNDER PREORDER LINEARIZATION. THE TABLE VARIES ONLY Seg AND κ. "ACTIVE SEGS." IS $\lceil 536{,}870{,}933/\mathsf{Seg} \rceil$. "STORED ELEMS./SEG." IS THE NUMBER OF CODED FIELD ELEMENTS IN ONE HELPER'S ADVERTISED RESPONSIBILITY FOR ONE ACTIVE SEGMENT, $s + \kappa \cdot \mathsf{RowsPerClaim}$. "STATE" IS $\lceil 536{,}870{,}933/\mathsf{Seg} \rceil \cdot (s + \kappa \cdot \mathsf{RowsPerClaim}) \cdot 32$ BYTES. THE CODED UPDATE WORK APPLIES THE LOCAL PATCH TO THE STORED ELEMENTS IN TOUCHED SEGMENTS. TIMES ARE LOCAL SEQUENTIAL HANDLER TIMES INCLUDING TRANSITION VALIDATION, NEW-LABEL COMPUTATION, DELTA DERIVATION, AND DATA-FRAGMENT UPDATE; THEY ARE ARITHMETIC MEANS FROM THE ADAPTIVE SAMPLING PROTOCOL ABOVE.

10.3. End-to-End BORGLOG Costs

All BORGLOG experiments use a sparse Merkle tree of height $h = 50$, and hence capacity $\mathsf{N} = 2^{50}$. This choice is deliberately conservative. For a Bitcoin-scale capacity comparison, 4 MiB of serialized 32-byte authenticated values occupies 2^{17} positions, so the tree can hold 2^{33} such blocks. At Bitcoin's ten-minute block interval, this capacity would last more than 160,000 years, far beyond the intended lifetime of a Bitcoin-scale archive.

The experiments below test whether BORGLOG is practical across its principal operations: append processing and storage, point and range openings in the presence of Byzantine responses, and maintainer repair, onboarding, and reconfiguration.

Append processing. Append processing remains fast across all measured coded-array configurations. Table 5 varies the append batch size over 1, 2, 4, and 8 MiB and varies (Seg, κ) to show how the workload and coded-array configuration affect one maintainer's processing time and data-fragment growth. One independent repetition starts a fresh maintainer from an empty log and processes 64 consecutive accepted batches of the given size. Each benchmark process first executes and discards one complete 64-batch warmup sequence, then records between 10 and 160 independent sequences under the stopping rule above. The mean of the 64 within-sequence batch timings is one independent timing sample; the individual batch timings are retained but are not treated as independent. The table reports the mean across the retained sequence samples together with the mean increase in the maintainer's advertised data fragment. The workload sizes count serialized 32-byte authenticated values and are distinct from the segment size Seg. The timed operation includes active-prefix extension and materialization of the newly active maintained state; it does not assume a pre-extended suffix. Within each fixed workload and configuration, consecutive batch times remain nearly constant as the log grows because each append processes the newly active suffix and affected Merkle labels without reprocessing the full existing prefix. The measurements therefore characterize the cost of later batches of the same size, not only appends near an empty log.

Across all reported configurations, 1 MiB batches take between 59 and 148 ms, while 8 MiB batches take between 469 ms and 1.192 s. Within each row, doubling the batch size approximately doubles the processing time, showing that the cost scales with the newly appended data. The segment size affects processing time through the number of newly materialized segments. For example, with $\kappa = 8$, a 4 MiB batch materializes 4096 segments and takes 275 ms at $\mathsf{Seg} = 64$, but materializes 2048 segments and takes 257 ms at $\mathsf{Seg} = 128$. Even the largest measured batch takes less than 1.2 seconds under every reported configuration.

Append processing is substantially cheaper than a general rewrite. At $(\mathsf{Seg}, \kappa) = (128, 8)$, appending 131,072 values takes 257 ms, compared with 1.469 s for rewriting the same number of active values in Table 3. Fresh suffix positions have

Seg	κ	Append workload; entry: time (ms) / fragment size (MiB)			
		1 MiB	2 MiB	4 MiB	8 MiB
64	8	66 / 0.500	135 / 1.000	275 / 2.000	544 / 4.000
64	16	67 / 0.625	133 / 1.250	273 / 2.500	544 / 5.000
64	32	112 / 1.063	225 / 2.125	449 / 4.250	919 / 8.500
64	64	147 / 2.031	296 / 4.063	586 / 8.125	1178 / 16.250
128	8	63 / 0.500	125 / 1.000	257 / 2.000	515 / 4.000
128	16	61 / 0.375	121 / 0.750	251 / 1.500	506 / 3.000
128	32	62 / 0.563	125 / 1.125	249 / 2.250	508 / 4.500
128	64	148 / 1.031	299 / 2.063	589 / 4.125	1192 / 8.250
256	8	64 / 0.500	127 / 1.000	254 / 2.000	514 / 4.000
256	16	59 / 0.250	117 / 0.500	242 / 1.000	481 / 2.000
256	32	60 / 0.313	118 / 0.625	236 / 1.250	478 / 2.500
256	64	62 / 0.531	124 / 1.063	249 / 2.125	496 / 4.250
512	8	78 / 0.500	156 / 1.000	312 / 2.000	619 / 4.000
512	16	61 / 0.250	120 / 0.500	240 / 1.000	489 / 2.000
512	32	59 / 0.188	117 / 0.375	234 / 0.750	477 / 1.500
512	64	59 / 0.281	121 / 0.563	239 / 1.125	469 / 2.250
1024	8	83 / 0.500	165 / 1.000	323 / 2.000	646 / 4.000
1024	16	64 / 0.250	126 / 0.500	256 / 1.000	511 / 2.000
1024	32	60 / 0.125	118 / 0.250	236 / 0.500	479 / 1.000
1024	64	60 / 0.156	118 / 0.313	242 / 0.625	481 / 1.250
4096	8	92 / 0.500	179 / 1.000	350 / 2.000	686 / 4.000
4096	16	99 / 0.250	190 / 0.500	376 / 1.000	746 / 2.000
4096	32	69 / 0.125	133 / 0.250	264 / 0.500	532 / 1.000
4096	64	66 / 0.063	126 / 0.125	249 / 0.250	498 / 0.500

TABLE 5. PER-MAINTAINER APPEND-PROCESSING COST. WORKLOADS CONTAIN 32,768, 65,536, 131,072, AND 262,144 AUTHENTICATED 32-BYTE VALUES. EACH ENTRY REPORTS *mean processing time in milliseconds / mean newly materialized fragment size in MiB*. ONE INDEPENDENT TIMING SAMPLE IS THE MEAN OVER A FRESH SEQUENCE OF 64 CONSECUTIVE BATCHES. PROCESSING INCLUDES ACTIVE-PREFIX EXTENSION AND INSTALLATION OF THE NEW SUFFIX.

public default old values, and whole newly exposed segments can be materialized directly; only boundary segments and upper Merkle labels require ordinary delta updates.

Append storage. The storage added by an append is the increase in one maintainer's advertised data fragment. If the append grows the maintained array from L_t to L_{t+1}, this increase is

$$\left(\left\lceil \frac{L_{t+1}}{\mathsf{Seg}} \right\rceil - \left\lceil \frac{L_t}{\mathsf{Seg}} \right\rceil\right)(s + \kappa\mathsf{RowsPerClaim})\mathsf{FieldBytes}.$$

The maintained array includes both leaves and newly active Merkle labels, so appending U values can make $L_{t+1} - L_t$ nearly $2U$. When the balanced-storage condition holds, each maintainer stores about a $2/\kappa$ share of this newly active maintained state. When $\mathsf{Seg} < \kappa^2$, the condition fails and auxiliary row-label storage can dominate, making the advertised data fragment larger.

The 4 MiB column exhibits both cases. Along $\mathsf{Seg} = \kappa^2$, the configurations $(64, 8)$, $(256, 16)$, $(1024, 32)$, and $(4096, 64)$ add 2.000, 1.000, 0.500, and 0.250 MiB, respectively, to one advertised data fragment. In particular, at $(\mathsf{Seg}, \kappa) = (4096, 64)$, appending 4 MiB of log values adds only 0.250 MiB to the measured maintainer's data fragment, including its share of the newly active Merkle labels. Thus doubling κ halves the stored fragment growth when the segment size is chosen to satisfy the balanced-storage condition. At fixed $\mathsf{Seg} = 1024$, increasing κ from 8 to 32 reduces the fragment growth from 2.000 to 0.500 MiB, but $\kappa = 64$ violates the condition and raises the growth to 0.625 MiB because each claim must store at least one auxiliary row label.

Point openings. Point openings remain efficient across all measured configurations, both without faults and with a large number of corrupted helper responses. Table 6 measures a fresh opening of one authenticated value and its Merkle path, separately reporting one helper's work, the total helper-response size, and the client's $\mathsf{Combine} + \mathsf{Verify}$ work after responses arrive. Because the Merkle height h is fixed, the path length and point-opening cost do not grow with the current prefix length. The reconstruction threshold κ, together with the selected Byzantine-error budget errTol_t, determines the number of required responses, download, and client reconstruction or correction work. The segment size Seg affects helper work through evaluation of the local primary slice when constructing an $\mathsf{OpenShare}$. Helpers construct their responses in parallel; the helper column therefore reports one helper's work rather than aggregate helper work or the sum of all helper times.

The corrupted-response experiments in Tables 6 and 7 set $\mathsf{errTol}_t = \kappa$, set $\mathsf{missTol}_t = 0$, and collect $3\kappa = \kappa + 2\mathsf{errTol}_t$ replies. We select a fixed set of κ helpers as corrupted and change one coded contribution from each. Scalar contributions are changed by adding a nonzero field offset; for a full-segment contribution, nonzero offsets are added across its returned field elements. In the point-opening experiment, corrupted contributions are distributed round-robin across active Merkle path nodes. In the range-opening experiment, they are distributed round-robin across the full-segment and point reads used for the range query. In both experiments, this distribution forces the client to discover corrupted helpers at different coded-array reads within the query, rather than concentrating every corruption in one read.

Without corrupted responses, every measured point open-

		Helper work and response size			Client Combine + Verify ms	
Seg	κ	Helper work ms	κ set helpers; KiB	3κ set helpers; KiB	κ set client ms	3κ set client ms
64	8	0.193	8 / 9.8	24 / 29.3	0.303	1.435
64	16	0.182	16 / 19.5	48 / 58.6	0.396	10.678
64	32	0.178	32 / 39.1	96 / 117.3	0.684	105.828
64	64	0.173	64 / 78.2	192 / 234.6	1.700	796.059
256	8	0.248	8 / 9.8	24 / 29.3	0.293	1.439
256	16	0.211	16 / 19.5	48 / 58.6	0.389	10.688
256	32	0.193	32 / 39.1	96 / 117.3	0.681	105.915
256	64	0.179	64 / 78.2	192 / 234.6	1.715	796.213
1024	8	0.510	8 / 9.8	24 / 29.3	0.287	1.425
1024	16	0.335	16 / 19.5	48 / 58.6	0.390	10.705
1024	32	0.253	32 / 39.1	96 / 117.3	0.688	107.185
1024	64	0.206	64 / 78.2	192 / 234.6	1.726	805.832
4096	8	1.525	8 / 9.8	24 / 29.3	0.280	1.413
4096	16	0.861	16 / 19.5	48 / 58.6	0.384	10.714
4096	32	0.512	32 / 39.1	96 / 117.3	0.689	105.686
4096	64	0.332	64 / 78.2	192 / 234.6	1.692	795.757

TABLE 6. FRESH POINT-OPENING COSTS FOR ONE AUTHENTICATED VALUE AT MERKLE HEIGHT $h = 50$. THE κ-HELPER COLUMNS USE NO CORRUPTED HELPERS. THE 3κ-HELPER COLUMNS USE RESPONSES FROM 3κ HELPERS WITH κ CORRUPTED HELPERS. RESPONSE-SIZE COLUMNS REPORT THE TOTAL ACROSS THE INDICATED HELPER SET. CLIENT TIMINGS MEASURE LOCAL Combine + Verify AFTER RESPONSES ARRIVE. HELPER WORK INCLUDES CONSTRUCTING AND SERIALIZING ONE RESPONSE.

ing takes at most 1.525 ms at one helper and 1.726 ms at the client. Along $\mathsf{Seg} = \kappa^2$, increasing κ from 8 to 64 raises one helper's work only from 0.193 to 0.332 ms, client Combine+Verify from 0.303 to 1.692 ms, and total response size from 9.8 to 78.2 KiB. More importantly, point openings remain practical under the largest tested Byzantine-error budget. With $\mathsf{errTol}_t = \kappa$, one third of the 3κ responses are corrupted; nevertheless, every measured configuration completes client Combine + Verify in less than 0.81 s. At the largest balanced configuration, $(\mathsf{Seg}, \kappa) = (4096, 64)$, the client recovers and verifies the opening in 795.757 ms from 192 responses totaling 234.6 KiB, despite 64 of those responses being corrupted.

The fixed-κ rows isolate the effect of Seg: at $\kappa = 8$, raising Seg from 64 to 4096 increases one helper's work from 0.193 to 1.525 ms, while the fault-free client time changes only from 0.303 to 0.280 ms and the total download remains 9.8 KiB. Thus κ and the error budget determine client reconstruction work, whereas Seg primarily changes the work performed by each helper.

Remark 4 (Batched point openings)**.** *Table 6 measures one client request at a time. When requests from several clients touch the same segment, each helper can use FFT-style batch evaluation of its primary slice at the requested points, amortizing the* Seg*-dependent work across clients. We do not implement or evaluate this cross-client batching.*

Range openings. Table 7 gives $\kappa = 64$ BORGLOG contiguous range-opening measurements for the range-opening path described in Section 7. The benchmark fixes the balanced point $(\mathsf{Seg}, \kappa) = (4096, 64)$ and varies only the returned byte range. In the table, full-segment reads decode complete coded-array segments, while point reads decode individual boundary leaves or authentication labels not covered by a full-segment read.

For large contiguous ranges, most requested maintained cells are covered by whole segments. For each such segment, a helper returns the coded slice it already stores; for a boundary point read, it evaluates its local primary slice. Serving the request therefore consists primarily of assembling and serializing stored slice values, plus evaluating the small number of boundary points. The helper does not reconstruct the returned range, verify Merkle paths, or run error correction; those costs are in the client Combine + Verify columns. Across κ correct helpers, the full-segment slices have the aggregate size of one segment, keeping the reconstruction payload close to the returned leaves and the Merkle labels needed to verify them. Increasing κ divides this payload among more helpers rather than increasing it.

For the 128 MiB query, each helper returns a 4112.7 KiB coded share. Thus any $\kappa = 64$ correct responses provide about 257 MiB, only $2.01\times$ the requested range. The requested leaves account for 128 MiB; most of the remainder consists of active Merkle labels needed to construct and verify the range opening. Because the client does not know which helpers are corrupted, the experiment collects 3κ responses to correct up to $\mathsf{errTol}_t = \kappa$ errors. These additional responses provide the redundancy needed for error correction; they do not change the $2.01\times$ reconstruction overhead. Internally, the query uses 2048 full-segment coded-array reads and only 27 point reads for boundary cells. The "Array point reads" column counts these internal single-cell reads, not separate authenticated point openings. Even for the 128 MiB range, one helper constructs and serializes its complete response in only 6.307 ms.

The downloaded authentication material also makes corrupted responses efficient to handle. With no corrupted responses, Algorithm 1 uses ordinary Reed–Solomon interpolation from κ replies and certifies the decoded labels through the Merkle tree. If a check fails, it invokes the implementation's cubic Berlekamp–Welch decoder only for the corresponding coded-array read, uses that correction to identify previously unknown corrupted helpers, and treats their later responses as erasures. Every full correction

Range MiB	Full-segment reads	Array point reads	Per-helper KiB	Helper work ms	0 corrupted helpers (s)	64 corrupted helpers (s)
1	16	41	33.9	0.031	0.164	2.842
4	64	37	130.0	0.100	0.653	5.763
8	128	35	258.3	0.200	1.351	6.734
32	512	31	1028.9	1.237	6.706	13.663
128	2048	27	4112.7	6.307	30.249	42.510

TABLE 7. CONTIGUOUS RANGE OPENINGS AT MERKLE HEIGHT $h = 50$, $\mathsf{Seg} = 4096$, AND $\kappa = 64$, AFTER 16 COMMITTED 8 MIB APPEND BATCHES. BOTH CLIENT COLUMNS PROCESS 192 HELPER RESPONSES; THE CORRUPTED CASE CONTAINS 64 CORRUPTED HELPERS. PER-HELPER SIZE AND WORK INCLUDE CONSTRUCTING AND SERIALIZING ONE RESPONSE. THE LAST TWO COLUMNS REPORT LOCAL $\mathsf{Combine} + \mathsf{Verify}$ TIME AFTER RESPONSES ARRIVE AND EXCLUDE NETWORK TRANSMISSION AND HELPER WORK.

therefore discovers at least one new corrupted helper, so an entire range opening requires at most errTol_t full corrections rather than one for every segment or point read. For fixed parameters, this is a bounded additive cost over the erasure-only path: for the 128 MiB query, 64 corrupted responses among 192 increase client time from 30.249 s to 42.510 s, only a $1.41\times$ overhead.

Repair and reconfiguration. Repair and supported reconfiguration remain efficient because they avoid decoding the complete current authenticated vector. Repair/onboarding materializes only the target advertised responsibility, while reconfiguration reuses an existing responsibility over the same authenticated vector. Table 8 quantifies both operations after the log contains 393,216 inserted 32-byte values, or 12 MiB. The maintained array also contains the active Merkle labels, so 12 MiB denotes the inserted log-value prefix rather than the complete maintained array. These operations depend on the current state, not on how the inserted values were divided into append batches. Fragment size, receiver work, one helper's work, and total helper download are normalized per 4 MiB of inserted values, so the measured totals are three times the reported entries.

This normalization follows the construction. At fixed (Seg, κ), repair and same-segment reconfiguration repeat the same operation for every active segment, while segment-double repeats its local operation for adjacent segment pairs. In each case, the total download is the sum of the corresponding segment payloads. Extending the prefix adds complete segments on which the same work is repeated; only fixed setup costs and the final partially filled segment do not scale with the number of active segments. Even at the largest segment size in this table, $\mathsf{Seg} = 4096$, the maintained array induced by the measured prefix contains 193 active segments. The 12 MiB experiment therefore exercises many repetitions of the segment operation and supports reporting a per-4-MiB cost for longer prefixes at the same parameters.

Repair/onboarding starts with no usable local state and materializes one current advertised responsibility. Reconfiguration starts from an existing responsibility and measures the adjacent threshold changes $\kappa \to 2\kappa$ and $2\kappa \to \kappa$, together with the coupled transition $(\mathsf{Seg}, \kappa) \to (2\mathsf{Seg}, 2\kappa)$. Each group separates receiver work, one helper's work, and total helper data downloaded. For $2\kappa \to \kappa$, the helper column measures one helper computing target evaluations, while the total download also includes the other source primary slice transferred directly. The coupled transition applies the local segment-double procedure and requires neither helper work nor helper data.

Repair is practical at the balanced operating point. At $(\mathsf{Seg}, \kappa) = (4096, 64)$, repairing one maintainer's responsibility for the entire 12 MiB prefix takes 247.8 ms of receiver work, 3.9 ms for one helper, and 0.753 MiB of total download. In every row, repair download equals the target fragment size. The parameter sweep explains why balance matters: at $\kappa = 64$, increasing Seg from the unbalanced value 128 to 4096 reduces the normalized fragment and download from 4.126 to 0.251 MiB per 4 MiB of inserted values and reduces normalized receiver work from 661.7 to 82.6 ms.

Reconfiguration is similarly fast because it reuses the maintainer's existing responsibility. Over the complete 12 MiB prefix at $(\mathsf{Seg}, \kappa) = (4096, 64)$, changing $\kappa \to 2\kappa$ takes 204.9 ms of receiver work, 8.4 ms for one helper, and 0.747 MiB of total download; changing $2\kappa \to \kappa$ takes 51.3 ms, 2.1 ms, and 0.561 MiB, respectively. The coupled segment-double transition takes 92.1 ms at the receiver with no helper work or download. The two κ-rescale operations download only the missing target material, while the zero-download result follows from the local segment-double construction.

The final sweep tests how far this practical behavior persists as segments grow. The balanced-storage condition prevents auxiliary state from dominating, but it does not imply that arbitrarily large segments improve every operation. Table 9 fixes $\kappa = 64$ and varies Seg from 128 through 65536 to measure storage, append processing, repair, and reconfiguration on both sides of the boundary $\mathsf{Seg} = \kappa^2 = 4096$.

The balanced and moderately larger configurations remain efficient. From $\mathsf{Seg} = 4096$ to 16384, a maintainer processes a 4 MiB append in 249–294 ms, while normalized repair work is 82.6–170.2 ms per 4 MiB of inserted values. Increasing Seg to 65536 raises these costs to 874 ms and 551.1 ms, respectively. Storage no longer improves over this range: fragment size changes only from 0.251 MiB at $\mathsf{Seg} = 4096$ to 0.255 and 0.271 MiB at $\mathsf{Seg} = 16384$ and 65536. Although the number of active segments falls from 193 to 49 and then 13, the greater work within each segment eventually outweighs that reduction for append processing and repair.

Reconfiguration remains fast throughout the balanced-or-larger rows: every normalized receiver time at $\mathsf{Seg} \geq 4096$

(a) Repair/onboarding and $\kappa \to 2\kappa$ reconfiguration

Seg	κ	Fragment size per 4 MiB (MiB)	Repair/onboard at (Seg, κ)			Reconfig $\kappa \to 2\kappa$		
			Receiver (ms)	Helper (ms)	Download (MiB)	Receiver (ms)	Helper (ms)	Download (MiB)
128	8	2.000	530.3	33.4	2.000	47.4	6.9	0.938
128	16	1.500	153.5	16.1	1.500	141.0	6.2	1.938
128	32	2.250	222.5	18.3	2.250	516.1	6.2	3.938
128	64	4.126	661.7	23.9	4.126	2127.7	7.2	7.939
256	8	2.001	588.6	43.4	2.001	25.1	4.7	0.469
256	16	1.000	168.3	9.4	1.000	71.0	3.7	0.969
256	32	1.250	143.0	9.6	1.250	260.4	3.7	1.969
256	64	2.126	345.7	12.1	2.126	1070.3	4.9	3.970
1024	8	2.003	931.2	94.8	2.003	32.0	11.8	0.469
1024	16	1.001	270.7	16.0	1.001	18.9	2.2	0.243
1024	32	0.501	116.9	3.4	0.501	65.3	2.0	0.493
1024	64	0.626	118.5	3.5	0.626	270.5	3.1	0.993
4096	8	2.010	2351.9	60.1	2.010	58.6	37.8	0.471
4096	16	1.005	616.4	41.1	1.005	22.3	5.6	0.243
4096	32	0.503	202.3	6.5	0.503	17.1	1.6	0.124
4096	64	0.251	82.6	1.3	0.251	68.3	2.8	0.249

(b) $2\kappa \to \kappa$ reconfiguration and coupled segment-double

Seg	κ	Fragment size per 4 MiB (MiB)	Reconfig $2\kappa \to \kappa$			Reconfig $\mathsf{Seg} \to 2\mathsf{Seg},\ \kappa \to 2\kappa$		
			Receiver (ms)	Helper (ms)	Download (MiB)	Receiver (ms)	Helper (ms)	Download (MiB)
128	8	2.000	175.8	18.2	1.375	34.4	–	0
128	16	1.500	47.0	6.7	1.188	75.8	–	0
128	32	2.250	140.8	5.9	2.063	235.0	–	0
128	64	4.126	515.2	5.5	4.001	860.8	–	0
256	8	2.001	195.6	28.8	1.375	30.6	–	0
256	16	1.000	24.6	4.5	0.719	38.0	–	0
256	32	1.250	70.9	3.4	1.094	117.8	–	0
256	64	2.126	261.5	3.0	2.032	430.5	–	0
1024	8	2.003	257.5	80.1	1.377	28.1	–	0
1024	16	1.001	31.6	11.5	0.720	18.6	–	0
1024	32	0.501	18.7	1.9	0.368	30.5	–	0
1024	64	0.626	65.5	1.2	0.555	110.5	–	0
4096	8	2.010	207.7	44.9	1.382	29.1	–	0
4096	16	1.005	57.9	37.1	0.722	18.7	–	0
4096	32	0.503	22.0	5.0	0.369	16.1	–	0
4096	64	0.251	17.1	0.7	0.187	30.7	–	0

TABLE 8. MAINTAINER REPAIR, ONBOARDING, AND RECONFIGURATION AFTER 393,216 INSERTED 32-BYTE LOG VALUES (12 MIB). FRAGMENT SIZE, RECEIVER TIME, HELPER TIME, AND DOWNLOAD ARE NORMALIZED PER 4 MIB OF INSERTED VALUES; MEASURED TOTALS ARE THREE TIMES THE REPORTED ENTRIES. "RECEIVER" IS THE LOCAL WORK OF THE RECOVERING OR RECONFIGURING MAINTAINER, "HELPER" IS THE WORK OF ONE HELPER SERVING THE REQUEST, AND "DOWNLOAD" IS THE TOTAL MODELED HELPER DATA RECEIVED, EXCLUDING TRANSPORT FRAMING. THE κ-RESCALE GROUPS ARE ADJACENT FACTOR-TWO THRESHOLD CHANGES IN OPPOSITE DIRECTIONS. THE SEGMENT-DOUBLE GROUP CHANGES (Seg, κ) TO $(2\mathsf{Seg}, 2\kappa)$ WHILE KEEPING s FIXED; THE TRANSITION IS COMPUTED LOCALLY AND USES NO HELPER PAYLOAD.

is below 70 ms per 4 MiB of inserted values. Reducing the number of active segments initially reduces the reconfiguration work. From $\mathsf{Seg} = 4096$ to 16384, the $64 \to 128$ time falls from 68.3 to 18.1 ms and the coupled segment-double time falls from 30.7 to 19.1 ms, while the $128 \to 64$ time changes only from 17.1 to 19.3 ms. At $\mathsf{Seg} = 65536$, all three are still only 21.2, 28.5, and 29.5 ms, respectively. Thus no segment size minimizes append, repair, and all reconfiguration operations simultaneously; Seg must be selected for the expected operation mix rather than from active-segment count alone.

11. Related Work

Vector commitments and authenticated data structures provide short public digests and verifiable openings [11], [17]. Updatable VCs and Cauchyproofs refresh stale openings after committed updates [11]–[13]. Sparse Merkle trees give hash-based membership over large sparse domains [15], [18], while stateless-client and accumulator work moves witnesses to transactions, users, or compact accumulators [19]–[22]. EDVC instead distributes the state used to produce fresh witnesses.

Gao et al. [14] introduce distributed vector commitments (DVCs) for a vector of maximum length N partitioned into M equal subvectors. Machine P_k exclusively holds v_k and is responsible for openings to its positions, while a designated coordinator assembles the global commitment and shared proof material from the machines' contributions. Their DVC construction supports succinct proofs, batching, aggregation, additive homomorphism, commitment-and-proof updates, and faster bulk proof precomputation by running the fixed subvector owners in parallel. Once all position proofs have been precomputed, a point read downloads the requested value and an $O(\log M)$-size proof in one response. This avoids a threshold download, but opening service remains

Seg	Segment bytes	Active segments	Fragment size/4 MiB	Append	Repair at 64	Reconfig $64 \to 128$	Reconfig $128 \to 64$	Reconfig $\mathsf{Seg} \to 2\mathsf{Seg},\ 64 \to 128$
128	4 KiB	6145	4.126 MiB	589 ms	661.7 ms	2127.7 ms	515.2 ms	860.8 ms
1024	32 KiB	769	0.626 MiB	242 ms	118.5 ms	270.5 ms	65.5 ms	110.5 ms
4096	128 KiB	193	0.251 MiB	249 ms	82.6 ms	68.3 ms	17.1 ms	30.7 ms
16384	512 KiB	49	0.255 MiB	294 ms	170.2 ms	18.1 ms	19.3 ms	19.1 ms
65536	2048 KiB	13	0.271 MiB	874 ms	551.1 ms	21.2 ms	28.5 ms	29.5 ms

TABLE 9. LARGE-SEGMENT SENSITIVITY AT FIXED $\kappa = 64$. "APPEND" IS THE MEAN PROCESSING TIME FOR ONE 4 MIB BATCH IN THE 64-BATCH SEQUENCE OF TABLE 5. FRAGMENT SIZE, REPAIR, AND RECONFIGURATION ARE MEASURED AT THE SAME 12 MIB INSERTED-VALUE PREFIX AS TABLE 8 AND NORMALIZED PER 4 MIB OF INSERTED VALUES. "ACTIVE SEGMENTS" IS THE ACTUAL COUNT AT THAT PREFIX. REPAIR/ONBOARDING STARTS WITH NO USABLE LOCAL STATE. RECONFIGURATION STARTS FROM EXISTING MAINTAINED STATE AND MEASURES BOTH ADJACENT SAME-SEGMENT κ-RESCALE DIRECTIONS AND THE LOCAL SEGMENT-DOUBLE TRANSITION TO $(2\mathsf{Seg}, 128)$.

dependent on one machine: if P_k is unavailable or returns an invalid response, the primitive provides no alternative authenticated source. Correspondingly, its security definition treats the machines collectively as one prover and defines neither κ-out-of-n recovery nor recovery from missing or Byzantine responses. The small online response relies on substantial fixed-capacity state and preprocessing: cached position proofs occupy $N+M$ group elements, each machine's proving and update keys require $O((N/M)\log M)$ storage, an uncached proof costs $O(N/M)$ to generate, and generating and refreshing all proofs cost $O((N/M)\log(N/M))$ and $O(N/M + \log M)$, respectively. These costs scale with maximum capacity rather than the populated part of the vector, so succinct proof size does not provide EDVC compactness. The fixed layout also defines no advertisements or caller-selected registries, heterogeneous responsibilities, sparse-vector scaling, extension beyond the preallocated capacity, repair, onboarding, or reconfiguration. Its use of *maintainability* concerns refreshing precomputed proofs; appendix variants reduce that cost without adding the missing availability and lifecycle properties. Borg instead codes the growing sparse-Merkle maintained state across caller-selected registries, making openings threshold-recoverable and the state compact, repairable, and reconfigurable under an unchanged anchor. Whereas their construction uses a structured pairing-based setup and pairing-based hardness assumptions, Borg assumes only collision-resistant hash functions and random oracles.

Data-availability protocols make encoded pieces of one proposed payload checkable under a payload commitment, retrievable, or both. Fraud proofs and sampling use authenticated pieces of a proposed block; EIP-4844 commits blobs and PeerDAS samples authenticated blob pieces; VID/AVID disperse one encoded message, and committing AVID adds fragment checks. Semi-AVID-PR adds retrievability certificates for one committed payload. Erasure-coding proof systems make encoded fragments independently checkable against one block commitment; DAS with repair and Walrus extend this fixed-payload setting with repair or storage for encoded payloads [1]–[8], [10]. These systems authenticate or recover fragments of the encoded object named by one block, blob, or storage reference. EDVC instead codes the maintained state of one evolving authenticated vector. Its client-visible operations reconstruct fresh openings under a named anchor; its maintenance operations materialize the data fragments used for repair and onboarding. A client can authenticate any position or range already in the active prefix under the current anchor. In this respect, EDVC generalizes the fixed-payload abstraction: instead of distributing one separately named payload per invocation, it maintains the opening state for the entire growing history.

iDDA studies audited archival storage for an ever-growing database [9]. It exposes provisioned space as a node-level parameter, and its non-uniform extension allows different nodes to contribute different amounts. A node stores a corresponding evolving shard and can continue using that capacity as the database grows. Audits check that the claimed space is devoted to useful archive data; recoverability is tied to the aggregate space of audited nodes, and replication security relates rewards to actual storage consumed. Its construction nevertheless relies on a distinguished trusted data publisher that maintains the archive data structure and supplies node state during joins and updates. The paper identifies eliminating this trust efficiently as a practical direction and discusses an IVC-based approach in its full version. Conversely, EDVC exposes a different resource interface and decentralizes this stateful publisher role: each advertisement names a coded responsibility whose size follows from its coding parameters and the current vector length, no single online party is assumed to hold the full vector-commitment prover state; each maintainer node processes updates locally; and any sufficiently-large set of honest nodes suffices for retrievals and reconfiguration.

Feist et al. [23] introduce robust distributed arrays as a networking layer for DAS. Their construction hashes nodes into a fixed $k_1 \times k_2$ grid and replicates each chunk across one column; a read succeeds after receiving one position-valid response from an honest node in the corresponding cell. Its proof assumes static corruptions and random honest placement. A cell without an honest node makes its assigned positions unretrievable, and the definition explicitly accepts such failures for up to εm positions per party and time. The construction also provides no reconfiguration of its fixed (m, k_1, k_2) layout. Our coded array instead provides κ-out-of-n recovery for every position under the registry fault condition, tolerates Byzantine and missing replies, and supports extension, repair, and parameter reconfiguration. It can similarly store authenticated DAS codewords without full-chunk replication, at the cost of requiring κ correct coded replies rather than one honest replica.

Reconfigurable erasure codes. Conventional erasure codes fix both the number of data fragments and the reconstruction threshold. Changing either parameter can always be done by reconstructing the represented data and

then materializing every data fragment required by the new parameters. PACEMAKER reduces the resulting I/O through advance fragment placement and transition scheduling [24]; BART balances conversion work across maintainers [25]; and Okapi separates the placement of represented data from the sets encoded together, avoiding rewrites of unchanged data [26]. These are system-level improvements to a transition that replaces one complete encoded representation with another.

Other work changes the code or fragment assignment for particular parameter transitions. Elastic Reed–Solomon codes support increasing or decreasing κ while keeping fixed the number of additional coded fragments used for fault tolerance. They reuse existing data fragments under source and target parameters selected in advance [27]. StripeMerge handles the specific transition that combines two encoded arrays with reconstruction threshold κ into one with threshold 2κ, again keeping the number of additional coded fragments fixed. Its zero-communication case requires a special fragment assignment: the fragments holding the represented data from the two inputs must be held by distinct maintainers, while each corresponding pair of additional coded fragments must already be held by the same maintainer. Finding such pairings exactly is infeasible at realistic scale, and its practical heuristics still move data fragments [28].

Convertible codes are the closest theoretical comparison. They jointly design the encoding before and after a parameter change. A party performing the change can retain unchanged parts of existing data fragments, download selected parts of other fragments, compute the missing fragments under the new parameters, and send them to the target maintainers [29], [30]. This can reduce access and communication relative to reconstructing all represented data. Nevertheless, the operation still produces a complete collection of data fragments under one new parameter choice, and the supported old and new parameter choices must be fixed when the code is constructed.

Our coded array does not require such a complete transition. Its encoding does not fix the number of maintainers: each caller chooses the maintainers in its local registry according to its fault budgets. A maintainer can therefore join or leave without changing any existing data fragment. For a supported factor-two change to κ, one maintainer reconfigures only its advertised responsibility, reusing its primary and auxiliary data and materializing missing target values through the existing repair procedures. Neither the caller nor a maintainer reconstructs the complete fixed-size array, other maintainers need not reconfigure, and advertisements under both parameter choices may coexist under the same public anchor. The coupled transition $(\mathsf{Seg}, \kappa) \to (2\mathsf{Seg}, 2\kappa)$ is performed entirely locally, without helper communication.

12. Discussion and Future Work

Fragment archival nodes. Even without integrating BORGLOG into consensus, it enables an archival role between a full archival node (keeping full history with $O(L)$ storage) and a pruned node (not supporting retrieval of history). Each BORGLOG maintainer node can retain and advertise a coded fragment of the archive, at some supported operating point (κ, Seg), using local storage just $O(L/\kappa)$. Advertisements need not pass through a central directory: maintainers may publish them on social media or in any other public venue, and clients can shop among compatible offers when assembling their own local registries. Maintainers that retain the auxiliary rows can support efficient repair and onboarding for other maintainers, while a maintainer interested only in serving client retrievals may drop them. Under the balanced-storage condition, this removes one of two equal-sized leading storage components and therefore halves its leading storage. Even if every maintainer drops the auxiliary rows, a node can still reconstruct its responsibility by retrieving and replaying the log; the auxiliary rows provide a substantially more efficient recovery path. When a primary-only maintainer reaches its storage limit, it can apply the supported coupled transition $(\mathsf{Seg}, \kappa) \to (2\mathsf{Seg}, 2\kappa)$ entirely locally and without helper traffic. This halves its currently occupied fragment storage, allowing the log to double in length before reaching the same limit again. This provides a concrete, non-invasive deployment path for existing blockchains such as Bitcoin: BORGLOG can follow the consensus-determined block sequence as a separate archival and retrieval layer without changing the consensus rules or block format.

Consensus-integrated availability. The current BORGLOG deployment appends a block after consensus has fixed its position. A future protocol could instead make data availability a condition for voting. Strong validators download and execute the complete candidate block and update their Borg fragments directly. Weak validators cannot afford this work: they obtain only their assigned Borg fragments from strong validators and receive the blockchain state diff together with a SNARK proof that the diff results from correctly executing the block. Both vote only after obtaining their assigned fragments and verifying the state transition.

In a proof-of-stake system, the validator set can therefore serve as a public registry of Borg maintainers. If the responsibilities represented by a voting quorum satisfy Borg's threshold condition, its consensus certificate also shows that enough distinct fragments are held to reconstruct the block. Validators need not retain the complete block, and a validator that loses its fragment can recover it through repair. The SNARK proves correct execution; Borg keeps the block data available. Specifying which Borg responsibilities a voting quorum must cover, how validators transfer fragments, and how repair interacts with voting is left to future work.

Persistent discounted data. BORGLOG could provide long-term storage for blobs a la EIP-4844 or, more generally, for discounted data whose native availability guarantee expires after a bounded period [2], [3]. Once consensus fixes the data, maintainers can append it to BORGLOG and continue storing their coded responsibilities after ordinary nodes are permitted to prune it. Past data then remains authenticated and recoverable from a threshold-valid registry

rather than disappearing when the native retention window ends.

Here too, the proof-of-stake validator set is a particularly natural public source of maintainer advertisements: validator keys and stake identify accountable participants from which clients can assemble their local registries. Advertisements could be bound to validator keys, and the retrievability audits of Section 6.4 could support penalties for stakers that fail to demonstrate continued possession of their advertised responsibilities. This requires choosing κ relative to the expected concentration of storage under one operator. A coalition retaining fewer than κ distinct primary fragments cannot answer audits for additional responsibilities, but an operator retaining κ fragments can reconstruct the challenge polynomial and simulate arbitrarily many advertised responsibilities. Designing stake-weighted assignments, audit parameters, and penalty rules that address such common control is left to future work.

Distributed application state. Borg is not limited to historical block data. Its logical vector could represent a live authenticated application state, such as Bitcoin's UTXO set or an account-based execution state such as an EVM world state, with the corresponding maintained state distributed across a group of maintainers. We envision independently formed groups whose members advertise compatible responsibilities for a particular state, allowing clients to form a threshold-valid registry for that state. No individual member would need to store the complete state, yet clients could retrieve current state entries with authenticated openings, while maintainers update, repair, and reconfigure their fragments. Such a deployment would require adapting the block format to carry compact authenticated information from which maintainers can update their Borg state and verify the resulting anchor. A succinct state-transition proof could establish that this update follows from correct execution. Designing this block format, the representation of different state models, and the formation of state-specific maintainer groups is left to future work.

Updatable AVID and independent instances. Borg can also make AVID-style dispersal updatable. Each AVID instance can be represented as a separate range of the Borg vector. After initially dispersing a payload, its dealer can update selected values by broadcasting only an authenticated diff, allowing maintainers to update their existing fragments without redispersing the unchanged portion of the payload. Multiple independent instances, potentially controlled by different dealers, can occupy disjoint ranges and evolve separately while sharing one repairable and reconfigurable Borg storage layer. The current Borg anchor then authenticates the latest contents of every range. This idea extends beyond AVID to any collection of independently controlled objects maintained inside one evolving authenticated distributed store. The storage mapping is direct; specifying update authorization and composing it with AVID's agreement and termination guarantees is left to future work.

Decentralized Byzantine key–value stores. For a bounded, publicly indexed key space of size K, Borg directly gives an authenticated key–value store by assigning each key to one vector position (and using the default value for absent keys). This store is fully decentralized: any maintainer may advertise a responsibility and onboard by downloading only the fragment it will store, with optimal download, while a maintainer may leave without participating in the installation of its replacement. Clients choose and replace their own local registries rather than relying on a fixed storage committee. A threshold-valid registry recovers each requested value and its authenticated opening despite Byzantine and nonresponding maintainers within the selected fault budgets. Membership, the reconstruction threshold, and the supported coding parameters can all be reconfigured without recommitting the key–value state, and each advertised responsibility uses only $O(K/\kappa)$ leading storage under the compactness conditions of this paper. To our knowledge, Borg is the first key–value store to combine decentralized join and leave, Byzantine fault tolerance, and reconfigurable erasure-coded storage under one authenticated state.

The remaining problem is a large sparse key universe. Borg's current active-prefix representation makes storage depend on the represented key range, rather than only on the number of populated keys. Constructing an authenticated and reconfigurable key–value store with the same κ-out-of-n guarantees, but with maintainer storage proportional to the number of live key–value pairs, is left as an open problem.

13. Conclusion

BORG realizes EDVC for large evolving vectors. Maintainers independently advertise coded responsibilities, and each client forms its own threshold-valid local registry from compatible advertisements, without a fixed global maintainer set or central fragment allocator. Within the selected fault budgets, clients can recover authenticated openings even when some selected maintainers are Byzantine or fail to respond. No maintainer stores the complete vector-commitment prover state. Compactness ensures that each maintainer stores only a coded share of the current active prefix rather than the complete state. This share becomes smaller as the reconstruction threshold increases: doubling the threshold halves the leading storage, while unused vector capacity does not increase it. BORG supports local fragment updates, authenticated retrieval, repair, onboarding, and reconfiguration of registry membership and coding parameters without recommitting the represented vector.

When specialized to append-only public logs, BORG yields BORGLOG, a non-invasive overlay for distributed storage of large data archives. It follows the ordinary published entry stream without requiring the source log to publish BORG-specific proofs or coordinate with maintainers. Storage and load are spread across numerous maintainer nodes. A public commitment binds the evolving history, and maintainers keep a local coded fragment that can be used both to answer client queries and to follow the stream through local updates. Clients can retrieve and authenticate any part of the archive by querying maintainers.

Our experiments show that BORGLOG is practical for blockchain-scale archives. When instantiated with Bitcoin-scale parameters, namely 4 MiB blocks, BORGLOG maintainers can process new blocks in under one second, and clients can verifiably retrieve any block in a few seconds even when some maintainers are malicious. BORGLOG also reduces maintainer storage: at $(\mathsf{Seg}, \kappa) = (4096, 64)$, and for Bitcoin's current sub-1 TiB blockchain, a maintainer stores at most 64 GiB if it keeps auxiliary material for repair and onboarding, and at most 32 GiB if it drops that auxiliary material and only serves clients.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. 2209194 and 2613424.

References

[1] M. Al-Bassam, A. Sonnino, and V. Buterin, "Fraud and data availability proofs: Maximising light client security and scaling blockchains with dishonest majorities," 2018. [Online]. Available: https://arxiv.org/abs/1809.09044

[2] V. Buterin, D. Feist, D. Loerakker, G. Kadianakis, M. Garnett, M. Taiwo, and A. Dietrichs, "EIP-4844: Shard blob transactions," Ethereum Improvement Proposals, no. 4844, Feb. 2022. [Online]. Available: https://eips.ethereum.org/EIPS/eip-4844

[3] D. Ryan, D. Feist, F. D'Amato, H.-W. Wang, and A. Stokes, "EIP-7594: PeerDAS – peer data availability sampling," Ethereum Improvement Proposals, no. 7594, Jan. 2024. [Online]. Available: https://eips.ethereum.org/EIPS/eip-7594

[4] C. Cachin and S. Tessaro, "Asynchronous verifiable information dispersal," in *24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005)*. IEEE, 2005, pp. 191–202. [Online]. Available: https://doi.org/10.1109/RELDIS.2005.9

[5] N. Alhaddad, L. Reyzin, and M. Varia, "Committing AVID with partial retrieval and optimal storage," Cryptology ePrint Archive, Paper 2024/685, 2024. [Online]. Available: https://eprint.iacr.org/2024/685

[6] K. Nazirkhanova, J. Neu, and D. Tse, "Information dispersal with provable retrievability for rollups," in *Proceedings of the 4th ACM Conference on Advances in Financial Technologies (AFT 2022)*, ser. AFT '22. New York, NY, USA: Association for Computing Machinery, 2023, pp. 180–197. [Online]. Available: https://doi.org/10.1145/3558535.3559778

[7] D. Boneh, J. Neu, V. Nikolaenko, and A. Partap, "Data availability sampling with repair," Cryptology ePrint Archive, Paper 2025/1414, 2025. [Online]. Available: https://eprint.iacr.org/2025/1414

[8] G. Danezis, G. Giuliari, E. Kokoris Kogias, M. Legner, J.-P. Smith, A. Sonnino, and K. Wüst, "Walrus: An efficient decentralized storage network," 2025. [Online]. Available: https://arxiv.org/abs/2505.05370

[9] E. Shi, R. Silver, and C. Mu, "Decentralized data archival: New definitions and constructions," in *17th Innovations in Theoretical Computer Science Conference (ITCS 2026)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 362. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2026, pp. 116:1–116:22. [Online]. Available: https://doi.org/10.4230/LIPIcs.ITCS.2026.116

[10] N. Alhaddad, S. Duan, M. Varia, and H. Zhang, "Succinct erasure coding proof systems," Cryptology ePrint Archive, Paper 2021/1500, 2021. [Online]. Available: https://eprint.iacr.org/2021/1500

[11] D. Catalano and D. Fiore, "Vector commitments and their applications," in *Public-Key Cryptography – PKC 2013*, ser. Lecture Notes in Computer Science, vol. 7778. Berlin, Heidelberg: Springer, 2013, pp. 55–72. [Online]. Available: https://doi.org/10.1007/978-3-642-36362-7_5

[12] E. N. Tas and D. Boneh, "Vector commitments with efficient updates," in *5th Conference on Advances in Financial Technologies (AFT 2023)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 282. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2023, pp. 29:1–29:23. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.AFT.2023.29

[13] Z. Luo, Y. Jia, A. V. Ospina Gracia, and A. Kate, "Cauchyproofs: Batch-updatable vector commitment with easy aggregation and application to stateless blockchains," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 1947–1963. [Online]. Available: https://doi.org/10.1109/SP61157.2025.00247

[14] R. Gao, H. Wang, Z. Wan, and Y. Hu, "Distributed vector commitments and their applications," in *Proceedings of the 35th USENIX Security Symposium (USENIX Security '26)*. Baltimore, MD, USA: USENIX Association, 2026, prepublication version. [Online]. Available: https://www.usenix.org/conference/usenixsecurity26/presentation/gao

[15] R. C. Merkle, "A certified digital signature," in *Advances in Cryptology – CRYPTO '89*, ser. Lecture Notes in Computer Science, vol. 435. Springer, 1990, pp. 218–238. [Online]. Available: https://doi.org/10.1007/0-387-34805-0_21

[16] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960. [Online]. Available: https://doi.org/10.1137/0108018

[17] A. Miller, M. Hicks, J. Katz, and E. Shi, "Authenticated data structures, generically," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: ACM, 2014, pp. 411–424. [Online]. Available: https://doi.org/10.1145/2535838.2535851

[18] R. Dahlberg, T. Pulls, and R. Peeters, "Efficient sparse merkle trees: Caching strategies and secure (non-)membership proofs," Cryptology ePrint Archive, Paper 2016/683, 2016. [Online]. Available: https://eprint.iacr.org/2016/683

[19] V. Buterin, "The stateless client concept," Ethereum Research forum post, 2017. [Online]. Available: https://ethresear.ch/t/the-stateless-client-concept/172

[20] D. Boneh, B. Bünz, and B. Fisch, "Batching techniques for accumulators with applications to IOPs and stateless blockchains," in *Advances in Cryptology – CRYPTO 2019, Part I*, ser. Lecture Notes in Computer Science, vol. 11692. Cham: Springer, 2019, pp. 561–586. [Online]. Available: https://doi.org/10.1007/978-3-030-26948-7_20

[21] M. Christ and J. Bonneau, "Limits on revocable proof systems, with implications for stateless blockchains," in *Financial Cryptography and Data Security*, ser. Lecture Notes in Computer Science, vol. 13951. Cham: Springer, 2024, pp. 54–71, fC 2023. [Online]. Available: https://doi.org/10.1007/978-3-031-47751-5_4

[22] T. Dryja, "Utreexo: A dynamic hash-based accumulator optimized for the bitcoin UTXO set," Cryptology ePrint Archive, Paper 2019/611, 2019. [Online]. Available: https://eprint.iacr.org/2019/611

[23] D. Feist, G. Herold, M. Simkin, and B. Wagner, "Robust distributed arrays: Provably secure networking for data availability sampling," 2025. [Online]. Available: https://arxiv.org/abs/2504.13757

[24] S. Kadekodi, F. Maturana, S. J. Subramanya, J. Yang, K. V. Rashmi, and G. R. Ganger, "PACEMAKER: Avoiding HeART attacks in storage clusters with disk-adaptive redundancy," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 369–385. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/kadekodi

[25] K. Cheng, H. Puyang, X. Li, P. P. C. Lee, Y. Hu, J. Li, and T.-Y. Wu, "Toward load-balanced redundancy transitioning for erasure-coded storage," *IEEE Transactions on Parallel and Distributed Systems*, vol. 36, no. 5, pp. 889–902, May 2025. [Online]. Available: https://doi.org/10.1109/TPDS.2025.3547872

[26] S. Athlur, T. Kim, S. Kadekodi, F. Maturana, X. Ramos, A. Merchant, K. V. Rashmi, and G. R. Ganger, "Okapi: Decoupling data striping and redundancy grouping in cluster file systems," in *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. Boston, MA: USENIX Association, Jul. 2025, pp. 897–914. [Online]. Available: https://www.usenix.org/conference/osdi25/presentation/athlur

[27] 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, Feb. 2024. [Online]. Available: https://doi.org/10.1109/TNET.2023.3303865

[28] 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)*. IEEE, 2021, pp. 483–493. [Online]. Available: https://doi.org/10.1109/ICDCS51616.2021.00053

[29] 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, Jul. 2022. [Online]. Available: https://doi.org/10.1109/TIT.2022.3155972

[30] ——, "Bandwidth cost of code conversions in distributed storage: Fundamental limits and optimal constructions," *IEEE Transactions on Information Theory*, vol. 69, no. 8, pp. 4993–5008, Aug. 2023. [Online]. Available: https://doi.org/10.1109/TIT.2023.3265512