Fully Distributed Multi-Point Functions for PCGs and Beyond

Amit Agarwal Srinivasan Raghuraman Peter Rindal

September 23, 2026

Abstract

We give fully distributed multi-point function (DMPF) constructions that share a sparse map over a domain of size N. Setup takes secret-shared indices; each later expansion takes new payloads for those indices and returns shares of the map. Our main construction, Waterfall Cuckoo, samples public hashes before assigning the inputs to bins. An occupancy scan followed by bounded repair finds the assignment; a hidden permutation conceals it during routing. Sparse DPFs then provide $4N$ or $3N$ leaf work per expansion, replacing the tN work of t independent DPFs. We prove correctness and semi-honest simulation in the ideal-subprotocol model, with an explicit routing-error bound.

For the random sparse supports in Ring-LPN PCGs, we also give Reverse Cuckoo, a $2N$-expansion specialization. It programs hashes around a hidden placement and leaks information about the support: in the ideal hash model, outputs are weighted by their number of valid placements. We characterize the idealized leakage, state the additional decisional Ring-LPN assumption, and evaluate leakage-aware decoding attacks. A separate analysis bounds the concrete hash evaluator's batch correctness error below 2^{-40} at the filtered degree-2^{20} parameter point.

Our measurements use an earlier implementation of this specialized Reverse-Cuckoo pipeline. They predate the mask, support-filter, permutation, and evaluator-seed updates, and the AES rekeying described here. At degree 2^{20} it produces 1.81 million Goldilocks-field OLEs per second, with 12.99 MB communicated per expansion. Against the reported FHE baseline at the same field and degree, these figures give 6.57× higher throughput and 13.60× lower expansion communication. Setup costs and comparisons with the sum-of-DPFs baseline are reported separately. Waterfall's costs are modeled; gains for other fields, rings, and PCG constructions remain application-dependent.

1 Introduction

Distributed point functions (DPFs) let two parties hold short keys whose evaluations share a point function [GI14, BGI16, BGI15]. Applications include pseudorandom correlation generators (PCGs), private information retrieval, and private set operations. Many of these applications need several hidden positions at once. Running t independent DPFs solves that problem, but expanding a length-N output costs $O(tN)$ local work.

A distributed multi-point function (DMPF) can reduce this expansion cost. The challenge in our setting is key generation: neither party knows the complete index list. Cuckoo layouts assign each input to one of several candidate bins and use a sparse DPF for each bin [BGI19, SGRR19, BGH+25]. The layout reduces total leaf work, but constructing the secret assignment requires

oblivious table access, collision handling, and repair. We focus on the total nonlinear work of this distributed setup.

Generic construction. Our main construction is Waterfall Cuckoo. It samples public hashes first and combines an inexpensive occupancy scan with a fixed number of complete augmentations. After deduplication and dummy padding, the candidates have the same uniform distribution for every input support. A hidden-permutation scatter then initializes the sparse DPFs. We prove correctness and semi-honest simulation in the ideal-subprotocol model, with a once-per-setup routing error and separate subprotocol losses.

Setup fixes the addresses, routing, and DPF leaf seeds. Expansion supplies only new payloads. The default $t = 16$ profile uses 64 physical bins and $4N$ leaf work. A second profile uses 320 bins and $3N$ leaf work. The profiles share one protocol and proof, exposing a setup–expansion tradeoff rather than separate constructions.

Ring-LPN specialization. Reverse Cuckoo instead chooses the placement first and programs compatible public hashes. This reduces leaf expansion to $2N$ but leaks information about the support. We retain it only for random Ring-LPN supports under an explicit leakage-robust decisional assumption. The attack study examines support search, decoding, and algebraic approaches; it is evidence for that assumption, not a generic security proof. The concrete degree-two hash analysis gives more than 40 bits of batch correctness for the filtered $N_{\mathsf{R}} = 2^{20}$ caller distribution.

Measured performance. The end-to-end measurements use Reverse Cuckoo over two prime fields. They predate the mask, support-filter, permutation, and evaluator-seed updates, and the AES rekeying described here; they do not validate the complete current recipe. At $n = 2^{20}$, the Goldilocks field $q = 2^{64} - 2^{32} + 1$ gives 1.81 million OLEs/s and 12.99 MB per expansion. The Fp31 field gives 2.71 million OLEs/s and 6.51 MB. Against the reported FHE baseline at the same degree and fields, throughput improves by 6.57× and 8.75×, respectively; expansion communication falls by 13.60× and 27.12×. These comparisons exclude one-time setup.

Section 13 gives setup costs, OLE-to-triple normalization, and the smaller-domain sum-of-DPFs comparison, where faster expansion need not mean lower communication. Waterfall's costs are modeled, not measured here.

Organization. We first define the reusable DPF and DMPF interfaces and develop Waterfall as the generic construction. Bucketing and big-state alternatives describe other setup–state tradeoffs. The Ring-LPN application then motivates Reverse Cuckoo, its leakage assumption, and the measured specialized pipeline.

2 Preliminaries

2.1 Notation

For integers $m \le n$, let $[m, n] := \{m, \dots, n\}$ and $[m, n) := \{m, \dots, n-1\}$. In particular, $[0, n) = \{0, 1, \dots, n-1\}$, whereas $[n] := [1, n]$ is one-based. We write index ranges explicitly when addressing bins, blocks, or tree nodes. The computational security parameter is κ; λ controls statistical error.

Our concrete examples use $\kappa = 128$ and $\lambda = 40$. The parties are P_0, P_1. We use $:=$ for deterministic assignment and $\leftarrow$ for sampling. An indicator $[E]$ is one when predicate E holds and zero otherwise.

We denote a sharing of x by $[\![x]\!]$ and the share held by P_i by $[\![x]\!]_i$. Binary sharing means $x = [\![x]\!]_0 \oplus [\![x]\!]_1$. For a finite additive group $\mathbb{G}$, additive sharing means $x = [\![x]\!]_0 + [\![x]\!]_1 \in \mathbb{G}$. The notation $[\![x]\!] \in \mathbb{G}$ specifies the reconstructed value's domain. A shared public constant uses a fixed local sharing known to both parties. Ideal circuit and permutation calls return fresh random shares unless their interfaces state otherwise. The DPF and DMPF functionalities instead let the corrupt party choose its output share and return the complementary share to the honest party.

2.2 Functions

We make use of the following helper functions (inputs may be secret-shared unless noted).

- $\mathsf{decomposition}_w : [0, w^D) \to [0, w)^D$ returns the D base-w digits of an integer, for public integers $w \geq 2$ and $D \geq 1$. Thus $a = \sum_{i=1}^{D} w^{i-1} b_i$. Protocol boxes specify the digit order when assigning these outputs.
- $\mathsf{unitVector}_S : (\alpha \in S,\ \beta \in \mathbb{G}) \to \mathbb{G}^S$, where $v = \mathsf{unitVector}_S(\alpha, \beta)$ satisfies $v_\alpha = \beta$ and $v_j = 0$ for $j \in S \setminus \{\alpha\}$.
- $\mathsf{append} : X^n \times X^m \to X^{n+m}$ concatenates two sequences. We also write $x \| y$ for the same operation.
- $\mathsf{sort} : X^n \times \{\text{total order } \prec\} \to (X^n, \pi)$. Given a sequence and a comparator, outputs the *stably* sorted sequence (ascending under $\prec$) and the permutation $\pi \in S_n$ such that sorted $= x_{\pi(1)}, \ldots, x_{\pi(n)}$. For tuples, the default is lexicographic order on the first component.
- $\mathsf{convert}_{\mathbb{G}} : \{0,1\}^\ell \to \mathbb{G}$ is a fixed, deterministic map used to turn a leaf's ℓ-bit string into a group element. Two standard instantiations we rely on: (i) if $\mathbb{G} = (\mathbb{Z}_p, +)$ or $\mathbb{F}_p$, interpret the bit string as an integer and reduce modulo p; (ii) for an arbitrary prime-order group, use a fixed hash-to-group encoding $H : \{0,1\}^\ell \to \mathbb{G}$ (e.g., a standard hash-to-field followed by a canonical map). The choice is public and fixed by parameters; security only requires that outputs are computationally indistinguishable from random in $\mathbb{G}$ given random inputs.
- $\mathsf{shuffle} : X^n \to X^n$ samples a uniform $\pi \leftarrow S_n$ and returns $(x_{\pi(1)}, \ldots, x_{\pi(n)})$. On shared inputs, this denotes an ideal functionality: reconstruct the records internally, sample π, and return fresh shares of the permuted records. Neither party receives π. A named reusable instance Shuf samples and retains π on Setup; each Expand applies that same permutation to a new shared vector and returns fresh shares. Both methods return the permuted vector; repeated setup or expansion before setup returns $\perp$. Aligned record fields use the same permutation.
- $\mathsf{reshare}([\![x]\!] \in \mathbb{G}) \to [\![x]\!]$ denotes the ideal identity circuit with fresh additive output shares. It reconstructs x internally, samples $r \leftarrow \mathbb{G}$, and returns $r, x - r$. No reconstructed value is opened.
- permute denotes an ideal privately controlled permutation: one designated party supplies π, the parties supply shared records, and the functionality returns fresh shares of their permutation. Only the designated party knows π. A named instance retains it for inverse calls. Composing one such instance per party gives the two serial hidden passes used by Waterfall.

- $\bowtie : X_1^n \times \cdots \times X_k^n \to (X_1 \times \cdots \times X_k)^n$, denote the "zip" operator $\bowtie(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(k)}) = ((x_1^{(1)}, \ldots, x_1^{(k)}), \ldots, (x_n^{(1)}, \ldots, x_n^{(k)}))$. The inverse $\diamond$ unzips a list of k-tuples back into k aligned arrays of length n. We will also use them as infix binary operators.
- $\mathsf{powerSet} : 2^U \to 2^{2^U}$ is the usual power-set operator, returning the set of all subsets of U.
- $\mathsf{G} : \{0,1\}^\kappa \to (\{0,1\}^\kappa)^w$ is the tree PRG for branching factor w. Each child seed includes its tag as its least significant bit. The protocol extracts these bits; G has no separate auxiliary output.

2.3 Related Work

Function secret sharing and DPFs. Function Secret Sharing (FSS) provides short keys that secret-share the evaluation of a function across two (or more) parties. The special case for point functions—Distributed Point Functions (DPFs)—originates with Gilboa-Ishai and subsequent improvements [GI14, BGI16]. Later works explored programmability and larger payloads, along with protocols for efficient *distributed* key generation without a trusted dealer [BGIK22, DS17]. These lines supply the core building blocks we rely on (standard and sparse DPFs) and motivate our fully distributed DMPF keygen.

DPFs in PIR, anonymity, and aggregation. DPFs have been used as a backbone for two-server PIR and anonymous write/read systems (e.g., Riposte), where a client encodes a unit vector query/update via DPF keys [CGBM15]. In private telemetry and measurement, Prio and successors (VDAF) use DPF-style ideas to aggregate client data while preserving privacy and enabling robustness/verification [CGB17, DPRS23]. Recent systems also study aggregation of high-dimensional sparse vectors with DPF-based encodings [AFK+25] and incremental/offline-online PIR updates compatible with DPF pipelines [MZRA22]. Our setting differs: we compress and *multiply* sparse structures arising inside PCGs—hence the need for DMPFs rather than per-point DPF summation.

DMPFs and big-state constructions. DMPFs can be viewed as an instance of earlier FSS-for-batch-codes constructions [BGI19]. More recently, DMPFs were explicitly identified and optimized, with cuckoo-style schemes and big-state variants providing sublinear domains for sub-instances [BGH+25]. Our Waterfall construction retains the cuckoo-style sparse layout but uses an input-independent candidate matrix with a fixed scan and bounded complete repair. We also give bucketing and big-state key-generation paths, and later isolate Reverse Cuckoo as a Ring-LPN-specific optimization rather than a generic DMPF.

PCGs from LPN/Ring-LPN and beyond. PCGs unify many preprocessing correlations for MPC and were introduced as a unifying abstraction by Boyle et al. [BCG+19]. Ring-LPN PCGs first achieved silent OLE/triples from structured noise [BCG+20], with later generalizations to arbitrary fields and to $\mathbb{Z}/p^k\mathbb{Z}$ [LXYY25a, LXYY25b]. Beyond LPN-based assumptions, Bombar et al. introduced programmable PCGs from the quasi-abelian syndrome decoding (QA-SD) problem [BCCD23]. Recent high-throughput Boolean PCGs (e.g., FOLEAGE) highlight how practical pipelines are gated by multi-point encodings in their sparsity steps [BBC+24]. Our DMPFs reduce the number of domain traversals used to encode sparse vectors. Transferring this improvement to a particular PCG requires checking its noise distribution and security assumptions.

Implementation of PCGs. Silentium [Rie25] implements a PCG for Beaver triples. Our evaluation instead measures the Reverse-Cuckoo Ring-LPN pipeline over two prime fields, with the baselines and setup exclusions stated in Section 13. We do not claim the same gains for every PCG.

Cuckoo hashing and stash bounds. We rely on classical cuckoo hashing analysis and its stash refinements to bound the correctness error of fixed Waterfall configurations [PR04, KMW10, ADW12]. For set-based cryptographic protocols such as PSI, cuckoo hashing is now standard, and circuit/OPPRF-based PSI variants analyze related load and failure profiles [PSWW18, DRRT18, YY25].

3 Technical Overview

Our goal is to share a sparse map without revealing its indices. The parties first supply shares of at most t addresses. Later, they supply new payloads for those same addresses. Each expansion returns shares of a length-N map; duplicate addresses contribute their payload sum. A dummy address contributes nothing. The ideal functionality lets the corrupt party choose its output share; the honest party receives the complement to the specified map. The main cost is local expansion: a sum of t independent DPFs traverses the full domain t times.

Sparse DPFs. A single-point DPF uses a tree of pseudorandom seeds and public correction words to keep one secret path active. For a public subset $S \subseteq [0, N)$, our sparse DPF contracts paths with only one live child. It materializes only the leaves in S and their branching nodes. Setup expands this compressed tree and saves the leaf seeds. Each later expansion converts those seeds using a round-indexed generator and applies a payload correction. It processes $|S|$ leaves rather than traversing the full ambient domain.

The cuckoo layout. A cuckoo-based DMPF gives each address one candidate bin in each of w partitions [BGI19, SGRR19]. Each bin owns the public sparse set of addresses that can map to it. If the secret input rows receive distinct admissible bins, one sparse DPF per bin suffices. Summing the bin outputs reconstructs the multi-point map. Each address appears once per partition, so expansion visits wN leaves. The difficulty is finding the secret placement efficiently in MPC.

3.1 Waterfall Cuckoo DMPF

Waterfall first merges duplicate addresses and pads the list with dummies. It then samples public t-wise independent hashes before routing. Distinct real addresses have independent uniform candidates; dummy rows receive fresh uniform candidates. Consequently, the entire candidate matrix has the same distribution for every input support.

An inexpensive scan fills the partitions in order. A row that collides remains unplaced and tries the next partition. A small public number of complete augmentations then repairs the remaining matching, followed by stash compaction.

Setup hides the resulting placement with two privately controlled permutation passes and opens only the permuted placement. On successful routing, that opening is a uniform labeled injection. It supports an oblivious scatter into the sparse-DPF columns. The simulation proof keeps the hashes

unconditioned and charges routing failure once per setup state. It uses ideal MPC, permutation, and reusable-DPF functionalities; concrete realizations contribute their separate security losses.

At $t = 16$, the default profile uses four 16-bin partitions and three repairs, with no stash. It has $4N$ local leaf expansion. The lower-expansion profile uses partition sizes $(128, 128, 64)$ and two repairs, also without a stash. It saves one domain traversal per expansion but has larger setup and state.

3.2 Bucketing DMPF

For independent uniform input addresses, public intervals can divide the inputs into smaller padded groups. One DMPF handles each group. The tradeoff depends on the grouping circuit, the underlying DMPF, and an explicit overflow bound. Expansion must still materialize N outputs.

3.3 Big-State DMPF

The big-state approach [BGH+25] trades stored state for cheaper subsequent evaluation. We discuss a black-box key-generation path and a generic-MPC path for this setup–state tradeoff.

3.4 Integration into Ring-LPN and Stationary-LPN PCGs

A Ring-LPN PCG must share products of sparse polynomials. With P polynomials per party and t selected positions per polynomial, each cross-party product has t^2 indexed terms, possibly zero. For regular noise, one position lies in each block of length $L = N_{\mathsf{R}}/t$. The products reorganize into P^2t lists, each with t terms and a local domain of length $2L$. Replacing a sum of t DPFs on each list by a constant-expansion DMPF removes a factor of t from the dominant leaf work.

Stationary supports allow the address-dependent setup to be reused. Every expansion samples fresh coefficients and a fresh public mask vector. It converts cached leaves, communicates payload corrections, and performs ring arithmetic. Thus repeated expansion is not communication-free. Security across a polynomial number of expansions requires a joint stationary Ring-LPN assumption, separate from the routing argument.

3.5 Reverse Cuckoo

For these random Ring-LPN supports, we also consider a placement-first construction. Reverse Cuckoo chooses a hidden placement, then programs public hashes compatible with it. This gives the $2N$ specialized pipeline evaluated in the paper. Its descriptors leak information about the support: in the ideal hash model, a hash tuple is weighted by its number of valid placements. Hidden permutations do not remove this multiplicity bias. We therefore use Reverse Cuckoo under an explicit leakage-robust decisional Ring-LPN assumption. The degree-two evaluator's correctness analysis is separate from that assumption. Section 12 derives the leakage distribution and examines attacks that use it.

4 Single-Point DPF Constructions

Our applications reuse one secret index with many payloads. We therefore separate the index-dependent setup from each payload-dependent expansion. This separation also makes the persistent

state and the permitted reuse explicit.

Fix $\kappa, N \in \mathbb{N}$, an ordered public set $S \subseteq [0, N)$, and a finite additive group $\mathbb{G}$. We write $\mathbb{G}^S$ for the set of maps from S to $\mathbb{G}$. For $\alpha \in S$ and $\beta \in \mathbb{G}$, define the point map

$$\mathsf{pt}_S(\alpha, \beta)(x) := \begin{cases} \beta, & x = \alpha, \\ 0, & x \neq \alpha, \end{cases} \qquad x \in S.$$

The reusable DPF functionality maintains one index and a public expansion counter. Setup receives shares of the index. Expansion number q receives shares of one payload and produces an additive sharing of the corresponding point map. The corrupt party P_b, $b \in \{0, 1\}$, chooses its output share; the functionality supplies the honest party's complementary share. Payloads may depend on the transcript of earlier expansions.

Public parameters are $\kappa, N \in \mathbb{N}$, an ordered nonempty set $S \subseteq [0, N)$, and a finite additive group $\mathbb{G}$. The functionality maintains state (α, q), initially undefined.

$\mathcal{F}_{\mathsf{rDPF}}.\mathsf{Setup}(\llbracket \alpha \rrbracket \in S) \longrightarrow \mathsf{ready}$:

1. Reconstruct α from the two submitted shares.
2. If setup was already called, output $\perp$.
3. Store $(\alpha, q) := (\alpha, 0)$ and output ready.

$\mathcal{F}_{\mathsf{rDPF}}.\mathsf{Expand}(\llbracket \beta \rrbracket \in \mathbb{G}) \longrightarrow \llbracket y \rrbracket \in \mathbb{G}^S$:

1. If setup has not completed, output $\perp$.
2. Receive $y_b \in \mathbb{G}^S$ from the corrupt party P_b.
3. Reconstruct β from the two submitted shares.
4. Output $y_{1-b} := \mathsf{pt}_S(\alpha, \beta) - y_b$ to P_{1-b}.
5. Set $q := q + 1$ and retain α.

Figure 1: Reusable single-point DPF functionality over S.

A reusable DPF realization has the same interface:

$$\mathsf{DPF.Setup}_S(\llbracket \alpha \rrbracket \in S) \longrightarrow \mathsf{ready},$$
$$\mathsf{DPF.Expand}_S(\llbracket \beta \rrbracket \in \mathbb{G}) \longrightarrow \llbracket y \rrbracket \in \mathbb{G}^S.$$

The protocol instance maintains internal state $(\llbracket s \rrbracket, q)$, initially undefined. Party p stores only $(\llbracket s \rrbracket_p, q)$. Setup initializes $q := 0$, and each expansion increments it. Calls have valid inputs in their declared domains; in particular, setup requires $\alpha \in S$. No membership-test circuit is included. Repeated setup and expansion before setup return $\perp$.

Fix a polynomial $Q(\kappa)$. A protocol realizes $\mathcal{F}_{\mathsf{rDPF}}$ for Q expansions if the following condition holds. For each $b \in \{0, 1\}$ and every static semi-honest PPT adversary corrupting P_b, there exists a stateful PPT simulator. Given only public parameters and P_b's input shares, the simulator generates its view and submits its output share y_b to the functionality on each expansion. The joint simulated view and honest outputs must be computationally indistinguishable from their real

counterparts for at most $Q(\kappa)$ adaptively chosen payloads. The view includes persistent local state, messages, and output shares. The simulator proceeds online without rewinding or receiving future inputs.

For a nontrivial tree, setup performs the index-dependent oblivious transfers, expands the tree, and saves the leaf-seed shares. Expansion number q converts those seeds with L_q and applies $\mathsf{updateLeaves}$ to the new payload. Let $\epsilon_{\mathsf{DPF}}(Q)$ denote the simulation advantage for all Q adaptive expansions jointly. Theorem C.3 bounds this advantage under the tag-conditioned tree-PRG and joint leaf-PRF requirements stated in Appendix C. Correctness alone does not give this guarantee, which Waterfall requires when replacing ideal DPF calls by a concrete realization. The AES evaluators and rekeying schedules in Appendix C.4 remain separate instantiation claims.

For a singleton $S = \{x\}$, the point map is just the payload at x. Setup stores the initialized flag and counter, without a tree. Expansion invokes the ideal identity circuit $\mathsf{reshare}(\llbracket\beta\rrbracket)$ and returns its result at x. Empty sets require no DPF instance.

4.1 Standard DPF Protocol

We use the tree-based DPF approach of Gilboa–Ishai [GI14] and its refinements [BGI16, DS17]. A public branching factor $w \geq 2$ gives a complete tree with w^D leaves, where $D = \lceil \log_w n \rceil$. Only the first n leaves are returned. The input $\alpha \in [0, n)$ selects one root-to-leaf path.

Intuition. After correction, the parties hold equal seed shares at every off-path node. Those nodes expand identically and therefore cancel. The active node has tag shares whose XOR is one. Before correction, both parties expand their local seeds, producing w children per node. They sum child seeds by child number and compute one public correction per child number. The parent tags gate these corrections: inactive parents preserve equal child shares, while the active parent's children cancel except on the selected path.

For binary branching, the two corrections share their upper $\kappa - 1$ bits. Their low bits set the selected child's reconstructed tag to one and the other tag to zero. For $w > 2$, the correction procedure instead leaves one fresh shared seed on the selected child. Child numbers are one-based; the secret base-w digit is zero-based.

Leaf update. Each party converts its saved seed prefix using L_q and uses opposite signs in the output group. Off-path values cancel. The parties open a single correction γ and apply it using their saved leaf tags. For groups of characteristic other than two, $\mathsf{updateLeaves}$ first corrects the sign induced by the XOR-shared tags. Its shared selections and products are ideal circuit calls.

$\mathsf{correctionWord}(\llbracket z \rrbracket \in \{0,1\}^{w\times\kappa}, \llbracket \alpha \rrbracket \in [0,w)) \longrightarrow \sigma \in \{0,1\}^{w\times\kappa}$:
Party $p \in \{0,1\}$ computes:

1. If $w = 2$:
   (a) $\llbracket a \rrbracket := \mathsf{MSBs}(\llbracket z_{2-\llbracket \alpha \rrbracket} \rrbracket)$ by oblivious selection.
   (b) $\llbracket \sigma_1 \rrbracket := \llbracket a \rrbracket \mid\mid (\mathsf{LSB}(\llbracket z_1 \rrbracket) \oplus \llbracket \alpha \rrbracket \oplus 1)$
   (c) $\llbracket \sigma_2 \rrbracket := \llbracket a \rrbracket \mid\mid (\mathsf{LSB}(\llbracket z_2 \rrbracket) \oplus \llbracket \alpha \rrbracket)$
2. Else:
   (a) $\llbracket r \rrbracket \leftarrow \{0,1\}^{\kappa-1}$
   (b) For $j \in \{1,\ldots,w\}$, set $\llbracket \sigma_j \rrbracket := \llbracket z_j \rrbracket \oplus [j = 1 + \llbracket \alpha \rrbracket] \cdot (\llbracket r \rrbracket \mid\mid 1)$.
3. return $\mathsf{reveal}(\llbracket \sigma \rrbracket)$

$\mathsf{updateLeaves}_S(\llbracket y \rrbracket \in \mathbb{G}^S, \llbracket t \rrbracket \in \{0,1\}^S, \llbracket \beta \rrbracket \in \mathbb{G}) \longrightarrow \llbracket y \rrbracket \in \mathbb{G}^S$:
Party $p \in \{0,1\}$ computes:

1. $\llbracket \gamma \rrbracket := \llbracket \beta \rrbracket - \sum_{i\in S} \llbracket y_i \rrbracket$,
2. if $\mathbb{G}$ is not characteristic 2:
   (a) $c_p := \sum_{i\in S} \llbracket t_i \rrbracket_p \in \{0,\ldots,|S|\}$
   (b) $\llbracket d \rrbracket_p := (\lfloor c_p/2 \rfloor + p c_p) \bmod 2$
   (c) $\llbracket \gamma \rrbracket := (1 - 2\llbracket d \rrbracket) \cdot \llbracket \gamma \rrbracket$
3. $\gamma = \mathsf{reveal}(\llbracket \gamma \rrbracket)$
4. $\llbracket y_i \rrbracket := \llbracket y_i \rrbracket + (1-2p)\gamma \llbracket t_i \rrbracket$ for $i \in S$
5. return $\llbracket y \rrbracket \in \mathbb{G}^S$

Figure 2: correctionWord

Fix a round-indexed family of leaf PRGs

$$(\mathsf{L}_q)_{q\geq 0}, \qquad \mathsf{L}_q : \{0,1\}^{\kappa-1} \to \mathbb{G}.$$

The family may instead be instantiated by a domain-separated PRF $\mathsf{L} : \mathbb{N} \times \{0,1\}^{\kappa-1} \to \mathbb{G}$, where the second argument is the secret key and q is the public input, with $\mathsf{L}_q(s) := \mathsf{L}(q,s)$. Thus the public expansion number q selects the function applied to each saved leaf seed.

Public parameters are $\kappa, n \geq 1$, $w \geq 2$, a group $\mathbb{G}$, and $\mathsf{G} : \{0,1\}^\kappa \to (\{0,1\}^\kappa)^w$. Put $D := \lceil \log_w n \rceil$.
Party $p \in \{0,1\}$ stores state $(\llbracket s \rrbracket_p, q)$, initially undefined.

$\mathsf{DPF.Setup}_{[0,n)}(\llbracket \alpha \rrbracket \in [0,n)) \longrightarrow \mathsf{ready}$:

Party $p \in \{0,1\}$ computes:

1. If setup has already completed, return $\perp$.
2. If $n = 1$, store $(\llbracket s \rrbracket_p, q) := ((), 0)$ and return ready.
3. $\llbracket s \rrbracket \leftarrow \{0,1\}^{w \times \kappa}$
4. $\llbracket z \rrbracket := \llbracket s \rrbracket$
5. $\llbracket u \rrbracket := \bowtie(\llbracket s \rrbracket, \llbracket 1 \rrbracket^w)$
6. $\llbracket u' \rrbracket := ()$
7. $(\llbracket \alpha_1 \rrbracket, \ldots, \llbracket \alpha_D \rrbracket) := \mathsf{decomposition}_w(\llbracket \alpha \rrbracket) \in [0,w)^D$.
8. For $d = D, D-1, \ldots, 1$:
   (a) $\sigma := \mathsf{correctionWord}(\llbracket z \rrbracket, \llbracket \alpha_d \rrbracket)$
   (b) $\llbracket z \rrbracket := (0)^w$
   (c) if $d > 1$: for $(j, (\llbracket s \rrbracket, \llbracket t \rrbracket)) \in \bowtie((1, \ldots, w, 1, \ldots, w, \ldots), \llbracket u \rrbracket)$:
      i. $\llbracket s \rrbracket := \llbracket s \rrbracket \oplus \llbracket t \rrbracket \cdot \sigma_j$
      ii. $\llbracket s' \rrbracket_p := \mathsf{G}(\llbracket s \rrbracket_p)$
      iii. $\llbracket z \rrbracket := \llbracket z \rrbracket \oplus \llbracket s' \rrbracket$
      iv. $\llbracket u' \rrbracket := \llbracket u' \rrbracket \mid\mid \bowtie(\llbracket s' \rrbracket, \mathsf{LSB}(\llbracket s \rrbracket) \cdot (1)^w)$
   (d) If $d = 1$, process only the first n entries of $\llbracket u \rrbracket$. For entry $i \in [0,n)$, with child number $j := 1 + (i \bmod w)$ and seed/tag pair $(\llbracket s \rrbracket, \llbracket t' \rrbracket)$:
      i. $\llbracket s \rrbracket := \llbracket s \rrbracket \oplus \llbracket t' \rrbracket \cdot \sigma_j$
      ii. $\llbracket s_i \rrbracket := \llbracket s \rrbracket$
   (e) $\llbracket u \rrbracket := \llbracket u' \rrbracket$
   (f) $\llbracket u' \rrbracket := ()$
9. Store $(\llbracket s \rrbracket_p, q) := (\llbracket s \rrbracket_p, 0)$ and return ready.

$\mathsf{DPF.Expand}_{[0,n)}(\llbracket \beta \rrbracket \in \mathbb{G}) \longrightarrow \llbracket y \rrbracket \in \mathbb{G}^{[0,n)}$:

Party $p \in \{0,1\}$ computes:

1. If setup has not completed, return $\perp$.
2. If $n = 1$, set $\llbracket y(0) \rrbracket := \mathsf{reshare}(\llbracket \beta \rrbracket)$, increment q, and return $\llbracket y \rrbracket$.
3. for $i \in [0,n)$:
   (a) $\llbracket t_i \rrbracket_p := \mathsf{LSB}(\llbracket s_i \rrbracket_p)$
   (b) $\llbracket y_i \rrbracket_p := (1-2p) \cdot \mathsf{L}_q(\mathsf{MSBs}(\llbracket s_i \rrbracket_p))$
4. $\llbracket y \rrbracket := \mathsf{updateLeaves}_{[0,n)}(\llbracket y \rrbracket, \llbracket t \rrbracket, \llbracket \beta \rrbracket)$
5. Set $q := q + 1$ and return $\llbracket y \rrbracket$.

Figure 3: Reusable w-ary DPF realization.

Standard-tree correctness. At the root, the reconstructed parent tag is one. At every subsequent level, equal off-path seeds expand equally, so only the active parent's children contribute to the reconstructed child sums z. For binary branching, the correction cancels the off-path child and sets the two reconstructed tags to $1-\alpha_d, \alpha_d$. For larger branching, the corrected child sums are zero except at child $1+\alpha_d$, whose reconstructed seed is $r\|1$. Induction therefore gives equal off-path leaf seeds and one active leaf with reconstructed tag one. Because $\alpha < n$, truncating the padded tree to its first n leaves preserves that leaf. The sign calculation in the proof of Proposition 4.1 then shows that updateLeaves reconstructs the requested point map for each payload. This argument establishes correctness, not the reusable simulation guarantee defined above.

4.2 Sparse DPF

The standard DPF expands a full tree, even when the public set $S \subseteq [0, 2^D)$ contains few indices. The sparse construction retains only the leaves in S and the nodes where their paths branch. A seed passes unchanged through a chain with one live child. Each retained internal node has two children, so setup expands only $|S| - 1$ branching nodes.

Public parameters are $D \in \mathbb{N}$ and $S = (S_1 < \cdots < S_M) \subseteq [0, 2^D)$. For an index x, let $x_d := \lfloor x/2^{d-1} \rfloor \bmod 2$.
$\mathsf{partition}(\beta \in [M]^2) \longrightarrow (\delta, b) \in \{0, \ldots, D\} \times ([M]^2 \cup \{\perp\})^2$:
Require $1 \le \beta_1 \le \beta_2 \le M$.

1. if $\beta_1 = \beta_2$, return $(0, (\beta, \perp))$.
2. $\delta := 1 + \lfloor \log_2(S_{\beta_1} \oplus S_{\beta_2}) \rfloor$
3. $m := |\{i \in [\beta_1, \beta_2] : (S_i)_\delta = 0\}|$
4. $b := ((\beta_1, \beta_1 + m - 1), (\beta_1 + m, \beta_2))$
5. return (δ, b).

Figure 4: Public partition at the highest differing bit. The two returned intervals index the left and right child sets. A singleton returns depth zero.

We number bits from D at the root to 1 at the leaves. The public queue u_d contains nodes whose next split is at bit d; u_0 contains leaves. A queue entry $(j, \rho, b, [\![s]\!], [\![t]\!])$ records the child number $j \in \{1, 2\}$, its parent's split bit ρ, and its next child intervals b. Its seed still requires correction $\sigma_{\rho,j}$, controlled by the saved parent tag $[\![t]\!]$. For u_0, the sole address is $S_{b_{1,1}}$. Correction depth ρ and expansion depth d are different: an edge can skip several bits.

Public parameters are $\kappa, D \in \mathbb{N}$, $S = (S_1 < \cdots < S_M) \subseteq [0, 2^D)$ with $M \geq 1$, group $\mathbb{G}$, and $\mathsf{G} : \{0,1\}^\kappa \to (\{0,1\}^\kappa)^2$.
Party $p \in \{0,1\}$ stores state $([\![s]\!]_p, q)$, initially undefined.

$\mathsf{DPF.Setup}_S([\![\alpha]\!] \in S) \longrightarrow \mathsf{ready}$:

Party $p \in \{0,1\}$ computes:

1. If setup has already completed, return $\perp$.
2. If $M = 1$, store $([\![s]\!]_p, q) := ((), 0)$ and return ready.
3. For $i \in \{0, \ldots, D\}$, initialize $(u_i, [\![z_i]\!], [\![v_i]\!]) := ((), 0^{2\times\kappa}, 0)$.
4. $(d_*, b) := \mathsf{partition}((1, M))$
5. $[\![v_{d_*}]\!] := 1$
6. for $j \in \{1, 2\}$:
   (a) $(\delta', b') := \mathsf{partition}(b_j)$
   (b) $[\![s]\!] \leftarrow \{0,1\}^\kappa$
   (c) $[\![z_{d_*,j}]\!] := [\![s]\!]$
   (d) $u_{\delta'} := \mathsf{append}(u_{\delta'}, (j, d_*, b', [\![s]\!], [\![1]\!]))$
7. $([\![\alpha_1]\!], \ldots, [\![\alpha_D]\!]) := \mathsf{decomposition}_2([\![\alpha]\!])$.
8. for $d \in \{D, D-1, ..., 1\}$:
   (a) If $d \neq d_*$ and $u_d = ()$, continue.
   (b) for $(j, \rho, b, [\![s]\!], [\![t]\!]) \in u_d$:
      i. $[\![s]\!] := [\![s]\!] \oplus [\![t]\!] \cdot \sigma_{\rho,j}$
      ii. $[\![s']\!]_p := \mathsf{G}([\![s]\!]_p)$
      iii. $[\![z_d]\!] := [\![z_d]\!] \oplus [\![s']\!]$
      iv. $[\![v_d]\!] := [\![v_d]\!] \oplus \mathsf{LSB}([\![s]\!])$
      v. for $k \in \{1, 2\}$
         A. $(\delta, b') := \mathsf{partition}(b_k)$
         B. $u_\delta := \mathsf{append}(u_\delta, (k, d, b', [\![s'_k]\!], \mathsf{LSB}([\![s]\!])))$
   (c) $[\![r]\!] \leftarrow \{0,1\}^{2\times\kappa}$
   (d) $[\![z_d]\!] := [\![z_d]\!] \oplus (1 \oplus [\![v_d]\!]) \cdot [\![r]\!]$
   (e) $\sigma_d := \mathsf{correctionWord}([\![z_d]\!], [\![\alpha_d]\!])$
9. for $(j, \rho, b, [\![s]\!], [\![t]\!]) \in u_0$
   (a) $[\![s]\!] := [\![s]\!] \oplus [\![t]\!] \cdot \sigma_{\rho,j}$
   (b) $[\![s_{S_{b_{1,1}}}]\!] := [\![s]\!]$
10. Store $([\![s]\!]_p, q) := ([\![s]\!]_p, 0)$ and return ready.

$\mathsf{DPF.Expand}_S([\![\beta]\!] \in \mathbb{G}) \longrightarrow [\![y]\!] \in \mathbb{G}^S$:

Party $p \in \{0,1\}$ computes:

1. If setup has not completed, return $\perp$.
2. If $M = 1$, set $[\![y(S_1)]\!] := \mathsf{reshare}([\![\beta]\!])$, increment q, and return $[\![y]\!]$.
3. for $i \in S$:
   (a) $[\![t_i]\!]_p := \mathsf{LSB}([\![s_i]\!]_p)$
   (b) $[\![y_i]\!]_p := (1 - 2p) \cdot \mathsf{L}_q(\mathsf{MSBs}([\![s_i]\!]_p))$
4. $[\![y]\!] := \mathsf{updateLeaves}_S([\![y]\!], [\![t]\!], [\![\beta]\!])$
5. Set $q := q + 1$ and return $[\![y]\!]$.

Figure 5: Reusable sparse DPF realization.

Correction scheduling. Let d_* be the first split bit of S. Setup samples the two root-child seed shares directly and records their sums in z_{d_*}. Both children can be leaves, leaving every positive-depth queue empty. The root correction must nevertheless be computed.

At every visited split bit d, setup first processes u_d. A queued node consumes its parent's correction $\sigma_{\rho,j}$ before expansion. Since $\rho > d$, that correction already exists. The node then produces two children, whose uncorrected seeds contribute to z_d. Their correction is indexed by this split bit d, not by $d-1$. Each child waits in its next-split queue, or in u_0 if it is a leaf. Only after all nodes in u_d have expanded does setup compute σ_d.

The reconstructed bit v_d indicates whether the active path splits at d. It starts at zero, except for $v_{d_*} = 1$, and accumulates the corrected parent tags. If $v_d = 0$, both reconstructed child sums vanish. Setup masks those sums with fresh shared randomness before revealing a correction. This preserves the inactive nodes, whose parent tags reconstruct to zero. It also avoids revealing the absence of an active split through deterministically zero sums.

For example, take $S = \{0, 1, 4\}$ with $D = 3$. The root splits at bit 3; its left child waits in u_1 and its right child waits in u_0. Setup computes σ_3, skips bit 2, and then expands u_1 using σ_3. The new children contribute to z_1 before σ_1 is computed. At the end, leaves $0, 1$ consume σ_1 and leaf 4 consumes σ_3.

Correctness. The next proposition concerns the reconstructed map, not the distribution of either party's output share. The shared selections and products are evaluated by their ideal circuit functionalities.

Proposition 4.1 (Sparse leaf invariant)**.** *Fix $S \subseteq [0, 2^D)$ with $|S| > 1$ and $\alpha \in S$. For every choice of setup randomness, Figure 5 saves seed shares satisfying*

$$[\![s_x]\!]_0 = [\![s_x]\!]_1 \quad (x \neq \alpha), \qquad \mathsf{LSB}([\![s_x]\!]_0) \oplus \mathsf{LSB}([\![s_x]\!]_1) = [x = \alpha].$$

Consequently, every expansion reconstructs $\mathsf{pt}_S(\alpha, \beta)$, for every payload $\beta \in \mathbb{G}$.

Proof. Before any split, exactly one live subtree contains α. Inductively, every corrected inactive seed reconstructs to zero, whereas the active seed has reconstructed tag one. Equal local seeds expand identically, so inactive parents contribute zero to the reconstructed child sums. If the active parent splits at d, the sums $z_{d,1}, z_{d,2}$ therefore equal its two child seeds. The binary correction makes its off-path child zero and sets the tags to $1 - \alpha_d, \alpha_d$. All other children have reconstructed parent tag zero and retain reconstructed seed zero. If the active parent does not split, every processed parent is inactive; the masked correction again preserves their reconstructed zero seeds. Children can defer their correction because neither their seeds nor their saved parent tags change while queued. This proves the invariant at every split and, finally, at every leaf.

During expansion, equal off-path seeds receive equal values from L_q with opposite signs. Thus the provisional reconstructed map vanishes off α. Write $t_{p,x} := \mathsf{LSB}([\![s_x]\!]_p)$ and $c_p := \sum_{x \in S} t_{p,x}$ as integers. The tag invariant gives

$$c_0 - c_1 = 1 - 2t_{1,\alpha}, \qquad (\lfloor c_0/2 \rfloor + \lfloor c_1/2 \rfloor + c_1) \bmod 2 = t_{1,\alpha}.$$

Hence the bit d in $\mathsf{updateLeaves}$ equals $t_{1,\alpha}$. If $\gamma_0 := \beta - \sum_x y_x$ denotes the needed correction, the revealed value is $(1 - 2t_{1,\alpha})\gamma_0$. Its reconstructed update at α is $(1 - 2t_{1,\alpha})^2\gamma_0 = \gamma_0$ and is zero elsewhere. In characteristic two the sign conversion is unnecessary. The argument holds for each payload without changing the saved seeds. □

Cost and proof scope. The compressed binary tree has $|S|-1$ branching nodes. Setup samples the root children directly and calls G once per party at each of the remaining $|S|-2$ branching nodes. It generates one correction pair per distinct split bit, at most D pairs. Expansion applies L_q once per saved leaf and then $\mathsf{updateLeaves}$. The scheduling correction adds neither tree expansions nor correction pairs.

Each binary correction selection uses two OTs with block-length messages. Masking a visited sparse level uses two further OTs with two-block messages; the XOR of corrected parent tags supplies its secret control bit. A dense prefix can avoid masks at its complete levels: expand that prefix as a standard binary tree, then prune the remaining public subtrees. The prefix's corrected tags initialize activity at the subsequent sparse splits. For groups outside characteristic two, each instance also caches the sign-conversion correlations used by $\mathsf{updateLeaves}$. These costs belong to initialization; they do not add leaf conversions to repeated local expansion.

Proposition 4.1 proves correctness for valid inputs with $|S| > 1$; singleton correctness and ideal-call simulation follow from the identity-circuit interface. A simulation proof for $\mathcal{F}_{\mathsf{rDPF}}$ must additionally reproduce the corrupt party's view jointly with the honest output shares across expansions. The leaf invariant alone does not establish that distributional claim. Theorem C.3 proves the corresponding simulation under its stated generator requirements, without prescribing the corrupt output share.

5 Distributed Multi-Point Function

Section 4 defined a reusable DPF whose setup fixes one index and whose expansions supply many payloads. A distributed multi-point function (DMPF) provides the same interface for at most t indices. Running t independent DPFs realizes this interface, but costs $O(tN)$ local work.

Let $\mathcal{X}_N := [0, N) \cup \{N\}$, where N is a dummy address. For $A \in \mathcal{X}_N^t$ and $B \in \mathbb{G}^t$, define the map

$$\mathsf{mp}_{N,t}(A, B)(x) := \sum_{i \in [0,t): A_i = x} B_i, \qquad x \in [0, N).$$

Rows with address N make no contribution. Equal real addresses contribute their payload sum. Figure 6 defines the reusable ideal functionality targeted by the constructions below. Calls have inputs in their declared domains; no input-membership test is included in the protocol costs. As for the single-point functionality, the corrupt party P_b, $b \in \{0, 1\}$, chooses its output share.

Public parameters are $\kappa, N, t \in \mathbb{N}$ and a finite additive group $\mathbb{G}$. The functionality maintains state (A, q), initially undefined.

$\mathcal{F}_{\mathsf{rDMPF}}.\mathsf{Setup}([\![A]\!] \in \mathcal{X}_N^t) \longrightarrow \mathsf{ready}$:

1. Reconstruct A from the two submitted shares.
2. If setup was already called, output $\perp$.
3. Store $(A, q) := (A, 0)$ and output ready.

$\mathcal{F}_{\mathsf{rDMPF}}.\mathsf{Expand}([\![B]\!] \in \mathbb{G}^t) \longrightarrow [\![y]\!] \in \mathbb{G}^{[0,N)}$:

1. If setup has not completed, output $\perp$.
2. Receive $y_b \in \mathbb{G}^{[0,N)}$ from the corrupt party P_b.
3. Reconstruct B from the two submitted shares.
4. Output $y_{1-b} := \mathsf{mp}_{N,t}(A, B) - y_b$ to P_{1-b}.
5. Set $q := q + 1$ and retain A.

Figure 6: Reusable distributed multi-point functionality.

We use the same static semi-honest simulation convention as for $\mathcal{F}_{\mathsf{rDPF}}$. For either corrupt party P_b, the simulator receives only public parameters and that party's input shares. It generates the corrupt view and submits y_b to the functionality on each expansion. For every polynomial $Q(\kappa)$ and every static semi-honest PPT adversary, there must exist a stateful PPT simulator whose view, jointly with the honest outputs, is computationally indistinguishable from the real execution for at most $Q(\kappa)$ adaptively chosen payload vectors. The simulator proceeds online without rewinding or receiving future inputs. The view includes persistent local state, public values, messages, and output shares.

An efficient realization must address three costs:

1. *Local expansion.* The dominant leaf work should be linear in N, with a small factor independent of t.
2. *Succinct key generation.* Communication should be sublinear in N, ideally $O(t \log N)$.
3. *Deduplication.* Payloads at duplicate indices must be aggregated without revealing which indices coincide.

The sum-of-DPFs baseline is common in PCGs [BCG+20, BBC+24, LXYY25a, LXYY25b]; its tN leaf work can dominate correlation generation [BBC+24, PR]. Section 7 reviews the sparse-column layout; Section 8 gives Waterfall's generic realization with $4N$ and $3N$ profiles. Bucketing and big-state methods offer other setup–state tradeoffs in Section 9 and Section 10.

The later Reverse-Cuckoo specialization in Section 12 does not realize the generic functionality above. It uses random Ring-LPN supports and an additional leakage-robust decisional assumption to obtain $2N$ expansion.

6 Core MPC Subroutines

6.1 System Solver

The bin-solver protocol processes a binary linear system $M \in \mathbb{F}_2^{m \times c}$ with right-hand side $y \in \mathbb{G}^m$, where computation occurs in a characteristic-2 group $\mathbb{G}$. The inputs are secret-shared, and the protocol always outputs $x \in \mathbb{G}^c$. A reduced row with no pivot is skipped, regardless of its right-hand side; the protocol performs no consistency test and does not abort. Consequently $Mx = y$ is guaranteed when M has full row rank. If M is rank deficient, every retained pivot equation is satisfied, while a skipped dependent equation may be unsatisfied.

1. **Row Elimination:** For each row $i \in [m]$:
   (a) Find a pivot column j such that $M_{i,j} = 1$, and set $s_i := \mathsf{unitVector}(j, 1)$ if j exists and otherwise set $s_i = 0$.
   (b) Compute the active column $v := M \cdot s_i$, the vector corresponding to column j.
   (c) For each row $j \in [m] \setminus \{i\}$, eliminate the pivot bit. This is done by performing row operations. For each row that is active and not i, we subtract row i from it. This is done by:
      i. Update matrix row: $M_j := M_j \oplus v_j \cdot M_i$
      ii. Update RHS: $y_j := y_j \oplus v_j \cdot y_i$
2. **Combine Support Vectors:** Compute the vector $\sigma \in \mathbb{F}_2^c$ where $\sigma_j = 1$ means that column j of M and position j in x is used to solve the system. That is:
$$\sigma := \bigoplus_{s \in [m]} [s_i]$$
   Similarly, compute the set of "free variables" in x by taking the complement.
$$\overline{\sigma} := (1)^c \oplus \sigma$$
3. **Sample Free Variables:** To produce a random solution we first assign random values to the free bariables, i.e. $x \leftarrow \mathbb{G}^c \odot \overline{\sigma}$
4. **Back-substitute:** We then update the y values to cancel the random values assign to the free variables, i.e. $y := y \oplus M * x$.
5. **Recover Solution:** We then route the final values for x which are currently stored in y. The routing is simply the s permutation matrix, i.e. $x := x \oplus s \cdot y$.
6. **Output:** Return the final solution x.

Public parameters, a characturistic 2 group $\mathbb{G}$.
bin-solver($[\![M]\!] \in \mathbb{F}_2^{m \times c}, [\![y]\!] \in \mathbb{G}^m$).

Party $p \in \{0, 1\}$ computes:

1. for $i \in [m]$:
   (a) $[\![s_i]\!] :=$ unitVector$([\![j]\!], 1) \in \mathbb{F}_2^c$ s.t. $[\![M_{i,j}]\!] = 1$. If no j exists, let $s_i = 0$.
   (b) $[\![v]\!] := [\![M]\!] \cdot [\![s_i]\!] \in \mathbb{F}_2^m$
   (c) for $j \in [m] \setminus \{i\}$:
      i. $[\![M_j]\!] := [\![M_j]\!] \oplus [\![v_j]\!] \cdot [\![M_i]\!]$
      ii. $[\![y_j]\!] := [\![y_j]\!] \oplus [\![v_j]\!] \cdot [\![y_i]\!]$
2. $[\![\sigma]\!] := \bigoplus_{s \in [m]} [\![s_i]\!] \in \mathbb{F}_2^c$
3. $[\![\overline{\sigma}]\!] := (1)^c \oplus [\![\sigma]\!]$
4. $[\![x]\!] \leftarrow \mathbb{G}^c \odot [\![\overline{\sigma}]\!]$
5. $[\![y]\!] := [\![y]\!] \oplus [\![M]\!] \cdot [\![x]\!]$
6. $[\![x]\!] := [\![x]\!] \oplus [\![s]\!] \cdot [\![y]\!]$
7. output $[\![x]\!]$

Figure 7: Randomized binary system solver.

Figure 7 has the following complexity

- step 1a requires $m(2c - 3)$ AND gates, and $m2\log_2(c)$ rounds.
- step 1b requires $O(mc)$ OTs, $O(mc^2)$ bits of comm, and $O(m)$ rounds.
- step 1c requires $O(m^2)$ OTs, $O(mc^2)$ bits of comm, and $O(m)$ rounds.
- step 4 requires $O(c)$ OTs, $O(c \log |\mathbb{G}|)$ bits, and 1 round.
- step 5 requires $O(c \log |\mathbb{G}|)$ OTs, $O(mc \log |\mathbb{G}|)$ bits, and 1 round.
- step 6 requires $O(m \log |\mathbb{G}|)$ OTs, $O(mc \log |\mathbb{G}|)$ bits, and 1 round.

Therefore, overall we have $O(m \log c)$ rounds, $O(cm + c \log |\mathbb{G}|)$ OTs, and $O(mc^2 + mc \log |\mathbb{G}|)$ bits of communication.

Public parameters: a group $\mathbb{G}$.
$\mathsf{dedup}(\llbracket A \rrbracket \in \mathbb{G}^n, \llbracket B \rrbracket \in \mathbb{G}^n, \llbracket A' \rrbracket \in \mathbb{G})$.

Party $p \in \{0, 1\}$ computes:

1. for $i \in [n], j \in [i, n]$:$\llbracket c_{i,j} \rrbracket := \mathsf{eq}(\llbracket A_i \rrbracket, \llbracket A_j \rrbracket)$
2. for $i \in [n]$:
   (a) $\llbracket d_i \rrbracket := \neg \bigvee_{k \in [i-1]} \llbracket c_{k,i} \rrbracket$
   (b) $\llbracket c_i \rrbracket := \llbracket d_i \rrbracket \llbracket c_i \rrbracket$
   (c) $\llbracket A_i \rrbracket := \llbracket d_i \rrbracket \llbracket A_i \rrbracket \oplus \overline{\llbracket d_i \rrbracket} \llbracket A' \rrbracket$
   (d) for $j \in [i, n]$: $\llbracket B'_{i,j} \rrbracket := \llbracket B_i \rrbracket \llbracket c_{i,j} \rrbracket$
3. return $(\llbracket A \rrbracket, \llbracket B \rrbracket)$

Figure 8: Debuplicate protocol with values.

6.2 Dedup Protocol

The dedup protocol takes as input secret-shared keys $[A] \in \mathbb{G}^n$ and associated values $[B] \in \mathbb{G}^n$, along with a dummy key $[A'] \in \mathbb{G}$. The goal is to remove duplicate keys: all values associated with duplicate keys are added to the first occurrence, and subsequent duplicates are replaced with a dummy key and zero value.

1. **Equality Check:** For all $i \in [n]$ and $j \in [i, n]$, compute

$$[c_{i,j}] := \mathsf{eq}([A_i], [A_j])$$

   which returns a secret-shared bit indicating whether $A_i = A_j$.

2. **First Occurrence Detection and Deduplication:** For each $i \in [n]$:

   (a) Compute a bit indicating whether A_i is the first occurrence of its key:

$$[d_i] := \neg \bigvee_{k \in [i-1]} [c_{k,i}]$$

   This bit is 1 if and only if A_i has not occurred previously.

   (b) Define a control bit $[c_i]$ to enable value accumulation only for first occurrences:

$$[c_i] := [d_i] \cdot [c_i]$$

   (c) Replace duplicate keys with the dummy key $[A']$:

$$[A_i] := [d_i] \cdot [A_i] \oplus (\neg [d_i]) \cdot [A']$$

   (d) For $j \in [i, n]$, zero out duplicate values using the pairwise comparison bits:

$$[B_{i,j}] := [B_{i,j}] \cdot [c_{i,j}]$$

3. **Output:** Return the modified keys and values:

$$\text{return } ([A], [B])$$

The result is that for each set of equal keys, only the first occurrence remains, and all associated values are added to that index. All subsequent duplicates are replaced with the dummy key A' and their values are set to zero.

7 Cuckoo-Based DMPF

Cuckoo hashing assigns each input to one of a small number of candidate bins, with at most one input per bin [PR04]. Applied to DMPFs, it gives one sparse single-point function per bin [BGI19, SGRR19, BGH+25]. We review this layout because it separates two tasks: reducing local expansion and constructing the secret placement. Waterfall addresses the second task.

For this description, fix distinct addresses $A_1, \ldots, A_t \in [0, N)$; duplicate aggregation and dummy padding are handled by the full construction. Input rows use $i \in [t]$, hash choices use $j \in [w]$, and physical bins use $b \in [0, m)$. Public hashes $h_j : [0, N) \to [0, m)$ define a *position map* $P \in \{0,1\}^{[0,N)\times[0,m)}$ by

$$P_{x,b} := [b \in \{h_1(x), \ldots, h_w(x)\}].$$

A valid placement assigns distinct columns $c_1, \ldots, c_t \in [0, m)$ such that $P_{A_i,c_i} = 1$ for every $i \in [t]$. Not every hash tuple admits such a placement. Parameters must bound this failure probability over the sampled hashes for each fixed input set.

Given a valid placement, define $U \in \{0,1\}^{[0,N)\times[0,m)}$ by $U_{A_i,c_i} := 1$, with all other entries zero. Each column of U has weight at most one. For payloads $B \in \mathbb{G}^t$, define $D \in \mathbb{G}^{[0,N)\times[0,m)}$ by $D_{A_i,c_i} := B_i$, with all other entries zero. Thus each column of D is a point map or the zero map, and

$$\sum_{i\in[t]:A_i=x} B_i = \sum_{b\in[0,m)} D_{x,b}.$$

The public sparse set for column b is $S_b := \{x \in [0, N) : P_{x,b} = 1\}$. Write $S := (S_b)_{b\in[0,m)}$. Only entries in S_b need to be expanded in that column. Figure 9 illustrates the position map, placement, payloads, and sparse column view.

$$P = \begin{pmatrix} 1 & 0 & \mathbf{1} & 0 \\ 1 & 0 & 0 & 1 \\ 0 & \mathbf{1} & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & \mathbf{1} \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}, \; U = \begin{pmatrix} 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \; D = \begin{pmatrix} 0 & 0 & B_2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & B_1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & B_3 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \; D|_S = \begin{pmatrix} 0 & \shortmid & B_2 & \shortmid \\ 0 & \shortmid & \shortmid & 0 \\ \shortmid & B_1 & 0 & \shortmid \\ \shortmid & 0 & \shortmid & 0 \\ 0 & \shortmid & 0 & \shortmid \\ \shortmid & 0 & \shortmid & B_3 \\ 0 & 0 & \shortmid & \shortmid \\ \shortmid & \shortmid & 0 & 0 \end{pmatrix}$$

Figure 9: Example for $N = 8$, $t = 3$, $m = 4$, $w = 2$, with addresses $(A_1, A_2, A_3) = (2, 0, 5)$ and columns $(c_1, c_2, c_3) = (1, 2, 3)$. Rows and columns are numbered from zero. Bold entries mark the placement. $D|_S$ displays each column restricted to S_b; marks $\shortmid$ denote omitted entries, which are zero in D.

In the example, summing the columns gives $(B_2, 0, B_1, 0, 0, B_3, 0, 0)$. The public sparse sets are

$$S_0 = \{0, 1, 4, 6\}, \quad S_1 = \{2, 3, 5, 6\}, \quad S_2 = \{0, 2, 4, 7\}, \quad S_3 = \{1, 3, 5, 7\}.$$

Using full-domain DPFs for all m columns would cost mN leaf work. Restricting each column to its sparse set instead gives

$$\sum_{b \in [0,m)} |S_b| = \sum_{x \in [0,N)} \big|\{h_1(x), \ldots, h_w(x)\}\big| \leq wN.$$

Equality holds when every address has w distinct candidate bins, as in the example and in Waterfall's partitioned layout. For constant w, this reduces leaf work from tN to $O(N)$.

Address Translation. A standard DPF over a compressed column needs the rank of A_i within the increasingly ordered set S_{c_i}, rather than the ambient address A_i. That zero-based rank is

$$A'_i := \sum_{x \in [0, A_i)} P_{x, c_i} \in [0, |S_{c_i}|).$$

For example, $A_1 = 2$ has rank zero in S_1, whose compressed payload column is $(B_1, 0, 0, 0)$. Directly computing this rank with secret A_i, c_i requires a secret-index prefix count in the public position map.

The permutation-based layout of [BGH+25] supplies the bin and its local address together. Assuming $m \mid wN$, put $f := wN/m$ and use a permutation

$$\pi : [w] \times [0, N) \longrightarrow [0, m) \times [0, f).$$

Writing $(h_j(x), a_j(x)) := \pi(j, x)$, a point placed via choice j uses local address $a_j(x)$. Each bin has exactly f slots because π is a bijection. This avoids the prefix count, but requires evaluating the permutation on secret inputs in MPC.

Sparse DPF. Our sparse DPF accepts the ambient address $A_i \in S_{c_i}$ directly and returns a map indexed by that public set. It therefore removes the address-translation permutation from this layout. The remaining tasks are to sample suitable hashes and find a valid secret placement; the sparse DPF alone does not resolve either task.

Iterative Insertion. Up to this point we have ignored the question of how to choose the column $c_i \in \{h_1(A_i), ..., h_w(A_i)\}$ that the pair (A_i, B_i) will be mapped to. Moreover, note that this must be performed within MPC as the A_i, B_i values will be seceret shared. In the plaintext setting the standard approach is to iteratively construct the set c which is initially empty. Starting with A_1, we arbitrarily choose $c_1 \in \{h_1(A_1), ..., h_w(A_1)\}$. For $A_i \in \{A_2, ..., A_t\}$, we again choose some $c_i \in \{h_1(A_i), ..., h_w(A_i)\}$. If the value of c_i is already being used by come c_j, then we "evict" A_j from column c_i and place A_i there instead. We then reinsert A_j using some other hash function value and continue this eviction processes until all items are inserted or some threshold of attempts are made, at which point the insert abort.

In MPC, the occupied-bin lookup, eviction, and stopping condition depend on secret addresses. Implementing these operations directly can dominate setup. Waterfall Cuckoo retains the sparse-column layout but uses a fixed circuit: an occupancy scan places most rows, and a bounded number of complete augmentations repairs the placement. The next section specifies that circuit and bounds its routing error.

8 Waterfall Cuckoo DMPF

8.1 Intuition

Waterfall Cuckoo combines the partitioned sparse-DPF layout of Section 7 with a fixed MPC-friendly routing rule. Its setup has four stages.

Normalize the rows. Setup retains the first row containing each real address and replaces every other address by the dummy address N. Each expansion sums duplicate payloads into the retained row and assigns zero to every dummy row. The fixed row order supplies the priority used by the placement algorithm.

Sample candidates directly. For each partition, setup samples a public t-wise independent hash function. It evaluates the hash on every real address and replaces every dummy-row result by fresh uniform randomness. The resulting candidate matrix is exactly uniform for every support. The public hashes are sampled before routing and are neither conditioned on placement success nor retried.

Place the rows. The rows scan the partitions in order. In each partition, the first live row proposing an unoccupied bin is placed there; rows that collide remain live and fall to the next partition. After this initial waterfall stage, $\mathsf{WaterfallGen}_{\mathbf{d},c,r}$ performs r canonical complete augmentations. Each augmentation searches for an alternating path from the currently unplaced rows to a free main bin and flips one such path. It then compacts any remaining rows into the c stash positions. A row left unplaced after these steps causes a correctness error; the protocol neither aborts nor retries.

Materialize and expand. The placement is hidden by an exactly uniform secret permutation and then opened. On successful routing, the opened placement is a uniform injection, independent of the true candidate bins. The opened injection permits a public scatter of the secret records. Setup initializes one reusable sparse DPF for each resulting column. Later expansions supply only new payloads and reuse all routing and DPF state.

Fix partition sizes $\mathbf{d} = (d_0, \ldots, d_{w-1})$, stash capacity c, and repair budget r. Require $w, t, N \geq 1$, $c, r \geq 0$, and $d_s = 2^{k_s}$ with $0 \leq k_s \leq \max\{1, \lceil \log_2 N \rceil\}$. Define

$$\Delta_s := \sum_{u<s} d_u, \qquad m := \sum_{s\in[0,w)} d_s + c.$$

Require $m \geq t$. These parameters expose three distinct costs. The repeated local expansion is $(w + c)N + \mathrm{poly}(t)$, because each partition and stash entry contributes one full-domain traversal. The stored position map has m physical columns. The one-time MPC setup cost depends on the partition sizes and on r. Our default profile uses fewer physical columns and $4N$ local expansion; the lower-expansion profile uses $3N$ at a larger setup cost.

Formally, a *Waterfall profile* is $\Gamma = (\mathbf{d}, c, r)$, and

$$\mathsf{WaterfallGen}_{\mathbf{d},c,r} : \prod_{s\in[0,w)} [0, d_s)^t \longrightarrow \{0,1\}^{t\times m} \tag{1}$$

is the deterministic circuit described below. Its output O is a partial placement: every row and column has weight at most one; a selected main position for row i is one of $(s, C_{i,s})_{s\in[0,w)}$; and every stash position is admissible. Define

$$\mathsf{Good}_\Gamma(C) := \big[\forall i \in [0,t),\ \|O_i\|_1 = 1\big], \qquad O := \mathsf{WaterfallGen}_{\mathbf{d},c,r}(C). \tag{2}$$

The correctness and simulation proofs use only these interface properties and the probability of $\neg\mathsf{Good}_\Gamma$.

8.2 Direct hash sampling

Let $\ell := \max\{1, \lceil \log_2 N \rceil\}$ and fix a public injection $\iota : [0, N) \hookrightarrow \mathbb{F}_{2^\ell}$. Extend its evaluator by $\iota(N) := 0$; injectivity is required only on the real domain. For $k \leq \ell$, let trunc_k return the first k coordinates in a fixed binary basis. Distribution $\mathcal{H}_{t,k}$ samples $\alpha \leftarrow \mathbb{F}_{2^\ell}^t$ and returns the restriction to $[0, N)$ of

$$h_\alpha(x) := \mathrm{trunc}_k\left(\sum_{j=0}^{t-1} \alpha_j \iota(x)^j\right). \tag{3}$$

For any $\tau \leq t$ distinct inputs, the Vandermonde evaluation map from α to the corresponding τ field values is surjective with equal fiber sizes. The values are therefore independent and uniform. Truncation preserves independence and makes every output uniform in $\mathbb{F}_2^k \cong [0, 2^k)$.

Let $d_s = 2^{k_s}$. For a deduplicated address vector A^*, setup samples independent $h_s \leftarrow \mathcal{H}_{t,k_s}$ and $U_{i,s} \leftarrow [0, d_s)$, and sets

$$C_{i,s} := \begin{cases} h_s(A_i^*), & A_i^* \neq N, \\ U_{i,s}, & A_i^* = N. \end{cases} \tag{4}$$

The direct family need not treat N as a distinct field point because a dummy row never contributes a hash evaluation to C.

Lemma 8.1 (Uniform candidate matrix)**.** *For every fixed deduplicated address vector A^*, the matrix C is uniform in $\prod_{s\in[0,w)}[0,d_s)^t$.*

Proof. The real addresses are distinct, so their values within each partition are independent and uniform by t-wise independence. The hash seeds are independent across partitions. Every dummy entry is selected from an independent cell of U. These values cover all entries of C. □

This construction handles arbitrary N without enlarging the DMPF domain to a power of two. Power-of-two candidate ranges are used only to implement the hash outputs and secret-index circuits efficiently.

8.3 Protocol

The protocol has one address-dependent setup and a reusable expansion. The protocol box declares the retained state; named DPF instances retain their own leaf seeds and expansion counters. The notation $\mathsf{eq}(x,y)$ denotes a Boolean equality circuit, and reveal opens a shared value to both parties.

For public parameters $\kappa, N, t \in \mathbb{N}$, a finite additive group $\mathbb{G}$, and a Waterfall profile $\Gamma = (\mathbf{d}, c, r)$, let $m := \sum_{s\in[0,w)} d_s + c$. Party $p \in \{0,1\}$ stores its local component of $\mathsf{st}_\Gamma = (\llbracket\delta\rrbracket, \llbracket e\rrbracket, h, P, S)$, initially undefined. The instance owns named reusable DPF instances $\mathsf{DPF}^{\mathsf{col},j}$ over S_j whenever $S_j \neq \emptyset$.

$\mathsf{WaterfallCuckoo}_\Gamma.\mathsf{Setup}(\llbracket A\rrbracket \in \mathcal{X}_N^t) \longrightarrow \mathsf{ready}$.

Party $p \in \{0,1\}$ computes:

1. If setup has already completed, return $\perp$.
2. For $i \in [0,t)$, set $\llbracket e\rrbracket_{i,i} := 1$. For $i < j$, set $\llbracket e\rrbracket_{i,j} := \mathsf{eq}(\llbracket A\rrbracket_i, \llbracket A\rrbracket_j)$ and $\llbracket e\rrbracket_{j,i} := \llbracket e\rrbracket_{i,j}$.
3. For $i \in [0,t)$, set $\llbracket\delta\rrbracket_i := (1 - \mathsf{eq}(\llbracket A\rrbracket_i, N)) \bigwedge_{j<i}(1 - \llbracket e\rrbracket_{i,j})$.
4. For $i \in [0,t)$, set $\llbracket A\rrbracket_i^* := N \oplus \llbracket\delta\rrbracket_i(N \oplus \llbracket A\rrbracket_i)$.
5. For $s \in [0,w)$, sample public $h_s \leftarrow \mathcal{H}_{t,k_s}$.
6. For $(i,s) \in [0,t) \times [0,w)$, sample $\llbracket U\rrbracket_{i,s} \leftarrow [0,d_s)$.
7. For $(i,s) \in [0,t) \times [0,w)$, set $\llbracket C\rrbracket_{i,s} := \llbracket U\rrbracket_{i,s} \oplus \llbracket\delta\rrbracket_i(\llbracket U\rrbracket_{i,s} \oplus h_s(\llbracket A\rrbracket_i^*))$.
8. $\llbracket O\rrbracket := \mathsf{WaterfallGen}_{\mathbf{d},c,r}(\llbracket C\rrbracket)$.
9. Define $P \in \{0,1\}^{[0,N)\times m}$ by $P_{x,\Delta_s+h_s(x)} := 1$ for $(x,s) \in [0,N) \times [0,w)$ and $P_{x,\Delta_w+k} := 1$ for $(x,k) \in [0,N) \times [0,c)$, with all remaining entries zero.
10. For $j \in [0,m)$, let $S_j := \{x \in [0,N) : P_{x,j} = 1\}$ and $S := (S_j)_{j\in[0,m)}$.
11. $\llbracket A\rrbracket' := \mathsf{WaterfallScatter}_{\mathbf{d},c}.\mathsf{Setup}(\llbracket A\rrbracket^*, \llbracket\delta\rrbracket, \llbracket O\rrbracket, P, S)$.
12. For $j \in [0,m)$ with $S_j \neq \emptyset$, invoke $\mathsf{DPF}^{\mathsf{col},j}.\mathsf{Setup}_{S_j}(\llbracket A\rrbracket'_j)$.
13. Store $\mathsf{st}_\Gamma := (\llbracket\delta\rrbracket, \llbracket e\rrbracket, h, P, S)$ and return ready.

$\mathsf{WaterfallCuckoo}_\Gamma.\mathsf{Expand}(\llbracket B\rrbracket \in \mathbb{G}^t) \longrightarrow \llbracket y\rrbracket \in \mathbb{G}^{[0,N)}$.

Party $p \in \{0,1\}$ computes:

1. If setup has not completed, return $\perp$. Otherwise, read st_Γ.
2. For $i \in [0,t)$, set $\llbracket B\rrbracket_i^* := \llbracket\delta\rrbracket_i \sum_{j\in[0,t)} \llbracket e\rrbracket_{i,j} \llbracket B\rrbracket_j$.
3. $\llbracket B\rrbracket' := \mathsf{WaterfallScatter}_{\mathbf{d},c}.\mathsf{Expand}(\llbracket B\rrbracket^*)$.
4. For $j \in [0,m)$ with $S_j \neq \emptyset$, set $\llbracket y\rrbracket'_j := \mathsf{DPF}^{\mathsf{col},j}.\mathsf{Expand}_{S_j}(\llbracket B\rrbracket'_j) \in \mathbb{G}^{S_j}$.
5. For $x \in [0,N)$, set $\llbracket y\rrbracket(x) := \sum_{j:x\in S_j} \llbracket y\rrbracket'_j(x)$.
6. Return $\llbracket y\rrbracket$.

Figure 10: The reusable Waterfall-Cuckoo distributed multi-point protocol. Setup samples and routes one candidate matrix without retry. Expansion reuses the resulting address-dependent state for any polynomial number of payload vectors.

All index-dependent DPF OTs and tree expansions occur during setup. Expansion uses the cached equality matrix to aggregate duplicate payloads and each DPF's internal counter for leaf conversion. The permutation correlations are used only during setup.

8.4 Placement generator

The initial waterfall stage uses one secret occupancy array per partition. Set $a_i^{(0)} := 1$ for every row. For partition s, initialize $T_s^{(0)} := 0^{d_s}$ and scan $i = 0, \ldots, t-1$. Secret-index reads and gated unit vectors implement

$$\begin{aligned} o_i^{(s)} &:= T_s^{(i)}[C_{i,s}], & \ell_i^{(s)} &:= a_i^{(s)} o_i^{(s)}, \\ u_i^{(s)} &:= a_i^{(s)}(1 - o_i^{(s)}), & a_i^{(s+1)} &:= \ell_i^{(s)}, \\ E_i^{(s)} &:= \mathsf{unitVector}_{[0,d_s)}(C_{i,s}, u_i^{(s)}), & T_s^{(i+1)} &:= T_s^{(i)} \oplus E_i^{(s)}. \end{aligned} \tag{5}$$

The bit $u_i^{(s)}$ places row i in partition s, while $a_i^{(s+1)}$ records that it remains live. The update is valid because $u_i^{(s)} = 1$ only when the selected cell was previously zero. The resulting main-position matrix is

$$O^{(0)}_{i,\Delta_s+b} := u_i^{(s)}[C_{i,s} = b]. \tag{6}$$

After the scan, $q_i := a_i^{(w)}$ identifies the overflow rows and

$$R_i := \sum_{j<i} q_j \tag{7}$$

is their stable rank. Stash compaction places the first c remaining rows:

$$O_{i,\Delta_w+k} := q_i[R_i = k], \qquad k \in [0, c). \tag{8}$$

The case $r = 0$ stops after this initial stage; it is a parameter choice, not a separate scheme.

For complete repair, view the main positions as the right vertices of the bipartite candidate graph

$$G_C := \big([0,t),\ [0,\Delta_w),\ \{(i, \Delta_s + C_{i,s}) : (i,s) \in [0,t) \times [0,w)\}\big). \tag{9}$$

Starting from the initial matching $O^{(0)}$, one canonical augmentation treats every unmatched row as a root of an alternating-path search. It expands the first discovered but unexpanded row in public row order, scans that row's partitions in public order, and records the first parent of each newly discovered owner. The first free candidate ends the search. If one is found, the circuit flips its unique parent path; otherwise it leaves the matching unchanged. Thus one complete round repairs one unmatched row whenever any augmenting path exists. Repeating the round r times repairs at most r rows. Recompute q_i from the repaired matching and apply the same stash-rank and compaction formulas to the remaining rows. All loop bounds and tie-breaking rules are public, so this is a deterministic circuit. Appendix A gives the gate-level circuit and its invariant proof.

Every intermediate and final matrix satisfies

$$\sum_{j \in [0,m)} O_{i,j} \le 1, \qquad \sum_{i \in [0,t)} O_{i,j} \le 1, \tag{10}$$

and, for the position map defined in Figure 10,

$$A_i^* \neq N \ \wedge\ O_{i,j} = 1 \quad \Longrightarrow \quad P_{A_i^*,j} = 1. \tag{11}$$

Proposition 8.2 (Complete-repair criterion)**.** *Sample* $C \leftarrow \prod_{s\in[0,w)}[0,d_s)^t$ *and put*

$$p_\Gamma := \Pr_C[\neg\mathsf{Good}_\Gamma(C)].$$

Let L *be the number of rows left unmatched by the initial waterfall stage. If* G_C *has a matching of size* t *and* $L \le r + c$*, then* $\mathsf{WaterfallGen}_{\mathbf{d},c,r}$ *returns a valid placement. Hence*

$$p_\Gamma \le \Pr[L > r + c] + \Pr[G_C \text{ has no matching of size } t]. \tag{12}$$

Proof. If at most c rows remain, stash compaction places all of them. Otherwise, the current main-bin matching has size below $t - c$. Berge's lemma gives an augmenting path whenever G_C has a matching of size t. A complete repair round searches from every unmatched root, finds and flips such a path, and increases the matching size by one. Induction over r rounds leaves at most c unmatched rows whenever $L \le r + c$. □

8.5 Oblivious record scatter

The secret matrix O is a partial injection of the t labeled rows into the m columns. Directly multiplying every record by every entry would require tm conditional moves of full records. Waterfall instead hides the selected columns by two serial permutation networks.

Let $\mathsf{Inj}(t,m)$ be the set of injections $[0,t) \to [0,m)$. Party p samples a uniform permutation $\sigma_p \leftarrow \mathbb{S}_m$ and privately programs one Waksman network. The two passes apply

$$\rho := \sigma_1 \circ \sigma_0. \tag{13}$$

Conditioned on either party's permutation, the other permutation remains uniform. Composition by the fixed permutation is a bijection on $\mathbb{S}_m$. Thus for every $o, v \in \mathsf{Inj}(t,m)$,

$$\Pr[\rho \circ o = v \mid \sigma_p] = \frac{(m-t)!}{m!} = \frac{1}{(m)_t}. \tag{14}$$

The opened matrix

$$\widehat{O}_{i,\rho(j)} := O_{i,j} \tag{15}$$

is therefore an exactly uniform labeled injection on $\mathsf{Good}_\Gamma(C)$, for every fixed true placement and either party's permutation. This statement is in the ideal permutation hybrid, before the opening, where the returned shares hide the honest permutation. It is essential that O includes the physical placements of dummy rows; otherwise the opening would reveal the number of real rows.

The opened matrix scatters the shared activity bits and masked addresses using public wiring and linear operations. Applying the inverse serial passes restores their true columns. The destination of row i is

$$z_i := \sum_{j\in[0,m)} jO_{i,j} \in [0,m). \tag{16}$$

Setup initializes a reusable DPF for each z_i. During every expansion,

$$[\![v]\!]_i := \mathsf{DPF}^{\mathsf{sc},i}.\mathsf{Expand}_{[0,m)}([\![B]\!]_i), \qquad [\![B]\!]'_j := \sum_{i\in[0,t)} [\![v]\!]_i(j). \tag{17}$$

For every column, let

$$D_j := \begin{cases} \min S_j, & S_j \neq \emptyset, \\ 0, & S_j = \emptyset, \end{cases} \qquad A'_j := \bar{A}_j + (1 - \chi_j) D_j. \tag{18}$$

A real row preserves its address and payload. An unused column or physical dummy receives a public representative and zero payload.

For public $N, t \in \mathbb{N}$, partition sizes $\mathbf{d} = (d_s)_{s\in[0,w)}$, and stash capacity c, let $m := \sum_{s\in[0,w)} d_s + c$. The public sparse sets satisfy $S = (S_j)_{j\in[0,m)}$ and $S_j = \{x : P_{x,j} = 1\}$. Party $p \in \{0,1\}$ stores its local component of $\mathsf{st_{sc}} = (\llbracket z \rrbracket, (\mathsf{DPF}^{\mathsf{sc},i})_{i\in[0,t)})$, initially undefined. Each $\mathsf{DPF}^{\mathsf{sc},i}$ is reusable over $[0,m)$.

$\mathsf{WaterfallScatter}_{\mathbf{d},c}.\mathsf{Setup}\big(\llbracket A \rrbracket \in \mathcal{X}_N^t, \llbracket \delta \rrbracket \in \{0,1\}^t, \llbracket O \rrbracket \in \{0,1\}^{t\times m},$
$P \in \{0,1\}^{[0,N)\times m}, S \in (2^{[0,N)})^m\big) \longrightarrow \llbracket A \rrbracket' \in [0,N)^m.$

Party $p \in \{0,1\}$ computes:

1. If setup has already completed, return $\perp$.
2. Sample $\sigma_p \leftarrow \mathbb{S}_m$ and privately program Waksman pass p for σ_p. Let $\llbracket \rho \rrbracket := \sigma_1 \circ \sigma_0$ denote the two serial hidden passes.
3. $\llbracket \widehat{O} \rrbracket := \mathsf{permute}_{\llbracket \rho \rrbracket}(\llbracket O \rrbracket)$ along its column dimension.
4. $\widehat{O} := \mathsf{reveal}(\llbracket \widehat{O} \rrbracket)$.
5. For $i \in [0,t)$, set $\llbracket z \rrbracket_i := \sum_{j\in[0,m)} j \llbracket O \rrbracket_{i,j} \in [0,m)$.
6. For $i \in [0,t)$, set $\llbracket \widetilde{A}_i \rrbracket := \llbracket \delta \rrbracket_i \llbracket A \rrbracket_i$.
7. For $k \in [0,m)$, set $\llbracket \widehat{\chi}_k \rrbracket := \sum_{i\in[0,t)} \widehat{O}_{i,k} \llbracket \delta \rrbracket_i$.
8. For $k \in [0,m)$, set $\llbracket \widehat{A}_k \rrbracket := \sum_{i\in[0,t)} \widehat{O}_{i,k} \llbracket \widetilde{A}_i \rrbracket$.
9. $(\llbracket \chi \rrbracket, \llbracket \bar{A} \rrbracket) := \mathsf{permute}^{-1}_{\llbracket \rho \rrbracket}(\llbracket \widehat{\chi} \rrbracket, \llbracket \widehat{A} \rrbracket)$.
10. For $j \in [0,m)$, set $D_j := \min S_j$ if $S_j \neq \emptyset$ and $D_j := 0$ otherwise.
11. For $j \in [0,m)$, set $\llbracket A \rrbracket'_j := \llbracket \bar{A}_j \rrbracket + (1 - \llbracket \chi \rrbracket_j) D_j$.
12. For $i \in [0,t)$, invoke $\mathsf{DPF}^{\mathsf{sc},i}.\mathsf{Setup}_{[0,m)}(\llbracket z \rrbracket_i)$.
13. Store $\mathsf{st_{sc}} := (\llbracket z \rrbracket, (\mathsf{DPF}^{\mathsf{sc},i})_{i\in[0,t)})$ and return $\llbracket A \rrbracket'$.

$\mathsf{WaterfallScatter}_{\mathbf{d},c}.\mathsf{Expand}\big(\llbracket B \rrbracket \in \mathbb{G}^t\big) \longrightarrow \llbracket B \rrbracket' \in \mathbb{G}^m.$

Party $p \in \{0,1\}$ computes:

1. If setup has not completed, return $\perp$.
2. For $i \in [0,t)$, set $\llbracket v \rrbracket_i := \mathsf{DPF}^{\mathsf{sc},i}.\mathsf{Expand}_{[0,m)}(\llbracket B \rrbracket_i) \in \mathbb{G}^{[0,m)}$.
3. For $j \in [0,m)$, set $\llbracket B \rrbracket'_j := \sum_{i\in[0,t)} \llbracket v \rrbracket_i(j)$.
4. Return $\llbracket B \rrbracket'$.

Figure 11: Reusable permutation-assisted scatter into the unified position map. Setup fixes the destinations and initializes their DPF states. Expansion supplies only a new payload vector.

The forward permutation carries the t-bit columns of O. The inverse pass carries records of width $1 + \lceil \log_2(N+1) \rceil$. The same switch correlations support both directions by domain-separated PRG expansion [PRRS25]. The correlated permutations act only during setup.

8.6 Correctness and simulation

For $\Gamma = (\mathbf{d}, c, r)$, define

$$p_\Gamma := \Pr_{C \leftarrow \prod_{s\in[0,w)} [0,d_s)^t} [\neg \mathsf{Good}_\Gamma(C)]. \tag{19}$$

Fix polynomials $B_0(\kappa)$ and $Q(\kappa)$. The protocol initializes B_0 independent Waterfall states and performs at most Q expansions per state. Let $\epsilon_{\mathsf{corr}}(B_0, Q)$ and $\epsilon_{\mathsf{sub}}(B_0, Q)$ bound the combined

correctness error and simulation advantage of the invoked MPC, permutation, PRG, and reusable DPF subprotocols.

We prove simulation in the hybrid model with ideal circuit, permutation, and reusable DPF calls. Each call exposes only its specified outputs; secret circuit and permutation outputs use fresh additive or XOR shares. Locally derived values and reused shares retain their prescribed dependencies. The implementation state of an ideal subprotocol is internal to that functionality. Its concrete realization is covered by its own simulator, not simulated again in the Waterfall hybrid. At a DPF call, the corrupt party supplies its own output share according to Figure 1. A concrete DPF simulator supplies that share when we replace a concrete call by its ideal functionality. These choices may depend on the preceding view; we require no independent output-sharing law.

Proposition 8.3 (Waterfall correctness)**.** *For arbitrary setup inputs $A^{(b)} \in \mathcal{X}_N^t$ and payload vectors $B^{(b,q)} \in \mathbb{G}^t$, all $B_0 Q$ outputs equal $\mathsf{mp}_{N,t}(A^{(b)}, B^{(b,q)})$ except with probability at most*

$$B_0 p_\Gamma + \epsilon_{\mathsf{corr}}(B_0, Q).$$

The routing term is paid once per cached state, not once per expansion.

Proof. Condition on $\mathsf{Good}_\Gamma(C)$ and correct execution of the reused subprotocols. The placement invariants imply that every row has one distinct admissible column. For a real row, its selected main candidate equals $h_s(A_i^*)$; every stash column admits all real addresses. The inverse permutation passes cancel the forward passes, and the reusable DPF for z_i places B_i^* in the selected column during every expansion. Activity bits preserve real addresses and replace physical dummies and unused columns by public representatives with zero payload. The final pointwise sum therefore equals $\mathsf{mp}_{N,t}(A, B)$. A union bound over the B_0 independently routed states proves the claim. □

Theorem 8.4 (Generic Waterfall-Cuckoo simulation)**.** *Against either static semi-honest corrupted party, Waterfall Cuckoo realizes $\mathcal{F}_{\mathsf{rDMPF}}$ for B_0 independently initialized states and at most Q adaptive expansions per state. For every PPT adversary, there is a PPT simulator whose distinguishing advantage satisfies*

$$\mathrm{Adv}_{\mathsf{WF}}(B_0, Q) \le B_0 p_\Gamma + \epsilon_{\mathsf{sub}}(B_0, Q). \tag{20}$$

The simulator receives the corrupted party's input shares and submits its output share to the functionality, as specified after Figure 6. The claim uses no LPN or syndrome-decoding assumption.

Proof. Fix the corrupted party p. In the ideal-subprotocol hybrid, the simulator samples the public hashes with their original law and computes P, S locally. It samples private outputs of the ideal setup calls according to their sharing interfaces and evaluates local operations on those shares. For the placement opening, it samples a uniform labeled injection $V \leftarrow \mathsf{Inj}(t, m)$. If party p holds opening share $\widehat{O}_p$, the simulated honest message is $V \oplus \widehat{O}_p$.

On successful routing this opening has exactly the required distribution by Equation (14). The simulator does not condition or retry the public hashes. Couple its opening to the real opening whenever routing succeeds; replacing the entire transcript of a failed state costs at most p_Γ by Lemma 8.1.

During expansion, the simulator samples the fresh circuit-output shares for duplicate aggregation and computes all local sums. It runs the corrupt party's share-selection strategy at each row-to-column and column DPF call. It sums the resulting column maps exactly as the protocol does and submits that sum as its DMPF output share. On successful routing, correctness makes

the honest output its required complement. No output map is prescribed to the simulator. Appendix A.8 gives the call-by-call argument, including the joint distribution of the honest output and corrupt view. The simulator retains its state for subsequent adaptive expansions.

The hybrid simulator therefore uses only the corrupted party's inputs and the simulated history. A union bound charges failed routing once for each of the B_0 states, not once per expansion. Replacing the ideal calls by their concrete realizations and subprotocol simulators adds at most $\epsilon_{\mathsf{sub}}(B_0, Q)$, proving the claim. □

8.7 Profiles and concrete cost

For every partition s, the sets $\{S_{\Delta_s+b} : b \in [0, d_s)\}$ partition $[0, N)$. Every stash column has $S_j = [0, N)$. Hence

$$\sum_{j\in[0,m)} |S_j| = (w + c)N, \tag{21}$$

and the repeated local expansion work is

$$(w + c)N + \mathrm{poly}(t). \tag{22}$$

Table 1 reports the two main profiles at $t = 16$, $B_0 = 256$, $N = 2^{20}$, $\kappa = 128$, and $\mathbb{G} = \mathbb{Z}_{2^{64}}$. "Batch bits" means $-\log_2(B_0 p_\Gamma)$. The OT columns give a per-state upper bound on initial correlations for direct hashing, placement, scatter, and reusable DPF initialization. The batch size B_0 enters only the correctness bound. The DPF term reserves two OTs per shared correction selection, activity masks below the dense prefix, and cached sign-conversion correlations. A shared bit times a shared string uses two OTs; the string width contributes to the logical OT payload. For retained control bits, later multiplications expand cached OT strings with round-separated PRGs and send masked payloads, without obtaining new OTs. The model excludes public openings, OT-extension overhead, and local computation; it is not a measurement of setup time or total wire traffic. Appendix A.3 gives the exact occupancy, Hall-witness, and component-cost calculations.

profile	$\mathbf{d}$	(c, r)	m	expansion	total OTs	OT payload	batch bits
default $4N$	$(16, 16, 16, 16)$	$(0, 3)$	64	$4N$	39,052	942,888 bits	43.37
lower-expansion $3N$	$(128, 128, 64)$	$(0, 2)$	320	$3N$	55,358	3,623,410 bits	41.13

Table 1: Main Waterfall profiles. Both use the same complete-repair generator, protocol, and simulation proof. OT payload sums the modeled selected-message lengths; it is not total wire traffic.

The compact $4N$ profile is the default. Compared with the $3N$ profile, it uses fewer physical columns and OTs, and carries substantially less OT payload. The $3N$ profile is preferable only when a state is expanded often enough that removing one full-domain traversal from every local expansion dominates its larger setup and 320-column position map.

Alternative profiles. The same generator admits other setup–state tradeoffs. Stopping after the initial waterfall stage ($r = 0$) gives the simplest but least compact profile. The asymmetric $4N$ profile $\mathbf{d} = (32, 32, 16, 8)$, $c = 0$, $r = 2$ uses $m = 88$ columns and slightly fewer OTs, but more payload and a smaller correctness margin. A shallow one-repair circuit replaces complete search

by one fixed depth-two repair and a two-entry stash; it minimizes setup at the cost of $6N$ local expansion. Appendix A.4 records these alternatives without changing the main construction.

9 Bucketing DMPF

Bucketing is useful when the input addresses are independent and uniform. Choose a public bucket count $k \mid N$, bucket capacity c, and $M := N/k$. Address A_i belongs to bucket $\lfloor A_i/M \rfloor$ and has local address $A_i \bmod M$. A padded Group-By operation collects the records for each bucket. One DMPF per bucket produces the corresponding interval of the output map.

If any bucket receives more than c rows, the construction may omit inputs and leak information through the Group-By opening. For independent uniform addresses and $c < t$, a union bound gives

$$\Pr[\text{overflow}] \le k \Pr[\mathrm{Bin}(t, 1/k) > c] \le k \binom{t}{c+1} k^{-(c+1)}.$$

The probability is zero for $c \ge t$. Choose k, c to make this bound at most the desired $2^{-\lambda}$. The choice $k \asymp t/\log t$ is a scaling heuristic, not by itself a concrete security bound.

Let $C_{\mathsf{setup}}(M, c)$ and $C_{\mathsf{expand}}(M, c)$ be the costs of the chosen DMPF. The total costs include k such calls, the Group-By cost, and $\Theta(N)$ output materialization. They are not $O(t)$ independently of N. The box below shows a single payload evaluation; it uses the reusable DMPF interface but does not specify a separate reusable Group-By implementation.

Public parameters are $N, t, k, c \ge 1$, $k \mid N$, and an additive group $\mathbb{G}$. Put $M := N/k$.
$\mathsf{BucketingDMPF}(\llbracket A \rrbracket \in [0,N)^t, \llbracket B \rrbracket \in \mathbb{G}^t) \longrightarrow \llbracket y \rrbracket \in \mathbb{G}^{[0,N)}$:

Party $p \in \{0,1\}$ computes:

1. For $i \in [0,t)$, set $\llbracket A'_i \rrbracket := \lfloor \llbracket A \rrbracket_i / M \rfloor$.
2. For $i \in [0,t)$, set $\llbracket A''_i \rrbracket := \llbracket A \rrbracket_i \bmod M$.
3. $(\llbracket A \rrbracket^*, \llbracket B \rrbracket^*) := \triangleleft\triangleright \left(\mathsf{Group\text{-}By}_{c,t,k}(\llbracket A \rrbracket', \llbracket A \rrbracket'' \bowtie \llbracket B \rrbracket)\right)$, using record group $\mathbb{Z}_M \times \mathbb{G}$.
4. For $j \in [0,k)$:
   (a) Invoke $\mathsf{DMPF}^j.\mathsf{Setup}(\llbracket A \rrbracket^*_j \in [0,M)^c)$.
   (b) $\llbracket z \rrbracket_j := \mathsf{DMPF}^j.\mathsf{Expand}(\llbracket B \rrbracket^*_j)$.
5. For $j \in [0,k)$ and $x \in [0,M)$, set $\llbracket y \rrbracket(jM + x) := \llbracket z \rrbracket_j(x)$.
6. Return $\llbracket y \rrbracket$.

Figure 12: Bucketing with padded groups and one DMPF per bucket.

9.1 Group-By Operation

For keys $A \in [0,N)^t$, payload records $B \in \mathbb{G}^t$, and capacity p, Group-By returns N groups of exactly p records. When every key occurs at most p times, group i contains its input records and enough zero records to fill the group, in hidden random order.

Let f_i be the count of key i as an integer in $[0,t]$. The count modulus is 2^ℓ for $\ell := \lceil \log_2(t+1) \rceil$; in particular, it represents the allowed boundary count $f_i = p$ without wrapping. For each key,

create p auxiliary records, of which $p - \min(f_i, p)$ have label i and selection bit one. The others have label zero and selection bit zero. Append the real records, all with selection bit one, and shuffle. Opening the labels and selection bits identifies the groups, but not which selected records are real: selected padding records also have bit one.

Public parameters are $N, t, p \geq 1$ and an additive record group $\mathbb{G}$. Put $\ell := \lceil \log_2(t+1) \rceil$.
$\mathsf{Group\text{-}By}_{p,t,N}(\llbracket A \rrbracket \in [0,N)^t, \llbracket B \rrbracket \in \mathbb{G}^t) \longrightarrow \llbracket G \rrbracket \in (\mathbb{G}^p)^N$:
The parties compute:

1. $\llbracket f \rrbracket := \sum_{j \in [0,t)} \mathsf{unitVector}_{[0,N)}(\llbracket A \rrbracket_j, 1) \in \mathbb{Z}_{2^\ell}^{[0,N)}$.
2. For $i \in [0, N)$, set $\llbracket d_i \rrbracket := (0)^{\min(\llbracket f \rrbracket_i, p)} \| (1)^{p - \min(\llbracket f \rrbracket_i, p)}$.
3. $\llbracket A \rrbracket' := 0 \cdot \llbracket d \rrbracket_0 \| \cdots \| (N-1) \cdot \llbracket d \rrbracket_{N-1} \| \llbracket A \rrbracket$.
4. $\llbracket B \rrbracket' := 0^{Np} \| \llbracket B \rrbracket$.
5. $\llbracket d \rrbracket' := \llbracket d \rrbracket_0 \| \cdots \| \llbracket d \rrbracket_{N-1} \| 1^t$.
6. $(\llbracket A \rrbracket', \llbracket B \rrbracket', \llbracket d \rrbracket') := \mathsf{shuffle}(\llbracket A \rrbracket' \bowtie \llbracket B \rrbracket' \bowtie \llbracket d \rrbracket')$.
7. $(A', d') := \mathsf{reveal}(\llbracket A \rrbracket' \bowtie \llbracket d \rrbracket')$.
8. For $i \in [0, N)$, let $\llbracket G \rrbracket_i$ be the first p records $\llbracket B \rrbracket'_j$ with $A'_j = i$ and $d'_j = 1$.
9. Return $\llbracket G \rrbracket$.

Figure 13: Padded Group-By. Selection bit one includes both input and padding records. Overflow is truncated rather than causing an explicit abort.

Correctness and opening distribution. If $f_i \leq p$ for every key, each group has f_i input records and $p - f_i$ selected zero records. Exactly t auxiliary records are unselected. The opened pairs are therefore a uniform permutation of a fixed multiset: p copies of $(i, 1)$ for each $i \in [0, N)$, and t copies of $(0, 0)$. This describes the joint opening, not independent distributions for its two fields. In the ideal-shuffle hybrid, it hides the group sizes on the no-overflow event. Fresh output shares hide the retained payload records. On overflow, a group may have more than p selected records; truncation keeps the output shape fixed, and the entire failed transcript is charged to the caller's overflow bound.

Frequency evaluation and construction of the auxiliary records require secret-index circuits. The explicit arrays already contain $Np + t$ records; a dense frequency implementation processes Nt indicator entries. These costs must be included before comparing bucketing with another DMPF.

10 Improved Big-State DMPF

Boyle et al. [BGH+25] introduced two approaches to DMPF: an OKVS-based construction and the *big-state* construction. The OKVS variant uses *oblivious key-value stores* to encode correction information for multiple indices, achieving good performance for very large t. By contrast, the big-state construction targets small to moderate weights t (roughly $3 \leq t \leq 70$), as arise in pseudorandom correlation generators. The work primarily considers a trusted dealer that knows A, B in plaintext, constructs the keys, and distributes them to the evaluation parties. That interface differs

from distributed generation with secret-shared inputs. The authors also give a fully distributed protocol for their big-state construction, which we review next.

The key insight of the big-state construction is to generalize the standard single point DPF protocol of Section 4 such that the "tag bit" vector that marks the active child on each level becomes a set of t *tag vectors*. Instead of each node i carrying a single bit τ_i that determines which child is "active," each node i in the evaluation tree maintains t tag bits $\tau_{1,i}, ..., \tau_{t,i}$. As before, each tag vector is a unit vector, i.e. $|\tau_j| = 1$. However, there are now t check vectors and therefore there can be t active children on each level.

The parties expand the tree from root to leaf. At each level $d \in [1, D]$, the parties hold a *single* random secret shared vector of seeds $[\![s^{d-1}]\!] \in \{0,1\}^{\kappa \times 2^{d-1}}$ for the previous level. s^{d-1} when viewed as a vector over $\{0,1\}^\kappa$ elements has hamming weight (at most) t. Party $p \in \{0,1\}$ expand their seed shared $[\![s_i^{d-1}]\!]_p \in \{0,1\}^\kappa$ via the PRG $\mathsf{G} : \{0,1\}^\kappa \to \{0,1\}^{\kappa \times 2 + t \times 2}$ to obtain a total of 2^d children seeds $[\![s']\!] \in \{0,1\}^{\kappa \times 2^d}$ and tags $[\![\tau']\!] \in \{0,1\}^{2^d}$. However, s'_i in general has hamming weight $2t$ instead of just the t active children that we desire. In particular, each active child and its sibling will have a random seed value and tag bit while all others are zero. The challenge is to then compute t correction values $L_1, ..., L_t \in \{0,1\}^{\kappa \times 2 + t \times 2}$ such that the sibling at index $i \oplus 1$ is corrected to have $s^d_{i \oplus 1} = 0, \tau^d_{i \oplus 1} = 0$ while the active child i has its tag bit corrected to $\tau^d_i = 1$.

Observe that the parent level has t unit vectors $[\![\tau_1^{d-1}]\!], ..., [\![\tau_t^{d-1}]\!]$ which encode the active parents. We want to use these to obliviously select the child values for each. For the single point case of $t = 1$ there is a "trivial" method of just computing the sum of all left and right children. However, when $t > 1$ this approach does not work as we get only one sum as opposed of selecting t pairs of children. For example, with $t = 2$

$$\begin{aligned} \tau_1^{d-1} &:= 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0 \\ \tau_2^{d-1} &:= 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0 \\ s' &:= 000000ab0000cd00 \end{aligned}$$

The left sum would be $a + c$ while the right sum is $b + d$ but instead we want two pairs (a, b) and (c, d). Once we have these pairs we can compute the desired correction values using a constant sized MPC protocol.

To obtain these pairs, we compute the inner product of each unit vector $[\![\tau_i^{d-1}]\!] \in \{0,1\}^{2^{d-1}}$ with the weight-at-most-t vector $[\![s']\!]$, treating each sibling pair as one element. Boyle et al. [BGH+25] give a protocol called UnitSparseInnerProd for this task. This routine improves on the trivial linear-time solution only when $n \gg 2^{30}$. Figure 14 reproduces their approach. It computes the inner product one bit at a time, rather than operating directly on seed-and-tag elements in $\{0,1\}^{2\kappa+2t}$. The general case therefore uses $2\kappa + 2t$ calls to UnitSparseInnerProd, one for each bit of the seed and tag vectors.

The idea is to reduce the general inner product problem between unit vector $u := \tau_i^{d-1} \in \{0,1\}^m$ and weight at most t vector $v \in \{0,1\}^m$ into many tiny checks using randomized set partitions. Concretely, UnitSparseInnerProd repeatedly partitions the index set $[m]$ into disjoint blocks $P_{i,1}, \ldots, P_{i,T} \subset [m]$ such that each block covers a disjoint portion of the vector. For each block $P_{i,j}$, the parties compute the XOR of the bits of $[\![u]\!]$ restricted to $P_{i,j}$, denoted $[\![x_{i,j}]\!]$, and simultaneously apply a random linear map $R_{i,j}$ to the corresponding coordinates of $[\![v]\!]$ to obtain $[\![y_{i,j}]\!]$. Since $[\![u]\!]$ is a unit vector, then exactly one block will contain the non-zero position. The

For public parameters $\kappa, n, w \in \mathbb{N}$, group $\mathbb{G}$, PRG $\mathsf{G} : \{0,1\}^\kappa \to \{0,1\}^{2t+2\kappa}$, $D := \lceil \log_w(n) \rceil$.

bigstate-DPF: On input $[\![A]\!] \in [0,n)^t, [\![B]\!] \in \mathbb{G}^t$.
Party $p \in \{0,1\}$ computes:

1. $[\![\pi]\!] := \mathsf{sort}([\![A]\!]), [\![A]\!] := \mathsf{perm}([\![\pi]\!], [\![A]\!]), [\![B]\!] := \mathsf{perm}([\![\pi]\!], [\![B]\!])$.
2. $[\![s_0^0]\!] \leftarrow \{0,1\}^\kappa$
3. $[\![\tau_0^0]\!] := [\![(1) \mid\mid (0)^{t-1}]\!]$
4. for $d \in [1, D]$
   (a) $[\![s_i']\!]_p := \mathsf{G}([\![s_i^{(d-1)}]\!]_p)$ for $i \in [2^d]$
   (b) $[\![L_{k,\ell}]\!] := \mathsf{UnitSparseInnerProd}([\![\tau_{*,k}^{d-1}]\!], [\![s_{*,l}']\!])$ for $k \in [t], \ell \in [2t + 2\kappa]$
   (c) for $k \in [t]$:
      i. $([\![z_0]\!] \mid\mid [\![z_1]\!] \mid\mid [\![b_0]\!] \mid\mid [\![b_1]\!]) := L_k$ where $z_i \in \{0,1\}^\kappa, b_i \in \{0,1\}^t$
      ii. $[\![\sigma_k]\!] := \bowtie([\![z_{[\![\alpha_d]\!]\oplus 1}]\!] \cdot (1,1), ([\![b_{0,k}]\!] \oplus [\![\alpha_d]\!] \oplus 1, [\![b_{1,k}]\!] \oplus [\![\alpha_d]\!])) \in \{0,1\}^{2\times\kappa}$
      iii. for $i \in [0, 2^d)$: $[\![s_i^d]\!] || [\![\tau_i^d]\!] := [\![s_i']\!] \oplus [\![\tau_{i/2}^{d-1}]\!] \sigma_{k,i \mod 2}$
   (d) if $d = D$:
      i. convert the leaves to $\mathbb{G}$.
      ii. use another instance of $\mathsf{UnitSparseInnerProd}$ to compute the leaf correction values.
      iii. update the leaves.

UnitSparseInnerProd: On input $[\![u]\!] \in \{x \in \{0,1\}^m \mid |x| \le 1\}$ and $[\![v]\!] \in \{x \in \{0,1\}^m \mid |x| \le t\}$.
The parties $p \in \{0,1\}$ compute:

1. select $s, s' \in \mathbb{N}$ s.t. $(1 - (1 - 1/T)^{t-1})^s (1 - 2^{-s'})^s = \mathsf{negl}$ where $T := \lceil \log_2(e)t \rceil \approx 1.44t$
2. for $i \in [s]$, let $P_i := (P_{i,1}, ..., P_{i,T})$ where $P_{i,j} \leftarrow \mathsf{powerSet}([m])$ s.t. $P_{i,j} \cap P_{i,j'} = \emptyset$ for all $j \neq j'$ and $\bigcup_j P_{i,j} = [m]$
3. for $i \in [s], j \in [T]$:
   (a) $R_{i,j} \leftarrow \{0,1\}^{s \times |P_{i,j}|}$
   (b) $[\![x_{i,j}]\!] := \bigoplus_{k \in P_{i,j}} [\![u_k]\!]$
   (c) $[\![y_{i,j}]\!] := R_{i,j} \cdot [\![v_{P_{i,j}}]\!]$
   (d) $[\![c_{i,j}]\!] := ([\![y_{i,j}]\!] = 0) \wedge [\![x_{i.j}]\!]$
4. output $\neg \bigvee_{i \in [s], j \in [t]} [\![c_{i,j}]\!]$

Figure 14: Big-state distributed multi-point function (DMPF) protocol [BGH+25], before substituting our optimized $\mathsf{UnitSparseInnerProd}$ protocol.

check $[\![c_{i,j}]\!] := ([\![y_{i,j}]\!] = 0) \wedge [\![x_{i,j}]\!]$ ensures that the block containing the non-zero entry of $[\![u]\!]$ also contains the matching entry of $[\![v]\!]$. By repeating this randomized partitioning s times with fresh $R_{i,j}$ matrices, the protocol guarantees with high probability that the true inner product is preserved while false matches are eliminated. Finally, the parties combine the results and output $\neg \bigvee_{i\in[s],j\in[T]} [\![c_{i,j}]\!]$ as the decision bit.

Intuitively, this procedure attempts to "hash" the weight-t vector $[\![v]\!]$ into small buckets and check consistency with the unit vector $[\![u]\!]$, without revealing the positions of the non-zeros. While information-theoretically sound, the construction requires $O(s \cdot T)$ sub-checks where $T \approx 1.44t$, and the randomized partitioning must be repeated $s = \Theta(\log n)$ times to achieve negligible error. As a result, UnitSparseInnerProd is only asymptotically better than a trivial linear-time inner product when $n \gg 2^{30}$, and in practice it is slower than simply performing the direct comparison. This inefficiency is the main obstacle preventing Boyle et al.'s big-state DMPF from being practically competitive in the fully distributed setting.

10.1 Improved UnitSparseInnerProd

We improve the basic idea of [BGH+25] with a protocol that operates directly on strings rather than one bit at a time. Here $u \in \{0,1\}^m$ is a *unit* vector, and $v \in \{0,1\}^{\sigma\times m}$ has at most t nonzero σ-bit entries. Given secret shares $[\![u]\!], [\![v]\!]$, we compute $[\![\langle u, v\rangle]\!]$ using only: (i) applying public linear maps to $[\![u]\!]$ and $[\![v]\!]$; and (ii) $O(\mathrm{poly}(t))$ nonlinear ops.

Fix constants $c > 2$ and let $\varepsilon := 2^{-\lambda}$ be the statistical security error. Let $B := \lceil ct \rceil$ be the bucket size and $R := \Theta(\lambda)$ be the repetition factor. For each repetition $r \in [R]$, sample a public (pairwise-independent) hash $h_r : [m] \to [B]$ which maps positions into buckets. Let $H_r \in \{0,1\}^{B\times m}$ be the bucket matrix with $(H_r)_{b,k} = 1$ iff $h_r(k) = b$. We compress the input vectors as:

$$[\![U_r]\!] := H_r \cdot [\![u]\!] \in \{0,1\}^B, \qquad [\![V_r]\!] := H_r \cdot [\![v]\!] \in \{0,1\}^{\sigma\times B}.$$

These are just public matrix-vector multiplies (bucketed sums).

For each $r \in [R]$, compute the scalar

$$[\![y_r]\!] := [\![U_r]\!]^\top [\![V_r]\!] = \sum_{b\in[B]} [\![U_{r,b}]\!] \cdot [\![V_{r,b}]\!],$$

which uses $B = O(t)$ secret-shared multiplications (nonlinear) per repetition, requiring a total of $2RB$ OTs. Return the final answer as the bitwise majority of $\{[\![y_r]\!]\}_{r\in[R]}$. This can be computed via a $R\sigma$ bit injection on $[\![y]\!]$ and the summing each bit position to compute the frequency. Finally, it must be compared with the threshold requiring σ comparisons of R length strings. In total this requires approximately $2RB + 4R\sigma$ OTs.

Why this works. Let $j^\star$ be the (unknown) index with $u_{j^\star} = 1$. Then U_r is one-hot at bucket $b^\star := h_r(j^\star)$, so

$$y_r = (U_r)^\top (V_r) = (V_r)_{b^\star} = \bigoplus_{k\in h_r^{-1}(b^\star)} v_k \in \mathbb{F}_2^\sigma.$$

If no support element of v collides with $j^\star$ in repetition r, then $y_r = v_{j^\star} = \langle u, v\rangle$. Each repetition has $\Pr[\text{collision at } b^\star] \le (t-1)/B \le 1/c < 1/2$. Thus in expectation *most* repetitions are collision-free. By Chernoff, with $R = \Theta(\lambda)$, the bitwise majority of $\{y_r\}$ equals $v_{j^\star}$ with probability at least

$1-\varepsilon$. All maps from $[\![u]\!]$ and $[\![v]\!]$ to $[\![U_r]\!], [\![V_r]\!]$ are *linear*, their size is $B = O(t)$, and the only nonlinear work is $O(R \cdot B) = O(t\lambda)$ secret multiplications plus $O(R \cdot \sigma)$ bitwise majorities (still $O(\text{poly}(t))$).

10.2 Sketch with Detection

We now show that this can be further optimized such that $c > 1$ by adding a mechanism to detect if a given bucket has exactly 1 non-zero element of v in it. As before $B := \lceil ct \rceil$. Pick R independent public hash functions $h_r : [m] \to [B]$ mapping indices to bins.

For fingerprints, pick public pairwise-independent matrices $\psi_r(k) \in \{0,1\}^{\rho \times \sigma}$ for each $k \in [m]$. For each repitions $r \in [R]$ and bucket $b \in [B]$ we derive values:

$$[\![U_b^r]\!] := \bigoplus_{k \in h_r^{-1}(b)} [\![u_k]\!] \in \{0,1\},$$
$$[\![V_b^r]\!] := \bigoplus_{k \in h_r^{-1}(b)} [\![v_k]\!] \in \{0,1\}^\sigma,$$
$$[\![F_b^r]\!] := \bigoplus_{k \in h_r^{-1}(b)} \psi_r(k)[\![v_k]\!] \in \{0,1\}^\rho.$$

As before, $U^r \in \{0,1\}^B$ is a unit vector. This repetitions is "good" if the bucket that contains the one from u also contains at most 1 non-zero from v. We additionally compute the fingerprint $F_b^r \in \{0,1\}^\rho$ of the same subset. Each term in the fingerprint summation is randomized using $\psi_r(k)$. Let $j^\star \in [m]$ denote the (unknown) position where $u_{j^\star} = 1$ and let $b^\star := h_r(j^\star)$. We now compute the inner products between

$$[\![V_\star^r]\!] := \langle [\![U^r]\!], [\![V^r]\!] \rangle \in \{0,1\}^\sigma,$$
$$[\![F_\star^r]\!] := \langle [\![U^r]\!], [\![F^r]\!] \rangle \in \{0,1\}^\rho.$$

These terms can all be computed using just $2RB$ OTs, two for each bit of U. To detect if the given bucket is good we muct compute the *expected* fingerprint matrix from $[\![u]\!]$:

$$[\![\Psi^r]\!] := \bigoplus_{k \in [m]} [\![u_k]\!]\psi_r(k) \in \{0,1\}^{\rho \times \sigma}.$$

This summation is over matrices. Then compute $[\![\widehat{F}^r]\!] := [\![\Psi^r]\!] \cdot [\![V_\star^r]\!] \in \{0,1\}^\rho$ using σ OTs. Finally perform a single equality test

$$[\![s^r]\!] := \left([\![F_\star^r]\!] = [\![\widehat{F}^r]\!]\right).$$

If $s^r = 1$, the bucket is (with overwhelming probability) a singleton containing $v_{j^\star}$, so we accept $[\![y_r]\!] := [\![V_\star^r]\!]$ as the output for this repetition.

Aggregation. Output the first accepted candidate. This can be done using a binary tree over the $[\![s]\!]$ vector to obtain the unit vector $[\![s]\!]$ with 1 at the first non-zero position. The final output is then computed as

Failure probability. Let $q := \Pr[\text{collision in one repetition}]$; with pairwise-independent h_r and $B = \lceil ct \rceil$,

$$q = 1 - \left(1 - \tfrac{1}{B}\right)^{t-1} \le \frac{t-1}{B} \le \frac{1}{c}.$$

A run fails if *all* repetitions collide (no singleton) or we accept on a collided bucket due to a fingerprint coincidence. By a union bound,

$$\Pr[\text{fail}] \le \underbrace{q^R}_{\text{all collide}} + \underbrace{R \cdot 2^{-\rho}}_{\text{some collision}s\text{passes fingerprint}} \le c^{-R} + R \cdot 2^{-\rho}.$$

In particular, if you set $\epsilon = 2^{-\lambda}$, it suffices to choose

$$R \ge \frac{\lambda + 1}{\log_2 c} \qquad \text{and} \qquad \rho \ge \lambda + \lceil \log_2(2R) \rceil$$

so that $\Pr[\text{fail}] \le 2^{-\lambda}$.

Failure probability. With $B = \lceil ct \rceil$, the per-repetition collision probability satisfies $q = 1 - (1 - 1/B)^{t-1} \le 1/c$. Over R independent repetitions, $\Pr[\text{fail}] \le c^{-R} + R\,2^{-\rho}$; e.g., choose $R \ge (\lambda + 1)/\log_2 c$, $\rho \ge \lambda + \lceil \log_2(2R) \rceil$. Works for any $c > 1$.

Cost. Per repetition: (i) linear sketching into $B = O(t)$ buckets; (ii) B masked multiplications to form $[\![V^r_\star]\!]$ plus B to form $[\![B^r_\star]\!]$; (iii) $\rho \cdot \sigma$ bit-mults to compute $[\![\widehat{B}^r]\!]$; and (iv) one ρ-bit equality. Total nonlinear work is $O\big(R(B + \rho\sigma)\big) = O\big(tR + \rho\sigma R\big)$; since $R = \Theta(\lambda)$ and $\rho = \lambda + O(\log \lambda)$, this is $O\big(t\lambda + \sigma\lambda^2\big)$, with all preprocessing linear and independent of m.

Comparison with majority-only aggregator. Without fingerprints, set $B = \lceil ct \rceil$ with $c > 2$ so $p = 1 - q > 1/2$ and take $R \ge \ln(1/\epsilon)/(2(p - \frac{1}{2})^2)$ to get $\Pr[\text{fail}] \le e^{-2(p-\frac{1}{2})^2 R}$. Costs are $2RB$ OTs of length σ plus an $O(R\sigma)$ majority. With fingerprints, the bound becomes $\Pr[\text{fail}] \le c^{-R} + R\,2^{-\rho}$, works for any $c > 1$, and per repetition uses $2B$ OTs of length $(\sigma + \rho)$ and 2σ OTs of length ρ (plus one ρ-bit equality), often enabling smaller B and lower total work for the same $\epsilon = 2^{-\lambda}$.

Comparison with the Original UnitSparseInnerProd Per repetition $i \in [s]$ and block $j \in [T]$, compute $[\![x_{i,j}]\!] \in \{0,1\}$, $[\![y_{i,j}]\!] \in \{0,1\}^s$, then test ($[\![y_{i,j}]\!] = 0^s$) and AND with $[\![x_{i,j}]\!]$. A zero-test on an s-bit vector costs $O(s)$ nonlinear ops, this is $O(\sigma s)$ per (i,j). Across all sT pairs:

$$\#\ \text{OTs} = O(\sigma\, s^2\, T) \quad \text{with} \quad T = \lceil \log_2(e)\, t \rceil \approx 1.44\, t, s = \Theta(\log m),$$

i.e., $O\big(\sigma\, t\, (\log m)^2\big)$. This is constrasted with our which is just $O(\sigma t)$.

Despite our improvements the concrete cost of this approach remains high as this proceedure must be run on each level of the tree.

10.3 Non-blackbox Bigstate Keygen

Idea. Let $\mathsf{KeyGen}_{\mathrm{BS}}(A, B; r)$ denote the *trusted-dealer* big-state key generation algorithm of [BGH+25], which on inputs $A \in [0, N)^t$, $B \in \mathbb{G}^t$ and randomness r outputs two evaluation keys (K^0, K^1). We realize a fully distributed key generation by evaluating this function *inside MPC*: the parties hold secret shares $[\![A]\!], [\![B]\!]$, run a generic MPC for the circuit of $\mathsf{KeyGen}_{\mathrm{BS}}$, and at the end *selectively output* K^p to party $p \in \{0, 1\}$. All steps of the dealer algorithm are preserved; we only swap the PRG primitive for an MPC-friendly instantiation.

Protocol (high-level, inside MPC).

1. **Sort & permute** the t indices (as in dealer): $[\![\pi]\!] := \mathsf{sort}([\![A]\!])$, then $[\![A]\!] := \mathsf{perm}([\![\pi]\!], [\![A]\!])$, $[\![B]\!] := \mathsf{perm}([\![\pi]\!], [\![B]\!])$.
2. **Initialize root**: sample $[\![s_0^0]\!] \leftarrow \{0, 1\}^\kappa$; set the tag vector $[\![\tau_0^0]\!] := [\![(1) \mid\mid (0)^{t-1}]\!]$.
3. **Decompose** $[\![A]\!]$ into binary bits per depth: $([\![\alpha_D]\!], \ldots, [\![\alpha_1]\!]) := \mathsf{decomposition}_2([\![A]\!]) \in \{0, 1\}^{t \times D}$.
4. **For each depth** $d = 1, \ldots, D$**:**
   (a) **PRG expansion (AM–PRF).** For each nonzero parent seed $[\![s_i^{d-1}]\!]$ (the big-state invariant ensures at most t per level), compute
   $$([\![s'_{2i}]\!], [\![s'_{2i+1}]\!], [\![b'_{1,i}]\!], [\![b'_{2,i}]\!]) \;:=\; \mathsf{AMPRG}([\![s_i^{d-1}]\!]),$$
   yielding two κ-bit child seeds and two t-bit tag masks (all as shares).
   (b) **Dealer-style corrections.** Using *dealer logic* (no $\mathsf{UnitSparseInnerProd}$), compute the per-$k$ correction word for this level: parse the PRG output to $([\![z_1]\!], [\![z_2]\!], [\![b_1]\!], [\![b_2]\!])$ and set
   $$[\![\sigma_{k,d}]\!] := \bowtie\Big([\![z_{\,[\![\alpha_d]\!][k] \oplus 1}]\!] \cdot (1, 1),\ ([\![b_{1,k}]\!] \oplus [\![\alpha_d]\!][k] \oplus 1,\ [\![b_{2,k}]\!] \oplus [\![\alpha_d]\!][k])\Big),$$
   exactly as in the trusted-dealer spec of [BGH+25].
   (c) **Apply corrections & push tags/seeds** to obtain $[\![s_*^d]\!]$ and $[\![\tau_*^d]\!]$ (one child remains active per k).
5. **Leaves & payload injection.** Compute per-leaf masks from $\mathsf{MSBs}([\![s_i^D]\!])$ and inject $[\![B_k]\!]$ via $\mathsf{updateLeaves}$ t times, as in the dealer.
6. **Selective outputs.** Reveal to party p exactly the components that define its evaluation key K^p (root seed share, per-depth $\sigma_{k,d}$ words, final balancing term), while keeping the other party's key hidden.

Security. This is a straight secure evaluation of the dealer program. In the semi-honest (resp. malicious) model, security reduces to: (i) the PRF security of AM–PRF (as a replacement for G), and (ii) the security of the underlying MPC compiler. Thus the resulting key distribution is computationally indistinguishable from the trusted-dealer output of [BGH+25].

Complexity. Let $D = \lceil \log_2 n \rceil$.

- **PRG cost.** By the big-state invariant, at most t parents are active per depth. We perform $O(t)$ PRG calls per level, hence $O(tD)$ total PRG calls. If an AM–PRF round uses $c_{\text{mul}} = O(\kappa)$ multiplications, $R = O(1)$ rounds per block, and we need $\lceil (2\kappa + 2t)/w \rceil$ blocks per call, the arithmetic-multiplication cost is

$$O\Big(tD \cdot R \cdot c_{\text{mul}} \cdot \Big\lceil \frac{2\kappa + 2t}{w} \Big\rceil\Big),$$

  plus linear operations.
- **Sorting $[\![A]\!]$.** Sorting t indices with a data-oblivious network costs $O(t \log^2 t)$ comparisons (Boolean multiplications). Or can be implemented using a specialized protocol with $O(t \log t)$ time[PRRS25, CHI+19].
- **Corrections/updates.** Per depth, $O(t(\kappa + t))$ linear ops and $O(t)$ bit-mults (for parity/lsb gates) suffice; total $O(tD)$ nonlinear gates beyond the PRG.
- **No UnitSparseInnerProd.** We *do not* execute the distributed inner product routine; all "dealer arithmetic" (select the active child per k using $\alpha_d[k]$) happens *inside* the MPC in the same way the dealer would.

Discussion. This "non-blackbox" path is attractive when: (i) you already have a fast 2PC/MPC stack for word operations; (ii) t is in the practical big-state regime (say 3–70) so $O(t \log n)$ AM–PRF calls dominate; and (iii) you prefer engineering simplicity over custom distributed sub-protocols. Compared to [BGH+25]'s distributed route (which incurs an n-dependent UnitSparseInnerProd), this approach eliminates the n dependence and replaces it with PRF rounds over words. We believe that this approach could be competitive with our reverse cuckoo hashing approach. Overall fewer PRG calls will be made but these PRG calls will be more expensive and require MPC evaluation of the PRG for every expansion.

11 Application to PCGs: Ring-LPN and Stationary LPN

PCGs in a nutshell. A pseudorandom correlation generator (PCG) uses short seeds to generate a long sample from a target two-party correlation $\mathcal{C}$. Its interfaces are

$$\mathsf{PCG.Gen}(1^\lambda) \longrightarrow (k_0, k_1), \qquad \mathsf{PCG.Expand}(p, k_p) \longrightarrow R_p, \quad p \in \{0, 1\}.$$

Expansion is local and communication-free. Security concerns a semi-honest corrupt party that knows its seed, not an observer given only the outputs [BCG+20, Defs. 2.5–2.6].

We describe an interactive correlation-generation protocol built from distributed sparse products and local ring arithmetic. Its reusable setup fixes the sparse supports and initializes the DMPFs. Each subsequent batch refreshes the coefficients and public masks, invokes the DMPF expansions, and computes the OLE outputs. Unlike PCG.Expand, these batches communicate; reuse amortizes only the address-dependent setup. Security requires the joint stationary assumption below and the distributed subprotocol guarantees.

The DMPF functionality guarantees a private sharing of the sparse product, but does not require independently uniform output shares. The application must separately establish the target correlation distribution and security given a corrupt party's view.

11.1 Ring-LPN overview and its use in PCGs

Let $\mathbb{R} := \mathbb{F}_q[X]/(X^{N_\mathsf{R}} + 1)$. For the NTT-based application, assume $2N_\mathsf{R} \mid (q-1)$. The negacyclic NTT then maps ring multiplication to coordinate-wise multiplication over $\mathbb{F}_q$. For a specified sparse-noise distribution $\mathcal{D}$ on $\mathbb{R}$, decisional Ring-LPN assumes

$$(r, re + f) \approx_c (r, u), \qquad r, u \leftarrow \mathbb{R}, \quad e, f \leftarrow \mathcal{D}.$$

The samples on the right are independent. The noise distribution and parameter family are part of the assumption [BCG+20].

From sparse products to OLEs. Party $p \in \{0,1\}$ samples $e_{p,0}, e_{p,1} \leftarrow \mathcal{D}$. The parties sample a public $r \leftarrow \mathbb{R}$ and define $x'_p := e_{p,0} + re_{p,1}$. They need shares of

$$\begin{aligned} y' &:= x'_0 x'_1 \\ &= e_{0,0}e_{1,0} + r(e_{0,0}e_{1,1} + e_{0,1}e_{1,0}) + r^2 e_{0,1}e_{1,1}. \end{aligned}$$

Given those shares, local NTTs give $x_p := \mathsf{NTT}(x'_p)$ and $[\![y]\!] := \mathsf{NTT}([\![y']\!])$ satisfying $x_0 \odot x_1 = [\![y]\!]_0 + [\![y]\!]_1$. This is the OLE correlation used by the PCG. Two independent OLE correlations yield one Beaver triple by local conversion.

Write $A_{p,i} \in [0, N_\mathsf{R})^t$ and $B_{p,i} \in \mathbb{F}_q^t$ for a sparse polynomial's positions and coefficients. For equally long lists A, B, define $\mathsf{makePoly}(A, B) := \sum_j B_j X^{A_j}$. Cartesian sums of positions and products of coefficients give

$$A'_{i,j} := A_{0,i} \boxplus A_{1,j}, \qquad B'_{i,j} := B_{0,i} \otimes B_{1,j}.$$

Both are lists of length t^2; repeated positions contribute their coefficient sum. Their polynomial reduces to $e_{0,i}e_{1,j}$ modulo $X^{N_\mathsf{R}} + 1$. A DMPF over $[0, 2N_\mathsf{R})$ directly constructs its shared coefficient map. Public ring operations then form the weighted sum above.

More than two noise polynomials. For a public polynomial count $P \geq 2$, put $r_0 := 1$ and sample $r_i \leftarrow \mathbb{R}$ for $i \in [1, P)$. Both parties use this same mask vector. The two-polynomial example generalizes to

$$x'_p := \sum_{i \in [0,P)} r_i e_{p,i}, \qquad y' := \sum_{i,j \in [0,P)} r_i r_j e_{0,i} e_{1,j}.$$

All P noise polynomials enter one ring sample per party. They are not split into independent two-polynomial instances.

Regular product lists. For regular noise, let $t \mid N_\mathsf{R}$ and $L := N_\mathsf{R}/t$. Each party holds P sparse polynomials, with one selected position in each of t blocks. For $p \in \{0,1\}$, $a \in [0, P)$, and $j \in [0, t)$, sample an offset $X_{p,a,j} \leftarrow [0, L)$. Its absolute ring position is $jL + X_{p,a,j}$. Sample each coefficient independently from $\mathbb{F}_q$, including zero. We call the selected position sets the supports; the actual nonzero supports can be smaller. For a polynomial pair (a, b) and $k \in [0, t)$, define the ordered list

$$A_{a,b,k} := \left(X_{0,a,j} + X_{1,b,(k-j) \bmod t}\right)_{j \in [0,t)} \in [0, 2L)^t. \tag{23}$$

For row j, multiply the two coefficients and the sign

$$(-1)^{\lfloor (j+((k-j) \bmod t))/t \rfloor}.$$

This sign accounts for negacyclic wraparound of the block positions. After DMPF expansion, multiplication by X^{kL} and reduction modulo $X^{N_{\mathsf{R}}}+1$ also handle offsets that cross a block boundary. Summing over k gives the polynomial product.

Thus all P^2 cross-party polynomial products use $P^2 t$ DMPF instances, each with at most t real points and domain length $2L$. Regularity reorganizes the product terms; it does not reduce their number. Our implementation uses $(P, t) = (4, 16)$. For a wN DMPF expansion without a stash, the total leaf work is $2wP^2 N_{\mathsf{R}}$, rather than the $2P^2 t N_{\mathsf{R}}$ work of a sum of t DPFs on each product list.

11.2 Stationary LPN and repeated expansion

Stationary noise retains the supports and refreshes their coefficients. The stationary-decoding literature motivates this reuse [KPRR25]; the ring application requires a joint decisional assumption for the samples it actually exposes.

Fix a polynomial expansion bound $Q(\kappa)$. Sample the supports once from the prescribed distribution. At expansion $u \in [0, Q)$, sample independent coefficients from $\mathbb{F}_q$ to form $e_{p,i}^{(u)}$. Put $r_0^{(u)} := 1$ and independently sample $r_i^{(u)} \leftarrow \mathbb{R}$ for $i \in [1, P)$. Write $\mathbf{r}^{(u)} := (r_i^{(u)})_{i \in [0,P)}$. For an honest party, the required statement is

$$\left(\mathbf{r}^{(u)}, \sum_{i \in [0,P)} r_i^{(u)} e_{p,i}^{(u)} \right)_{u \in [0,Q)} \approx_c \left(\mathbf{r}^{(u)}, U^{(u)} \right)_{u \in [0,Q)}, \qquad U^{(u)} \leftarrow \mathbb{R}.$$

The replacements are independent across expansions and of the masks. Marginal pseudorandomness for each expansion is not sufficient. For multiple OLE instances, the assumption concerns their full joint sequence as well. The Reverse-Cuckoo application additionally exposes descriptors, as formalized in Assumption 12.3.

The product-address lists $A'_{i,j}$ remain fixed. Setup initializes their reusable DMPFs. Each expansion forms new coefficient products $B'^{(u)}_{i,j}$, expands the cached DMPFs, and computes

$$y'^{(u)} := \sum_{i,j \in [0,P)} r_i^{(u)} r_j^{(u)} \mathsf{makePoly}(A'_{i,j}, B'^{(u)}_{i,j}) \pmod{X^{N_{\mathsf{R}}} + 1}.$$

The NTT yields the new OLE batch. The same public mask vector is used for both parties within an expansion. It is freshly sampled across expansions in this model.

Sampling and applying the masks. The NTT is a bijection. Sampling independent field elements in its output coordinates therefore samples a uniform ring mask without an additional NTT. At each expansion, the implementation regenerates those coordinates and their pairwise products, retaining the allocated storage. For P masks with $r_0 = 1$, the current routine samples $(P-1)N_{\mathsf{R}}$ field elements and computes $P^2 N_{\mathsf{R}}$ coordinate products per party. This work does not change the supports or repeat the DMPF setup.

Reuse removes repeated address-dependent setup, not the linear leaf work. Each expansion still converts the cached leaf seeds, exchanges payload corrections, and performs ring arithmetic. The generic DMPF argument and the stationary Ring-LPN assumption are separate: the former hides arbitrary valid input indices through its ideal interface; the latter justifies the pseudorandomness of the resulting correlation sequence.

11.3 Impact on Related PCG Constructions

Sparse encodings also appear in programmable PCGs for Boolean circuits, arbitrary finite fields, and Galois rings [BBC+24, BCCD23, LXYY25a, LXYY25b]. Waterfall is a candidate replacement when a construction needs the multi-point sharing functionality defined here. The benefit depends on the domain sizes, support distribution, and cost of the surrounding arithmetic.

Such a transfer requires a separate argument for stationary reuse and, if Reverse Cuckoo is used, descriptor leakage. The assumption analyzed here is specific to the stated Ring-LPN distribution. We have not measured end-to-end gains for these other applications.

12 Reverse Cuckoo for Ring-LPN

Waterfall Cuckoo is our generic DMPF construction. This section gives a smaller, application-specific alternative for the random sparse supports in our Ring-LPN PCG. Reverse Cuckoo reduces the dominant local expansion from $3N$ or $4N$ to $2N$: instead of sampling hashes and then finding a placement, it samples a hidden placement and programs public hashes around it. The price is support leakage. We first describe the $K = 1$ construction, then derive its output distribution in the ideal hash model. The concrete evaluator requires a separate correctness analysis and leakage assumption.

Here K counts independently generated placement-first proposals passed to an optional public debiasing rule; $K = 1$ simply publishes the sole proposal. We lead with $K = 1$ because it needs no proposal generation or comparison beyond the single published descriptor tuple. Larger K adds setup work to reduce the idealized leakage, but does not reduce the $2N$ expansion cost. The attack analysis therefore starts with the unmitigated $K = 1$ channel.

12.1 Concrete feature evaluator

Reverse Cuckoo must program a public hash after the input rows have been shuffled. We use a degree-two feature evaluator whose only nonlinear layer is one batch of bit multiplications. Let $n := \lceil \log_2(N+1) \rceil$, let $\mathsf{bin}_N : [0, N] \to \mathbb{F}_2^n$ be the canonical encoding, and write $\nu := \mathsf{bin}_N(N)$. A seed expands to independent uniform matrices

$$A_0, A_1 \in \mathbb{F}_2^{n \times q'}, \qquad A_2 \in \mathbb{F}_2^{(n+q') \times q}.$$

For $x \in [0, N]$, define

$$a(x) := \mathsf{bin}_N(x) A_0, \qquad b(x) := \mathsf{bin}_N(x) A_1, \qquad c(x) := a(x) \odot b(x),$$

where $\odot$ is coordinate-wise multiplication in $\mathbb{F}_2^{q'}$. The anchored feature vector and evaluator are

$$\Phi(x) := \big(\mathsf{bin}_N(x) \oplus \nu,\, c(x) \oplus c(N)\big), \qquad F(x) := \Phi(x) A_2. \tag{24}$$

Thus $F(N) = 0$. Each partition uses an independent seed, sampled jointly after the shuffled rows are fixed. Under the ideal-cipher model for the seed expander, the three matrices are uniform.

The two matrix products defining a and b, the anchoring, and the final compression are linear over $\mathbb{F}_2$. Evaluating one row therefore requires only the q' secure bit products in $a(x) \odot b(x)$. The implementation sets

$$q' = q = d + \lambda + 8, \tag{25}$$

up to byte alignment. The fixed eight-bit margin is part of this evaluator; it does not change the independently chosen statistical parameter λ. The exact reason for the margin is given after the protocol.

12.2 The $K = 1$ construction

Fix a domain $[0, N)$, a maximum support size t, two partitions of size d, and $m := 2d$. We take d to be a power of two and write $\ell := \log_2 d$. Setup receives $[\![A]\!] \in \mathcal{X}_N^t$, and each expansion receives $[\![B]\!] \in \mathbb{G}^t$. As elsewhere, the public symbol $N \notin [0, N)$ denotes a dummy address and has payload zero.

For each partition $s \in [0, 2)$, instantiate Equation (24) with an independent seed, and denote the resulting evaluator by $F_s : [0, N] \to \mathbb{F}_2^q$. Let $\mathsf{enc}_d : [0, d) \to \mathbb{F}_2^\ell$ be binary encoding and let dec_d denote its inverse. A public descriptor $R_s \in \mathbb{F}_2^{q \times \ell}$ defines the hash

$$h_s(x) := \mathsf{dec}_d\big(F_s(x) R_s\big) \in [0, d). \tag{26}$$

Place first. Setup deduplicates the addresses, appends $m - t$ copies of N, and applies a uniform secret shuffle. Write $\bar{A}_r$ for row $r \in [0, m)$ after the shuffle. The protocol retains the shuffle and applies it to each later deduplicated payload vector. Write B'_r for the corresponding shuffled payload. Row $r = sd + b$, for $s \in [0, 2)$ and $b \in [0, d)$, is interpreted as the physical bin (s, b). Since the dummy rows are indistinguishable and the shuffle is uniform, the real rows occupy a uniform injection into the m bins.

Program one partition at a time. For partition s, form the secret matrix and public target

$$M_s := \begin{bmatrix} F_s(\bar{A}_{sd}) \\ \vdots \\ F_s(\bar{A}_{sd+d-1}) \end{bmatrix} \in \mathbb{F}_2^{d \times q}, \qquad Y_d := \begin{bmatrix} \mathsf{enc}_d(0) \\ \vdots \\ \mathsf{enc}_d(d-1) \end{bmatrix} \in \mathbb{F}_2^{d \times \ell}.$$

Invoke ℓ parallel instances of bin-solver to sample $[\![R]\!]_s$ satisfying the independent rows of

$$M_s R_s = Y_d. \tag{27}$$

The solver natively skips a row that reduces to zero, regardless of its right-hand side. Hence a dummy row, for which $F_s(N) = 0$, imposes no constraint. Reveal R_0, R_1 after both systems have been solved.

If a real record $x = \bar{A}_{sd+b}$ occupies row $sd + b$, then its equation is $F_s(x) R_s = \mathsf{enc}_d(b)$ and therefore $h_s(x) = b$. Thus its shuffled row is a valid cuckoo position under the revealed hashes.

Build the sparse columns. The public position map $P \in \{0,1\}^{[0,N)\times m}$ is

$$P_{x,sd+h_s(x)} = 1 \quad \text{for every } x \in [0,N),\ s \in [0,2), \tag{28}$$

with all other entries zero. Column $j \in [0,m)$ has sparse domain $S_j := \{x \in [0,N) : P_{x,j} = 1\}$. Setup replaces each dummy in column j by a public representative of S_j and initializes one reusable sparse DPF over S_j. The corresponding payload is always zero. Expanding all columns and adding them at their original addresses yields $\sum_{i\in[0,t)} \mathsf{unitVector}_{[0,N)}(A_i, B_i)$.

The complete construction is summarized below. Every invoked operation is defined earlier: eq, a standard reusable oblivious shuffle, $\mathsf{bin\text{-}solver}$, and the reusable sparse DPF.

For public parameters $\kappa, N, t, d, q, q' \in \mathbb{N}$ and a finite additive group $\mathbb{G}$, let $m := 2d$.
Party $p \in \{0,1\}$ stores its local component of $\mathsf{st}_{\mathsf{RC}} = (\llbracket\delta\rrbracket, \llbracket e\rrbracket, R, P, S)$, initially undefined. The instance owns a named reusable oblivious shuffle Shuf on m rows and named reusable DPF instances DPF^j over S_j whenever $S_j \neq \emptyset$.

$\mathsf{ReverseCuckoo}_1.\mathsf{Setup}(\llbracket A\rrbracket \in \mathcal{X}_N^t) \longrightarrow \mathsf{ready}$.

Party $p \in \{0,1\}$ computes:

1. If setup has already completed, return $\perp$.
2. For $i \in [0,t)$, set $\llbracket e\rrbracket_{i,i} := 1$. For $i < j$, set $\llbracket e\rrbracket_{i,j} := \mathsf{eq}(\llbracket A\rrbracket_i, \llbracket A\rrbracket_j)$ and $\llbracket e\rrbracket_{j,i} := \llbracket e\rrbracket_{i,j}$.
3. For $i \in [0,t)$, set $\llbracket\delta\rrbracket_i := (1 - \mathsf{eq}(\llbracket A\rrbracket_i, N)) \bigwedge_{j<i}(1 - \llbracket e\rrbracket_{i,j})$ and $\llbracket A\rrbracket_i^* := N \oplus \llbracket\delta\rrbracket_i(N \oplus \llbracket A\rrbracket_i)$.
4. Append $m-t$ copies of $(N,0)$ to $(\llbracket A\rrbracket^*, \llbracket\delta\rrbracket)$ and set $(\llbracket\bar{A}\rrbracket, \llbracket\chi\rrbracket) := \mathsf{Shuf.Setup}(\llbracket A\rrbracket^* \| N^{m-t}, \llbracket\delta\rrbracket \| 0^{m-t})$.
5. Jointly sample and reveal independent evaluator seeds for $s \in [0,2)$, and instantiate F_s by Equation (24).
6. For $s \in [0,2)$, form $\llbracket M\rrbracket_s[b,*] := F_s(\llbracket\bar{A}\rrbracket_{sd+b})$ for every $b \in [0,d)$.
7. For $s \in [0,2)$, compute $\llbracket R\rrbracket_s \leftarrow \mathsf{bin\text{-}solver}(\llbracket M\rrbracket_s, Y_d)$.
8. Reveal R_0, R_1 and define h_s by Equation (26).
9. Define P by Equation (28) and $S_j := \{x \in [0,N) : P_{x,j} = 1\}$ for every $j \in [0,m)$.
10. For $j \in [0,m)$, set $D_j := \min S_j$ if $S_j \neq \emptyset$ and $D_j := 0$ otherwise, and set $\llbracket A\rrbracket'_j := D_j \oplus \llbracket\chi\rrbracket_j(D_j \oplus \llbracket\bar{A}\rrbracket_j)$.
11. For $j \in [0,m)$ with $S_j \neq \emptyset$, invoke $\mathsf{DPF}^j.\mathsf{Setup}_{S_j}(\llbracket A\rrbracket'_j)$.
12. Store $\mathsf{st}_{\mathsf{RC}} := (\llbracket\delta\rrbracket, \llbracket e\rrbracket, R, P, S)$ and return ready.

$\mathsf{ReverseCuckoo}_1.\mathsf{Expand}(\llbracket B\rrbracket \in \mathbb{G}^t) \longrightarrow \llbracket y\rrbracket \in \mathbb{G}^{[0,N)}$.

Party $p \in \{0,1\}$ computes:

1. If setup has not completed, return $\perp$. Otherwise, read $\mathsf{st}_{\mathsf{RC}}$.
2. For $i \in [0,t)$, set $\llbracket B\rrbracket_i^* := \llbracket\delta\rrbracket_i \sum_{j\in[0,t)} \llbracket e\rrbracket_{i,j} \llbracket B\rrbracket_j$.
3. Set $\llbracket B\rrbracket' := \mathsf{Shuf.Expand}(\llbracket B\rrbracket^* \| 0^{m-t})$.
4. For $j \in [0,m)$ with $S_j \neq \emptyset$, set $\llbracket y\rrbracket^{(j)} := \mathsf{DPF}^j.\mathsf{Expand}_{S_j}(\llbracket B\rrbracket'_j)$.
5. For $x \in [0,N)$, set $\llbracket y\rrbracket(x) := \sum_{j:x\in S_j} \llbracket y\rrbracket^{(j)}(x)$.
6. Return $\llbracket y\rrbracket \in \mathbb{G}^{[0,N)}$.

Figure 15: Reusable $K = 1$ Reverse-Cuckoo specialization. Setup fixes the addresses, programmed hashes, shuffle, and DPF leaf seeds. Expansion supplies only a new payload vector.

12.3 Correctness of the programmed systems

Fresh matrices do not make the evaluator rows independent. Its degree-two structure instead permits an exact relation-by-relation analysis. Fix the real points $x_1, \ldots, x_r$ assigned to one partition. Stack their evaluator outputs and public row labels in

$$H_A := \begin{bmatrix} F(x_1) \\ \vdots \\ F(x_r) \end{bmatrix} \in \mathbb{F}_2^{r\times q}, \qquad Y \in \mathbb{F}_2^{r\times \ell}.$$

A nonzero $\alpha \in \mathbb{F}_2^r$ is harmful if $\alpha^\top H_A = 0$ but $\alpha^\top Y \neq 0$.

Lemma 12.1 (Survival of a fixed relation)**.** *Write* $\bar{x}_i := \mathsf{bin}_N(x_i)$ *and define*

$$L_\alpha := \bigoplus_i \alpha_i(\bar{x}_i \oplus \nu), \qquad Q_\alpha := \bigoplus_i \alpha_i \bar{x}_i^\top \bar{x}_i \oplus \left(\bigoplus_i \alpha_i\right) \nu^\top \nu.$$

If $L_\alpha \neq 0$*, then the output relation survives with probability* 2^{-q}*. If* $L_\alpha = 0$*, then* Q_α *is alternating and has even rank* ρ*. In this case the survival probability is exactly*

$$\left(\frac{1}{2} + 2^{-\rho-1}\right)^{q'} + \left(1 - \left(\frac{1}{2} + 2^{-\rho-1}\right)^{q'}\right) 2^{-q}. \tag{29}$$

Proof. The diagonal of Q_α is L_α. When $L_\alpha = 0$, one product coordinate of the α-sum equals $u^\top Q_\alpha v$ for independent uniform u, v and is zero with probability $1/2 + 2^{-\rho-1}$. All q' product coordinates vanish with the first term of Equation (29). Otherwise the lifted relation is nonzero, and uniform A_2 maps it to zero with probability 2^{-q}. If $L_\alpha \neq 0$, the lifted relation is always nonzero, giving 2^{-q}. □

For example, the four vertices of an affine plane have $\rho = 2$, so their relation survives the nonlinear lift with probability $(5/8)^{q'}$. The eight vertices of an affine three-cube have $\rho = 0$, and their degree-two relation survives every seed. These facts rule out the usual argument that the rows may simply be treated as independent uniform strings. They do not, however, determine the caller-level failure probability: all related points must first occur in the Ring-LPN batch, enter one partition, and receive labels with nonzero XOR.

We also account for a local support filter: each party resamples its four regular supports until their total number of occupied residues modulo 128 is at least 61. Residues are counted separately in each polynomial, before coefficient cancellation. The parties filter independently, before sampling the shuffle and evaluator seeds.

Proposition 12.2 (Ring-LPN batch correctness)**.** *Assume a uniform oblivious shuffle, uniform Ring-LPN block offsets, and ideal uniform matrices* A_0, A_1, A_2*. For the implementation-shaped batch with* $N_{\mathsf{R}} = 2^{20}$*,* $P = 4$*,* $t = 16$*,* $w = 2$*,* $d = 16$*, and* $q' = q = 64$*, the probability that any programmed system is inconsistent is less than* $2.379 \cdot 10^{-13} < 2^{-41.934}$*. If both parties instead apply the local support filter above, this probability is less than* $4.739 \cdot 10^{-13} < 2^{-40.940}$*.*

The bound counts every possible zero-linear-moment relation by weight and quadratic rank, then adds the ordinary rank loss in the final compression. Appendix B gives the counting argument. Filtering changes each polynomial-pair contribution by at most a factor of 2.044; the final-compression bound remains unchanged. At the former widths $q' = q = 56$, the corresponding bound was only $2^{-37.694}$. The fixed eight-bit margin in Equation (25) therefore restores more than 40 bits of batch correctness without repurposing λ. The protocol needs no explicit abort: a skipped inconsistent equation is included in this correctness error. This proposition is specific to the Ring-LPN caller; the AnyField distribution requires a separate analysis.

The dependence on N_{R} is material. At the same evaluator widths, our present proof gives only 34.213 bits for $N_{\mathsf{R}} = 2^{18}$ and 26.218 bits for $N_{\mathsf{R}} = 2^{16}$. Deterministic affine-cube relations dominate both bounds, so increasing q or q' alone cannot improve them. Accordingly, only the $N_{\mathsf{R}} = 2^{20}$ Ring-LPN row carries the stated 40-bit correctness claim. The smaller rows in the evaluation are performance scaling measurements.

Cost. Every address appears once in each of the two partition maps, so $\sum_j |S_j| = 2N$. After setup, local sparse-DPF expansion is therefore $2N$ leaf operations, independent of t. Setup performs one shuffle, two $d \times q$ binary solves with ℓ right-hand-side bits, sparse-set construction, and sparse-DPF generation. These are exactly the stages measured in Section 13.1.

12.4 What the descriptor reveals

The placement is hidden, but the public hashes are not distributed like independent hashes. The difference has a simple exact description.

Fix a set $A \subseteq [0, N)$ of t distinct addresses. In the ideal hash model, let U sample w independent uniform functions $h_s : [0, N) \to [0, d)$; the construction above uses $w = 2$. For a sampled hash tuple h, define

$$Z_A(h) := |\{\tau : A \hookrightarrow [0, w) \times [0, d) : \ \tau(x) = (s, b) \Longrightarrow h_s(x) = b\}| \,. \tag{30}$$

Thus $Z_A(h)$ is the number of injective cuckoo placements of A supported by h. There are $(wd)_t$ injections, and any fixed injection agrees with a uniform h with probability d^{-t}, so

$$\mathbb{E}_{h \leftarrow U}[Z_A(h)] = \frac{(wd)_t}{d^t}. \tag{31}$$

The $K = 1$ Reverse-Cuckoo experiment chooses a uniform injection τ first and then samples a uniform hash tuple consistent with τ. Summing over all hidden injections compatible with a fixed output h gives

$$\Pr[H = h \mid A]_{\mathsf{RC}_1} = \Pr[H = h]_U \, \frac{Z_A(h)}{\mathbb{E}_U[Z_A(H)]}. \tag{32}$$

This is the complete leakage channel in the ideal model: Reverse Cuckoo *size-biases* the ordinary hash distribution by its number of valid placements.

Equation (32) also isolates a tempting but incorrect simulation argument. Sampling U conditioned on $Z_A(H) > 0$ gives equal weight to every feasible hash tuple. Reverse Cuckoo instead gives a feasible tuple weight proportional to $Z_A(h)$. Hidden placement randomness is already summed into $Z_A(h)$; it does not remove the multiplicity bias.

The role of dummies. The $m - t$ dummies make the real placement a uniform injection into all m bins and provide slack. They impose no hash equation because $F_s(N) = 0$ and no output because their payload is zero. Increasing the slack smooths the placement-count distribution, but it does not change Equation (32): the descriptor is still weighted by $Z_A(h)$.

Graph intuition for two partitions. For $w = 2$, represent each address $x \in A$ by an edge from $h_0(x)$ in the left partition to $h_1(x)$ in the right partition. A valid placement orients every edge toward a distinct endpoint. A tree component with e edges contributes $e+1$ placements, a unicyclic component contributes two, and an overfull component contributes zero. The global value $Z_A(h)$ is the product of these component contributions. Pair collisions are useful because they change the component structure and generally reduce $Z_A(h)$, but the exact channel is a higher-order matching statistic, not a collection of independently leaked support bits.

Why this is not a generic DMPF. For arbitrary chosen inputs, Equation (32) depends on A. A party can test public collisions among candidate addresses and distinguish supports. Consequently Reverse Cuckoo does not realize the ordinary DMPF functionality for arbitrary inputs in the standard simulation model. It realizes a leakage-augmented functionality that reveals one sample from the channel in Equation (32). Waterfall Cuckoo exists precisely to avoid this qualification.

The binary feature evaluator adds algebraic structure beyond the ideal random-function model. Proposition 12.2 bounds the resulting consistency error for the Ring-LPN caller, but it does not turn the published descriptors into ideal independent hashes. Our security assumption below therefore includes the descriptors produced by the concrete evaluator. Equation (32) remains the clean idealized leakage model used by the attack study.

12.5 Why sparse Ring-LPN may tolerate the leakage

In the PCG, A is not an adversarial DMPF input. It has the form in Equation (23). Each party holds $P = 4$ regular support lists, and every list has one offset in each of $t = 16$ blocks. At the claimed $N_{\mathsf{R}} = 2^{20}$ point, each block has length $L = N_{\mathsf{R}}/t = 2^{16}$. Against a semi-honest corrupt party, that party's four lists are known and the honest party's $Pt = 64$ offsets remain hidden. Before the public Ring-LPN samples are used, those unknown offsets range over a space of size $L^{Pt} = 2^{1024}$.

Conditioning on a public Reverse-Cuckoo descriptor changes the support prior by

$$\Pr[A \mid H = h] \;\propto\; \Pr_{\mathsf{reg}}[A]\, Z_A(h). \tag{33}$$

It does not add candidate supports, turn the error dense, or reveal a linear equation in its positions. It reweights an already sparse family by a nonlinear matching count. Three features make this plausibly benign for LPN:

1. *Sparsity is preserved.* Every posterior candidate still has one position in each regular block. The decoder's basic combinatorial obstacle remains finding a tiny support inside a huge ambient ring.

2. *The leakage is geometric, not linear.* It describes collisions, Hall obstructions, and placement multiplicity among hash images. A linear test of the Ring-LPN syndrome cannot directly consume the statement "this support graph has more matchings."

3. *The coefficients remain hidden.* Reverse Cuckoo depends on positions, not on the independent uniform field coefficients. Even a correct partial support hypothesis must still satisfy the large-field Ring-LPN equations.

This is intuition, not a reduction. Sparse decoding is often robust to partial support information, and stationary syndrome decoding provides a particularly strong analogy for the failure of linear tests, but neither fact proves that the matching weights are harmless. A nonlinear decoder may bias its search toward high-likelihood supports. The relevant question is therefore concrete: does the leakage appreciably reduce the best known Ring-LPN decoding exponent?

In the stationary setting, the descriptor and support are reused while only the coefficients change. Repeated expansions do not reveal fresh independent placement samples, but an attacker may amortize arbitrary preprocessing of the single descriptor. Our assumption permits that preprocessing.

Attack or diagnostic	$K = 1$ conclusion
Linear tests, Fourier tests, and BKW	Fresh error coefficients cannot be cancelled directly. Leakage can only guide a dual word away from the posterior support; this is exactly a conditional dual-avoidance problem. The SSD theorem supplies the no-leakage analogy, but does not cover adaptive leakage-guided selection.
Stationary output-span and module tests	Reusing one descriptor does not create fresh placement leakage, but permits arbitrary preprocessing. Cross-expansion attacks remain outside the fixed linear-test argument; fresh independent ring masks are security-critical.
Likelihood-ordered support search and quadratic solving	Exact toy searches confirm that the channel is exploitable. A direct degree-two encoding at the production point has roughly $4N$ support variables, 131,072 placement variables, and about 10^{13} quadratic support monomials. This defeats naive dense XL, not structured solving.
Regular Prange and posterior-guided ISD	Pairwise, sequential collision-conditioned, and complete-matching selectors were tested. The best repeatable production estimate is $2^{127.123}$ rather than 2^{128}; complete-posterior sampling found no further gain, with a particle-degeneracy caveat.
Equation-coupled lists and blockwise beam search	Exact matching gives a clear advantage on reduced instances, including when coupled to syndrome equations. Full enumeration costs 2^{192} per polynomial at production parameters, while the bounded beam remains prohibitive for $L = 4096$; neither yields a production attack.
Syndrome projections and Wagner-style merging	The tested projected-bucket attack loses to the finer flat merge. Guessing intermediate projection shares repays the filtering gain in repetitions; more elaborate representation reuse remains unexcluded.
Regular Stern, MMT, and BJMM	Stern costs $2^{146.526}$; even an optimistic MMT estimate is at least $2^{145.55}$ after granting the measured leakage gain. The representation attacks remain dominated.
Sparse-factor statistical decoding	The best projected leakage weight is zero, and complete matching propagation gives no measured gain. The ordinary conservative estimate at mean folded weight is $2^{128.450}$; low-weight-tail rejection is nevertheless required.
Concrete feature and structured-rank audits	Reduced experiments detect neither stable leakage beyond ideal placement multiplicity nor a rank advantage for guided information sets. These are diagnostics, not a pseudorandomness or production-rank proof.

Table 2: Coverage of the Reverse-Cuckoo attack study. All numerical claims use the stronger $K = 1$ channel.

12.6 Concrete attack evidence for $K = 1$

We evaluated the channel primarily at the reduced point $(P, t, L) = (4, 16, 4096)$, for which $N_{\mathsf{R}} = tL = 2^{16}$. This point retains the implemented batching and Reverse-Cuckoo parameters while making the leakage-aware experiments tractable. It is not the full $N_{\mathsf{R}} = 2^{20}$ correctness point above. Analytic calculations that scale to larger rings are identified separately. All bit counts below are attack work exponents, and every leakage-aware algorithm is free to ignore the descriptor. The experiments use the stronger $K = 1$ channel throughout.

Table 2 separates the attack families we considered from diagnostics that only describe the channel. A negative entry means that the particular attack did not improve the best known exponent; it does not constitute a generic lower bound.

Exact small-instance experiments do find multi-bit improvements from the leakage. This is important: the channel is exploitable, and its effect cannot be dismissed merely because it is

nonlinear. At production parameters, however, none of the implemented attacks improves upon the roughly one-bit gain of leakage-guided regular Prange. The quadratic formulation and the stationary cross-expansion analysis identify additional attack surfaces, but we do not claim to have closed them.

The most important structured attack reduces the ring modulo $X^{128} + a$. The expected total number of occupied residues is 60.383. This counts selected positions, not nonzero coefficients after folding. The published conservative statistical-decoding model gives 128.450 bits at that mean. Low folded weights are materially weaker: averaging the same optimistic model over unconditioned noise gives 122.482 bits. We therefore require the local support filter defined before Proposition 12.2. It accepts each party's sample with probability 0.501914 and raises the averaged conservative estimate to 130.572 bits. The implementation applies this filter during setup for the Goldilocks profile $N_{\mathsf{R}} = 2^{20}$, $P = 4$, $t = 16$. Each rejection resamples all four supports; later expansions retain the accepted supports. Other parameter sets retain the unfiltered sampler and are not certified by this estimate. The rejection is a security requirement, not a Reverse-Cuckoo mitigation. The filter tests only selected positions; it does not reject zero coefficients or prevent cancellation after folding. Appendix B.4 accounts for its effect on programmed-hash correctness without assuming that the accepted offsets remain independent.

For leakage adaptation after factor reduction, a degree-128 dual word can be chosen to vanish on any desired set of 127 out of 512 reduced coordinates. We projected the complete pair posterior into these coordinates and optimized the resulting global zero-set distribution. Uniform selection is best for both expected work and average signal. A final matching-aware experiment propagated complete support particles through the reduction; it also found no gain, but suffered low particle diversity. Thus the obvious factor attack does not benefit from the leakage, while a fundamentally better joint matching-aware decoder remains unexcluded.

The study therefore covers every attack family listed in Table 2, but with different levels of evidence: some are production cost estimates, some are reduced experiments, and some are structural formulations or open audits. It does not prove security against every decoder, a new structured solver for the quadratic system, a stronger cross-expansion module attack, or a new algebraic attack on the concrete feature evaluator. The scripts, deterministic checks, raw CSVs, and fuller methodology are recorded in `experiments/rev_cuckoo_lpn_attack_study.md`.

12.7 Security statement and scope

The PCG needs a distinguishing assumption, not merely hardness of recovering a sparse support. We state it for the joint sequence of samples and the concrete published descriptors.

Fix a polynomial count $P \geq 2$, block count $t \mid N_{\mathsf{R}}$, and a field $\mathbb{F}_q$. Define

$$\mathbb{R} := \mathbb{F}_q[X]/(X^{N_{\mathsf{R}}} + 1).$$

Fix a corrupted party $p \in \{0, 1\}$, a number of expansions Q, and a total adversarial work budget T. The experiment $\mathsf{RC\text{-}LPN}_b$ has challenge bit $b \in \{0, 1\}$.

1. Independently sample the parties' regular supports, applying the local support filter stated before Proposition 12.2. Retain these supports for all Q expansions.

2. Run the concrete $K = 1$ descriptor-generation procedure on every product list in Equation (23), with fresh shuffle and evaluator randomness. Let H contain all public evaluator seeds and programmed descriptors. Give H and party p's supports to the adversary.

3. For each expansion $u \in [0, Q)$, sample independent coefficients from $\mathbb{F}_q$ on both parties' supports, including zero. Write $e_{v,a}^{(u)}$ for party v's polynomial $a \in [0, P)$. Give party p's coefficients to the adversary.

4. For each expansion u, put $r_0^{(u)} := 1$ and sample $r_i^{(u)} \leftarrow \mathbb{R}$ independently for $i \in [1, P)$. Write $\mathbf{r}^{(u)} := (r_i^{(u)})_{i \in [0,P)}$. If $b = 0$, set

$$Z^{(u)} := \sum_{i \in [0,P)} r_i^{(u)} e_{1-p,i}^{(u)}.$$

   If $b = 1$, sample $Z^{(u)} \leftarrow \mathbb{R}$ independently. Give the adversary every pair $(\mathbf{r}^{(u)}, Z^{(u)})$.

5. The adversary outputs a bit. Its work, including preprocessing of H before receiving the samples, is at most T.

The supports, filter, descriptors, and corrupted-party information have the same distribution in both experiments. Only the honest party's noisy ring samples are replaced; in the random experiment, those replacements are independent of the hidden supports and descriptors.

The coefficient law matches the implementation; no zero rejection is required. For comparison with nonzero-coefficient models, couple the samplers until the first zero coefficient. Across both parties and Q expansions, the statistical distance is at most $2PtQ/q$, approximately $Q\,2^{-57}$ for the Goldilocks profile with $P = 4$ and $t = 16$. This bounds the change in any fixed attack's success probability, not its work exponent, and does not establish a hardness ordering between the laws. The reported nonzero-coefficient attack estimates remain comparisons under that model; the assumption below uses uniform field coefficients.

Assumption 12.3 (Leakage-robust decisional Ring-LPN)**.** *For the chosen parameter family and polynomially bounded T, Q, the advantage*

$$\epsilon_{\mathsf{RC}}(T, Q) := \sup_{\mathcal{A}:\mathrm{time}(\mathcal{A}) \le T} |\Pr[\mathcal{A}(\mathsf{RC\text{-}LPN}_0) = 1] - \Pr[\mathcal{A}(\mathsf{RC\text{-}LPN}_1) = 1]|$$

is negligible in the computational security parameter. The concrete point studied here has

$$(N_{\mathsf{R}}, P, t, d, q', q_{\mathsf{hash}}) = (2^{20}, 4, 16, 16, 64, 64),$$

with the Goldilocks coefficient field $q = 2^{64} - 2^{32} + 1$. Here q_{hash} denotes the feature-compression width called q earlier in this section.

At fixed concrete parameters, $\epsilon_{\mathsf{RC}}(T, Q)$ is a conjectured advantage bound, not a consequence of the attack work estimates. Support-recovery experiments provide evidence about some attacks on this assumption; they do not establish decisional security.

This is the additional pseudorandomness assumption needed to expose the Reverse-Cuckoo descriptors in the Ring-LPN application. A full PCG security proof must also compose the distributed generation subprotocols and reusable DPF realization with the target correlation functionality. The separate hash-correctness bound applies to the filtered caller distribution.

Table 3: RevCuckoo DMPF time breakdown (u64 field; linear sec. 40; cuckoo sec. 2; 3 expansion calls; 1 set). Setup is measured up through `sparseDpf done` and `negate done`; time shown is the max of Party 0/1 timestamps.

n	t	Setup [ms]	Avg. expand [ms]	Total [ms]	#Expands
2^{16}	16	72.90	1.37	77	3
2^{18}	16	172.90	6.37	194	3
2^{20}	16	516.20	27.27	598	3
2^{16}	128	201.80	2.40	209	3
2^{18}	128	363.80	7.07	385	3
2^{20}	128	988.50	38.50	1104	3

13 Evaluation

We measure an implementation of the *Reverse Cuckoo* DMPF (Section 12) integrated into a Ring-LPN OLE pipeline (Section 11). Waterfall's modeled costs appear separately in Section 8.7. Section 13.1 measures the DMPF alone. Section 13.2 measures the end-to-end Ring-LPN OLE pipeline and compares it with OLE and Beaver-triple baselines.

All timings predate the AES rekeying in Appendix C.4, the switch to exact-uniform permutations, and the use of independent evaluator seeds for each product list. The OLE measurements also predate the per-expansion mask refresh in Section 11.2 and omit the support-rejection filter used by the current parameter recipe. These results characterize the measured revision, not the complete current recipe.

Environment. Benchmarks used an i7-13700H CPU with 32 GB DDR5 memory and local sockets with sub-millisecond latency and bandwidth above 10 Gbps. All code was compiled with full optimizations (`-O3`, LTO) and native CPU tuning (`-march=native`). Results are single-threaded per party.

13.1 Reverse-Cuckoo DMPF

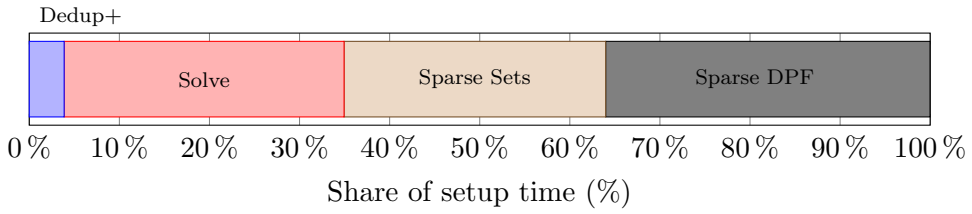

Figure 16: Reverse-Cuckoo DMPF setup breakdown at $n = 2^{18}, t = 128$. Dedup+ includes deduplication, permutation, and hashing. Solve is the binary system solver. Sparse Sets constructs the public leaf domains; Sparse DPF initializes the trees and leaf values.

Table 3 reports setup times of 72.90–988.50 ms and average expansion times of 1.37–38.50 ms across the measured cases. Expansion time increases with the domain size. Setup remains more expensive than one expansion in every row, so reuse amortizes a substantial initial cost.

13.2 Ring-LPN OLE pipeline

We generate OLE correlations over two prime fields. Goldilocks has modulus $q = 2^{64} - 2^{32} + 1$, whose form permits efficient modular arithmetic [Pol22, Blo22]. Fp31 has modulus $q = 15 \cdot 2^{27} + 1$; we use Montgomery reduction for this field.

Protocol	Field	n	Setup time (s)	Median (op/s)	Setup comm	Expand comm
SumDmpf	Goldilocks	2^{16}	–	37,686	–	12.4 MB
RcDmpf	Goldilocks	2^{16}	2.628	885,621	149.52 MB	12.99 MB
RcDmpf	Goldilocks	2^{18}	3.558	1,579,180	158.92MB	12.99 MB
RcDmpf	Goldilocks	2^{20}	12.490	1,811,012	170.44 MB	12.99 MB
[Kel20]	Goldilocks	2^{20}	–	275,548	–	176.65 MB
SumDmpf	Fp31	2^{16}	–	47,250	–	12.14 MB
RcDmpf	Fp31	2^{16}	2.258	1,260,307	143.04 MB	6.51 MB
RcDmpf	Fp31	2^{18}	3.793	2,221,559	152.43 MB	6.51 MB
RcDmpf	Fp31	2^{20}	11.583	2,709,498	163.95 MB	6.51 MB
[Kel20]	Fp31	2^{20}	–	309,588	—	176.56 MB

Table 4: Ring LPN PCG performance and communication for generating OLE correlations. For each field and vector length n, we report the one-time setup latency (seconds), the median per-expansion throughput over 10 expansions (op/s), and the total communication summed over both parties: one-time setup bytes and per-expansion bytes. Communication is shown in MB (decimal). SumDmpf refers to the "sum of DPFs" approach and RcDmpf is our new Reverse Cuckoo DMPF. [Kel20] generates Beaver triples and therefore we generated $n' = n/2$ triples for their numbers.

The $n = 2^{16}$ and $n = 2^{18}$ Reverse-Cuckoo rows measure performance scaling; they are not claimed as 40-bit correctness parameter sets for the present degree-two feature evaluator. Proposition 12.2 and Appendix B certify the corresponding filtered $n = 2^{20}$ caller distribution, not the measured unfiltered sampler.

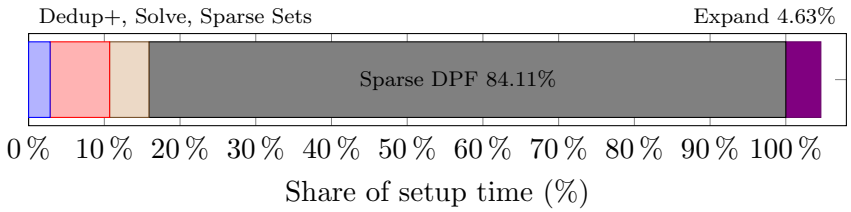

Figure 17: Reverse-Cuckoo Ring-LPN setup breakdown at $n = 2^{20}$, with one expansion shown relative to setup time. Dedup+ includes deduplication, permutation, and hashing. Solve is the binary system solver. Sparse Sets constructs the public leaf domains; Sparse DPF initializes the trees and leaf values. The expansion measurement predates the fresh-mask update.

Results summary. At $n = 2^{20}$, measured throughput is 1.81 million OLEs/s in Goldilocks and 2.71 million OLEs/s in Fp31. Relative to the measured [Kel20] baseline, these are throughput factors of 6.57 and 8.75. Per-expansion communication is smaller by factors of 13.60 and 27.12.

These ratios exclude Reverse-Cuckoo setup and use the conversion of one Beaver triple to two OLEs stated in the table caption.

At $n = 2^{16}$, the throughput factors over SumDmpf are 23.50 and 26.67. Goldilocks expansion communication is slightly higher, not lower: 12.99 MB versus 12.4 MB. Fp31 uses 6.51 MB versus 12.14 MB.

Where setup time goes. In the $n = 2^{20}$ PCG breakdown, sparse-tree initialization accounts for 84.11% of setup time, compared with 7.86% for solving and 5.19% for constructing sparse sets. The remaining 2.84% covers deduplication, permutation, and hashing. These are measured implementation costs, distinct from the logical OT counts used to compare Waterfall profiles.

Setup amortization. Consider the Goldilocks row at $n = 2^{20}$. Setup takes 12.49 s and communicates 170.44 MB. After Q expansions, total measured communication in MB is

$$170.44 + 12.99Q.$$

For example, $Q = 10$ gives 300.34 MB. Each expansion still communicates. The throughput column excludes setup, so an application should account for its intended number of expansions. We do not infer total-byte savings over the smaller-domain SumDmpf row.

Field choice. Fp31 setup communicates 163.95 MB at $n = 2^{20}$; Goldilocks setup communicates 170.44 MB. Fp31 also has higher measured throughput and a smaller expansion payload. These differences illustrate the arithmetic tradeoff. A smaller field is appropriate only when the application's correctness and security requirements permit it.

References

[ADW12] Martin Aumüller, Martin Dietzfelbinger, and Philipp Woelfel. Explicit and efficient hash families suffice for cuckoo hashing with a stash. *CoRR*, abs/1204.4431, 2012.

[AFK+25] Hilal Asi, Vitaly Feldman, Hannah Keller, Guy N Rothblum, and Kunal Talwar. Preamble: Private and efficient aggregation of block sparse vectors and applications. *arXiv preprint arXiv:2503.11897*, 2025.

[BBC+24] Maxime Bombar, Dung Bui, Geoffroy Couteau, Alain Couvreur, Clément Ducros, and Sacha Servan-Schreiber. Foleage: F 4 ole-based multi-party computation for boolean circuits. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 69–101. Springer, 2024.

[BCCD23] Maxime Bombar, Geoffroy Couteau, Alain Couvreur, and Clément Ducros. Correlated pseudorandomness from the hardness of quasi-abelian decoding. Cryptology ePrint Archive, Paper 2023/845, 2023.

[BCG+19] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. Cryptology ePrint Archive, Paper 2019/448, 2019.

[BCG+20] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from ring-lpn. In *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part II 40*, pages 387–416. Springer, 2020.

[BCG+22] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Nicolas Resch, and Peter Scholl. Correlated pseudorandomness from expand-accumulate codes. Cryptology ePrint Archive, Paper 2022/1014, 2022.

[BGH+25] Elette Boyle, Niv Gilboa, Matan Hamilis, Yuval Ishai, and Yaxin Tu. Improved Constructions for Distributed Multi-Point Functions . In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 2414–2432, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.

[BGI15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 337–367. Springer, 2015.

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1292–1303, 2016.

[BGI19] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing for batch codes. Cryptology ePrint Archive, Paper 2019/273, 2019.

[BGIK22] Elette Boyle, Niv Gilboa, Yuval Ishai, and Victor I Kolobov. Programmable distributed point functions. In *Annual International Cryptology Conference*, pages 121–151. Springer, 2022.

[Blo22] Remco Bloemen. The goldilocks prime. `https://xn--2-umb.com/22/goldilocks/`, 2022. Discussion of the Goldilocks prime and roots-of-unity properties.

[CGB17] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX symposium on networked systems design and implementation (NSDI 17)*, pages 259–282, 2017.

[CGBM15] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy*, pages 321–338. IEEE, 2015.

[CHI+19] Koji Chida, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Naoto Kiribuchi, and Benny Pinkas. An efficient secure three-party sorting protocol with an honest majority. *Cryptology ePrint Archive*, 2019.

[DG75] Philippe Delsarte and Jean-Marie Goethals. Alternating bilinear forms over GF(q). *Journal of Combinatorial Theory, Series A*, 19(1):26–50, 1975.

[DPRS23] Hannah Davis, Christopher Patton, Mike Rosulek, and Phillipp Schoppmann. Verifiable distributed aggregation functions. *Cryptology ePrint Archive*, 2023.

[DRRT18] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. Pir-psi: scaling private contact discovery. *Proceedings on Privacy Enhancing Technologies*, 2018.

[DS17] Jack Doerner and Abhi Shelat. Scaling oram for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 523–535, 2017.

[GI14] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *Advances in Cryptology–EUROCRYPT 2014: 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings 33*, pages 640–658. Springer, 2014.

[Kel20] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[KMW10] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, 39(4):1543–1561, 2010.

[KPRR25] Vladimir Kolesnikov, Stanislav Peceny, Srinivasan Raghuraman, and Peter Rindal. Stationary syndrome decoding for improved PCGs. Cryptology ePrint Archive, Paper 2025/295, 2025.

[LXYY25a] Zhe Li, Chaoping Xing, Yizhou Yao, and Chen Yuan. Efficient pseudorandom correlation generators for any finite field. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 145–175. Springer, 2025.

[LXYY25b] Zhe Li, Chaoping Xing, Yizhou Yao, and Chen Yuan. Efficient pseudorandom correlation generators over z/pk z. In *Annual International Cryptology Conference*, pages 207–240. Springer, 2025.

[MZRA22] Yiping Ma, Ke Zhong, Tal Rabin, and Sebastian Angel. Incremental {offline/online}{PIR}. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1741–1758, 2022.

[Pol22] Polygon Zero Team. Plonky2: Fast recursive arguments with plonk and fri. `https://docs.rs/crate/plonky2/latest/source/plonky2.pdf`, 2022. See §2 "Field selection" for the choice $p = 2^{64} - 2^{32} + 1$ (Goldilocks) and fast reduction.

[PR] Lance Roy Peter Rindal. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. `https://github.com/osu-crypto/libOTe`.

[PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[PRRS25] Stanislav Peceny, Srinivasan Raghuraman, Peter Rindal, and Harshal Shah. Efficient permutation correlations and batched random access for two-party computation. In *IACR International Conference on Public-Key Cryptography*, pages 76–109. Springer, 2025.

[PSWW18] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based psi via cuckoo hashing. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 125–157. Springer, 2018.

[Rie25] Vincent Rieder. Silentium: Implementation of a pseudorandom correlation generator for beaver triples. *Cryptology ePrint Archive & International Workshop on Foundations and Applications of Privacy-Enhancing Cryptography*, 2025.

[SGRR19] Phillipp Schoppmann, Adrià Gascón, Leonie Reichert, and Mariana Raykova. Distributed vector-ole: Improved constructions and implementation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1055–1072, 2019.

[YY25] Foo Yee Yeo and Jason HM Ying. Private set intersection and other set operations in the third party setting. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 7723–7742, 2025.

A Waterfall Cuckoo Details and Alternative Profiles

This appendix gives the circuit-level complete-repair generator, the exact failure calculations behind Table 1, and the OT accounting used for its concrete totals. It then records a shallow single-repair alternative. All profiles use the top-level protocol and simulation theorem from Section 8.

A.1 Complete-repair circuit

The circuit represents the current matching by row-to-partition bits $M_{i,s}$ and the cached candidate unit vectors $D_{i,s} := \mathsf{unitVector}_{[0,d_s)}(C_{i,s}, 1)$. The physical placement is recovered as $O_{i,\Delta_s+b} = M_{i,s}D_{i,s}[b]$. At the start of each repair round, the circuit materializes from M an occupancy vector T_s and an h-bit owner array J_s for every partition, where $h := \lceil \log_2 t \rceil$. Thus $T_s[b] = 1$ exactly when bin (s, b) is matched, and $J_s[b]$ holds its matched row. A secret-index mux reads $(T_s[C_{i,s}], J_s[C_{i,s}])$. Every priority choice below is implemented by the prefix circuit

$$\mathsf{first}(x)_i := x_i \bigwedge_{j<i} (1 - x_j).$$

Let $q_i := 1 - \sum_s M_{i,s}$ identify all unmatched rows. One repair uses a public t-step multi-source reachability search initialized with every row for which $q_i = 1$. At each step it expands the first visited but unexpanded row in public row order and reads all w candidate-owner records. A free record terminates the search. An occupied record inserts its owner unless that row was already visited, and stores the current row and partition as the owner's parent. Since a row is expanded at most once, t steps exhaust every row reachable from any unmatched root. The circuit then follows the selected parent chain for t fixed steps and flips the corresponding row-to-partition assignments. Gating every write by the selected target makes all unused steps no-ops.

$\mathsf{WaterfallGen}_{\mathbf{d},c,r}(\llbracket C\rrbracket \in \prod_{s\in[0,w)}[0,d_s)^t) \longrightarrow \llbracket O\rrbracket \in \{0,1\}^{t\times m}$, where $m := \sum_{s\in[0,w)} d_s + c$.

Party $p \in \{0,1\}$ computes:

1. Compute $\llbracket a\rrbracket, \llbracket \ell\rrbracket, \llbracket u\rrbracket$, the initial matching $\llbracket M\rrbracket$, and $\llbracket q\rrbracket$ by the occupancy scan in Equation (5). Cache $\llbracket D\rrbracket_{i,s} := \mathsf{unitVector}_{[0,d_s)}(\llbracket C\rrbracket_{i,s}, 1)$.
2. For $v \in [r]$:
   (a) Materialize $(\llbracket T\rrbracket_s, \llbracket J\rrbracket_s)_{s\in[0,w)}$ from $\llbracket M\rrbracket$ and the cached decoders, and set $\llbracket q\rrbracket_i := 1 - \sum_s \llbracket M\rrbracket_{i,s}$ for every $i \in [0,t)$.
   (b) Initialize $\llbracket V\rrbracket := \llbracket q\rrbracket$, $\llbracket E\rrbracket := 0^t$, and zero parent arrays $\llbracket \pi\rrbracket \in [0,t)^t$ and $\llbracket \eta\rrbracket \in [0,w)^t$. Reset the target-selected flag to zero.
   (c) For $a \in [0,t)$, let $\llbracket f\rrbracket := \mathsf{first}(\llbracket V\rrbracket \wedge (1 - \llbracket E\rrbracket))$ and add $\llbracket f\rrbracket$ to $\llbracket E\rrbracket$. For each $s \in [0,w)$, read the occupancy bit and owner at the candidate of $\llbracket f\rrbracket$ from $(\llbracket T\rrbracket_s, \llbracket J\rrbracket_s)$. Select the first free candidate if no target has yet been selected. Otherwise, insert each newly seen owner into $\llbracket V\rrbracket$ in partition order. Store $(\sum_{i\in[0,t)} i\llbracket f\rrbracket_i, s)$ in its parent record on first discovery. Gate every read and write by an active selected row and an unset target flag.
   (d) Starting from the row and partition of the selected free candidate, perform t gated backtracking steps. At each step, assign the current row to the current partition, clear its previous partition assignment, and, if $\llbracket q\rrbracket_i = 0$ for the current row i, move to its stored parent record. Reaching a root ends the active path; if no free candidate was selected, leave $\llbracket M\rrbracket$ unchanged.
3. Recompute $\llbracket q\rrbracket_i := 1 - \sum_s \llbracket M\rrbracket_{i,s}$ and $\llbracket R\rrbracket_i := \sum_{j<i} \llbracket q\rrbracket_j$. Set $\llbracket O\rrbracket_{i,\Delta_s+b} := \llbracket M\rrbracket_{i,s}\llbracket D\rrbracket_{i,s}[b]$ and $\llbracket O\rrbracket_{i,\Delta_w+k} := \llbracket q\rrbracket_i[\llbracket R\rrbracket_i = k]$.
4. Return $\llbracket O\rrbracket$.

Figure 18: Gate-level organization of the canonical complete-repair generator. All loops have public bounds; muxes, unit-vector decoders, prefix priority, and gated writes expand to Boolean products and XORs.

Lemma A.1 (Complete-search invariant)**.** *Before the search selects a free bin, V consists exactly of the unmatched roots and the matched owners discovered from expanded rows. Every discovered non-root row stores the first edge by which it was discovered, and the parent records form a forest rooted at the unmatched rows. If no free bin has been selected after t steps, every row reachable from an unmatched root has been expanded. If the search selects a free bin, the parent records define a simple augmenting path from one root to that bin.*

Proof. Initially, V contains exactly the unmatched roots. Expanding a visited row examines all of its candidate edges. A free endpoint completes an alternating path. An occupied endpoint extends the path through its uniquely matched owner. The visited mask inserts each owner once, so its first stored parent is never overwritten and the parent relation is a forest. Conversely, the circuit adds a non-root row only through such an occupied candidate edge. Induction proves the discovery claim. Unless a free endpoint terminates the search, each of the t steps expands one previously unexpanded reachable row; there are at most t rows. Backtracking follows one tree of the parent forest to its unmatched root and therefore flips exactly one simple augmenting path. □

The path flip preserves the row and column inequalities in Equation (10) and increases the main matching size by one. Candidate decoders ensure Equation (11). This proves the invariant used by Proposition 8.2.

A.2 Exact failure calculations

Let L_s be the number of live rows entering partition s, with $L_0 = t$. If $L_s = n$ and the n independent candidates occupy j bins, then $L_{s+1} = n - j$. Write $\left\{{n \atop j}\right\}$ for a Stirling number of the second kind. Then

$$\Pr[L_{s+1} = n - j \mid L_s = n] = \frac{(d_s)_j \left\{{n \atop j}\right\}}{d_s^n}. \tag{34}$$

Dynamic programming over this transition gives the exact overflow tail

$$\Pr[L_w > r + c] = \sum_{\ell > r+c} \Pr[L_w = \ell]. \tag{35}$$

For a fixed set of q rows, the probability that their candidates occupy exactly j bins in partition s is

$$\Pr[|N_s(Q)| = j] = \frac{(d_s)_j \left\{ {q \atop j} \right\}}{d_s^q}. \tag{36}$$

Hall's theorem fails only if some q rows have at most $q-1$ total neighbors. A union bound over the row set and its per-partition neighbor counts gives

$$\Pr[G_C \text{ has no size-}t \text{ matching}] \le \sum_{q=1}^{t} \binom{t}{q} \sum_{\substack{j_0,\dots,j_{w-1} \ge 0 \\ \sum_s j_s \le q-1}} \prod_{s \in [0,w)} \frac{(d_s)_{j_s} \left\{ {q \atop j_s} \right\}}{d_s^q}. \tag{37}$$

The implementation evaluates Equations (35) and (37) with exact rational arithmetic before converting the final bounds to bits.

One useful necessary check comes from identical candidate vectors. Put $D := \prod_s d_s$. If some candidate vector occurs more than w times, those rows have only w neighbors and violate Hall's condition. Exact occupancy counting gives

$$\Pr[\text{maximum candidate multiplicity} > w] = 1 - \frac{t!}{D^t} [z^t] \left(\sum_{j=0}^{w} \frac{z^j}{j!} \right)^D. \tag{38}$$

This check rules out overly small candidate spaces before the full Hall sum is evaluated.

For $t = 16$ and $B_0 = 256$, the complete-repair profiles give

	Hall bits	overflow bits
$(16, 16, 16, 16),\ c = 0, r = 3$	43.83	45.24
$(128, 128, 64),\ c = 0, r = 2$	41.15	47.07

Combining the two terms in Equation (12) gives 43.37 and 41.13 batch-correctness bits, respectively.

A.3 OT accounting

We compare profiles by $(Q_{\mathsf{OT}}, B_{\mathsf{OT}})$, where the first coordinate counts OTs and the second sums their payload lengths. A product of a shared bit and a shared λ-bit string costs two λ-bit OTs and contributes $(2, 2\lambda)$. XORs, public linear maps, openings, local PRG calls, depth, and rounds are free.

Let $k_s := \log_2 d_s$, $h := \lceil \log_2 t \rceil$, $D := \lceil \log_2 N \rceil$, $\ell := \max\{1, D\}$, $g := \log_2 |\mathbb{G}|$, $h_m := \lceil \log_2 m \rceil$, and $L_{\mathsf{sc}} := t + D + 2$. To evaluate all hash polynomials, compute powers of each secret $\iota(A_i)$ once and reuse them across partitions. Squaring is linear; only the $\nu_t := \lfloor (t-2)/2 \rfloor$ odd powers require a field multiplication.

Recursive polynomial-basis Karatsuba gives

$$K(1) := 1, \qquad K(\ell) := K(\lfloor \ell/2 \rfloor) + 2K(\lceil \ell/2 \rceil). \tag{39}$$

At $\ell = 20$, use $\mathbb{F}_{2^{20}} \cong \mathbb{F}_{16}[Y]/(Y^5 + Y^2 + 1)$ and evaluate the two degree-four operands at nine points of $\mathbb{F}_{16}$. Nine pointwise $\mathbb{F}_{16}$ products, each costing $K(4) = 9$ bit products, give an 81-product multiplier. Evaluation, interpolation, reduction, and basis conversion are binary linear maps. Put $M(20) := 81$ and $M(\ell) := K(\ell)$ otherwise. Hashing and dummy selection contribute

$$q_{\mathsf{hash}} := t\nu_t M(\ell) + wt, \qquad b_{\mathsf{hash}} := t\nu_t M(\ell) + t\sum_s k_s. \tag{40}$$

For a sparse set S_j, let $\sigma(S_j)$ count the binary trie depths that contain a split, and put

$$\Sigma(P) := \sum_{j:S_j \neq \emptyset} \sigma(S_j) \leq mD. \tag{41}$$

Each binary correction selects between shared child sums. It therefore costs two OTs, not one. We charge κ bits per selected message, including the implementation's full-block representation. At a sparse level, masking both child sums costs a further two OTs with 2κ-bit messages. The mask depends on secret activity and cannot be omitted merely because public splitting nodes exist.

The implementation first expands a complete binary prefix of depth $D_0 := \min\{D, h_m+2\}$, then prunes the remaining public subtrees. Activity is known to be one at each complete level; only the residual sparse levels need masking. The row-to-column trees are fully dense. For a conservative reservation bound, put

$$C_{\mathsf{DPF}} := mD + th_m, \qquad M_{\mathsf{DPF}} := m(D - D_0).$$

These count correction selections and activity masks, respectively. Skipping unused public levels can reduce consumption below this bound. Let $\chi = 0$ for groups of characteristic two and $\chi = 1$ otherwise. In the latter case, each DPF additionally caches one sign–payload product, costing two OTs with g-bit messages. Thus reusable DPF initialization contributes

$$Q_{\mathsf{DPF}} = 2\big(C_{\mathsf{DPF}} + M_{\mathsf{DPF}} + \chi(m+t)\big), \tag{42}$$

$$B_{\mathsf{DPF}} = 2\kappa C_{\mathsf{DPF}} + 4\kappa M_{\mathsf{DPF}} + 2g\chi(m+t). \tag{43}$$

The sign controls depend on cached tags, so their correlations are charged once, not once per expansion. This accounting does not change the leaf conversion or local expansion work.

If q_m is the number of switches in one recursive Waksman pass, then

$$q_1 := 0, \quad q_2 := 1, \quad q_m := 2\lfloor m/2 \rfloor + q_{\lfloor m/2 \rfloor} + q_{\lceil m/2 \rceil} \quad (m > 2).$$

The two serial privately controlled passes contribute

$$(2q_m, 2q_m L_{\mathsf{sc}}). \tag{44}$$

For deduplication, cache the first-occurrence bits and the equality bits gated by those first-occurrence bits. Expansion multiplies each retained payload by its activity bit and routes each later equal payload to its first occurrence. This requires at most $t + \binom{t}{2} = t(t+1)/2$ shared bit–payload products. Their controls depend only on setup addresses. The model charges the corresponding OT correlations once, with g-bit payload width; round-separated expansion reuses

them but still communicates masked payloads. The formulas below do not count those online messages. The common deduplication and activity path has

$$q_{\mathsf{common}} = \binom{t}{2} D + \frac{(t-1)(t-2)}{2} + tD + \frac{t(t+1)}{2} + 3t, \tag{45}$$

$$b_{\mathsf{common}} = \binom{t}{2} D + \frac{(t-1)(t-2)}{2} + t(D+1) + \frac{t(t+1)}{2} g + t + t(2D+1). \tag{46}$$

The optimized initial scan performs one mux-tree occupancy read, one gated unit-vector decoder, and one live-bit product per row and partition. Stash compaction adds one product per row. Thus the initial stage has

$$q_{\mathsf{basic}} = t \sum_s (2k_s + 1) + t, \tag{47}$$

$$b_{\mathsf{basic}} = t \sum_s (2d_s - 1) + t(c+1). \tag{48}$$

For complete repair, the scan caches raw and winning decoders and gates the final matching once. Its base profile is

$$q_{\mathsf{base}} = t \sum_s (2k_s + 2) + t, \qquad b_{\mathsf{base}} = t \sum_s (4d_s - 3) + t(c+1). \tag{49}$$

With packed control bits, one complete augmentation contributes

$$\begin{aligned} q_{\mathsf{reach}} &:= wt + t \sum_s k_s + t(t + 2h + wh + 3w + 2) \\ &\quad + (w-1) + (t-1)(2h + w) + t, \qquad (50) \\ b_{\mathsf{fwd}} &:= t\big((t-1)(2 + w(h+4)) + ht + 4w\big), \\ b_{\mathsf{back}} &:= (w-1)(t-1) + (t-1)\big((h + 2w + 1)(t-1) + 1\big) + tw, \\ b_{\mathsf{reach}} &:= t \sum_s (d_s - 1) + t(h+1) \sum_s (d_s - 1) + b_{\mathsf{fwd}} + b_{\mathsf{back}}. \qquad (51) \end{aligned}$$

Consequently,

$$q_{\mathsf{BA}} = q_{\mathsf{base}} + r q_{\mathsf{reach}}, \tag{52}$$

$$b_{\mathsf{BA}} = b_{\mathsf{base}} + r b_{\mathsf{reach}}. \tag{53}$$

Combining all components gives

$$Q_{\mathsf{OT}} = 2(q_{\mathsf{common}} + q_{\mathsf{BA}} + q_{\mathsf{hash}}) + 2q_m + Q_{\mathsf{DPF}}, \tag{54}$$

$$B_{\mathsf{OT}} = 2(b_{\mathsf{common}} + b_{\mathsf{BA}} + b_{\mathsf{hash}}) + 2q_m L_{\mathsf{sc}} + B_{\mathsf{DPF}}. \tag{55}$$

At $t = 16$, $N = 2^{20}$, $\kappa = 128$, and $\mathbb{G} = \mathbb{Z}_{2^{64}}$, these equations give $(39{,}052, 942{,}888)$ for the default $4N$ profile and $(55{,}358, 3{,}623{,}410)$ for the lower-expansion $3N$ profile, matching Table 1. Their dense-prefix depths are 8 and 11. The DPF terms are, respectively, $(4{,}448, 755{,}712)$ and $(19{,}520, 3{,}192{,}832)$. For a characteristic-two group of the same size, omit $2(m+t)$ OTs and $128(m+t)$ payload bits from each total.

A.4 Specialized single-repair profile

This setup-minimized alternative performs one fixed depth-two augmentation after the initial waterfall stage. If $L := \sum_i q_i \le c$, the stash holds all overflow rows. If $L = c + 1$, it selects the first overflow row, inspects the later candidates of the rows occupying its candidates, and applies the first free length-two path. Otherwise the excess rows remain unplaced. The protocol opens no failure bit; the simulation theorem charges this event to p_{1rep}.

$\mathsf{OneRepairWaterfallGen}_{w,d,c}(\llbracket C \rrbracket \in [0,d)^{t\times w}) \longrightarrow \llbracket O \rrbracket \in \{0,1\}^{t\times(wd+c)}$.
Party $p \in \{0,1\}$ computes:

1. Compute $\llbracket a \rrbracket, \llbracket \ell \rrbracket, \llbracket u \rrbracket$, the main-position matrix $\llbracket O \rrbracket^{(0)}$, $\llbracket q \rrbracket$, and $\llbracket R \rrbracket$ as in Equations (5), (6), and (7).
2. $\llbracket L \rrbracket := \sum_{i\in[0,t)} \llbracket q \rrbracket_i$, $\llbracket x \rrbracket_i := \llbracket q \rrbracket_i \wedge \mathsf{eq}(\llbracket R \rrbracket_i, 0)$ for $i \in [0,t)$, and $\llbracket f \rrbracket_{\mathsf{try}} := \mathsf{eq}(\llbracket L \rrbracket, c+1)$.
3. For $u \in [0,w)$, let $\llbracket b \rrbracket_u := \sum_{i\in[0,t)} \llbracket x \rrbracket_i \llbracket C \rrbracket_{i,u}$ and, for $i \in [0,t)$,
$$\llbracket v \rrbracket_{u,i} := \bigvee_{b\in[0,d)} \left(\mathsf{eq}(\llbracket b \rrbracket_u, b) \wedge \llbracket O \rrbracket^{(0)}_{i,ud+b}\right).$$
Write $\llbracket v \rrbracket_u$ for the row selected by the one-hot vector $(\llbracket v \rrbracket_{u,i})_{i\in[0,t)}$, and similarly write $\llbracket x \rrbracket$ for $(\llbracket x \rrbracket_i)_{i\in[0,t)}$.
4. For $(u,s) \in [0,w)^2$ with $u < s$, let $\llbracket b \rrbracket_{u,s} := \sum_{i\in[0,t)} \llbracket v \rrbracket_{u,i} \llbracket C \rrbracket_{i,s}$ and
$$\llbracket g \rrbracket_{u,s} := \llbracket f \rrbracket_{\mathsf{try}} \wedge \bigwedge_{i\in[0,t)} \bigwedge_{b\in[0,d)} \neg\left(\mathsf{eq}(\llbracket b \rrbracket_{u,s}, b) \wedge \llbracket O \rrbracket^{(0)}_{i,sd+b}\right).$$
5. In lexicographic order on (u,s), let $\llbracket z \rrbracket_{u,s} := \llbracket g \rrbracket_{u,s} \wedge \neg \bigvee_{(u',s')<(u,s)} \llbracket g \rrbracket_{u',s'}$ and $\llbracket z \rrbracket := \bigvee_{u<s} \llbracket z \rrbracket_{u,s}$.
6. For $(i,a,b) \in [0,t) \times [0,w) \times [0,d)$, set
$$\llbracket \Delta \rrbracket^{\mathsf{old}}_{i,a,b} := \bigoplus_{u<s} \llbracket z \rrbracket_{u,s} [a = u] \mathsf{eq}(\llbracket b \rrbracket_u, b)(\llbracket v \rrbracket_{u,i} \oplus \llbracket x \rrbracket_i),$$
$$\llbracket \Delta \rrbracket^{\mathsf{new}}_{i,a,b} := \bigoplus_{u<s} \llbracket z \rrbracket_{u,s} [a = s] \mathsf{eq}(\llbracket b \rrbracket_{u,s}, b) \llbracket v \rrbracket_{u,i},$$
$$\llbracket O \rrbracket^{(1)}_{i,ad+b} := \llbracket O \rrbracket^{(0)}_{i,ad+b} \oplus \llbracket \Delta \rrbracket^{\mathsf{old}}_{i,a,b} \oplus \llbracket \Delta \rrbracket^{\mathsf{new}}_{i,a,b}.$$
7. For $i \in [0,t)$, let $\llbracket q \rrbracket'_i := \llbracket q \rrbracket_i \wedge \neg(\llbracket x \rrbracket_i \wedge \llbracket z \rrbracket)$ and $\llbracket R \rrbracket'_i := \sum_{j<i} \llbracket q \rrbracket'_j$. For $(k,i) \in [0,c) \times [0,t)$, set $\llbracket O \rrbracket^{(1)}_{i,wd+k} := \llbracket q \rrbracket'_i \wedge \mathsf{eq}(\llbracket R \rrbracket'_i, k)$.
8. Return $\llbracket O \rrbracket^{(1)}$.

Figure 19: The specialized single depth-two repair followed by stash compaction.

A.5 Failure bound and parameters

For a live-count path $\boldsymbol{L} = (L_0, \ldots, L_w)$ under Equation (34), put $J_s := L_s - L_{s+1}$ and let $p(\boldsymbol{L})$ denote the path probability.

Proposition A.2 (One-repair failure bound)**.** *Fix $d \in \mathbb{N}$, and define*
$$\mathbf{d} := (d, \ldots, d) \in \mathbb{N}^w.$$

Then

$$p_{\mathsf{1rep}} \le \Pr[L_w \ge c+2] + \sum_{\boldsymbol{L}: L_w = c+1} p(\boldsymbol{L}) \prod_{s=1}^{w-1} \left(\frac{J_s}{d}\right)^s. \tag{56}$$

Proof. The first term covers every proposal with at least $c+2$ overflow rows. Fix a path ending in $c+1$ rows and let x be the selected overflow row. Its candidate in partition u is occupied by a row v_u placed in that partition. The v_u are distinct because a placed row owns only one position. Every candidate of v_u through partition u was tested by the waterfall and is occupied. For every later partition $s > u$, however, $C_{v_u,s}$ was never inspected because v_u had already left the live set. These cells are mutually independent and uniform, and are independent of the J_s occupied bins in partition s. Exactly s such untouched cells target partition s, one from each $v_0, \ldots, v_{s-1}$. All length-two paths are blocked with probability $\prod_{s=1}^{w-1}(J_s/d)^s$. Summing over the live-count paths proves the bound. □

For $t = 16$ and $B_0 = 256$, the compact proved profile is:

w	d	c	m	expansion	total OTs	OT payload	batch bits
4	16	2	66	$6N$	30,980	829,288 bits	45.23

Table 5: The retained specialized one-repair profile.

The compact point $(w, d, c) = (4, 16, 2)$ satisfies

$$\begin{aligned} \Pr[L_4 \ge 4] &< 2^{-53.2360}, \\ \Pr[L_4 = 3 \text{ and depth-two repair fails}] &< 2^{-60.6651}, \\ p_{\mathsf{1rep}} &< 2^{-53.2277}. \end{aligned} \tag{57}$$

Thus $B_0 p_{\mathsf{1rep}} < 2^{-45.2277}$. It uses 66 position columns and $6N$ local expansion.

A.6 OT count

Put $k := \log_2 d$, $h := \lceil \log_2 t \rceil$, $\omega := \lceil \log_2 w \rceil$, and $P := \binom{w}{2}$. For the OT-minimized realization, augment the occupancy scan from Section A.3 with an h-bit owner array $J_s \in [0,t)^d$ for each partition. Whenever row i wins with gated unit vector E_i, update the bit planes of J_s by the public binary encoding of i times E_i. This is linear. Reading $J_s[C_{i,s}]$ later returns the owner of an occupied candidate without an all-pairs equality scan.

During the scan, also cache the ungated unit vector for every $C_{i,s}$. Its decoder shares the same address bits and dependency level as the winning unit-vector decoder, so their string operands are concatenated in each OT. After the repair, one wide product gates the cached vector by the final row-to-partition matching bit; one coordinate is recovered linearly from the unit-vector sum. Together, the scan, both decoders, and final gate use $2k+2$ products with width sum $4d-3$ per row and partition.

The one-hot overflow root gives its h-bit row number by a public linear map. One mux tree, batched across all w candidate arrays, reads its candidates. The w owner reads use the corresponding k-bit candidate addresses. For a fixed victim, reads of all its later-partition candidates share the same row address and are likewise batched. The resulting P candidate addresses read

the occupancy arrays. Finally, a priority scan selects the first feasible pair, a mux selects its victim identifier, and two demux trees route the w-bit matching deltas to the root and victim rows. This gives

$$q_{\mathsf{1rep}} = wt(2k+2) + 2t + (w+2)h + k(w+P) + P + \omega + 3, \tag{58}$$

$$\begin{aligned} b_{\mathsf{1rep}} = {} & wt(4d-3) + t(2c+3) + wk(t-1) + wh(d-1) \\ & + Pk(t-1) + P(d-1) + 2P + 2^{\omega}h + 2w(t-1) + 1. \end{aligned} \tag{59}$$

At $(t, w, d, c) = (16, 4, 16, 2)$ these are $q_{\mathsf{1rep}} = 747$ and $b_{\mathsf{1rep}} = 5{,}095$. These expressions are conservative upper bounds; omitting the unused explicit failure test only lowers them. Including the common, direct-hash, scatter, column sparse-DPF, and cached row-to-column DPF terms from Equations (54) and (55) gives at most 30,980 OTs carrying 829,288 payload bits for $\mathbb{G} = \mathbb{Z}_{2^{64}}$.

A.7 Asymmetric complete-repair profile

The complete-repair generator also supports $\mathbf{d} = (32, 32, 16, 8)$, $c = 0$, and $r = 2$. This profile has $m = 88$, 41.20 bits of batch correctness, and $4N$ local expansion. Substitution in Equations (54) and (55) gives 38,158 OTs carrying 1,162,710 payload bits. It saves 894 OTs relative to the default profile, but adds 24 physical columns, about 26.8 KiB of OT payload, and a smaller correctness margin. We therefore retain it as a setup-versus-state alternative rather than a main profile.

A.8 Hybrid simulator for Waterfall

This subsection supplies the simulator used in Theorem 8.4. Fix a static semi-honest corrupted party $p \in \{0, 1\}$. Circuit, permutation, and reusable DPF calls are ideal. The simulator receives party p's setup input shares and, for each expansion, its payload shares. It maintains the simulated caller's shares and public state between calls. Ideal subprotocols expose no additional implementation state. At every ideal DPF expansion, the corrupt party selects its output share using its current view. The simulator runs this strategy on the simulated view; it does not replace the selected map by an independent sample. When composing concrete DPF realizations, their stateful simulators supply these strategies and simulate their own local states and messages.

Setup calls and opening. The simulator samples each public h_s from $\mathcal{H}_{t,k_s}$ and derives P and S. It follows the setup circuit in order. For each fresh secret output of an ideal circuit call, it samples the corrupted party's share with the prescribed uniform law. It computes local linear operations, public constants, and repeated uses of a share rather than resampling them. This simulates the normalization and placement calls without knowing the honest party's address shares.

Each ideal permutation pass returns freshly shared outputs while retaining its private permutation for the inverse call. The simulator samples the corrupted party's output shares for these calls. It also retains the corrupt party's sampled permutation and its inverse, which are part of that party's local view. At the opening, it samples $V \leftarrow \mathsf{Inj}(t, m)$ and sends the honest opening share $V \oplus \widehat{O}_p$. The public scatter is then a deterministic operation on V and the previously sampled shares. In particular, the simulator does not independently resample the public scatter's outputs. It continues with the inverse permutation outputs, computes the representative addresses, and supplies the setup responses of the row and column DPF functionalities. On successful routing their indices are valid, so those responses are ready.

For every fixed hash tuple and successful candidate matrix, the hidden honest permutation makes the real opening uniform on $\mathsf{Inj}(t, m)$. Couple that opening to V. The simulated hashes are still unconditioned samples: routing success need not be independent of the hashes. If routing fails, discard this coupling and charge the entire continuation to the failure event. This includes the unmatched row labels exposed by the opening. Its probability is p_Γ for every fixed setup input.

Expansion calls. The simulator retains the setup shares used for duplicate aggregation. It samples the fresh private outputs of the ideal multiplication calls and computes their specified local sums. For each row $i \in [0, t)$, let $v_{p,i} \in \mathbb{G}^{[0,m)}$ be the map chosen by the corrupt party at its row-to-column DPF call. The simulator computes the corrupt column-payload share $B'_{p,j} := \sum_{i\in[0,t)} v_{p,i}(j)$ for each $j \in [0, m)$. These are the inputs to its column-DPF calls. No row destination, equality bit, or setup state is sampled again.

For each nonempty S_j, let $Z_{p,j} \in \mathbb{G}^{S_j}$ be the corrupt output map selected at column DPF j. The simulator follows those selections and computes

$$Y_p(x) := \sum_{j:x\in S_j} Z_{p,j}(x), \qquad x \in [0, N). \tag{60}$$

It submits Y_p to $\mathcal{F}_{\mathsf{rDMPF}}$. Empty sparse sets require no DPF call. This procedure imposes no distribution on the selected maps beyond that of the corrupt strategy on its simulated view.

Honest outputs and reuse. On successful routing, each ideal DPF gives the honest party the complement to the specified point map. Summing those complements and applying Proposition 8.3 gives the honest DMPF output $\mathsf{mp}_{N,t}(A, B) - Y_p$. This is exactly the output of the top-level functionality. Thus the coupling preserves the joint corrupt view and honest output, not merely the marginal transcript. It holds for arbitrary choices of the corrupt subprotocol shares, including correlated, nonuniform choices.

Given a coupled history, use the same next inputs and corrupt random coins in both executions. Fresh circuit-output shares can be coupled identically; DPF share selections then follow the same strategy on the same view. The complement identity completes the honest output for the next call. Induction proves the hybrid claim for polynomially many adaptive expansions. Routing failure is fixed during setup and is charged once per state. Replacing concrete subprotocol calls by their ideal calls and online simulators adds their accumulated joint simulation loss $\epsilon_{\mathsf{sub}}(B_0, Q)$. This loss includes any distinguishable failure of those realizations; the hybrid routing error remains separate.

B Correctness Analysis of the Reverse-Cuckoo Evaluator

This appendix proves Proposition 12.2. The analysis is about consistency of the programmed binary systems. It is independent of the Reverse-Cuckoo leakage assumption in Section 12.4.

B.1 Caller distribution and shuffle factors

Use the notation of Equation (23). At the claimed Ring-LPN point,

$$N_{\mathsf{R}} = 2^{20}, \qquad P = 4, \qquad t = 16, \qquad L = N_{\mathsf{R}}/t = 2^{16}.$$

The $P^2t = 256$ triples (a, b, k) produce 256 Reverse-Cuckoo sets. Before deduplication, set $A_{a,b,k}$ contains the t sums

$$X_{0,a,j} + X_{1,b,(k-j) \bmod t}, \qquad j \in [0, t).$$

Before any support rejection, these points within one set are independent samples from the triangular distribution obtained by adding two uniform elements of $[0, L)$. The DPF domain is $[0, 2L)$, which we identify through binary encoding with $\mathcal{X} := \mathbb{F}_2^{17}$. Every point has probability at most $2/|\mathcal{X}|$. Different sets share the underlying offsets and are therefore correlated. The union bound below uses only expected relation counts; it does not assume independent failures across sets.

Fix a relation of weight k. Under a uniform shuffle into w partitions of d rows, the probability that all its points enter one partition is

$$p_{\mathsf{part}}(k) = \frac{w(d)_k}{(wd)_k}, \tag{61}$$

where $(a)_k = a(a-1)\cdots(a-k+1)$. Conditioned on this event, its labels are a uniform k-subset of $\mathbb{F}_2^{\log_2 d}$. Let $p_0(d, k)$ be the probability that their XOR is zero. A character sum gives

$$p_0(d,k) = \begin{cases} \dfrac{1}{d}, & k \text{ odd}, \\ \dfrac{\binom{d}{k} + (d-1)(-1)^{k/2}\binom{d/2}{k/2}}{d\binom{d}{k}}, & k \text{ even}. \end{cases} \tag{62}$$

The relation is harmful with the remaining probability $1 - p_0(d, k)$.

B.2 Counting all degree-two relations

Only relations with $L_\alpha = 0$ can disappear in the nonlinear lift. For real points, the sentinel coordinate also implies that such a relation has even weight. Let $B[k, s]$ denote the number of k-subsets of $\mathcal{X}$ whose XOR is zero and whose quadratic moment has rank $2s$. The rank-zero class consists of the weight-k words in $\mathrm{RM}(2, 17)^\perp$; the other classes retain nonzero quadratic moments.

The refinement can be computed without enumerating point sets. Let $K = \dim \mathrm{RM}(2, 17) = 1 + 17 + \binom{17}{2}$, let v_i be the number of alternating forms of rank $2i$, and let $P_{i,s}$ be the eigenmatrix of the binary alternating-forms association scheme. If $T_{k,i}$ is the sum of the weight-k Hamming Krawtchouk polynomial over the affine parts of a quadratic form of rank $2i$, then

$$B[k, s] = 2^{-K} \sum_i v_i P_{i,s} T_{k,i}. \tag{63}$$

We use the alternating-forms eigenmatrix of Delsarte and Goethals [DG75]. Equation (63) specializes the MacWilliams transform while retaining quadratic rank; direct enumeration in dimensions three and four verifies the implementation of the transform.

For any fixed k-subset, the probability that all its points occur among the t samples is at most $(t)_k(2/|\mathcal{X}|)^k$. Deduplication cannot increase the number of distinct relations. The expected number of weight-k, rank-$2s$ relations in the complete batch is therefore at most

$$P^2 t\,(t)_k B[k, s] \left(\frac{2}{|\mathcal{X}|}\right)^k. \tag{64}$$

For such a relation, the probability that the degree-two lift vanishes is

$$a_{q'}(2s) := \left(\frac{1}{2} + 2^{-2s-1}\right)^{q'}. \tag{65}$$

Combining Equations (61), (62), (64), and (65), then summing over $k \le t$ and s, gives the structural failure bound

$$\epsilon_{\mathsf{str}} \le \sum_{k,s} P^2 t\,(t)_k B[k,s] \left(\frac{2}{|\mathcal{X}|}\right)^k \frac{w(d)_k}{(wd)_k}\big(1 - p_0(d,k)\big) a_{q'}(2s). \tag{66}$$

This sum includes deterministic degree-two identities: when $s = 0$, $a_{q'}(0) = 1$.

For the numerical bounds, we sharpen the expected count of affine planes using the triangular distribution directly. Write $p(x)$ for its mass at $x \in \mathcal{X}$, and define its Walsh transform by $\widehat{p}(u) := \sum_x p(x)(-1)^{u\cdot x}$. The probability that four independent samples are distinct and have XOR zero is

$$\frac{1}{|\mathcal{X}|}\sum_u \widehat{p}(u)^4 - 3\left(\sum_x p(x)^2\right)^2 + 2\sum_x p(x)^4.$$

Multiplying by $P^2 t\binom{t}{4}$ bounds their expected count. The numerical structural sum also conservatively uses the full survival probability in Equation (29) instead of only $a_{q'}(2s)$. This counts some final-compression events twice but preserves an upper bound.

The largest terms for $w = 2$, $d = 16$, and the former intermediate width $q' = 56$ are shown below. Every omitted rank class contributes less than the last displayed row.

Table 6: Dominant structural terms at $q' = 56$.

Relation weight k	Quadratic rank $2s$	Batch contribution
4	2	$2^{-39.469}$
8	0	$2^{-42.217}$
6	4	$2^{-46.993}$
8	6	$2^{-50.366}$
10	8	$2^{-52.928}$

The first row consists of affine planes. A separate Walsh-transform calculation bounds their expected number before the shuffle by 3.791522 and gives the same $2^{-39.469}$ contribution. Rank-zero relations begin at weight eight and are affine three-cubes; their complete contribution is at most $2^{-42.217}$. Thus deterministic degree-two identities are included, not excluded by a sampling heuristic.

B.3 Final compression and complete bound

Fix the supports, shuffle, and lifted rows in one partition containing r real rows. Each nonzero lifted relation is killed by the independent uniform matrix A_2 with probability 2^{-q}. There are at most $2^r - 1$ such relations, so their union contribution is less than 2^{r-q}. This argument does not require the lifted rows to be independent.

Under a uniform shuffle, the number of real rows in one partition is hypergeometric before deduplication. Deduplication can only reduce this number. Averaging and taking a union bound over all sets and partitions gives

$$\epsilon_{\mathsf{cmp}} < P^2 t w \sum_{r=0}^{t} \frac{\binom{d}{r}\binom{(w-1)d}{t-r}}{\binom{wd}{t}} 2^{r-q}. \tag{67}$$

The structural and compression events cover every nonzero row relation: either it vanishes in the lift, or a nonzero lifted relation is killed by A_2. Hence $\epsilon_{\mathsf{batch}} \le \epsilon_{\mathsf{str}} + \epsilon_{\mathsf{cmp}}$.

Table 7: Analytic correctness bounds before support rejection for the $N_{\mathsf{R}} = 2^{20}$, $w = 2$, $d = 16$ Ring-LPN batch. Displayed exponents are rounded.

$q' = q$	Structural	Compression	Complete
56	$2^{-39.261}$	$2^{-38.288}$	$2^{-37.694}$
64	$2^{-42.007}$	$2^{-46.288}$	$2^{-41.935}$

Substituting $d = 16$ and $\lambda = 40$ in Equation (25) gives $q' = q = 64$, proving the unfiltered clause of Proposition 12.2. For comparison, the same calculation for $w = 3$, $d = 8$ gives only 32.577 bits at width 48, but 40.194 bits at the fixed-margin width 56.

The relation count also depends on the ring degree because $L = N_{\mathsf{R}}/t$ determines the point domain. Applying the same analytic calculation to the three Ring-LPN benchmark sizes gives the following bounds.

Table 8: Dependence of the $q' = q = 64$ proof on the Ring-LPN ring degree.

N_{R}	$L = N_{\mathsf{R}}/t$	DPF domain	Complete bound
2^{16}	2^{12}	2^{13}	$2^{-26.218}$
2^{18}	2^{14}	2^{15}	$2^{-34.213}$
2^{20}	2^{16}	2^{17}	$2^{-41.935}$

At the two smaller ring degrees, rank-zero affine cubes dominate the bound. Their relations vanish in every degree-two feature lift. Increasing the two evaluator widths therefore does not certify 40-bit correctness at those points; a stronger evaluator or a sharper caller-specific argument would be required.

The proof assumes a uniform shuffle, ideal random matrices derived from the seed, uniform Ring-LPN block offsets, and semi-honest execution. It does not analyze malicious bias in joint seed generation. It also does not apply to the AnyField caller, whose point distribution must be counted separately.

B.4 Conditioning on the folded-support filter

The weak-factor filter changes the support distribution, so the preceding bound cannot be applied unchanged. Each party independently resamples its four supports until their total folded support

weight is at least 61. The weight counts occupied residues separately in each polynomial, before coefficient cancellation. For party $p \in \{0,1\}$ and polynomial $a \in [0,P)$, define

$$D_{p,a} := \left|\{(jL + X_{p,a,j}) \bmod 128 : j \in [0,t)\}\right|, \qquad R_p := \left\{ \sum_{a\in[0,P)} D_{p,a} \geq 61 \right\}.$$

The shuffle and evaluator seeds remain independent of this filter. Here 128 divides L, so before rejection the $D_{p,a}$ are independent copies of the number D of occupied bins after 16 uniform draws into 128 bins.

Let $D_1, \ldots, D_4$ be independent copies of D, and put

$$\alpha := \Pr[D_1 + \cdots + D_4 \geq 61], \qquad \beta := \Pr[D_1 + D_2 + D_3 \geq 45].$$

These probabilities follow from an occupancy recurrence. If $u_j(v)$ is the probability of v occupied bins after j draws, then $u_0(0) = 1$ and

$$u_{j+1}(v) = \frac{v}{128} u_j(v) + \frac{129 - v}{128} u_j(v-1),$$

with out-of-range entries zero. Convolving u_{16} gives $\alpha \approx 0.5019144583$ and $\beta \approx 0.7174085984$.

Fix one polynomial's complete offset vector. It has at most 16 occupied residues, so the probability that the other three polynomials complete acceptance is at most β. Bayes' rule therefore bounds the density of this polynomial after local rejection by β/α times its original density. The two parties reject independently. Any nonnegative function of one polynomial from each party and independent setup coins consequently satisfies

$$\mathbb{E}[F \mid R_0 \cap R_1] \leq \left(\frac{\beta}{\alpha}\right)^2 \mathbb{E}[F]. \tag{68}$$

Every product list $A_{a,b,k}$ depends on just such a pair of polynomials. Apply Equation (68) to each list's structural union contribution and sum over all lists. This does not assume that different lists fail independently. The final-compression bound is pointwise in the supports and therefore remains unchanged. We obtain

$$\epsilon_{\mathsf{batch}|R_0\cap R_1} \leq \left(\frac{\beta}{\alpha}\right)^2 \epsilon_{\mathsf{str}} + \epsilon_{\mathsf{cmp}} < 4.739 \cdot 10^{-13} < 2^{-40.940}. \tag{69}$$

Here $(\beta/\alpha)^2 \approx 2.043025290$. Thus the proposed support-only filter preserves 40-bit programmed-system correctness at the stated 2^{20} ring degree, without changing the evaluator widths. This is a correctness bound, not a bound on descriptor leakage. A filter that also tests coefficients, or a different rejection rule, requires its own analysis.

C Reusable DPF Simulation

We first simulate two random-value hybrids of Figures 3 and 5. We then reduce the real protocol to these hybrids under explicit tree-generator and leaf-PRF requirements. The reductions concern the abstract generators in the figures; they do not establish those requirements for a particular AES evaluator. Fix a static semi-honest corrupt party P_b, $b \in \{0,1\}$, and put $h := 1 - b$. Shared selections, products, and digit decomposition are ideal circuit calls. Their fresh secret-output shares are uniform; the simulator computes local operations and repeated uses of those shares consistently.

C.1 Setup with independent honest active children

Define the *independent-child experiment* by changing only the honest party's expansion of an active parent: replace that invocation of G by w fresh independent κ-bit children. Root-child shares are already sampled directly. Keep every corrupt-party evaluation and every off-path expansion unchanged. The active node and the replacements are defined using the hidden index inside the experiment; the simulator is not given that index.

Lemma C.1 (Correction-word distribution)**.** *In the independent-child experiment, conditioned on the corrupt view before each correction, the public correction has the following law, independent of the secret child number:*

1. *For binary branching, sample $a \leftarrow \{0,1\}^{\kappa-1}$ and $\tau_1, \tau_2 \leftarrow \{0,1\}$ independently, and return $(a\|\tau_1, a\|\tau_2)$.*
2. *For $w > 2$, return w independent κ-bit blocks.*

At an active split, the corrected honest active seed has a uniform $(\kappa-1)$-bit prefix conditioned on the corrupt view after the correction and on the secret child number.

Proof. Fix the corrupt pre-correction view and the secret digit $u \in [0, w)$. At an active split, cancellation of off-path parents makes the reconstructed child sums $z_1, \ldots, z_w$ the XORs of the active parent's two sets of children. The honest children are independent uniform blocks, so these sums are independent uniform blocks even after fixing the corrupt children.

For $w = 2$, Figure 2 reveals

$$a = \mathsf{MSBs}(z_{2-u}), \qquad \tau_1 = \mathsf{LSB}(z_1) \oplus u \oplus 1, \qquad \tau_2 = \mathsf{LSB}(z_2) \oplus u.$$

These are independent uniform values. The prefix of z_{1+u} is unused by this disclosure. The corrected honest active prefix is that unused prefix XOR a value fixed by the corrupt children, their parent tag, and a. It therefore remains uniform after the disclosure. The active tag is determined by correctness and is not claimed to be hidden.

For $w > 2$, let $r \leftarrow \{0,1\}^{\kappa-1}$ be the fresh reconstructed mask. The correction is

$$\sigma_j = z_j \oplus [j = 1 + u](r\|1), \qquad j \in [1, w].$$

For every fixed r, this is a bijection on the uniform tuple of child sums. Thus the correction tuple is uniform and independent of r. Conditioned on that tuple, the corrected honest active prefix is r XOR the corrected corrupt active prefix, so it is uniform.

At a sparse binary level with no active split, the reconstructed child sums are zero. Figure 5 replaces them by the independent uniform mask r before computing the correction. The binary correction therefore has the same law at such a level. No active seed is changed there. Ideal circuit-output shares add only independent local shares; deterministic dependencies between local correction components are retained. □

The setup simulator now follows the public schedule. It samples the corrupt root shares and the fresh output shares of ideal circuit calls. At each visited split it samples the public correction according to Lemma C.1. It computes the corrupt party's local correction shares from its simulated state, and supplies their XOR complement as the honest opening message. It then performs the

corrupt party's actual local corrections and generator evaluations. For sparse trees, the visited split bits depend only on S; the correction law does not reveal whether the active path splits at a visited bit.

Induction over this schedule gives the exact corrupt setup-view distribution in the independent-child experiment, for either corruption choice. The simulator saves its computed leaf seeds and counter. It does not need the honest index to select a path or to construct those local seeds.

C.2 Expansion with a fresh honest active value

Fix a setup state satisfying the leaf invariant. For $x \in S$, write $t_p(x) := \mathsf{LSB}([\![s_x]\!]_p)$ and let $u_p(x) \in \mathbb{G}$ denote the signed leaf conversion of party p. Off the active address α, the two tags agree and $u_0(x) + u_1(x) = 0$.

Define the *fresh-active-value experiment* by replacing only the honest conversion $u_h(\alpha)$ in each expansion with a fresh uniform element of $\mathbb{G}$, independent of the history and current inputs. All corrupt conversions are still computed from the saved seeds with the public L_q. Honest off-path conversions remain their negatives. The replacement is a distributional hybrid; it is not an extra protocol step.

Lemma C.2 (Online leaf-update simulation)**.** *In the fresh-active-value experiment, the expansion messages and corrupt output can be simulated exactly from the corrupt local state and payload share. The honest output equals the complement prescribed by $\mathcal{F}_{\mathsf{rDPF}}$. The same simulator works online for polynomially many adaptive expansions without changing the saved seeds.*

Proof. Let $\eta := t_0(\alpha) - t_1(\alpha) \in \{-1, 1\}$, where the tags are interpreted as integers. The correctness calculation gives the opened value

$$\gamma = \eta\big(\beta - u_0(\alpha) - u_1(\alpha)\big).$$

Fix the preceding history, the current inputs, and the corrupt conversions. The remaining honest active value is uniform. Translation and multiplication by either sign are bijections of $\mathbb{G}$, so γ is uniform and independent of that fixed information.

The simulator computes all $u_b(x)$ and $t_b(x)$ from its saved state and samples $\gamma \leftarrow \mathbb{G}$. In characteristic two, the corrupt opening share is the locally computed $\gamma_b := [\![\beta]\!]_b - \sum_{x \in S} u_b(x)$. In other characteristics, the simulator computes the corrupt sign-bit share from its tags, and samples its fresh output share γ_b from the ideal sign-multiplication call. In either case, it supplies $\gamma - \gamma_b$ as the honest opening message. These are the exact opening-share distributions in the stated experiment.

It computes and submits the map

$$y_b(x) := u_b(x) + (1 - 2b)\gamma t_b(x), \qquad x \in S,$$

to $\mathcal{F}_{\mathsf{rDPF}}$. Off α, the two corrected values cancel. At α, their sum is $u_0(\alpha) + u_1(\alpha) + \eta\gamma = \beta$. The honest map is consequently $\mathsf{pt}_S(\alpha, \beta) - y_b$, exactly as returned by the functionality. This proves the joint-output claim. At the next call the simulator retains the same seeds, increments q, and uses fresh simulation randomness. The fresh honest active value has the same conditional law for every ensuing choice of payload; induction proves the adaptive claim. A singleton uses the ideal identity circuit instead: sample its corrupt output share and submit it to the functionality. □

C.3 Generator requirements and reductions

Let H_0 be the real generator-based protocol with ideal shared-circuit calls. Let H_1 make the independent-child replacements during setup, and let H_2 additionally make the fresh-active-value replacements during expansion. The lemmas above give an exact online simulator for H_2. We justify the transitions using the following two primitive requirements.

For the tree generator, define $\mathsf{G}_\theta(k) := \mathsf{G}(k\|\theta)$ for $k \in \{0,1\}^{\kappa-1}$ and $\theta \in \{0,1\}$. The challenge algorithm chooses θ before receiving either $\mathsf{G}_\theta(K)$ for $K \leftarrow \{0,1\}^{\kappa-1}$, or w independent uniform κ-bit blocks. It does not receive K. Let ϵ_{G} bound its distinguishing advantage. Its preceding state is independent of K, and it may evaluate the public generators on inputs of its choice. This requirement includes knowledge of the tag; ordinary PRG security for an unrestricted seed does not imply it.

For leaf conversion, use the PRF option $\mathsf{L}_q(k) := \mathsf{L}(q,k)$ from Section 4. Let $\epsilon_{\mathsf{L}}(Q)$ bound the advantage of distinguishing oracle access to $\mathsf{L}(\cdot,K)$ from a random function into $\mathbb{G}$, for a uniform secret key K and at most Q queries. The algorithm may also evaluate the public family under chosen keys. This is one joint requirement across rounds, not a separate assumption for each q. Both primitive bounds below are evaluated at the runtime of the external distinguisher plus one execution of the protocol and its ideal circuits.

Theorem C.3 (Reusable DPF simulation)**.** *Fix valid public parameters and a nonempty support S. Let a_S bound the number of active-parent calls to G after the directly sampled root children, maximized over $\alpha \in S$. Under the two primitive requirements above, the standard and sparse DPF protocols realize $\mathcal{F}_{\mathsf{rDPF}}$ against static semi-honest corruption for Q polynomially many adaptive expansions, with advantage*

$$\epsilon_{\mathsf{DPF}}(Q) \le a_S\epsilon_{\mathsf{G}} + \epsilon_{\mathsf{L}}(Q) + \epsilon_{\mathsf{circuit}}(Q),$$

where $\epsilon_{\mathsf{circuit}}(Q)$ is the accumulated joint simulation loss of the shared-circuit realizations. For a nontrivial standard tree, $a_S \le D-1$; for a sparse binary tree, $a_S \le \min\{D-1, |S|-2\}$. A singleton needs only the ideal identity circuit and incurs neither generator term.

Proof. First replace active-parent expansions in root-to-leaf order. Consider two adjacent experiments, differing only at the next active parent. All earlier active expansions have independent children. Lemma C.1 makes this parent's honest prefix a uniform secret K, independent of the preceding corrupt view. Its tag θ is fixed by the corrupt tag and the active path.

The reduction runs the external distinguisher and the surrounding execution, including the honest inputs. Unlike the final simulator, it therefore knows the active path. It leaves K implicit in the honest parent state, chooses θ, and uses its challenge as that parent's children. This embedding does not require K: at the preceding correction, the honest active prefix was uniform and unexposed. In the binary case its prefix was unused by the correction; in the wider case the fresh mask made it independent of the correction. Local ideal-output shares can be sampled without exposing that prefix. The parent is used only for this expansion. All off-path seeds are computable from the corrupt state, and all subsequent children are computable from the challenge. Earlier honest parents are not needed again. If the path has no such call, the adjacent experiments coincide.

The reduction follows all later corrections and expansions, and returns the external distinguisher's bit. It obtains honest output maps from correctness and the computed corrupt maps, preserving the joint external execution. A real challenge gives the earlier experiment; a uniform

challenge gives the later experiment. There are at most a_S transitions, each using one challenge, so their total loss is at most $a_S\epsilon_{\mathsf{G}}$.

In H_1, the final honest active prefix K is uniform and independent of the corrupt setup view. For H_1 to H_2, a second reduction leaves that prefix implicit and uses its PRF oracle for every honest active leaf conversion. It generates the setup view by the exact setup simulation above. It computes all corrupt conversions locally; honest off-path conversions are their negatives. On call q, it queries its oracle at q, applies the honest party's sign, and performs the leaf update. The hidden prefix has no other post-setup use. The reduction can generate the opening messages and both output maps without knowing that prefix.

A real PRF oracle gives H_1. A random-function oracle gives H_2, because the stored counter never repeats an input. The reduction makes at most Q queries and returns the external distinguisher's bit. It receives payloads as the execution proceeds, so their dependence on earlier honest outputs does not change the reduction. This transition costs $\epsilon_{\mathsf{L}}(Q)$, with no additional factor of Q. Neither reduction rewinds or aborts. The exact simulator for H_2 and composition with the shared-circuit simulators give the stated bound. □

C.4 Concrete AES evaluators

The theorem does not require fresh public randomness during expansion. A concrete AES instantiation must justify its tree evaluator, stored-seed representation, and leaf evaluator, accounting for forward and inverse queries, collisions, and domain separation.

Existing libOTe evaluator. For dense expansion, we retain an AES-based evaluator from earlier `libOTe` code [PR]. This is an inherited implementation choice, not a new tree-generator design. With public AES key k and seed $s \in \{0,1\}^{128}$, it computes

$$t := \mathsf{AES}_k(s), \qquad \big(\mathsf{AESRound}(t,s), \mathsf{add64x2}(t,s)\big).$$

Here $\mathsf{AESRound}$ is one AES round, including its round-key XOR; $\mathsf{add64x2}$ adds the two 64-bit lanes separately modulo 2^{64}. We conjecture that this optimization preserves DPF privacy. Related work on expand-accumulate codes conjectures punctured-function unpredictability for $(P(s) \oplus s, P(s) + s \bmod 2^\kappa)$ with ideal permutation P [BCG+22, Section 6.2, Conjecture 6.4]. The evaluator's additional AES round is intended to disrupt that candidate's direct bit and carry relations. Neither ideal AES nor the cited analysis proves this evaluator secure or instantiates the theorem above.

For sparse binary expansion, define $H_k(s) := \mathsf{AES}_k(s) \oplus s$ for $s \in \{0,1\}^{128}$. The children are $(H_k(s), H_k(s \oplus 1))$, where 1 is the block whose only set bit is the low bit. The dense kernel exports its corrected seed s through $\mathsf{AESRound}(s,s)$ before the sparse interface. The reusable sparse interface stores a 127-bit seed prefix and a separate tag; leaf conversion pads the prefix with a zero low bit. These representation steps are also part of the concrete instantiation. The rekeying schedules below retain the existing child-generation formulas.

Internal-node rekeying. Setup derives purpose-separated public 128-bit roots from its agreed public coins. Dense and sparse expansion use separate roots, as do Waterfall's column and value-scatter DPFs. Fix one such root R. Write $\langle c, j \rangle$ for a fixed injective 128-bit encoding of a 64-bit

purpose label c and a 64-bit counter j. The labels node, leaf, and next below are distinct. Internal counter position j uses

$$k_j := \mathsf{AES}_R\big(\langle \mathsf{node}, \lfloor j/1024 \rfloor \rangle\big).$$

One parent producing two children consumes one position. Sparse traversal starts at zero and continues across trees and levels, including dense-to-sparse boundary expansions. Dense recomputation restores each node's position in public per-tree, per-level counter ranges. Thus revisiting a node preserves its key. Each chunk contains at most 1,024 distinct parent positions; neither counter assignment depends on the secret active path.

Cached-leaf rekeying. Each cached-leaf stream has a separate public root R_0. Number its leaves $j = 0, 1, \ldots$ in public row order, continuing across row boundaries. Expansion q derives its keys and advances its root by

$$k_{q,j} := \mathsf{AES}_{R_q}\big(\langle \mathsf{leaf}, \lfloor j/1024 \rfloor \rangle\big), \qquad R_{q+1} := \mathsf{AES}_{R_q}(\langle \mathsf{next}, 0 \rangle).$$

For a saved prefix $s \in \{0,1\}^{127}$, conversion applies $H_{k_{q,j}}(s\|0)$ before coefficient conversion and the party's sign. For $\mathbb{G} = \mathbb{Z}_{2^{64}}$, coefficient conversion takes the low 64 bits. Other coefficient contexts use their specified conversion maps. The extra public context j is part of this concrete evaluator; the figures use the abstract notation L_q. The roots advance deterministically, so rekeying adds no DMPF messages or OTs and requires no fresh public coins during expansion. The implementation reuses each expanded AES key across a chunk and retains eight-node batching.

Rekeying is intended to limit same-key collision effects. It does not establish the generator requirements or remove collisions within a chunk. A concrete proof must handle the public schedules jointly with setup openings, dense export, and all leaf conversions. Distinct purpose encodings do not by themselves prove independence across derived roots or expansion rounds.

Disclaimer. Case studies, comparisons, statistics, research and recommendations are provided "AS IS" and intended for informational purposes only and should not be relied upon for operational, marketing, legal, technical, tax, financial or other advice. Visa Inc. neither makes any warranty or representation as to the completeness or accuracy of the information within this document, nor assumes any liability or responsibility that may result from reliance on such information. The Information contained herein is not intended as investment or legal advice, and readers are encouraged to seek the advice of a competent professional where such advice is required.

These materials and best practice recommendations are provided for informational purposes only and should not be relied upon for marketing, legal, regulatory or other advice. Recommended marketing materials should be independently evaluated in light of your specific business needs and any applicable laws and regulations. Visa is not responsible for your use of the marketing materials, best practice recommendations, or other information, including errors of any kind, contained in this document.