Fully Distributed Multi-Point Functions for PCGs and Beyond

Amit Agarwal Srinivasan Raghuraman Peter Rindal

December 19, 2025

Abstract

We introduce new *Distributed Multi-Point Function* (DMPF) constructions that make multi-point sharing as practical as the classic single-point (DPF) case. Our main construction, *Reverse Cuckoo*, replaces the "theoretical" cuckoo insertions approach to DMPFs with a MPC-friendly linear solver that circumvents the concrete inefficiencies. Combined with our new sparse DPF construction, we obtain the first fully distributed and efficient DMPF key generation that avoids trusted dealers and integrates cleanly with standard two-party MPC.

Applied to pseudorandom correlation generators (PCGs), our DMPFs remove the dominant "sum of t DPFs" bottleneck. In Ring-LPN and Stationary-LPN pipelines (Crypto 2020, 2025), this translates to *an order of magnitude more Beaver triples per second* with *an order of magnitude less communication* compared to the status quo by Keller et al (Eurocrypt 2018). The gains persist across fields and rings ($\mathbb{F}_{p^k}$, $\mathbb{Z}_{2^k}$ for $k \geq 1$) and are complementary to existing PCG frameworks: our constructions drop in as a black-box replacement for their sparse multi-point steps, accelerating *all* PCGs that rely on such encodings.

We provide a complete protocol suite (deduplication, hashing, linear solver, sparse DPF instantiation) with a semi-honest security proof via a straight-line simulator that reveals only hash descriptors and aborts with negligible (cuckoo-style) probability. A prototype implementation validates the asymptotics with strong concrete performance improvements.

1 Introduction

Distributed Point Functions (DPFs) have emerged as a core primitive for function secret sharing[BGI15, BGI16], enabling two parties to hold compact keys that expand into additive shares of a unit vector of length N [GI14, BGI16]. This remarkable ability to compress a sparse vector into two short keys has found diverse applications: pseudorandom correlation generators (PCGs) for secure computation [BCG$^+$20, LXYY25b, LXYY25a], efficient two-server private information retrieval (PIR) [DRRT18, MZRA22], private set intersection (PSI) [RR22, RS21, Zha23], oblivious RAM [DS17], and federated analytics [CGB17, DPRS23].

Many such applications require not just a single hidden position, but several positions at once. For example, Ring-LPN based PCGs [BCG$^+$20] or the recent FOLEAGE protocol [BBC$^+$24] rely on compact encodings of sparse structures with Hamming weight t, while batched PIR queries and PSI protocols involve sets of multiple elements. A natural generalization is the *Distributed Multi-Point Function* (DMPF), which shares a vector with up to t non-zero entries. Naïvely, one can run t independent DPFs, but this inflates cost by a factor of t, yielding $O(Nt)$ work instead of the $O(N)$ ideal.

Prior DMPF approaches highlighted two main obstacles. First, efficient *distributed* key generation is difficult: early proposals often assumed that one party learns all indices, or that a trusted dealer provides keys [BGH+25]. Second, optimized cuckoo-hash based schemes [SGRR19] reduced asymptotic overhead but introduced iterative insertion and eviction subroutines. When the indices are secret-shared, such procedures become expensive MPC protocols that dominate concrete performance.

Our contribution. We present new, *fully distributed* DMPF constructions that close this gap and enable scalable, MPC-friendly key generation. Our central tool is the *Reverse Cuckoo* paradigm: instead of fixing hash functions first and searching for placements, we randomly assign items to buckets and then solve for hash functions consistent with this assignment. This inversion eliminates iterative eviction loops and yields a one-shot linear-algebraic subroutine that integrates smoothly into MPC. We also develop complementary bucketing and big-state approaches, and show how all these constructions integrate into PCGs from Ring-LPN and Stationary LPN.

Performance. Our implementation achieves multi-million $\mathbb{F}_q$-OLE/sec online throughput with modest, n-independent per-expansion communication. For the 64 bit "Goldilocks" prime[Ham15, Pol22], $q = 2^{64} - 2^{32} - 1$, generates rates climb from 0.89×10^6 to 1.58×10^6 to 1.81×10^6 OLEs/s as n grows from 2^{16} to 2^{20}, with per-expansion communication ≈ 13 MB and setup times 2.63-12.49 s and 149-170 MB setup communication. For a 31-bit prime *Fp31* is 1.4-1.5$\times$ faster online (e.g., 2.71×10^6 ops/s at $n{=}2^{20}$) and uses slightly less setup bandwidth. Comparing our DMPF with the status-quo "sum of t DPFs," we see over an order-of-magnitude higher throughput (e.g., 885,621 vs. 37,686 OLEs/s at $n{=}2^{16}$, Goldilocks).

Compared to [Kel20], the state of art for Beaver triple generation based on FHE, which achieves 275,548-309,588 OLEs/s at $n{=}2^{20}$, our online throughput is 6-10$\times$ higher with substantially lower per-expansion communication. Moreover, the recent PCG based implementation of [Rie25] using sum of DPFs achieves an lower throughput of just 22,800 OLEs per second, although on different hardware and larger prime. In conclusion, generating Beaver triples is now an order of magnitude more efficient. See Section 11 for a detailed comparison.

2 Preliminaries

2.1 Notation

For integers $m, n \in \mathbb{Z}$, we denote by $[m, n]$ the set of integers $\{m, m+1, m+2, \ldots, n\}$ and $[n]$ be the shorthand for $[1, n]$. We use κ to denote the statistical security parameter, e.g., $\lambda = 40$. Parties are denoted as P_0, P_1. We use $x := 3$ to assign the variable x the value 3. We use $x = 3$ to denote mathematical equality or in the case of an alogirthm the predicate that is 1 if x has value 3 and 0 otherwise.

We denote by $[\![x]\!]$ a secret sharing of the value x, and by $[\![x]\!]_i$ the share held by party P_i. We will assume that shares are either binary, meaning $x = [\![x]\!]_0 \oplus [\![x]\!]_1$, or arithmatic in the ring $\mathbb{Z}_{2^n}$, meaning $x = [\![x]\!]_0 + [\![x]\!]_1 \in \mathbb{Z}_{2^n}$. We note that other sharing types are possible. When ambigous, we use the notation $[\![x]\!] \in \mathbb{G}$ to denote that the value is shared over the group $\mathbb{G}$.

2.2 Functions

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

- $\mathsf{decomposition}_w : \mathbb{G} \to \mathbb{F}_w^n$, where $n := \log_w(|\mathbb{G}|)$, returns the base-$w$ digit vector $b = (b_1, \ldots, b_n)$ such that $a = \sum_{i=1}^n w^{i-1} b_i$. (Assume $w \geq 2$ and that $|\mathbb{G}| = w^n$ or we fix a canonical embedding of $\mathbb{G}$ into $\mathbb{Z}_{w^n}$.)

- $\mathsf{unitVector}_S : (\alpha \in S,\ \beta \in \mathbb{G}) \to \mathbb{G}^S$, where $v = \mathsf{unitVector}_S(\alpha, \beta)$ satisfies $v_\alpha = \beta$ and $v_j = 0$ for $j \in S \setminus \{\alpha\}$.

- $\mathsf{append} : X^n \times X^m \to X^{n+m}$, concatenation: $\mathsf{append}((x_1, \ldots, x_n), (y_1, \ldots, y_m)) = (x_1, \ldots, x_n, y_1, \ldots, y_m)$. We also use the notation $||$ to have the same meaning but as an infix operator.

- $\mathsf{sort} : X^n \times \{\text{total order } \prec\} \to (X^n, \pi)$. 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 $(x_{\pi(1)}, \ldots, x_{\pi(n)})$. In MPC we implement this as a data-oblivious random permutation (coin-tossing to agree on π).

- $\bowtie : X_1^n \times \cdots \times X_k^n \to (X_1 \times \cdots \times X_k)^n$, 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 $\diamondsuit$ 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.

- $\mathsf{G} : \{0,1\}^\kappa \to (\{0,1\}^\kappa)^b \times \{0,1\}^\sigma$ is the PRG used in the DPF tree. For branching factor b (binary b=2 in our base construction), $\mathsf{G}(s)$ outputs the b child seeds $(s_1, \ldots, s_b)$ and auxiliary bits/blocks $\xi \in \{0,1\}^\sigma$ (used for masks/correction words). We write $(s_1, \ldots, s_b, \xi) \leftarrow \mathsf{G}(s)$ and apply this independently at each internal node during expansion.

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 were explicitly identified and optimized in recent work, with cuckoo-style schemes and big-state variants providing sublinear domains for sub-instances [BGH+25]. Our Reverse Cuckoo turns the usual table-building on its head: we fix a target bin assignment first and then solve (and reveal) small linear systems that act as hash function descriptors; this avoids iterative evictions in MPC while preserving cuckoo-style load properties. We also give bucketing and big-state keygen paths that trade structure for very simple distributed key generation.

PCGs from LPN/Ring-LPN and beyond. PCGs unify many preprocessing correlations for MPC. 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]. 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 distributed DMPFs replace the status-quo "sum of t DPFs," eliminating the $O(\kappa)$ multiplicative overhead per correlation and aligning Ring-LPN PCG costs with the OT/VOLE case.

Implementation of PCGs. The Silentium system [Rie25] provides the first large-scale implementation of the Bt-PCG for Beaver triples, demonstrating feasibility of the Boyle et al. design and showing concrete speedups ($\approx 34\%$ over MP-SPDZ LowGear) together with dramatic communication savings. Silentium focuses on engineering challenges: synchronizing SUV generation, optimizing 128-bit NTTs of degree 2^{20}, and parallelizing PRG invocations. In contrast, our work advances the algorithmic foundation: DMPFs accelerate the sparse encodings underlying *all* PCGs, not just Bt-PCG, yielding order-of-magnitude gains both in throughput and in communication. Whereas Silentium demonstrates practicality of an existing construction, our results redefine the underlying primitives so that PCGs across domains inherit these improvements.

Cuckoo hashing and stash bounds. We rely on classical cuckoo hashing analysis and its "stash" refinements to argue negligible failure probability with small constant stash; these also justify our use of w independent systems and the permutation-consistency viewpoint [PR04, KMW10, ADW12]. For set-based cryptographic protocols (e.g., PSI), cuckoo hashing is now standard, and circuit/OPPRF-based PSI variants analyze similar load/failure profiles that our Reverse Cuckoo mirrors while removing kick-out dynamics in MPC [PSWW18, DRRT18, YY25].

3 Technical Overview

This section formalizes our DMPF target, recalls the DPF subroutine we build on, and then presents the three constructions we implement—*Cuckoo baseline*, *Reverse Cuckoo*, and *Bucketing*—together with the big-state keygen variants. We finish by detailing how these plug into Ring-LPN[BCG+20] and Stationary-LPN PCGs[KPRR25].

Target functionality and constraints. Given secret-shared index-payload pairs $(\llbracket A_1 \rrbracket, \ldots, \llbracket A_t \rrbracket) \in [0, N)^t$ and $(\llbracket B_1 \rrbracket, \ldots, \llbracket B_t \rrbracket) \in \mathbb{G}^t$, a Distributed Multi-Point Function (DMPF) outputs additive shares of

$$y \;=\; \sum_{i=1}^{t} \mathsf{unitVector}_{[0,N)}(A_i, B_i) \;\in\; \mathbb{G}^N,$$

with $\mathsf{unitVector}$ being defined to return a unit vector with value B_i at index A_i. Note that we require correctness under collisions ("equal A_i sum their B_i") and with no leakage beyond what is inherent in the functionality. The efficiency goals are: near-linear $O(N)$ expansion work, $O(t \log N)$ communication for key-generation, and oblivious deduplication.

DPF recap. The simplest case of a Distributed Point Function[GI14, BGI16, DS17] is when $t = 1$: we want to share a unit vector of length N with a single non-zero entry at position α. Classical DPF constructions realize this using a binary tree of depth $\log N$. Each party holds a short seed, and by expanding a PRG level by level down the tree they generate all N leaves. At each level, a small *correction word* ensures that the parties' expansions remain consistent while only one path — the one leading to the secret index α — contributes a non-zero value β at the leaf. All other leaves cancel to 0 when the two parties' outputs are added together. See Section 4.

Trivial baseline: sum of point functions. The most straightforward way to realize a Distributed Multi-Point Function with t active positions is simply to instantiate t independent DPFs, one for each (A_i, B_i). Each party receives t short keys, and upon expansion they obtain t length-N vectors that are added together to form the final output:

$$y \;=\; \sum_{i=1}^{t} \mathsf{DPF}(A_i, B_i).$$

This construction is functionally correct and requires no additional machinery. Its drawback is efficiency: expansion time, key size, and communication all scale linearly in t. Concretely, the work is $O(Nt)$, which is prohibitive in regimes where N is large and t grows with the security parameter or application size. Nonetheless, this "trivial sum" forms a useful conceptual baseline.

New Sparse DPF As a building block for our next construction, we generalize a basic $t = 1$ DPF to a *sparse DPF*. Here, the domain is restricted to a public subset $S \subseteq [0, N)$ of size n, and only leaves indexed by S are materialized. Conceptually, the DPF still follows the same tree expansion, but it prunes away all irrelevant branches. The resulting expansion cost is $O(n)$ rather than $O(N)$, even when $N \gg n$.

Cuckoo Baseline. A natural way[SGRR19] to extend from $t = 1$ to $t > 1$ is to try to "pack" the t active positions into a small number of sub-instances of a DPF[PR04, SGRR19]. Concretely, suppose we allow w candidate hash functions $h_1, \ldots, h_w : [0, N) \to [m]$ and choose $m \approx (1 + \varepsilon)t$ bins. Each input point (A_i, B_i) is assigned to one bin by evaluating $h_j(A_i)$ for some $j \in [w]$. Each bin corresponds to a *sub-DPF* whose domain is the set of positions that could hash there. Because these sets are much smaller than $[0, N)$, each sub-DPF can be realized as a *sparse DPF* over its valid subset. The final DMPF output is obtained by expanding all m sub-DPFs and summing their contributions into the global length-N vector.

This design is attractive in theory: it reduces the number of leaves each DPF must cover, and with proper load balancing, the total work is nearly $O(N)$ rather than $O(Nt)$. The assignment step—choosing which h_j each point uses—is exactly the same as building a cuckoo hash table[PR04], where each item has w possible slots and we must find a consistent placement.

The difficulty arises when the inputs (A_i, B_i) are secret-shared. To maintain privacy, the parties must run the cuckoo hashing *inside* MPC: for each element, evaluate its w candidate positions, attempt insertion, and if a collision occurs, evict an existing item and try its next candidate slot. This iterative "kick-out" process is simple in the clear, but under MPC it requires a complex sequence of oblivious comparisons, swaps, and table updates. The depth of this procedure makes it expensive to implement securely, and the randomness of eviction cascades is particularly ill-suited to efficient circuit or OT-based MPC. We attempted to optimize this step but were not satified.

For this reasons, although the cuckoo-baseline approach achieves the right asymptotics, in practice its distributed key-generation cost is dominated by the MPC required to realize cuckoo hashing and address translation. This motivates our alternative constructions that avoid iterative eviction entirely.

3.1 Reverse Cuckoo DMPF (Main Construction)

The key idea of Reverse Cuckoo is to avoid the iterative "kick-out" process of normal cuckoo hashing. Instead of fixing hash functions and searching for a valid placement, we reverse the process: first assign each input to a bin, and then solve for hash functions consistent with that assignment. Let us warmup with the (insecure) case $w = 1$. The construction proceeds in five steps:

1. **Assign to bins.** Each input (A_i, B_i) is padded and permuted into an array of length $m = (1+\varepsilon)t$. The resulting indices A'_i represent the target bin assignment for each point.

2. **Solve for hash functions.** We now construct hash functions that reproduce this assignment.

   2.a) Assign every position $j \in [0, N)$ a random binary feature vector $h_j \in \{0, 1\}^q$.

   2.b) Collect the the "active" vector h_{A_i} into a matrix H and solve for a *random* solution to the linear system
   $$H \cdot s = A',$$
   so that applying s to h_{A_i} yields the designated bin A'_i. This is done in MPC using our bin-solver protocol. The solution s is secret-shared until the next step.

3. **Reveal s.** Once a consistent solution is found, the parties reveal s. This makes the resulting hash family public, while inputs A, B (hopefully) remain hidden.

4. **Compute the position map.** Using the revealed s, we define a public position matrix $P \in \{0,1\}^{N \times m}$ that maps each global index in $[0, N)$ to exactly one of the m bins, i.e. $P_{j,i} = 1$ means position $j \in [0, N)$ is mapped to bin $i \in [m]$.

5. **Sparse DPF instantiations.** Finally, for each bin $j \in [m]$, the parties run a *sparse DPF* over the small domain $S_j = \{i \in [0, N) : P_{i,j} = 1\}$ with input $(\llbracket A'_j \rrbracket, \llbracket B'_j \rrbracket)$. The m sparse vectors are then summed to produce the final output y.

Security considerations. The outline above—solving a single global system—would leak too much. In particular, revealing a single s exposes correlations between all indices A'_i, leaking more than the ideal functionality. To repair this, we partition into w independent systems, each of size $d = m/w$. Each block has its own feature matrix and solution s_ℓ. Revealing s_ℓ gives only the hash family for that block, and the overall assignment is consistent with a valid cuckoo hashing placement except with the negligible cuckoo failure probability. Thus, for every revealed s_ℓ, there exists a valid permutation of the inputs into bins, and the scheme inherits the standard security guarantees of cuckoo hashing in MPC.

Costs and pipeline shape. The Reverse Cuckoo setup consists of: (i) deduplication, (ii) w linear solves on $d \times q$ binary matrices, (iii) computing the sparse sets S_j, and (iv) m sparse DPF evaluations. Online expansion then requires only local DPF leaf work and vector summations, achieving essentially $O(N + \mathsf{poly}(t))$. The critical property is that the N terms is independent of t and that the main keygen cost is solving a system for which we give an optimized construction.

3.2 Bucketing DMPF (uniform-key regime)

When keys are uniform (or close), we show that one can partition the inputs into $k \approx t/\log t$ buckets of size $M = N/k$ so that Chernoff bounds keep per-bucket load to $O(\log t)$ w.h.p. We then run one DMPF per bucket over domain M and concatenate. Both compute and communication are $O(t)$ overall (bucketing, grouping/reindexing, and concat are linear). This is particularly attractive as a black-box DMPF when uniformity (or light skew) holds. We formalize the $\mathsf{Group\text{-}By}$ subroutine (frequency counts $\llbracket f_i \rrbracket$ and padded grouping) and its realizations via DPFs or binary AND-trees.

3.3 Big-State DMPF (two keygen paths)

In the big-state paradigm of Boyle et al. [BGH+25], we give two key-generation methods: (1) a black-box route inspired by Reverse Cuckoo that reuses the linear-algebraic assignment to precompute the state structure, and (2) a very simple generic-MPC path leveraging AltMod-style PRGs to distribute the otherwise non-distributed keygen. Both aim to minimize distributed key-generation cost while preserving the asymptotic/structural benefits of big-state.

3.4 Integration into Ring-LPN and Stationary-LPN PCGs

Why DMPFs matter here. To motivate our application domain, recall that the current status quo for Ring-LPN based PCGs is to fall back to the trivial "sum of DPFs" approach[BCG+20, BBC+24, LXYY25a, LXYY25b]. Concretely, generating n correlations requires instantiating $t = O(\kappa)$ DPFs for each correlation, resulting in $O(n \cdot \kappa)$ calls to a PRF. This term dominates the runtime in existing implementations [PR], and represents the main efficiency barrier to scaling Ring-LPN PCGs.

Where the κ overhead comes from. The extra factor of κ is not an artifact of poor implementation but inherent to the structure of Ring-LPN. At a high level, we begin with $t = O(\kappa)$ sparse polynomials that define the error distribution. What we actually need to share are the supports of their product. Naïvely multiplying two t-sparse polynomials produces a t^2-sparse product. If we then apply "regular noise"—the standard trick of spreading non-zeros more uniformly—we can reduce this back down by a factor of t, leaving an $O(Nt)$-sparse structure.

For OT and VOLE correlations, this regularization trick is enough to reach $O(N)$ complexity: the initial support size is t, and regular noise reduces the work by exactly a factor of t. In contrast, for Ring-LPN we start with t^2 non-zeros, so regular noise only shaves off one t, leaving us with $O(Nt)$ work. Thus the core bottleneck is the need to efficiently compress and share a *multi-point* structure of size t for each correlation.

Our contribution. This is exactly where our DMPF constructions enter. By providing an efficient, fully distributed way to realize a sparse multi-point function, we eliminate the need to instantiate t independent DPFs per correlation[BCG+20, BBC+24, LXYY25a, LXYY25b]. In effect, our DMPFs collapse the $O(n \cdot \kappa)$ PRF calls of the naïve approach down to $O(n)$, finally putting Ring-LPN based PCGs on the same asymptotic footing as PCGs for OT and VOLE.

Stationary LPN. The advantage is even greater in the stationary-LPN regime[], where the support of the sparse error polynomials remains fixed across many expansions and only the coefficients change. Here the heavy distributed key-generation for the DMPF can be done once and amortized across arbitrarily many expansions. Each subsequent expansion then costs only $O(N)$ local PRG work, making the scheme extremely competitive in practice.

3.5 Complexity and Empirics (at a glance)

Reverse Cuckoo setup cost is: one dedup; w linear solves over $d \times q$; sparse-set computation; and m sparse-DPFs. Online expand is a handful of ms and stable across n, t; in measured runs, setup dominates (solve/sparse-sets/sparse-DPF roughly split the time budget), and median per-expansion throughput scales to multi-Mops/s across $n \in \{2^{16}, 2^{18}, 2^{20}\}$ and both 64-bit Goldilocks and 31-bit primes. Full tables and breakdown plots appear in §11.1 and §11.2.

4 Single Point DPF Constructions

We now review the standard DPF construction first presented in [GI14, BGI16, DS17] generalize this protocol to allow efficient expansion over a sparse domain. The ideal functionality we target is presented in Figure 1 which differs from prior works in several ways. Unlike some prior works we do not separate the DPF protocol into an key generation phase and then a non-interactive expansion phase. This is to bring the presentation of the paper in line with practice. In particular, by combining the two phases we eliminate the need to perform the tree expansion twice. The input to the functionality is a secret shared index $\alpha\alpha$ and a shared value $[\![\beta]\!]$. The output is a secret

For public parameters $N \in \mathbb{N}$, domain $S \subseteq [0, N)$ and group $\mathbb{G}$.

On input $[\![\alpha]\!] \in [0, N), [\![\beta]\!] \in \mathbb{G}$, computes:

1. Let $y := \mathsf{unitVector}_{\mathbb{G}^S}(\lfloor \alpha \rfloor_S, \beta)$.
2. Sample $[\![y]\!]$.
3. Output $[\![y]\!]_i$ to party i.

Figure 1: (Single-Point) DPF Functionality over domain $S \subseteq [0, N)$ for some $N \in \mathbb{N}$.

shared vector y such that at index α the vector has value β while all other positions are zero, i.e. $y_\alpha = \beta$.

Additionally, we allow the domain of the point function to be over some possibly sparse set $S \subseteq [0, N)$. The output vector y is then defined at each position y_i for $i \in S$. [GI14, BGI16, DS17] only considered the setting with $S = [0, 2^D]$. However, we will required this generalized definition.

4.1 Standard DPF Protocol

In this section we recall the classic tree–based Distributed Point Function (DPF) construction of Gilboa–Ishai [GI14] (and its subsequent refinements [BGI16, DS17]) and explain how the generalised w-ary version given in Figure 3 realises the ideal functionality of Figure 1. This will be instructive for our new "sparse" version of the protocol presented in Section 4.2. For clarity we first focus on the binary case $w = 2$ and then note the (minor) changes required for larger branching factors.

High-level intuition. Each party locally expands a sequence of PRG seeds along the unique root-to-leaf path that leads to the index α. The binary tree has 2^D leaves. The tree has the invariant:

> The d'th level of the tree is an XOR secret sharing of a unit vector $s = \mathsf{unitVector}(\alpha', r_d)$ and $t := \mathsf{unitVector}(\alpha', 1)$ where s_i, t_i is associated with node i, $r_d \in \mathbb{F}_{2^\kappa}$ is some random value and $\alpha' := \mathsf{prefix}(\alpha, d)$ is the d most significant bits of α, i.e. $\mathsf{prefix}(\alpha, d) := \lfloor \alpha / 2^{D-d} \rfloor$.

The root level containing a single node can be set to a random sharing. Each level d after that is constructed by the parties locally applying a PRG to their local share at each node. This results in two new shares for the children of the node. For nodes that were a sharing of zero, the children nodes will also be sharing of zero, i.e. for any $[\![z]\!]$ with $z = 0$ it holds that $[\![z']\!]_0 := \mathsf{G}([\![z]\!]_0), [\![z']\!]_1 := \mathsf{G}([\![z]\!]_1)$ is also a sharing of zero.

For the "active" node at index $\alpha' := \mathsf{prefix}(\alpha, d)$, the parties will hold $[\![s_{\alpha'}]\!]$ with $s_{\alpha'}$ being some random value. Party P_i will compute

$$([\![s'_{1+2\alpha'}]\!]_i, [\![s'_{2+2\alpha'}]\!]_i) := \mathsf{G}([\![s_{1+\alpha'}]\!]_i)$$

where s' is the sharing for the next level. Unfortunently, the fact that both children are random breaks the invariant that s' is a unit vector with non-zero at index $\mathsf{prefix}(\alpha, d+1)$. To restore the invariant, the parties must then obliviously update their sharing such that the only the child at index $\mathsf{prefix}(\alpha, d+1)$ is non-zero. This is acheived with so called *correction words* (the vectors σ_d, τ_d

in Figure 3) and with the help of the t vector which is 1 at index α'. The remarkable fact of this protocol is that it only requires $O(1)$ communication per level, independent of the size of the level.

Detailed walk-through. Figure 3 performs the following steps. We will omit the root node of the tree and begin with the next level which we label as the D'th level. The leaf level of the tree will be referred to the 1'st level. We will use zero indexing for the s vector.

1. **Initial seed.** Each party samples two κ-bit seed share $[\![s]\!] \in \{0,1\}^{2\times\kappa}$ which forms the nodes for the D'th level. Each level is first constructed with two sibling nodes having random s values and then we will correct the sibling node not on the α path to be zero.

   The current level will be stored in the $[\![u]\!]$ tuple consisting of the shared unit vector $[\![s]\!]$ and a shared "tag" vector $[\![t]\!]$. $t_i = 1$ if s_i is current a random value and otherwise $t_i = 0$.

2. **Path decomposition.** The shared index is written in base 2: $([\![\alpha_D]\!], \dots, [\![\alpha_1]\!]) := \mathsf{decomposition}_2([\![\alpha]\!])$

3. **Depth iteration.** For each depth $d = D, \dots, 2$:

   (a) *Left Right Sums.* The parties compute the sum of all left children $[\![z_1]\!]$ and the sum of the right children $[\![z_2]\!]$ for the current level, i.e. $[\![z_1]\!] := \oplus_i [\![s_{2i}]\!], [\![z_2]\!] := \oplus_i [\![s_{2i+1}]\!]$.

   (b) *Reveal correction words.* Let $\alpha' = \mathsf{composition}_2(\alpha_D, ..., \alpha_d)$ be the current prefix of α. The parties want to update the value of $[\![s_{\alpha'\oplus 1}]\!]$ to be zero without revealing α. First observe that $s_{\alpha'\oplus 1} = z_{1+(\alpha_d \oplus 1)}$. The core idea is to compute $[\![\sigma]\!] := [\![z_{1+([\![\alpha_d]\!]\oplus 1)}]\!]$ using a pair of oblivious transfers to selectively reveal either $[\![z_1]\!]$ or $[\![z_2]\!]$ depending on $[\![\alpha_d]\!]$. The parties then reveal σ and update their shares approperately.

   (c) *Updating the shares.* To update the shares, the core idea is to have the parties add $[\![t_i]\!] \cdot \sigma$ into $[\![s_i]\!]$. Recall that $t_i = 1$ for all s_i that are not zero, i.e. $i \in \{\alpha', \alpha' \oplus 1\}$. This will result in only $[\![s_{\alpha'}]\!]$ having a random value while all others are zero and therefore the invariant is restored.

   (d) *Expanding the next level.* Party P_p then locally apply a PRG to each of their share $[\![s_i]\!]_p$ to obtain two new shares

   $$([\![s'_{2i}]\!]_p, [\![s'_{2i+1}]\!]_p) := \mathsf{G}([\![s_i]\!]_p)$$

   The new shared vector $[\![s']\!]$ almost follows the invariant except that again, the two siblings on the active path are non-zero instead of just the one on the active path. The invariant is the restored on the next iteration.

   (e) *Updating the tags.* Although the s shares have been fixed and used to expand the next level, the parties also need to compute the tag shares for the next level. The existing tag t is one at both index α' as well as $\alpha \oplus 1$. The idea is that we will choose the value of σ such that the least significant bit of s_α to be 1 and then define the next tag bits $[\![t'_{2i}]\!] := [\![t'_{2i+1}]\!] := \mathsf{LSB}([\![s_i]\!])$.

   However, as describe this is not possible as the value of σ is already fully determined if we want $s_{\alpha\oplus 1} = 0$. Therefore the full protocol is to reveal two values of σ, one for the left children σ_1 and one for the right children σ_2. They will share the same first $\kappa - 1$ bits but the LSB of each is chosen such that $s_{\alpha\oplus 1} = 0$ and $s_\alpha = 1$.

   Note that this effectively reveals the least signficiant bit of the input to the PRG G and therefore reduced the security level by one bit. However, due to implementation considerations this reduction of security is considered worth while. Alternatively, one could define G to output $2\kappa + 2$ bits to compensate.

(f) *Next iteration* The parties then use $[\![s']\!], [\![t']\!]$ as the current value for the next next iteration, i.e. $([\![s]\!], [\![t]\!]) := [\![(]\!]s', [\![t']\!])$.

4. **Leaf processing ($d = 1$).** The parties repeat the steps again one more time for $d = 1$ except that the shares are not expanded using the PRG. For every leaf index i the parties now hold a XOR shared value $[\![s_i]\!]$ as well as the tag $[\![t_i]\!]$ such that $s = \mathsf{unitVector}(\alpha, r), t = \mathsf{unitVector}(\alpha, 1)$, for some $r \in \{0,1\}^{\kappa}$. The leaf seeds must then be converted into the desired group $\mathbb{G}$. This is acheived using the the $\mathsf{convert} : \{0,1\}^{\kappa-1} \to \mathbb{G}$ user defined function which samples pseudorandom element of $\mathbb{G}$ given a uniform input. As discussed above, the LSB of s_α is forced to be 1 and therefore is not safe to input into $\mathsf{convert}$. Therefore we define the converted leaf shares as $[\![y_i]\!] := \mathsf{convert}_{\mathbb{G}}(\mathsf{MSBs}([\![s_i]\!]))$ where MSBs returns all but the least significant bit. Finally, if $\mathbb{G}$ is not charactuistic 2, it is neccessare for one party to negate their values of y to ensure additive reconstruction.

5. **Leaf Correction.** The final step is to reveal the difference between the sum of the leaf values and the desired value β. For characturistic 2 groups this is as simple as revealing $\gamma = \beta \oplus \bigoplus_i y_i$ and then adding $[\![t_i]\!] \cdot \gamma$ into each output share $[\![y_i]\!]$. For non-characturistic 2 groups the parties must conditionally negate γ and update y_i appropriately.

Generalisation to $w > 2$. For larger branching factors the only substantive change is that σ becomes length w and each σ_i must appear uniform instead of having a common $\kappa - 1$ prefix. The complexity therefore scales linearly with w. A ternary choice $w = 3$ has been advocated in recent work (i.e., Foleage [BBC+24]) where it works nicely with their base 3 indexing required for thier PCG.

4.2 Sparse DPF

The standard DPF above expands a *full* depth-D tree with 2^D leaves, paying work proportional to the domain size even when the target set $S \subseteq [0, 2^D)$ contains only a handful of indices. When $|S| \ll 2^D$ we observe that it is possible to substantually optimize the construction. Our *sparse DPF* achieves this by pruning every internal node whose entire sub-tree contains at most one live leaf; in our new construction such nodes carry the same PRG seed to their unique child and need *no* correction words.

Fix a depth D so that every index in S has a D-bit binary representation. We conceptually embed S in the complete binary tree $\{0,1\}^D$ but logically constract all nodes with one child. Hence all internal nodes have exactly two children but these children might reside at more than one level down.

To efficiently expand such a sparse tree, it is important not explicitly represent nodes with a single child. Instead, when a node is expanded, we must be able to determine the next level that each of the children will be expanded or if it is a leaf node. To keep tract of the state of the tree, we use the u_d vector which stores all shares that need to be expanded at level d. As before, level D will be the closest to the root while level 1 will be the closest to the leaves.

For each level, u_d will be a list of tuples $(j, \rho, b, [\![s]\!], [\![t]\!])$ where

```
$\mathsf{correctionWord}([\![z]\!] \in \{0,1\}^{w\times\kappa}, [\![\alpha]\!] \in \{0,1\})$:

  1. if $w = 2$: $[\![\sigma]\!] := \bowtie(\mathsf{MSBs}([\![z_{[\![\alpha_d]\!]\oplus 1}]\!]) \cdot (1,1), (\mathsf{LSB}([\![z_1]\!]) \oplus [\![\alpha_d]\!] \oplus 1, \mathsf{LSB}([\![z_2]\!]) \oplus [\![\alpha_d]\!])) \in \{0,1\}^{2\times\kappa}$,
     else: $[\![\sigma]\!] := [\![z]\!] \oplus \mathsf{unitVector}([\![\alpha_d]\!], [\![r]\!] \mid\mid 1) \in \{0,1\}^{w\times\kappa}$ where $[\![r]\!] \leftarrow \{0,1\}^{\kappa-1}$
  2. return $\mathsf{reveal}([\![\sigma]\!])$


$\mathsf{updateLeaves}_S([\![y]\!] \in \mathbb{G}^S, [\![t]\!] \in \{0,1\}^S, [\![\beta]\!] \in \mathbb{G})$:

  1. $[\![\gamma]\!] := [\![\beta]\!] - \sum_{i\in S}[\![y_i]\!]$,
  2. if $\mathbb{G}$ is not characteristic 2:
     (a) $c_p := \sum_{i\in S}[\![t_i]\!]_p \in [0, |S|)$
     (b) $[\![d]\!]_p := c_p/2 + pc_p \mod 2$
     (c) $[\![\gamma]\!] := (1 - 2[\![d]\!]) \cdot [\![\gamma]\!]$
  3. $\gamma = \mathsf{reveal}([\![\gamma]\!])$
  4. $[\![y_i]\!] := [\![y_i]\!] + (1-2p)\gamma[\![t_i]\!]$ for $i \in S$
  5. return $\mathsf{reveal}([\![y]\!])$
```

Figure 2: correctionWord,

- $j \in \{0,1\}$ denote if the current node is a left or right child.
- $\rho \in [D]$ denotes the level of the parent.
- $b \in [|S|]^2$ denote the index range that this node corresponds to, i.e. $S_{[b_1,b_2]}$.
- $[\![s]\!] \in \{0,1\}^k$ is the current node value.
- $[\![t]\!] \in \{0,1\}$ is the tag which is 1 when s is set to random.

The procedure `partition` (Figure 4) takes an index range $\beta \in [|S|]^2$ and returns the highest bit index $\delta \in [D]$ that still splits the current interval of leaves into a non-empty left and right part, together with those two index ranges l, r such that $[l_1, l_2]||[r_1, r_2] = [\beta_1, \beta_2]$. Given the range β this function tells us that this range needs to be split at level δ with subranges $[l_1, l_2], [r_1, r_2]$ for the left and right children. This will enable the expansion to prune all internal nodes that have only one child.

The algorithm can be read in five logical phases. For clarity we refer to the line numbers that appear in Figure 5.

- **Book-keeping initialisation (steps 1,2).** Each depth $d \in [0, D]$ stores
  - a bucket u_d that will hold the *live* seeds on that level,
  - a running sum $z_{d,1}$ and $z_{d,2}$ of all left and right s values at level d

  All structures start empty/zero and are added to in a breadth first manner.

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

On input $[\![\alpha]\!] \in [0,n), [\![\beta]\!] \in \mathbb{G}$, party $p \in \{0,1\}$ computes:
  1. $[\![s]\!] \leftarrow \{0,1\}^{w\times\kappa}, [\![z]\!] := [\![s]\!]$
  2. $[\![u]\!] := \bowtie([\![s]\!], (1)^w), [\![u']\!] := ()$
  3. $([\![\alpha_D]\!], ..., [\![\alpha_1]\!]) := \mathsf{decomposition}_w([\![\alpha]\!]) \in \mathbb{Z}_w^D$.
  4. for $d \in [D, 1]$:
     (a) $\sigma := \mathsf{correctionWord}([\![z]\!], [\![\alpha_d]\!])$
     (b) $[\![z]\!] := (0)^w$
     (c) if $d > 1$: for $(j, ([\![s]\!], [\![t]\!])) \in \bowtie((1,2,1,2,...), [\![u]\!])$:
         i. $[\![s]\!] := [\![s]\!] \oplus [\![t]\!] \cdot \sigma_j$
         ii. $[\![s']\!]_p := \mathsf{G}([\![s]\!]_p)$
         iii. $[\![z]\!] := [\![z]\!] \oplus [\![s']\!]$
         iv. $[\![u']\!] := [\![u']\!] \mid\mid \bowtie([\![s']\!], \mathsf{LSB}([\![s]\!]) \cdot (1)^w))$
     (d) if $d = 1$: for $(i, j, ([\![s]\!], [\![t']\!])) \in \bowtie([0,n), (1,2,1,2,...), [\![u]\!])$ :
         i. $[\![s]\!] := [\![s]\!] \oplus [\![t']\!] \cdot \sigma_j$
         ii. $[\![t_i]\!] := \mathsf{LSB}([\![s]\!])$
         iii. $[\![y_i]\!]_p := (1-2p) \cdot \mathsf{convert}_{\mathbb{G}}(\mathsf{MSBs}([\![s]\!]_p))$
     (e) $[\![u]\!] := [\![u']\!], [\![u']\!] := ()$.
  5. return $\mathsf{updateLeaves}_{[0,n)}([\![y]\!], [\![t]\!], [\![\beta]\!])$
```

Figure 3: w-arity (Single-Point) Distributed Point Function (DPF) protocol realizing Figure 1.

```
$\mathsf{partition}(\beta \in \mathbb{N}^2, S \subset \mathbb{N})$:
  1. if $\beta_1 = \beta_2$, return $(0, (\beta, \perp))$.
  2. let $\delta := \mathsf{max}\left(\{d \mid \{s_d \mid s \in \{S_{\beta_1}, ..., S_{\beta_2}\}, (s_D, ...., s_1) := \mathsf{bitDecomposition}(s)\} = \{0,1\}\}\right)$.
  3. let $b \in \mathbb{N}^{2\times 2}$ s.t. $b_1 := (\beta_1, \beta_1 + m), b_2 := (\beta_1 + m + 1, \beta_2)$ where $m := |\{s \mid s_\delta = 0\}|$.
  4. return $(\delta, b)$.
```

Figure 4: partition, returns the largest bit index δ that partitions the subset $\{S_{\beta_1}, ..., S_{\beta_2}\}$ into two parts, along with the partitions intervals b_1, b_2. If no parition exists, $\delta = 0, b = (\beta, \perp)$. For example, $\beta = (2,7), S = \{0001, 0010, 0100, 0101, 1000, 1001, 1010, 1101, 1111\}$ results in $(\delta, b) = (4, ((2,4),(5,7)))$ representing the ranges $\{0010, 0100, 0101\}, \{1000, 1001, 1010\}$.

```
For public parameters $D \in \mathbb{N}, S \subseteq [0, 2^D)$ and group $\mathbb{G}$ for $|S| > 1$.

$\mathsf{sparse\text{-}DPF}([\![\alpha]\!] \in [0, 2^D), [\![\beta]\!] \in \mathbb{G})$ : Party $p \in \{0,1\}$ computes:
  1. $u_i := \emptyset, [\![z_i]\!] := \{0\}^2, [\![v_i]\!] = 1$ for $i \in [0, D]$.
  2. $S := \mathsf{sort}(S)$
  3. $(\delta, b) := \mathsf{partition}((1, |S|), S)$
  4. for $j \in \{0,1\}$:
      (a) $(\delta', b') := \mathsf{partition}(b_j, S)$.
      (b) $[\![s]\!] \leftarrow \{0,1\}^\kappa, [\![z_{\delta,j}]\!] := [\![s]\!], [\![v_\delta]\!] = 1$
      (c) $u_{\delta'} := \mathsf{append}(u_{\delta'}, (j, \delta, b', [\![s]\!], [\![1]\!]))$
  5. $([\![\alpha_D]\!], ..., [\![\alpha_1]\!]) := \mathsf{bitDecomposition}([\![\alpha]\!])$.
  6. for $d \in \{D, D-1, ...., 1\}$:
      (a) if $|u_d| = 0$, continue.
      (b) $[\![z_d]\!] := [\![z_d]\!] \oplus (\neg[\![v_d]\!])[\![r]\!]$ where $[\![r]\!] \leftarrow \{0,1\}^{2\times\kappa}$
      (c) $\sigma := \mathsf{correctionWord}([\![z_d]\!], [\![\alpha_d]\!])$
      (d) 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. $[\![z_{d-1}]\!] := [\![z_{d-1}]\!] \oplus [\![s']\!], [\![v_{d-1}]\!] := [\![v_{d-1}]\!] \oplus \mathsf{LSB}([\![s]\!])$
          iv. for $k \in \{1, 2\}$
              A. $(\delta, b') := \mathsf{partition}(b_k, S)$
              B. $u_\delta := \mathsf{append}(u_\delta, (k, d-1, b', [\![s'_k]\!], \mathsf{LSB}([\![s]\!])))$
  7. for $(j, \rho, b, [\![s]\!], [\![t]\!]) \in u_0$
      (a) $[\![s]\!] := [\![s]\!] \oplus [\![t]\!] \cdot \sigma_{\rho,j}$
      (b) $[\![t_{b_1}]\!] := \mathsf{LSB}([\![s]\!])$
      (c) $[\![y_{b_1}]\!] := (1 - 2p) \cdot \mathsf{convert}_{\mathbb{G}}(\mathsf{MSBs}([\![s]\!]))$
  8. return $\mathsf{updateLeaves}_S([\![y]\!], [\![t]\!], [\![\beta]\!])$
```

Figure 5: Sparse (Single-Point) DPF.

- **Partitioning the root (steps 3,4)** We call $(\delta, b) := \mathsf{partition}((1, |S|), S)$ to find the top-most bit δ at which the tree first branches (presumably D). For each of the children, the leaves $S_{[b_{1,1},b_{1,2}]}$ live under the left child and $S_{[b_{2,1},b_{2,2}]}$ under the right child. Each of these nodes will be assigned a random $[\![s]\!]$ value. Partition is then called again $(\delta', b') := \mathsf{partition}(b_j, S)$ to determine the level that the j'th child will be updated at with it's correction word and then split again or assigned to a leaf. The relevant tuple for this child is then added to $u_{\delta'}$. Note that at this time these seeds have not yet been updated and therefore the invariant that each level represents a unit vector has not yet been achieved.

- **Top-down expansion (steps 5,6).** Write the secret index as a bit string $([\![\alpha_D]\!], \ldots, [\![\alpha_1]\!])$. For each depth $d = D, \ldots, 1$ we do:

  a) *Randomizing the sums.* In the event that the *active path* does not split on this level, the sums $[\![z_1]\!], [\![z_2]\!]$ will both be zero. As such, the correction word will also be zero which leaks this fact. To prevent this, we randomize the sums. $[\![v_d]\!] \in \{0, 1\}$ keeps tract of if this level requires randomization.

  b) *Compute and reveal correction words.* We construct the correction word using the same $\sigma := \mathsf{correctionWord}([\![z_d]\!], [\![\alpha_d]\!])$ subprotocol which returns two elements $\sigma_1, \sigma_2 \in \{0, 1\}^\kappa$.

  c) *Updating the shares.* Each tuple $(j, \rho, b, [\![s]\!], [\![t]\!]) \in u_d$ is updated by computing $[\![s]\!] := [\![s]\!] \oplus [\![t]\!] \cdot \sigma_j$. Recall that this maps the seed on the α path to being random subject to the LSB being 1 and maps its sibling to being zero. This restores the invarant that the s values at level d form a unit vector.

  d) *Computing Child values.* The children s values are then locally computed as $[\![s']\!]_p := \mathsf{G}([\![s]\!]_p)$. This level almost stasfies the invariant except that the sibling to the α path is also random which will be corrected in subsquent iterations. The z sums for the children are updated.

  e) *Partitioning the chidren.* Each of the children $k \in \{1, 2\}$ are then partitioned, $(\delta, b') := \mathsf{partition}(b_k, S)$ which tells us that this child will next be used at level δ and have sub-ranges $S_{[b'_{1,1},b'_{1,2}]}$ and $S_{[b'_{2,1},b'_{2,2}]}$ for its left and right children. The relavent tuple is then pushed onto the u_δ list to be processed at iteration δ.

     In the event that this value should be assigned to a leaf, $\mathsf{partition}$ will return $\delta = 0$ and the value will be stored in the special "leaf" list u_0.

- **Leaf processing (step 7).** Every tuple that now resides in u_0 corresponds to a *leaf* $i \in S$ and have yet to be updated to form a unit vector. In particular, two of the s values will be random. Each value is updated such that the leaf at α has an LSB of 1 and the other leaf is set to zero.

  Finally, the leaf values are locally converted into elements in $\mathbb{G}$ using the $\mathsf{convert}$ procedure. If neccessary, the shares are negated. The result is an additive sharing of a random unit vector $y = \mathsf{unitVector}(\alpha, r) \in \mathbb{G}^S$ and binary unit vector $t = \mathsf{unitVector}(\alpha, 1) \in \{0, 1\}^S$ for some random $r \in \mathbb{G}$.

- **Derandomizing the leaves (step 8).** Finally, the leaf values are updated so that the leaf value at index α is converted from r into the desired leaf value β. This is achieved using the previous $\mathsf{updateLeaves}$ subprotocol.

Complexity recap. Communication consists of $O(\kappa)$ bits per level of the tree that has a split. This is therefore at most $O(D\kappa)$ bits. In terms of computation, only the leaves and nodes with two children are processes. It is not hard to see that there are $|S|$ leaves and $|S|-1$ splits. Therefore the protocol only requires $O(|S|)$ work (calls to the PRG).
Generalisation to w-ary branching. Replacing binary with w-ary trees follows the same pattern: every internal node now has $1, \dots, w$ live children and correction words are needed only when at least two children survive. Figure 5 shows the algorithm for $w = 2$.

5 Distributed Multi-Point Function

Section 4 established how a DPF lets two parties share a vector that is zero everywhere except at a *single* secret location. In many real-world privacy-preserving tasks, however – think batched private look-ups, sparse PIR queries, or advanced PCG constructions – *several* positions must be populated at once. A naïve implementation would simply run t independent DPFs and add their outputs, but this inflates keys, computation, and communication by a factor of t.

In this section we show how to avoid that blow-up by moving to *Distributed Multi-Point Functions* (DMPFs), which secret-share a vector of length N containing up to t non-zero entries while retaining the compactness and efficiency that make DPFs attractive. Figure 6 formalises the ideal functionality we target: given secret-shared index-value pairs $(\llbracket A_1 \rrbracket, \dots, \llbracket A_t \rrbracket) \in [0, N)^t$ and $(\llbracket B_1 \rrbracket, \dots, \llbracket B_t \rrbracket) \in \mathbb{G}^t$, the parties obtain an additive sharing of the vector $\sum_{i=1}^{t} \mathsf{unitVector}(A_i, B_i)$.

Achieving this compactly raises three core challenges:

1. *Computational overhead.* The work should scale with N and be ideally be independent of t.

2. *Succint key generation.* The communication overhead must be sublinear in N, ideally just $O(t \log N)$.

3. *Index collisions & deduplication.* Duplicate indices must be merged without leaking which entries collided.

A trivial and common way [BCG+20, BBC+24, LXYY25a, LXYY25b] to implement a DMPF is with t instances of a DPF, one for each (A_i, B_i) pair. The final result is obtained by adding the DPFs together. However, this approach results in $O(Nt)$ work and therefore is far from the theoretical optimal of just $O(N)$. Indeed, implementations have shown that this is the main bottleneck for important applications [BBC+24, PR]. We address these challenges with the following constructions:

- *Section 6 Cuckoo-based baseline.* We review the classical cuckoo-hash approach[SGRR19, BGH+25]: each point is routed to one of severals buckets using pairwise-independent hash functions. We then further optimize it using our new sparse-DPF is instantiated per bucket. While conceptually simple, the scheme suffers from heavy key-generation cost, making it ill-suited for tight concrete efficiency.

- *Section 7 Reverse-Cuckoo DMPF.* We flip the viewpoint: instead of first choosing hash function and then assigning the points into bins, we first randomly assign the input points into bins and then solve for a set of compatible hash functions within MPC. Interestingly, the proof of security of this construction consists of the simulator performing cuckoo hashing.

For public parameters $N \in \mathbb{N}, t \in [1, n]$ and group $\mathbb{G}$.

On input $[\![A]\!] \in [0, N)^t, [\![B]\!] \in \mathbb{G}^t$, computes:

1. Let $y = \sum_{i \in [t]} \mathsf{unitVector}(A_i, B_i)$
2. Sample $[\![y]\!]$.
3. Output $[\![y]\!]$.

Figure 6: Distirbuted Multi-Point Function (DMPF) Functionality.

- *Section 8 Bucketing DMPF.* The Bucketing DMPF construction optimizes key generation and evaluation by partitioning the t input index-value pairs into multiple buckets, each handled independently—typically via a sparse DPF—enabling parallelism and reducing key size compared to running t separate DPFs. By selecting bucket parameters and assignment algorithms carefully, this approach balances efficiency with security, ensuring duplicate indices are resolved correctly without leaking information.

- *Section 9 Big-State DMPF.* We give two new key generation protocols for the Boyle-Gilboa-Hamilis-Ishai-Tu *big-state* paradigm[BGH$^+$25]. The first is blackbox and is inspired by our Reverse-Cuckoo construction. The other is a suprisingly simple idea to use generic MPC and the recent AltMod-based PRG to efficiently distribute the "non-distributed" key generation algorithm.

Throughout the section we give precise complexity tables, security proofs in the standard simulation model, and concrete parameter choices targeting 128-bit security. The end result is a toolkit that lets protocol designers pick the DMPF flavour – cuckoo, reverse-cuckoo, bucketing or big-state – that best matches their workload's size and latency constraints.

With the roadmap in place, we now dive into the first and simplest construction.

6 Cuckoo Based DMPF

The cuckoo DMPF adapts classic *cuckoo hashing* to split a t-point function into $m \approx (1 + \epsilon)t$ single-point sub-functions that we can implement with m independent DPF instances. The core idea was first presented in [SGRR19] in the context where one party knows A. However, it is relatively simple to generalize to a DMPF and was first fully described in [BGH$^+$25]. This is the first construction that in theory achieves our desired performance metrics and is closely related to our Reverse-Cuckoo construction. Therefore we review it here.

Cuckoo hashing, introduced by Pagh and Rodler [PR04], is a w-choice open-address hash table that achieves worst-case $O(1)$ look-ups and deletions while retaining linear-space overhead. Because its probe pattern is deterministic once the hash keys are fixed, the scheme has become a standard primitive inside privacy-preserving protocols, e.g., PSI, PIR, ORAM, batch codes, see [Yeo23].

t items $A_1, ..., A_t \in [0, N)$ can be inserted into a hash table of size $m \approx (1 + \epsilon)t$ bins such that there is at most one item in any bin and item α will reside at one of the bins indexed by the random functions $h_1(\alpha), ..., h_w(\alpha) \in [m]$. For approiate choices of m, w and taking h has a random variable, one can successfully construct such a hash table with overwhelming probaiblity. Typical choices is to set $w = 3$ and $m \approx 1.3t$ or $w = 2, m = 2.4t$ [DRRT18].

Consider an equivalent view where we have a matrix $P \in \{0,1\}^{[0,N)\times m}$ such that row P_α for $\alpha \in [0,N)$ has nonzeros at columns $h_1(\alpha), ..., h_w(\alpha)$, i.e. $P_{\alpha,h_j(\alpha)} = 1$. We refer to P as the position matrix as row P_α encodes the valid position for each α. Constructing a cuckoo hash table can then be thought of as finding a unique column $c_i \in [m]$ for each $A_i \in \{A_1, ..., A_t\}$ subject to $c_i \in \{h_1(A_i), ..., h_w(A_i)\}$ or equaivalently $P_{A_i,c_i} = 1$. For example, with $N = 8, t = 3, m = 4, w = 2$ and

$$P = \begin{pmatrix} 1 & 0 & \mathbf{1} & 0 \\ 1 & 0 & 0 & 1 \\ 0 & \mathbf{1} & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & \mathbf{1} \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

If we choose $(A_1, A_2, A_3) = (3, 1, 6)$ then a valid assignment would be for $c = (2, 3, 4)$ which we make bold above. It is also possible for no valid asigment to exists, e.g. in the case above this would correspond the senario where $P_{A_1} = P_{A_2} = P_{A_3}$ but could in general be a more complicated conflict. However, parameters are chosen such that this is unlikely.

Critically for us, the condition that each bin in a cuckoo hash table has one item is equivalent to there being at most one active position in each column of P. Switching our perspective we can observe the active position of each column forms a unit vector. That is,

$$\begin{pmatrix} 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

At this point is may be clear how to construct the DMPF. Given the input positions $A_1, ..., A_t$ we construct a cuckoo hash table for the set $A \subseteq [0,N)$ and obtain the column/bin assignemnts $c_1, ..., c_t$. We then construct m distirbuted single point function for where the c_i'th point function has index A_i and value B_i. In our example above the result would be

$$\begin{pmatrix} 0 & 0 & B_2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & B_1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & B_3 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

where each point function is expanded into the corresponding column. The final multi point function is obtained by summing the columns, e.g. the column vector $(B_2, 0, B_1, 0, 0, B_3, 0, 0)$ in our example.

However, it may not be appearent that this approach is any better than simply construct t points function instead of m. The critical observation is that is it publicly known which position of each column that can have values. For example, column 1 may only have a values at rows $0, 1, 5, 7$. Taking advantage of this sparsity we obtain

$$\begin{pmatrix} 0 & \shortmid & B_2 & \shortmid \\ 0 & \shortmid & \shortmid & 0 \\ \shortmid & B_1 & 0 & \shortmid \\ \shortmid & 0 & \shortmid & 0 \\ 0 & \shortmid & 0 & \shortmid \\ \shortmid & 0 & \shortmid & B_3 \\ 0 & 0 & \shortmid & \shortmid \\ \shortmid & \shortmid & 0 & 0 \end{pmatrix}$$

where position with $\shortmid$ can be ignored. The point function for the ith column can then be over a smaller range only consisting of the valid subset of positions. Overall, this reduces the number of possible points to $wN = O(N)$ and thereby reduce the overall amount of work.

Address Translation. The prior work of [BGH+25] suggested the use of a standard DPF for each column. However, this introduces a challenge in that the position $A_i \in [0, N)$ is not the correct position in the compressed column. For example, $A_1 = 3$ in the global address space $[0, N)$ but is in the first position of the compressed column $(B_1, 0, 0, 0)$. The parties therefore must translate the existing address A_i into a new address $A'_i = \sum_{j\in[0,A_i)} P_{c_i,j}$. However, computing the new address using this formula can not be acheived succinctly. [BGH+25] suggested a clever workaround by defining a permutation $\pi : [w] \times [0, N) \to [m] \times [0, f)$ where $f := wN/m$ and we assume m divides wN. The hash functions are then defined as $(h_j(A_i), A'_{i,j}) := \pi(j, A_i)$ which outputs the hash value $h_j(A_i)$ as well as the translated address $A'_{i,j}$ if A_i is placed in column $c_i = h_j(A_i)$. The correctness of this idea follows from π being a random permutation. In addition, each single point function now has exactly size f.

Although this approach works, it requires implementing a (pseudorandom) permutation within MPC. The natural choice would be to use a feistel construction but this requires many iterations of the feistel round function which complicates the implementation.

Sparse DPF. A natural application of our new sparse DPF is to directly implement the point function over the sparse columns. This approach completely eliminates the whole need to implement π within MPC and instead the hash functions can be implemented using simple techniques, e.g. as linear functions.

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.

To perform this process in MPC is presumably highly expensive. We considered several variants such as inserting many items in parallel but were unable to derive a satisfactory solution. Because of this we omit a formal protocol description of this approach.

7 Reverse Cuckoo DMPF

We solve the challenges of the cuckoo construction by completely eliminating the need to construct a cuckoo hash table c using the interative method. Instead, we first choose a random cuckoo hash table c and then to solve for consistent hash function $h_1, ..., h_w$.

Overview We will assume the reader is familiar with Section 6. The core idea of our construction is as follows.

- The idea is that we will first pad $[\![A]\!], [\![B]\!]$ to each be length m with dummy values and then permute the list in a random order. The positions that each real item lands in will be the cuckoo position that each point belongs to. That is, $c_i = \pi(i)$ where π is the random permutation. Let $[\![A']\!], [\![B']\!]$ now denote the permuted points.
- Our next goal is to construct hash function $h_1, ..., h_w$ such that for some $j \in [w]$ it holds that $h_j(A_i) = c_i$ without revealing anything about the actual value of A_i.
- We then reinterpret each index $[\![A_i]\!]$ as a binary vector in $\{0,1\}^{\log N}$ and partition $[\![A']\!]$ into w region denoted as $[\![M_1]\!], ..., [\![M_w]\!] \in \{0,1\}^{d \times \log N}$ where

  $$[\![M_i]\!] = \begin{bmatrix} [\![A'_{g+1}]\!] \\ ... \\ [\![A'_{g+d}]\!] \end{bmatrix}$$

  for $g := (i-1)d$ and $d := m/w$.
- The core idea will then be to construct a *linear* "hash function" which maps A'_i to its assigned position i. In particular, we want some vector h such that $\langle h, A'_i \rangle = i$. More speficially, we want w vectors $h_1, ..., h_w$ where for each A'_i and some random j it holds that $h_j(A'_i) = i$.

  Since the A'_is are in a random order, we can simply mandate that the first d values will use h_1, the second d values use h_2 and so on. Let us focus on h_1. The desired condition is simply

  $$[\![M_1]\!] \cdot [\![h'_1]\!] = \begin{bmatrix} 1 \\ 2 \\ ... \\ d \end{bmatrix}$$

We will choose a random h_1 which satisfies this. Moreover, we can perform all of this modulo d and therefore all other h_i can be derived using the same structure. Overall we obtain that

$$\langle [\![A'_i]\!], [\![h_{\lfloor i/d \rfloor}]\!] \rangle = i \mod d$$

The $\mod d$ can then be manually offset. That is, some index $\alpha \in \{0,1\}^{\log N}$ is hashed to positions

$$h'_j(\alpha) := (\langle \alpha, h_j \rangle \mod d) + 1 + (j-1)d$$

- Once the $[\![h_j]\!]$ values are solved for, the parties can reveal them and use h'_j to define the position matrix P. The rest of the protocol can the proceed as described before with the use of the Sparse DPF protocol for each column of P.

In summary, the parties first obliviously assign the t input points to $m = (1+\epsilon)t$ positions. The parties then solve a set of w linear systems to derive the hash function $h'_1, ..., h'_w$ which in turn defines the prosition matrix P which can be used in conjection with the sparse DPF to complete the protocol. The full protocol is described in Figure 7.

Invertibility. The description above assume that the system of equations M_i is invertable. However, it is possible to choose a set of inputs $A_1, ..., A_t$ for which their bit decomposition is not invertable. To address this issue we opt to first hash each index using some appropriately chosen hash function $H : \{0,1\}^{\log N} \to \{0,1\}^q$ where q is sufficiently large that over the random choices of H the resulting system is invertibable with overwhelming probaiblity, e.g. $q \geq d + \lambda$ for statistical security parameter λ. In Section 7.3 we give a concrete suggestion for H.

Duplication. Another complexity is that it is possible to have duplicate input indices $A_i = A_j$. In this case the ideal functionality will output $B_i + B_j$ at index A_i. However, the protocol as described only works for distinct A_i. Therefore we introduce a deduplication step at the start of the protocol which is explained in more detail in Section 7.2.

Next we will present the neccessary subprotocols before a formal description and proof of security.

Proof Sketch (semi-honest). We show that the view of either party in the real protocol is simulatable given only public parameters except with negligible probability.

Simulator. On input public params $(N, m, w, \dots)$:

1. Sample $h_1, \dots, h_w \leftarrow \{0,1\}^q$ independently and uniformly, and form the public hash family $h'_\ell(x) := (\langle H_\ell(x), h_\ell \rangle) + 1 + (\ell - 1)d$ and the induced position map P.

2. With probability equal to the cuckoo failure probability (negligible in λ for the chosen load), output abort and halt. Otherwise continue.

3. Simulate the MPC transcripts of dedup, bin-solver, and the local computations by sampling random messages consistent with (i) the public sizes and (ii) the final revealed h_ℓ. For sparse-DPF subinstances, output random keys that are distributed as real DPF keys (by the PRG/DPF security) and do not require knowledge of the hidden indices/values.

```
$\mathsf{RevCuckoo}([\![A]\!] \in (\{0,1\}^{\log N})^t, [\![B]\!] \in \mathbb{G}^t)$, party $p \in \{0,1\}$ computes:

  1. suggested parameters $w = 2, d = 2^{\lceil \log_2 t \rceil}, m = dw, q = d + \lambda$.
  2. $([\![A]\!], [\![B]\!]) := \mathsf{dedup}([\![A]\!], [\![B]\!], (N)_{i \in [t]})$.
  3. Define hash functions $H_1, ..., H_w : [0, N] \to \mathbb{Z}_2^q$ s.t. $H_i(N) = 0$.
  4. $[\![A]\!] := [\![A]\!] || (N)_{i \in [m-t]}$.
  5. $[\![B]\!] := [\![B]\!] || (0)_{i \in [m-t]}$.
  6. $([\![A]\!] \lhd\rhd [\![B]\!]) := \mathsf{shuffle}([\![A]\!] \bowtie [\![B]\!])$.
  7. Let $[\![M_i]\!] = \begin{bmatrix} H_i([\![A_{g+1}]\!]) \\ ... \\ H_i([\![A_{g+d}]\!]) \end{bmatrix}$ for $i \in [w]$ and $g := (i-1)d$.
  8. For $i \in [w]$, $[\![h_i]\!] := \mathsf{bin\text{-}solver}([\![M_i]\!], [\![y_i]\!])$ where $y_i := \begin{bmatrix} 0 \\ 1 \\ ... \\ d-1 \end{bmatrix}$
  9. $h := \mathsf{reveal}([\![h]\!])$.
 10. For $i \in [0, N), j \in [w]$, the parties compute $s_{i,j} := \langle H_j(i), h_j \rangle$
 11. Define $P \in \{0,1\}^{[0,N),m}$ s.t. $P_{i,(j-1)d+s_{i,j}} = 1$ for $i \in [0, N), j \in [w]$.
 12. For $j \in [m]$, let $S_j := \{i \mid P_{i,j} = 1\}$
 13. For $j \in [m]$, the parties invoke $\mathsf{sparse\text{-}DPF}([\![A]\!]_j, [\![B]\!]_j, S_j)$ and receive as output $[\![y'_j]\!]$.
 14. For $i \in [0, N)$, let $[\![y_i]\!] := 0$
 15. For $j \in [m], (i, [\![u]\!]) \in (S_j \bowtie [\![y_j]\!])$, let $[\![y_i]\!] := [\![y_i]\!] + [\![u]\!]$.
 16. output $[\![y]\!]$
 17. If one assumes stationary LPN, we can perform $O(t')$ work to update sparse-DPF with new leaf
     values. Then the DPFs can be re-expanded with $O(N)$ work. This step can be repeated as many
     times as you want.
```

Figure 7: Reverse-Cuckoo Distributed multi-point function realizing Figure 6.

Indistinguishability argument.

(1) **Distribution of the revealed hash descriptors $\boldsymbol{h_i}$.** In the real protocol (Fig. 7, Steps 7-9), each block $i \in [w]$ forms

$$M_i = \begin{bmatrix} H_i(A_{g+1}) \\ \vdots \\ H_i(A_{g+d}) \end{bmatrix} \in \{0,1\}^{d\times q}, \qquad y_i = (0,1,\ldots,d-1)^\top, \quad g = (i-1)d,$$

and runs $\mathsf{bin\text{-}solver}(M_i, y_i)$ to obtain a secret-shared solution $[\![h_i]\!] \in \{0,1\}^q$, which is then revealed as h_i. By construction, conditioned on $\mathrm{rank}(M_i) = d$ (which holds except with probability $\le 2^{-(q-d)}$ by our choice $q = d + \lambda$; see §7.3), the solution set $\{h \in \{0,1\}^q : M_i h = y_i\}$ is an affine subspace of dimension $q-d$, and our randomized solver outputs h_i *uniformly* over that subspace. Since neither the rows $H_i(A_{g+u})$ (i.e., M_i) nor anything about the A's is revealed, the adversary only sees the published h_i with the public sizes (d, q). Thus, from either party's view, each h_i is distributed as a random element of a hidden affine coset determined by the (unrevealed) M_i and public y_i. In the simulator, we therefore sample $h_i \leftarrow \{0,1\}^q$ uniformly and proceed, since the induced public uses of h_i are only through Steps 10-12:

$$s_{j,i'} = \langle H_j(i'), h_j \rangle, \qquad P_{i',(j-1)d+s_{j,i'}} = 1,$$

which depend on h_i but not on the hidden M_i; any distinguishing advantage would imply learning linear relations about the secret A's from h_i alone, contradicting that M_i (hence the defining coset) is never revealed.

(2) **Cuckoo validity / abort.** For the revealed hash family h'_ℓ, the existence of a valid placement of the t items into $m = (1+\varepsilon)t$ bins using w choices per item fails with negligible probability under standard cuckoo analysis. The real protocol only proceeds when a valid assignment exists (our construction fixes a permutation and solves for consistent h'); the simulator mirrors this by aborting with exactly the same negligible probability, so the abort bit is perfectly coupled.

(3) **Transcript simulation.** The dedup and $\mathsf{bin\text{-}solver}$ subprotocols are linear/boolean MPC over secret shares; they reveal no additional outputs beyond s_ℓ. Standard straight-line simulators for OT/garbled MPC can therefore produce transcripts indistinguishable from real, given s_ℓ and sizes. For each sparse-DPF, key indistinguishability of the underlying DPF (against a party who sees only its own key) lets us sample keys without the hidden (A'_j, B'_j). Since the public position map P is a function of the revealed s_ℓ only, no additional information about the inputs is leaked.

(4) **Correctness (brief).** When M_ℓ are full rank, each block produces hash parameters s_ℓ that send every A'_i to its designated bin; the sparse-DPFs routed by P then sum to $y = \sum_i \mathsf{unitVector}_{[0,N)}(A_i, B_i)$. Padding and dedup ensure uniqueness and proper summation under collisions.

Conclusion. The simulated view is computationally indistinguishable from the real view; any difference occurs only in the negligible cuckoo-failure event, which is identically reflected by the simulator's abort.

7.1 System Solver

The $\mathsf{bin\text{-}solver}$ protocol solves 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 outputs a secret-shared solution $x \in \mathbb{G}^c$ such that $Mx = y$.

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.

Figure 8 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 characturistic 2 group $\mathbb{G}$.
$\mathsf{bin\text{-}solver}([\![M]\!] \in \mathbb{F}_2^{m\times c}, [\![y]\!] \in \mathbb{G}^m)$, party $p \in \{0,1\}$ computes:

  1. for $i \in [m]$:
     (a) $[\![s_i]\!] := \mathsf{unitVector}([\![j]\!], 1) \in \mathbb{F}_2^c$ 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 8: Randomized binary system solver.

```
Public parameters: a group $\mathbb{G}$.
$\mathsf{dedup}([\![A]\!] \in \mathbb{G}^n, [\![B]\!] \in \mathbb{G}^n, [\![A']\!] \in \mathbb{G})$, party $p \in \{0,1\}$ computes:

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

Figure 9: Debuplicate protocol with values.

7.2 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.

7.3 Hashing Protocol

Our hashing layer is used only to ensure that, after shuffling, each block matrix M_i (formed from rows $H_i(A_{g+1}), \ldots, H_i(A_{g+d})$) is invertible over $\mathbb{F}_2$ with overwhelming probability. We do *not* require cryptographic strength (e.g., collision or preimage resistance). Instead, we require that—once the inputs are fixed and the hash seed is sampled—the resulting row sets is full rank with high probability.

Goal and minimal requirements. Let $H_i : \{0,1\}^{\log N} \to \{0,1\}^q$ be w independently seeded functions with $H_i(N) = 0$ for the dummy key. For each block $i \in [w]$ we form the $d \times q$ matrix

$$M_i \;=\; \begin{bmatrix} H_i(A_{g+1}) \\ \vdots \\ H_i(A_{g+d}) \end{bmatrix}, \qquad g := (i-1)d.$$

The *minimal requirement* is:

$$\mathrm{rank}_{\mathbb{F}_2}(M_i) \;=\; d \quad \text{for all } i \in [w].$$

Equivalently, the d rows are linearly independent over $\mathbb{F}_2$. This guarantees the linear systems we solve to define h_i have solutions (and, with our randomized treatment of free variables, yield a valid random solution when $q > d$).

A standard counting argument gives the following bound if, conditioned on fixed inputs, each row of M_i is distributed close to uniform and independently over $\{0,1\}^q$:

$$\Pr\big[\mathrm{rank}(M_i) < d\big] \;\le\; \sum_{j=0}^{d-1} 2^{j-q} \;\le\; 2^{-(q-d)}.$$

Thus choosing $q \ge d + \lambda$ yields failure probability at most $2^{-\lambda}$ per block. (Union-bounding over $w = O(1)$ blocks preserves $2^{-\lambda}$ up to a constant factor.)

Difference from a cryptographic hash.

- **What we need:** post-hoc *algebraic genericity* (full row rank) when the seed is sampled after inputs are fixed.
- **What we do not need:** collision resistance, preimage resistance, or random-oracle behavior.
- **Adversary model:** inputs (A_i) may be adversarially chosen, but the seed is sampled independently *after* inputs are fixed; our requirement is that, under this seeding, the induced M_i is full-rank with high probability.

This is closer in spirit to k-wise independent hashing or random linear embeddings than to cryptographic hashing.

Conservative parameter choice. Throughout we set

$$q \;=\; d + \lambda,$$

We also keep the H_i seeds independent across blocks and enforce $H_i(N) = 0$ so padding rows are neutral.

A Goldreich-style hash (seeded, non-cryptographic). We now describe a simple, efficient, Goldreich-PRG – inspired family that we conjecture meets the above rank requirement while remaining MPC-friendly.

Interface. A hash instance is parameterized by a seed and three binary matrices:

$$M_0 \in \{0,1\}^{q' \times \ell}, \quad M_1 \in \{0,1\}^{q' \times \ell}, \quad M_2 \in \{0,1\}^{q \times (\ell + q')},$$

where $\ell = \log N$ and q' is an *intermediate width.* Given $x \in \{0,1\}^\ell$, define:

1. $a = M_0 x \in \{0,1\}^{q'}$,
2. $b = M_1 x \in \{0,1\}^{q'}$,
3. $c = a \wedge b \in \{0,1\}^{q'}$ (bitwise AND),
4. $y = M_2 \cdot (x \,\|\, c) \in \{0,1\}^q$.

Rationale.

- The maps $x \mapsto M_0 x$ and $x \mapsto M_1 x$ are linear and yield rows close to uniform for random M_0, M_1.
- The coordinate-wise AND $a \wedge b$ injects mild nonlinearity reminiscent of Goldreich's generator, which helps break residual linear structure.
- The final compression $M_2(x\|c)$ produces the target width q while preserving enough randomness so that each block matrix M_i behaves as a random $d \times q$ binary matrix for rank analysis.

Dummy key constraint. We enforce $H(N) = 0$ by programming the seed so that (M_0, M_1, M_2) maps $x = N$ to 0^q. In practice this can be done by coset-shifting the rows of M_2 so that $H(N) = 0$ holds exactly without perturbing the distribution on non-dummy inputs.

Parameters and checks.

- Choose $q' \geq q$ (e.g., $q' = q$ or $q' = 2q$).
- Use independent seeds per block to avoid cross-block dependencies.

MPC-friendliness. The construction is linear except for the $a \wedge b$ step, which is a bitwise multiplication supported efficiently via our DpfMult primitive. The rest are matrix-vector products over $\mathbb{F}_2$. Communication is dominated by the single batched secure bit-multiplication over q' bits per input, which is negligible relative to the downstream Sparse-DPF bandwidth.

Takeaway. We only need seeded, post-hoc full-rank guarantees—not cryptographic hashing. A simple Goldreich-style hash with two linear sketches, a coordinate-wise AND, and a final linear compression provides (i) close-to-uniform row distributions conditioned on fixed inputs and fresh seeds, (ii) independence across rows and blocks, and hence (iii) full-rank M_i with probability $\geq 1 - 2^{-(q-d)}$ by setting $q \geq d + \lambda'$. We performed an evaluation that this does result in the desired properties for relavent parameters. We also note that other constructions could be used with minimal impact on overall performance.

Parameters: a bucket count k and maximum capacity c. For uniform keys, suggested parameter $k = t/O(\log t), c = O(\log t)$.

BucketingDMPF: On input $[\![A]\!] \in [0, N)^t$, $[\![B]\!] \in \mathbb{G}^t$, party $p \in \{0, 1\}$ computes:

1. Let $M := N/k$ denote the domain size of items within a fixed bucket. For simplicity, we assume k divides N.
2. For $i \in [t]$: $[\![A'_i]\!] := [\![A_i]\!]/M, [\![A''_i]\!] := [\![A_i]\!] \mod M$, note $A'_i \in [0, k)$.
3. $[\![A^*]\!] \triangleleft\triangleright [\![B^*]\!] := \mathsf{Group\text{-}By}_c([\![A']\!], [\![A'']\!] \bowtie [\![B]\!])$
4. For $j \in [k]$,
   (a) parties invoke $[\![z_j]\!] := \mathcal{F}_{\mathsf{dmpf}}([\![A^*_j]\!] \in (0, M]^c, [\![B^*_j]\!] \in \mathbb{G}^c) \in \mathbb{G}^M$.
5. Output $[\![y]\!] := [\![z_1]\!]||...||[\![z_k]\!] \in \mathbb{G}^N$.

Figure 10: Bucketing based Distributed multi-point function.

8 Bucketing DMPF

The Bucketing DMPF construction leverages the uniform distribution of non-zero entries in the input vectors to optimize protocol efficiency. By partitioning the input into multiple buckets or bins. Each bucket is then privately evaluated using a generic call to an underlying DMPF protocol, and the results are concatenated to produce the final output. This method improves scalability and performance, particularly when the non-zero points are clustered or follow a predictable distribution.

The protocol proceeds as follows. Given input vectors $[\![A]\!]$ and $[\![B]\!]$, the parties first choose a parameter k indicating the number of buckets. Each input index i is assigned to a bucket based on A_i modulo $M := N/k$, and the remainder of A_i is used to determine the position within the bucket. The parties then group and reindex the inputs accordingly, collecting all elements for each bucket. For each bucket, the parties invoke a standard DMPF protocol on the grouped inputs, obtaining a partial output for each bucket domain. Finally, all DMPF outputs are concatenated in order to construct the overall output vector corresponding to the full domain.

If the keys are drawn uniformly at random and we choose the number of buckets $k = t/\log t$, then the expected number of items per bucket is $\log t$. Using Chernoff bounds, with overwhelming probability, no bucket will contain more than $O(\log t)$ elements. Thus, setting the bin capacity parameter $c = O(\log t)$ ensures that no bin overflows in this regime.

With these parameters, the overall protocol achieves linear efficiency: There are $k = t/\log t$ buckets, each holding $O(\log t)$ items, so the total number of DMPF operations is $O(t)$. Each DMPF call is applied to a domain of size N/k and a batch of $O(\log t)$ items, so both computation and communication scale as $O(t)$ overall. Additional local operations, such as bucketing and output concatenation, likewise require $O(t)$ work. Thus, the bucketing approach is highly scalable for large t with uniform keys.

8.1 Group-By Operation

The Group-By protocol takes as input two secret-shared lists: a key vector $[\![A]\!] \in [0, N)^t$ and a value vector $[\![B]\!] \in \mathbb{G}^t$, as well as a padding parameter p. The protocol creates N groups (one for each key in $[0, N)$), each padded to length p, so that no information about the true group sizes is leaked beyond the bound p.

The protocol proceeds as follows:

1. For each $i \in [0, N)$, compute the frequency $[\![f_i]\!]$ of key i in $[\![A]\!]$.
2. For each key i, construct a length-p vector $[\![d_i]\!]$ with $[\![f_i]\!]$ zeros (for actual elements) and $p - [\![f_i]\!]$ ones (for padding).
3. Form new arrays:
   - Keys: Concatenate
     (a) for each i, $[\![f_i]\!]$ copies of i and $p - [\![f_i]\!]$ copies of 0.
     (b) the original keys in $[\![A]\!]$.
   - Values: 0 values for all padding positions, followed by the original values in $[\![B]\!]$.
   - Indicator bits: $[\![d_i]\!]$ for all keys, followed by 1 for each actual data element.

   Each new list has $Np + t$ elements.
4. Concatenate and randomly shuffle all tuples (key, value, indicator) so positions of actual and padded elements are hidden.
5. Reveal the key and indicator from each tuple to determine which are real (indicator $= 1$) and which are padding.
6. For each key i, collect all real values (indicator $= 1$, key $= i$) into group G_i of size p.
7. Return the collection $(G_0, \ldots, G_{N-1})$ of padded groups.

Implementation costs for Group-By are dominated by the need to compute the (secret-shared) frequency of each key, $[\![f_i]\!]$. This is accomplished using a unit vector indicator $\mathsf{unitVector}_{\mathbb{Z}_p^N}([\![A_i]\!], 1)$ for each input element and key. This unit vector can be implemented either via a Distributed Point Function (DPF) evaluation or by a binary tree of AND gates. The same applies to constructing the new key array $[\![A']\!]$ in step 3, where a copy of each key is assigned based on the corresponding frequency counters. Thus, both the frequency counting and array construction can be realized efficiently using either primitive, with the overall cost scaling as $O(Nt)$ for the binary tree approach or as $O(Nt)$ using DPF-based techniques.

Simulation. Simulating this protocol is trivial assuming A has maximum multiplicity of at most p. A' is distributed as a random permutation of $((0)^p, ..., (N-1)^p)||(0)^t$ and d' is the same random permutation of $(1)^{Np}||(0)^t$. That is, A' will have p copies of each index along with t additional copies of 0. These additional copies of 0 will have tag bit $d_i = 0$ while all others have tag bit $d_i = 1$. Simulation consists of uniformly sampling A', d' from this distirbution.

```
For input and padding size $t, p \in \mathbb{N}$ and key domain $[0, N)$.

$\mathsf{Group\text{-}By}_{p,t,N}([\![A]\!] \in [0,N)^t, [\![B]\!] \in \mathbb{G}^t)$:
    1. let $[\![f]\!] := \sum_{i\in[t]} \mathsf{unitVector}_{\mathbb{Z}_p^N}([\![A_i]\!], 1)$
    2. for $i \in [0,N)$ let $[\![d_i]\!] := (0)^{[\![f_i]\!]} || (1)^{p-[\![f_i]\!]} \in \{0,1\}^p$
    3. let $[\![A']\!] := (0) \cdot [\![d_0]\!] || ... || (N-1) \cdot [\![d_{N-1}]\!] || [\![A]\!] \in [0,N)^{Np+t}$
    4. let $[\![B']\!] := (0)^{Np} || [\![B]\!]$                 $\in \mathbb{G}^{Np+t}$
    5. let $[\![d']\!] := [\![d_0]\!] || ... || [\![d_{N-1}]\!] || 1^t$   $\in \{0,1\}^{Np+t}$
    6. let $([\![A']\!] \lhd\rhd [\![B']\!] \lhd\rhd [\![d']\!]) := \mathsf{shuffle}(\bowtie([\![A']\!], [\![B']\!], [\![d']\!]))$
    7. let $(A' \lhd\rhd d') := \mathsf{reveal}(\bowtie([\![A']\!], [\![d']\!]))$
    8. for $i \in [0,N)$ : let $[\![G_i]\!] := ([\![B'_j]\!] \mid d'_j = 1, A'_j = i) \in \mathbb{G}^p$
    9. output $[\![G]\!]$
```

Figure 11: Padded Group-By Protocol.

9 Improved Bigstate DMPF.

Boyle et al.[BGH+25] introduced two new 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 is tailored for small to moderate weights t (roughly $3 \le t \le 70$), which are precisely the regimes arising in pseudorandom correlation generators. This work has the major caveat that they focus on the trusted dealer setting where a central party construct the DPF keys and distirbuted them to the two evaluation parties. As such, they do not realize the API $\mathsf{DMPF}([\![A]\!], [\![B]\!])$ but instead requires the dealer to know A, B in plaintext. Never the less, [BGH+25] does 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 $s_{i\oplus 1}^d = 0, \tau_{i\oplus 1}^d = 0$ while the active child i has its tag bit corrected to $\tau_i^d = 1$.

Observe that the parent level has t unit vectors $[\![\tau_1^{d-1}]\!], ..., [\![\tau_t^{d-1}]\!]$ which encode the active

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

$\mathsf{bigstate\text{-}DPF}$: 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 $i \in [0, 2^d)$: $[\![s_i^d]\!] || [\![\tau_i^d]\!] := [\![s'_i]\!] \oplus [\![\tau_{i/2}^{d-1}]\!] \sigma_{k,i \bmod 2}$
     (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.

$\mathsf{UnitSparseInnerProd}$: On input $[\![u]\!] \in \{x \in \{0,1\}^m \mid |x| \leq 1\}$ and $[\![v]\!] \in \{x \in \{0,1\}^m \mid |x| \leq t\}$, 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 12: Big-state Distributed Multi-Point Function (DPF) protocol [BGH+25]. Does not include our optimized UnitSparseInnerProd protocol.

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 want to compute the inner product of the unit vectors $[\![\tau_i^{d-1}]\!] \in \{0,1\}^{2^{d-1}}$ with the weight (at most) t vector $[\![s']\!]$, where we view each pair of siblings in s' as a single element. [BGH+25] gave a protocol for doing this called UnitSparseInnerProd. Unfortunently, this routines is inefficient and only better than a trivial linear time solution when $n \gg 2^{30}$. Their approach is reproduced in UnitSparseInnerProd of Figure 12. The idea is to perform the inner product not on the seed+tag vector with $\{0,1\}^{2\kappa+2t}$ elements but one bit at a time. That is, compute the inner product between a unit vector $[\![\tau_i^{d-1}]\!] \in \{0,1\}^{2^{d-1}}$ and a vector $[\![v]\!]$ with weight at most t. Given this, one can compute the general case by using $2\kappa+2t$ 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 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.

9.1 Improved UnitSparseInnerProd

We improve the basic idea of [BGH+25] by giving a more efficient protocol that also directly work when the weight t vector has strings for elements, as opposed to bits. That is, $u \in \{0,1\}^m$ is a *unit* vector; $v \in \{0,1\}^{\sigma \times m}$ has Hamming weight $\leq t$ when viewed as vector of $\{0,1\}^\sigma$ elements. Given secret shares $[\![u]\!], [\![v]\!]$, we want $[\![\langle u, v\rangle]\!]$ with 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 $\Pr[\text{collision at } b^\star] \leq (t-1)/B \leq 1/c < 1/2$. 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 $O(\mathrm{poly}(t))$).

9.2 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:

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

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

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

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

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

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[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,

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

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

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

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

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

Cost. Per repetition: (i) linear sketching into $B = O(t)$ buckets; (ii) B masked multiplications to form $[\![V_\star^r]\!]$ plus B to form $[\![B_\star^r]\!]$; (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 $R \geq \ln(1/\epsilon)/(2(p-\frac{1}{2})^2)$ to get $\Pr[\text{fail}] \leq e^{-2(p-\frac{1}{2})^2 R}$. Costs are $2RB$ OTs of length σ plus an $O(R\sigma)$ majority. With fingerprints, the bound becomes $\Pr[\text{fail}] \leq c^{-R} + R\,2^{-\rho}$, 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 $\mathsf{UnitSparseInnerProd}$ 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.

9.3 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: $([\![\alpha_D]\!], \dots, [\![\alpha_1]\!]) := \mathsf{decomposition}_2([\![A]\!]) \in \{0,1\}^{t\times D}$.

4. **For each depth** $d = 1, \dots, D$**:**

   (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 $c_{\mathrm{mul}} = O(\kappa)$ multiplications, $R = O(1)$ rounds per block, and we need $\lceil (2\kappa + 2t)/w \rceil$ blocks per call, the arithmetic-multiplication cost is
$$O\Big(tD \cdot R \cdot c_{\mathrm{mul}} \cdot \Big\lceil \frac{2\kappa + 2t}{w} \Big\rceil\Big),$$
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 $\mathsf{UnitSparseInnerProd}$.** 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 $\mathsf{UnitSparseInnerProd}$), 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.

10 Application to PCGs: Ring-LPN and Stationary LPN

PCGs in a nutshell. A pseudorandom correlation generator (PCG) for a target two-party correlation $\mathcal{C}$ consists of a short-seed setup $\mathsf{PCG.Gen}(1^\lambda) \to (k_0, k_1)$ and a silent expansion algorithm $\mathsf{PCG.Expand}(\sigma, k_\sigma) \to R_\sigma$ whose outputs jointly emulate a sample from $\mathcal{C}$, even to an insider who knows one of the seeds. Formally, correctness and insider security are captured in [BCG+20, Def. 2.6], together with the reverse-sampleability notion of [BCG+20, Def. 2.5]. PCGs enable a tiny, one-time interactive setup followed by a bounded, communication-free expansion of k_i into n correlations. We also consider Stationary LPN what allows for an even smaller "key refresh" protocol that allows the parties to generate another set of n correlations, see below. We will not explicitly define $\mathsf{PCG.Gen}$ but instead describe the monotlitic protocol that allows the two parties to interactively compute $\mathsf{PCG.Gen}$. However, one can recast our protocol in this syntax if desired.

10.1 Ring-LPN overview and its use in PCGs

The Ring-LPN assumption works over a quotient ring $\mathbb{R} := \mathbb{F}_q[X]/F(X)$ (degree N), with "sparse" noise polynomials of Hamming weight t. Writing $e, f \leftarrow \mathsf{HW}_{\mathbb{R},t}$ for t-sparse polynomials and $r \leftarrow \mathbb{R}$, the distribution

$$(r,\ r \cdot e + f) \qquad \text{is computationally indistinguishable from} \qquad (r,\ u),\ u \leftarrow \mathbb{R},$$

as formalized in [BCG+20, Def. 3.1]. Concretely, $\mathbb{R}$ will be the polynomials modulo $F(X) = (X^N + 1)$. This choice allows efficient polynomial multiplications via a negacyclic NTT.

Boyle et al. showed how to realize a *silent* PCG for OLE (and authenticated triples) from Ring-LPN by packing many field OLEs into one ring instance and expanding with fast polynomial arithmetic; see their overview and the OLE/triples sections [BCG+20]. In brief:

- Party $p \in \{0,1\}$ samples two (or more) random t-sparse polynomials $(e_{p,0}, e_{p,1}) \in \mathbb{R}^2$. Concretely, this will represent sampling t-sparse vectors.
- The parties will sample a random polynomial $r \in \mathbb{R}$.

- Party p will compute $x'_p := e_{p,0} + r \cdot e_{p,1}$.
- Party p applies an NTT to obtain $x_p := \mathsf{NTT}(x'_p) \in \mathbb{F}_q^N$. A vector of N $\mathbb{F}_q$ elements.
- The parties *want* to compute a secret sharing

$$\begin{aligned}[\![y']\!] &= x'_0 \cdot x'_1 \in \mathbb{R} \\ &= (e_{0,0} + r \cdot e_{0,1}) \cdot (e_{1,0} + r \cdot e_{1,1}) \\ &= e_{0,0} \cdot e_{1,0} + r \cdot e_{0,1} \cdot e_{1,0} + e_{0,0} \cdot r \cdot e_{1,1} + r^2 \cdot e_{0,1} \cdot e_{1,1}\end{aligned}$$

- Given $[\![y']\!]$, the parties can then locally compute $[\![y]\!] := \mathsf{NTT}([\![y']\!])$. The critical property is that

$$x_0 \odot x_1 = [\![y]\!]_0 + [\![y_1]\!]$$

  where $\odot$ is componentwise multiplication over vectors $\mathbb{F}_q^N$. This is percisely an Oblivious Linear Evaluation (OLE) correlation. One interpretation of this transform is that the NTT is performing polynomails evaluation. If $[\![y']\!] = x'_0 \cdot x'_1 \in \mathbb{R}$, then for some $\alpha \in \mathbb{F}_q$ the same holds for $[\![y'(\alpha)]\!] = x'_0(\alpha) \cdot x'_1(\alpha) \in \mathbb{F}_q$ (for an appropriately chosen $\mathbb{R}$). Two such correlations can then be locally converted into N Beaver triples

$$[\![a]\!], [\![b]\!], [\![c]\!] \in \mathbb{F}_q^N \text{ s.t. } [\![a]\!] \odot [\![b]\!] = [\![c]\!]$$

- Let $A_{p,i} \subset [0, N)^t, B_{p,i} \in \mathbb{F}_q^t$ denote the positions and values of the non-zeros coefficients in $e_{p,i} \in \mathbb{R}$. Observe that the product polynomial $e_{0,i} \cdot e_{1,j}$ has position and values described as

$$\begin{aligned}A'_{i,j} :=& A_{0,i} \boxplus A_{1,j} \in [0, 2N)^{t^2}, \\ B'_{i,j} :=& B_{0,i} \otimes B_{1,j} \in \mathbb{F}_q^{t^2}\end{aligned}$$

  where $\boxplus$ is the cartisian addition and $\otimes$ is the cartisian product of the sets. That is,

$$e_{0,i} \cdot e_{1,j} = \mathsf{makePoly}(A'_{i,j}, B'_{i,j}) \mod F(X)$$

  where $\mathsf{makePoly}(P \in \mathbb{N}^t, V \in \mathbb{F}^t) = \sum_i V_i X^{P_i}$ returns the polynomails with the V coefficients at the P positons.

- Observe that $\mathsf{makePoly}([\![A'_{i,j}]\!], [\![B'_{i,j}]\!])$ can directly be constructed from a DMPF. In particular, $\mathsf{makePoly}([\![A'_{i,j}]\!], [\![B'_{i,j}]\!])$ is perciesely the same function as a DMPF when the polynomial is express as a vector of coefficients.

The outline above gives a concrete protocol for generating N OLE and Beaver triples.

1. The parties sample their sparse polynomails.
2. Each party construct the sparse represetation of their polynomails. Using generic MPC or similar, they compute the cartisian addition and product of the positons and values to obtains $[\![A'_{i,j}]\!], [\![B'_{i,j}]\!]$.

3. They use a DPMF of size t^2 for each (i,j) to obtain $\mathsf{makePoly}(A'_{i,j}, B'_{i,j})$ in secret shared form.

4. They reduce this modulo $F(X)$.

5. They perform local additions and FFT operations to obtain N OLE or $N/2$ Beaver Triples over $\mathbb{F}_q$.

One can further optimize this protocol by restricting the polynomials to being regular[BCG+20]. Given this restriction, makePoly can be implemented as t instances of a DMPF, each of size t. This can allow for a more efficient implement than a single DMPF instance of size t^2.

10.2 Stationary LPN / SSD and amortizing noise generation

Stationary LPN (a.k.a. Stationary Syndrome Decoding, SSD) captures the setting where the *support* of the noise is reused across many instances while the *payloads* (coefficients) are freshly sampled each time [KPRR25]. In our Ring-LPN notation from above, fix for each party $p \in \{0,1\}$ and selector $i \in \{0,1\}$ a support set

$$L_{p,i} \subseteq [0,N), \qquad |L_{p,i}| = t,$$

and let $A_{p,i}$ be the sorted list of positions in $L_{p,i}$. For the s-th expansion, sample fresh coefficient vectors $B^{(s)}_{p,i} \in \mathbb{F}_q^t$ and form

$$e^{(s)}_{p,i} \;:=\; \mathsf{makePoly}\big(A_{p,i},\, B^{(s)}_{p,i}\big) \;\in\; \mathbb{R},$$

while sampling an independent $r^{(s)} \leftarrow \mathbb{R}$. Security asserts that reusing $(L_{p,i})$ across many $s = 1,2,\ldots,q$ (with fresh coefficients and ring masks) remains pseudorandom against the relevant classes of attacks, so that the pairs $\big(r^{(s)},\, r^{(s)}\cdot e^{(s)}_{p,1} + e^{(s)}_{p,0}\big)$ are computationally indistinguishable from random in $\mathbb{R}^2$ for each s.

Consequence for the product terms. With stationary supports, the *product supports* are fixed once and for all:

$$A'_{i,j} \;:=\; A_{0,i} \;\boxplus\; A_{1,j} \;\in [0,2N)^{t^2}, \qquad B^{(s)}_{i,j} \;:=\; B^{(s)}_{0,i} \;\otimes\; B^{(s)}_{1,j} \;\in \mathbb{F}_q^{t^2},$$

and $e^{(s)}_{0,i} \cdot e^{(s)}_{1,j} = \mathsf{makePoly}\big(A'_{i,j},\, B^{(s)}_{i,j}\big) \bmod F(X)$. Thus, across $s = 1,\ldots,q$, the *addresses* $A'_{i,j}$ never change; only the *values* $B^{(s)}_{i,j}$ do.

How Stationary applies to our pipeline. The Ring-LPN PCG flow in Section 10.1 already factors the heavy work into: (i) building (and then multiplying) the sparse polynomials, and (ii) routing those sparse contributions into the ring (and then into the NTT domain). Stationarity lets us push nearly all position-dependent work *out of the per-expansion loop*:

1. **One-time (setup).**

   - Fix supports $A_{p,0}, A_{p,1}$ and derive the four fixed product supports $A'_{i,j}$.

- Precompute the position-only routing for each $A'_{i,j}$ with our new DMPF constructions: this fixes the level tags, permutations, and correction structure once. No payloads are embedded yet.

2. **Per expansion $s = 1, \dots, q$.**
   - Sample fresh coefficients $B^{(s)}_{p,i}$ (and fresh ring mask $r^{(s)}$); form $B^{(s)}_{i,j} = B^{(s)}_{0,i} \otimes B^{(s)}_{1,j}$.
   - *Value-only programming:* reuse the cached routing for each fixed $A'_{i,j}$ and inject the new $B^{(s)}_{i,j}$ values (no recomputation of tags or inner-product gadgets). This is exactly the "payload injection" stage in our DMPF box, repeated with new shares.
   - Reduce $\sum_{i,j} \mathsf{makePoly}(A'_{i,j}, B^{(s)}_{i,j})$ modulo $F(X)$ and proceed with the NTT and componentwise products to obtain the packed OLEs / triples.

Cost implications. Let $D = \lceil \log_2 N \rceil$ and recall $|A'_{i,j}| = t^2$.

- *Fresh (non-stationary):* each expansion repeats position discovery and routing, rebuilding the structure for every (i, j), plus programming values—this is where the original distributed approach paid an n-dependent price via $\mathsf{UnitSparseInnerProd}$.
- *Stationary:* we pay the position-routing cost *once*. Each expansion programs only the new values $B^{(s)}_{i,j}$ through the cached DPMF skeleton and then runs ring/NTT arithmetic. In our improved DPMF path, the per-expansion nonlinear work is $O(\mathrm{poly}(t))$ for the DMPF value path, independent of N; the arithmetic dominates (NTT and pointwise multiplies).

Practical takeaway. For degree-2 correlations (Ring-LPN $\to$ packed OLE/triples), Stationary LPN perfectly matches the factorization in our construction:

fix supports once $\Rightarrow$ precompute DPMF routing once $\Rightarrow$ per expansion: only new coefficients and ring arithmetic.

This reduces the online cryptographic work to value injection on a fixed sparse footprint and lets the NTT-backed Goldilocks arithmetic drive overall throughput, which is precisely the regime where our implementation attains multi-million ops/sec.

10.3 Impact on Related PCG Constructions

While our presentation above focused on the Ring-LPN and Stationary LPN settings, the same structural improvements offered by our DMPF constructions extend naturally to recent works on PCGs over Boolean circuits, arbitrary finite fields, and Galois rings. We briefly summarize these connections.

PCGs for Boolean and multi-party circuits. Bombar et al. introduce FOLEAGE [BBC+24], a PCG-based framework for efficient Boolean MPC using F_4-OLEs. Their protocol relies on a programmable PCG to stretch seeds into large families of correlations, but still requires recomputation of routing structure for each instance. Our DMPF path allows one to factor out this recomputation: the support positions (determined by F_4 embeddings and syndrome-decoding gadgets) can be cached, leaving only lightweight value injection per expansion. This aligns perfectly with the semi-honest, many-expansion setting FOLEAGE targets, and suggests further speedups in the multi-party case where $O(N^2)$ blowups are otherwise incurred.

PCGs over any finite field. Li et al. [LXYY25a] propose PCGs for OLEs and authenticated multiplication triples over arbitrary fields, using trace functions to lift the Ring-LPN/QA-SD paradigm into the binary case F_2. Ring LPN described above requires $\mathbb{F}_q$ where $q > n$ and therefore does not work for small fields. Their approach is programmable, and therefore already admits multi-party generalizations, but the per-expansion overhead remains dominated by reconstructing sparse cross-term supports. Substituting our DMPF-based support handling yields the same benefit as above: once the support routing is precomputed, subsequent expansions reduce to coefficient injection and NTT-backed arithmetic. This brings the “any field” PCG line into the same optimized regime as our Ring-LPN/Stationary pipeline.

PCGs over Galois rings. Li et al. [LXYY25b] present programmable PCGs for OLE and SPDZ$_{2^k}$ triples over arbitrary Galois rings, leveraging Hensel lifting and Galois theory. Their protocols inherit a key cost driver: repeatedly routing sparse product supports into a structured ring domain. Our DMPF constructions provide a direct plug-in replacement for this routing layer. Instead of rebuilding the support structure for each expansion, one can precompute the DMPF routing once (per support) and reuse it across expansions—precisely the benefit described in Section 10.2. This cuts the online cost down to payload injection and ring arithmetic, thereby improving both the asymptotics and the concrete constants of their Z_{2^k}-based PCGs.

Takeaway. In all three settings—Boolean circuits, arbitrary fields, and Galois rings—the new DMPF constructions eliminate repeated position-routing costs. This unifies the efficiency gains: the heavy, N-dependent cryptographic work is moved to a one-time setup, while each expansion costs only value injection plus local algebra. Thus, our ideas are broadly applicable and yield concrete improvements across the state-of-the-art PCG landscape.

11 Evaluation

We evaluate the practicality of our designs with a concrete implementation of the *Reverse Cuckoo* DMPF (Section 7) integrated into a RingLPN-based PCG pipeline (Section 10). In Section 11.1 we present just the cost to perform a Reverse Cuckoo DMPF with various parameters. In Section 11.2 we present the full end to end costs of generating prime field OLEs and Beaver triples via the Ring LPN protocol combined with our Reverse Cuckoo DMPF.

Environment. Benchmarks were run on a commodity laptop-class CPU (i7 13700H) with 32GB DDR5 memory. Communication was performed on a local socket with sub millisecond latency and +10Gbps. All code was compiled with full optimizations (`-O3`, LTO) and native CPU tuning (`-march=native`). Results are single-threaded per party.[1]

[1]Multi-core throughput scales near-linearly until memory bandwidth effects dominate, which we leave to extended results.

Table 1: 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

11.1 Reverse Cuckoo Dmpf

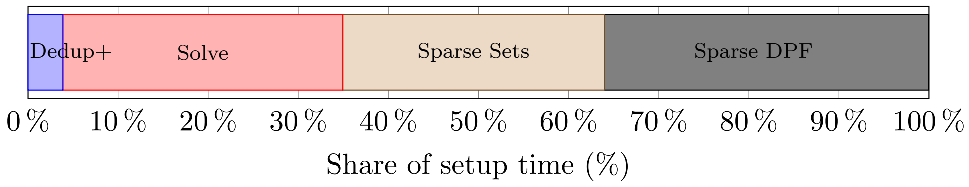

Figure 13: The realative costs of the Reverse Cuckoo DMPF sub proceedures. Data taken for $n = 2^{18}, t = 128$. Dedup+ denote deduplication, permuting and hashing. Solve is the binary system solver. Sparse Sets is the time requires to compute the sparse sets that are input to the Sparse Dpf. Sparse DPF is the time to evaluate the sparse tree and generate the leaf values.

Table 1 and Figure 13 report the setup and online (leaf expansion) performance of the Reverse Cuckoo DMPF across a range of parameters. As n and t increase, setup time grows as expected, but the per-expansion cost remains low and stable, with each online expansion taking just a few milliseconds even for large n. The results confirm that the bulk of computation is concentrated in the setup phase, while online expansions are efficient and suitable for high-throughput applications. This demonstrates the practicality of Reverse Cuckoo DMPF for scenarios that require fast, repeated expansions after a one-time setup cost.

11.2 Ring LPN PCG

We now evaluate our implementation of the Ring LPN-based pseudorandom correlation generator (PCG) for OLEs and Beaver triples over prime finite fields $\mathbb{F}_q$, as described in Section 10. Benchmarks are provided for our main construction (RcDmpf) as well as relevant baselines, focusing on throughput and communication for generating OLE correlations over different fields and vector lengths.

We conider two prime fields. Goldilocks is a 64 bit prime $q = 2^{64} - 2^{32} + 1$ which has special form and highly efficient modular operations[Pol22, Blo22]. Fp31 refers to the 31 bit prime $q = 15 \cdot 2^{27} + 1$ which we use montgomery reduction for.

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	–	MB 12.14
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

Figure 14: 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.

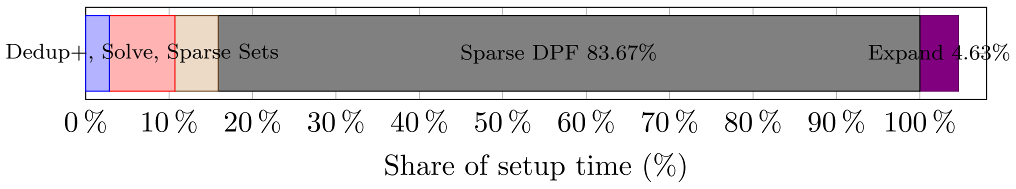

Figure 15: The realative costs of the Reverse Cuckoo DMPF based Ring LPN PCG compared to the online Expand time (4.6%). Data taken for $n = 2^{20}$. Dedup+ denote deduplication, permuting and hashing. Solve is the binary system solver. Sparse Sets is the time requires to compute the sparse sets that are input to the Sparse Dpf. Sparse DPF is the time to evaluate the sparse tree and generate the leaf values. Expand denote the time required to expand one set of leaves to generate the noise vectors.

Results summary. Table 14 reports wall-clock setup time, median expansion throughput, and (two-party) communication. We observe three consistent trends. First, the online throughput improves with n: for Goldilocks the median rises from 0.89×10^6 to 1.58×10^6 to 1.81×10^6 ops/s as n grows from 2^{16} to 2^{20}; for $\mathbb{F}_{p_{31}}$ it rises from 1.26×10^6 to 2.22×10^6 to 2.71×10^6 ops/s. For larger n the current implementation runs out of memory when allocating space to evaluate all of the Sparse DMPFs. Increased memory or a more careful implementation could mitigate this.

Second, the $\mathbb{F}_{p_{31}}$ pipeline is consistently faster online (about 1.4×-1.5× higher median ops/s than Goldilocks at the same n) while also using less setup communication (e.g., 1.69 GB vs. 2.02 GB at n=2^{20}). This difference would be larger if not for the exceptional efficency of the Goldilocks field[Pol22, Blo22]. Third, per-expansion communication is modest (≈6-13 MB) and does not grow with n.

We also compare to the SumDmpf approach and [Kel20]. The SumDmpf baseline provides lower throughput (e.g., $37,686$ ops/s at $n = 2^{16}$ in Goldilocks, and $47,250$ ops/s in Fp31) and lacks a setup phase, making it less practical for applications requiring high rates. [Kel20], measured in terms of OLE generation, achieves 275,548 (Goldilocks) and 309,588 (Fp31) ops/s at $n = 2^{20}$ with substantially higher per-expansion communication than our RcDmpf. In both fields, our RcDmpf protocol achieves an order of magnitude higher throughput and lower communication than these alternatives, especially in the online (expansion) phase.

Where setup time goes. The stacked bar in Figure 13 (Reverse Cuckoo DMPF) and the Goldilocks $n{=}2^{20}$ bar with a single expansion overlay (Figure 15) tell a consistent story: *Sparse DPF expansion of the sparse tree dominates setup.* For Goldilocks at $n{=}2^{20}$, Sparse DPF accounts for ≈82% of setup time, the linear *Solve* step is ≈8%, the *Sparse Sets* construction is ≈5%, and *Dedup+* (dedup/perm/hash) is ≈3% (the remainder is minor bookkeeping). The difference between the two bar charts is large because the PCG parameters are more sparse and many instances are evaluated in concurrently, hurting memory locality.

Online vs. offline cost at $n = 2^{20}$ (Goldilocks). For $n = 2^{20}$ in the Goldilocks field, our RcDmpf implementation achieves a median expansion throughput of $1,811,012$ ops/s, with a one-time setup cost of 12.49 s. Each expansion requires 12.99 MB of communication, while setup communication totals 170.44 MB. For comparison, [Kel20] achieves $275,548$ ops/s and 176.65 MB expand comm but no reported setup. The SumDmpf approach is much slower ($37,686$ ops/s, 12.4 MB comm). Thus, after as few as 10 expansions, the cumulative online cost is still less than the setup, and RcDmpf outperforms both alternatives on throughput and total bytes.

Impact of field choice. Switching to $\mathbb{F}_{p_{31}}$ improves online throughput at every n. For example, at $n = 2^{20}$ the RcDmpf achieves 2.71 M ops/s and 6.51 MB/expansion in Fp31, compared to 1.81 M ops/s and 12.99 MB/expansion in Goldilocks. Setup bytes are also lower in Fp31 (163.95 MB vs. 170.44 MB). The SumDmpf and [Kel20] baselines also show slightly better throughput in Fp31 but remain far below RcDmpf. Since per-expansion bytes are stable (and even lower in Fp31), using $\mathbb{F}_{p_{31}}$ is preferred for applications that tolerate a smaller field, giving higher speed and reduced bandwidth.

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.

[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.

[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.

[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.

[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.

[Ham15] Mike Hamburg. Ed448-goldilocks, a new elliptic curve. *Cryptology ePrint Archive*, 2015.

[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.

[RR22] Srinivasan Raghuraman and Peter Rindal. Blazing fast psi from improved okvs and subfield vole. In *Proceedings of the 2022 ACM SIGSAC conference on computer and communications security*, pages 2505–2517, 2022.

[RS21] Peter Rindal and Phillipp Schoppmann. Vole-psi: fast oprf and circuit-psi from vector-ole. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 901–930. Springer, 2021.

[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.

[Yeo23] Kevin Yeo. Cuckoo hashing in cryptography: Optimal parameters, robustness and applications. In *Annual International Cryptology Conference*, pages 197–230. Springer, 2023.

[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.

[Zha23] Shuqing Zhang. Efficient vole based multi-party psi with lower communication cost. *Cryptology ePrint Archive*, 2023.