Improved Constructions for Distributed Multi-Point Functions

Elette Boyle*, Niv Gilboa†, Matan Hamilis‡, Yuval Ishai§ and Yaxin Tu¶
*NTT Research and Reichman University, eboyle@alum.mit.edu
†Ben-Gurion University, gilboan@bgu.ac.il
‡Reichman University, matan@hamil.is
§Technion, yuvali@cs.technion.ac.il
¶Princeton University, yaxin.tu@princeton.edu

***Abstract*—A Distributed Point Function (DPF) is a cryptographic primitive used for compressing additive secret shares of a secret unit vector across two parties. Many DPF applications require compressed shares of a sparse *weight-t* vector, namely a Distributed *Multi*-Point Function (DMPF). Despite the strong motivation and prior optimization efforts, in most use cases the best practical implementation of DMPF is still a simple brute-force combination of t independent DPFs.**

We present new constructions and optimized implementations of DMPFs in different parameter regimes, providing significant efficiency savings over existing approaches. We showcase our new constructions within applications of pseudorandom correlation generators (PCGs) and 2-server private set intersection (PSI).

Incorporating our tools into the state-of-the-art PCG for "silent" generation of binary multiplication triples (FOLEAGE, Bombar *et al*, ePrint'24) yields a $\times 2.68$ improvement in throughput, with only $\times 1.4$ blowup in the seed size. On a single core of our benchmark machine, our implementation silently generates up to 22.1 million triples per second, outperforming even the best "non-silent" protocol (Roy, CRYPTO'22), which generates 16 million triples per second.

1. Introduction

Distributed Point Functions (DPF) [1], [2] are a cryptographic primitive enabling compressed additive secret-sharing of a length-N secret unit vector (or weight-1 vector) across two parties. This is often thought of as splitting a secret *point function* $f_{\alpha,\beta} : \{0,1\}^n \to \mathbb{G}$, which outputs β if the input is α and 0 otherwise, into two function shares described by short keys. Given the keys, the parties can locally generate additive shares (over the group $\mathbb{G}$) of $f_{\alpha,\beta}(x)$ on any input x.

Efficient DPFs can be constructed from any pseudorandom generator (PRG) with a λ-bit seed (typically $\lambda = 128$ in practice), where each DPF key size is of length $\approx \lambda n + \log|\mathbb{G}|$, and where both key generation and local evaluation require a small number of PRG invocation [2]. For example, even the full-domain evaluation (or "uncompressing" the length-N shared vector) requires fewer than $N = 2^n$ AES evaluations. The lightweight nature of DPFs has led to their adoption in a wide range of security applications, including pseudorandom correlation generators (PCG) [3], [4], [5], 2-server private set intersection (PSI) and private information retrieval (PIR) [2], [6], [7], Oblivious RAM [8], [9], secure aggregation [10] and much more.

Several such applications actually require a natural generalization of DPFs, where parties hold compressed shares of a secret sparse vector of weight t, or equivalently a multi-point function with t nonzero outputs. We refer to this primitive as a Distributed *Multi*-Point Function (DMPF).

Existing DMPFs. Within the literature, three existing DMPF construction approaches have been taken:

- *Naïve DMPF.* In its simplest form, an additively shared weight-t vector can be obtained by adding together t additively shared vectors of weight 1. In turn, one can build t-DMPF via t independent copies of standard DPF, resulting in $\times t$ cost in all metrics.
- *DMPF from batch codes.* More sophisticated constructions of DMPF have been shown based on probabilistic (combinatorial) batch codes (PBC) with appropriate parameters [11], [12], where the PBC is typically instantiated by cuckoo hashing [13], [14], [5].
- *SLAMP-FSS.* Concurrently, [15] introduces a construction of DMPF based on linear operations over large field, which will be compared to our constructions at the end of this section.

The PBC-based approach provides a (randomized) method for "bucketing" the weight-t vector into $(1+\epsilon)t$ shorter weight-1 vectors with redundancy, such that each weight-1 vector can be viewed as a DPF instance. The total length of the weight-1 vectors is $c \cdot N$ where in practical realizations c is a small integer like 2 or 3. This gives rise to an asymptotic advantage, in that the full-domain evaluation time scales with $c \cdot N$ rather than scaling with $t \cdot N$ as in naïve DMPF. However, it is hopeless to cut c down to 1 in the PBC-based DMPF. Moreover, the "bucketing" process consumes considerable computation resources, resulting in a questionable performance in typical parameter regimes.

Despite the strong motivation for optimization, and suggested asymptotic improvements via PBC, to date the most common implementations of DMPF remain the simple brute-force summation of t independent DPFs. Practical

advances beyond this $\times t$ naïve overhead have potential to yield direct application speedups.

1.1. Our Contributions

We initiate a systematic study of the DMPF primitive, including new improved constructions, optimized implementations, and a mapping of the parameter space to help developers identify which construction to use for best performance within a given application scenario. We additionally showcase our new constructions within two sample applications. We now give more details on each contribution.

New DMPF constructions. We provide (in Section 3) two new approaches built on the template of correcting pseudorandom strings on specific positions in an expanded pseudorandom tree, generalizing the ideas behind the tree-based DPF in [2].

- *OKVS-based DMPF.* One of our DMPF constructions makes use of Oblivious Key-Value Stores (OKVS), originally proposed as a primitive for private set intersection (PSI) protocols [16], and improved by a series of works [17], [18]. We use OKVS as a method to encode specific positions in the binary tree (the "keys") paired up with specific pieces of information needed for correction (the "values"), and include these encodings in the DMPF keys. The key size, key generation time and evaluation time of our OKVS-based DMPF are closely related to the OKVS instantiation. By applying the "random bands" RB-OKVS construction of [18], we obtain a DMPF scheme with fastest evaluation time for a wide range of practically useful parameters.
- *Big-state DMPF.* We propose another DMPF construction that directly generalizes the indicator bit in the standard tree-based DPF construction from [2] to an indicator *string*, to identify multiple nonzero entries of the shared vector. The costs of the big-state DMPF scheme grow quickly (quadratically) with the vector weight t; but, it provides dominating performance for full-domain evaluation for small-weight vectors, $3 \leq t \leq 70$. As discussed below, we found that big-state DMPF is a perfect fit to replace the use of DMPF in state-of-the-art PCG implementations [5]. We expect it to be useful for many other applications of DMPF that require small values of t.
  We additionally provide a distributed procedure for two parties to securely generate DMPF keys for a secret multi-point function with nonzero coordinates $\{x_1, \ldots, x_t\} \subset [N]$, where the parties begin with additive secret shares of the t values x_i. While the naïve DMPF provides the simplest such procedure, our protocol provides a solution for big-state DMPF that is significantly more efficient than distributed generation for the more complex PBC or OKVS DMPF constructions (see Section 3.5).

We remark that compared to the prior DMPF schemes, both our constructions feature a *one-time* traversal of a size-N binary tree for the full-evaluation vector share expansion, instead of t traversals in the naïve DMPF, or constant-many $c > 1$ traversals in the PBC-based DMPF. This feature is also present in the concurrent SLAMP-FSS.

Implementation & evaluation. We provide optimized implementations of all four DMPF approaches (naïve, PBC, OKVS, and Big-State) and benchmark their relative costs within different ranges of parameters t and N.

We reveal that the best choice of DMPF varies strongly based on parameter regimes, with each of the four constructions outperforming others within some region. For example, based on our experimental results, we identify that the optimal DMPF construction in terms of minimal FullEval time is largely independent of the choice of the domain size N, and varies for different ranges of t as follows:

- Naïve DMPF: For $t \leq 2$
- Big-state DMPF: For $3 \leq t \leq 70$
- OKVS-based DMPF: For $70 < t < 10^4$
- PBC-based DMPF: Comparable to OVKS starting at $t \geq 10^4$, and best starting around $t \approx 21{,}000$.

For the case of single-evaluation Eval, the picture is similar, but with an earlier switch between the big-state and OKVS. We refer the reader to Section 4 for further detail.

We remark that the dominating regimes for the Big-state and OKVS-based constructions constitute an important range for applications, as exemplified below.

Applications. As noted above, many existing applications of DPFs can benefit from the DMPF extension. For example, in the context of secure aggregation or multi-candidate voting [2], [10], if a DPF is used to change the value of a single (secret) entry, then DMPF can be used to change the values of up to t entries.

We chose to showcase the concrete efficiency benefits of our new DMPF constructions within two concrete applications: (1) pseudorandom correlation generators (PCGs) [3] for "silently" generating useful correlations, and (2) 2-server *unbalanced* private set intersection (PSI) [7].

These two applications have complementary requirements for their underlying DMPF, as well as different relevant parameter regimes (N, t). In the PCG application the domain size N is feasible and t is typically small. Here the most critical metric is the speed of full-domain evaluation DMPF.FullEval, which is used for fast PCG seed expansion, and a secondary metric is the efficiency of distributing the key generation algorithm DMPF.Gen. In the PSI application, N is typically infeasible (e.g., 2^{80}), t can be arbitrary, and the most critical metrics the are key size and the speed of single-point evaluation DMPF.Eval. We now give a brief account of the impact of our results on these two applications.

Pseudorandom correlation generators. A PCG is a general tool for a "silent" secure generation of many instances of a useful target correlation, such as oblivious transfer (OT), oblivious linear evaluation (OLE) or multiplication triple (MT). Current practical PCG constructions for OLE and MT, which follow the Ring-LPN based approach from [4], crucially rely on DMPF as an underlying primitive. We showcase the usefulness of our big-state DMPF construction

by improving the efficiency of FOLEAGE [5], the best current PCG for MT over $\mathbb{F}_2$, which yields the fastest current protocols for secure computation of Boolean circuits.

By replacing their use of the naïve DMPF with our big-state construction on their set of parameters (in particular, $t = 27$), we improve the overall throughput of binary MT generation $\times 2.68$, at the cost of $\times 1.4$ increase in seed size. An alternative DMPF proposed in [5] (without an implementation) uses PBC based on cuckoo-hashing to improve performance, noting that it lacks an efficient method for distributing the seed generation. Based on our optimized PBC-based DMPF benchmarks, we see that our big-state DMPF still provides higher throughput even against this alternative (see Section 5.1), while simultaneously admitting a reasonable cost for distributed seed generation.

Ultimately, on a single core of our benchmark machine our modified implementation silently generates up to 22.1 million binary MT instances per second. This throughput outperforms even the best "non-silent" alternative protocol for the same task [19], which generates 16 million binary MT instances per second. Moreover, unlike [19], the performance of the DMPF-based PCG approach degrades gracefully with the underlying MT or OLE field size, potentially being useful for secure arithmetic computations and more.

Private Set Intersection. Finally, a natural use case for DMPF is unbalanced PSI in the 2-server model considered in [7]. Here a client wants to find the intersection of its small set of t keywords (each of length n) with a potentially big (or even "streaming") set of keywords held by two remote servers. PSI in this setting can be useful for a variety of applications, including private searching on streaming data (e.g., finding documents containing one of t keywords), private contact discovery, measuring advertisement conversion rates, and more. Our improved big-state DMPF (resp., OKVS DMPF) constructions provide improved running times for small (resp., big) values of t.

Concurrent work. The concurrent work [15] introduces a DMPF construction named SLAMP-FSS, which shares a similar tree structure with our DMPF template. It provides an asymptotic analysis of key size and running time in terms of PRG invocations, but does not take into account other expensive operations (e.g., multiplications over a large field), and moreover lacks a practical evaluation. Upon closer inspection, SLAMP-FSS can be seen as an instance of our DMPF template in Section 3.1, with λ-bit strings interpreted as $\mathbb{F}_{2^\lambda}$ elements, and the length parameter $l \geq t\lambda$. More concretely, SLAMP-FSS is similar to an instance of our OKVS-based DMPF from Section 3.3, where OKVS is instantiated by the random-matrix OKVS ([16]) over field $\mathbb{F}_{2^\lambda}$. This does not cover our big-state DMPF or OKVS-based DMPF with a more time-efficient OKVS instantiation. While both SLAMP-FSS and the (random-matrix-)OKVS-based DMPF have almost the same number of PRG invocations in the evaluation algorithm, the PRG used in SLAMP-FSS has output length about t times longer than the PRG used in our OKVS-based construction, making PRG invocations more expensive. In addition, both constructions involve the same number of large field multiplications, which is not accounted for in [15], but we found these to be comparably expensive to PRG invocations in practice. For this reason, we optimized the evaluation time of OKVS-based DMPF by instantiating OKVS with recent practical constructions from [17], [18].

2. Preliminaries

2.1. Basic Notations

Throughout the paper, we use $[n]$ to denote the ordered list of numbers $1, 2, \ldots n$. For a list L of length l, we use $L[k]$ $(1 \leq k \leq l)$ to denote the kth entry of this list, and we use $L[i \ldots j]$ to denote the list $(L[i], L[i+1], \ldots, L[j])$. If the entries of L are in the domain of a function f, define $f(L) := (f(x))_{x \in L}$ to be the list obtained by entry-wise applying f to L. For $b \in \{0, 1\}$, we use $\bar{b}$ to denote $1 - b$. We use $[\![X]\!]_0$ and $[\![X]\!]_1$ to denote the random additive shares of an entity X whenever it is applicable.

(T, ϵ)-security. When discussing the concrete security of cryptographic primitives, we will use terminology of *(T, ϵ)-security* to refer to the adversarial advantage being at most ϵ for any adversary running in time T, where we consider the adversary to be a non-uniform probabilistic polynomial-time machine.

Point and multi-point functions. We will often represent weight-1 vectors of length n as a function with input domain $[n]$ and corresponding truth table. Namely, given a domain size N and Abelian group $\mathbb{G}$, a *point function* $f_{\alpha,\beta} : [N] \to \mathbb{G}$ for $\alpha \in [N]$ and $\beta \in \mathbb{G}$ evaluates to β on input α and to $0 \in \mathbb{G}$ on all other inputs. We denote by $\hat{f}_{\alpha,\beta} = (N, \hat{\mathbb{G}}, \alpha, \beta)$ the representation of such a point function, where $\hat{\mathbb{G}}$ denotes the description of the group $\mathbb{G}$. A *t-point function* $f_{A,B} : [N] \to \mathbb{G}$ for $A = \{\alpha_1, \cdots \alpha_t\} \subset N$ listed in ascending order and $B = (\beta_1, \cdots, \beta_t) \in \mathbb{G}^t$ evaluates to β_i on input α_i for $1 \leq i \leq t$ and to 0 on all other inputs. Let $\hat{f}_{A,B}(N, \hat{\mathbb{G}}, t, A, B)$ denote the representation of such a t-point function. Call the collection of all t-point functions for all t *multi-point functions*.

2.2. Distributed Multi-Point Functions

We begin by defining the notion of distributed multi-point functions (DMPF), which is a generalization of distributed point function (DPF [20], [2]).

Definition 1 (DMPF [12], [14], [2])**.** *A (2-party)* distributed multi-point function (DMPF) *is a triple of algorithms* $\Pi = (\mathsf{Gen}, \mathsf{Eval}_0, \mathsf{Eval}_1)$ *with the following syntax:*

1) $\mathsf{Gen}(1^\lambda, \hat{f}_{A,B}) \to (k_0, k_1)$*: On input security parameter* $\lambda \in \mathbb{N}$ *and point function description* $\hat{f}_{A,B} = (N, \hat{\mathbb{G}}, t, A, B)$*, the (randomized) key generation algorithm* Gen *returns a pair of keys* $k_0, k_1 \in \{0, 1\}^*$.
2) $\mathsf{Eval}_b(1^\lambda, k_b, x) \to y_b$*: On input key* $k_b \in \{0, 1\}^*$ *and input* $x \in [N]$ *the (deterministic) evaluation algorithm of server* b*,* Eval_b *returns* $y_b \in \mathbb{G}$.

3) $\mathsf{FullEval}_b(1^\lambda, k_b)$*: Returns* $\{\mathsf{Eval}_b(1^\lambda, k_b, x)\}_{x\in[N]}$.

We require Π *to satisfy the following requirements:*

Correctness: *For every* λ, $\hat{f}_{A,B} = (N, \hat{\mathbb{G}}, t, A, B)$ *such that* $B \in \mathbb{G}^t$, *and* $x \in [N]$

$$\Pr\left[\begin{array}{c}(k_0,k_1) \leftarrow \mathsf{Gen}(1^\lambda, \hat{f}),\\ \mathsf{Eval}_0(k_0,x) + \mathsf{Eval}_1(k_1,x) = f_{A,B}(x)\end{array}\right] = 1$$

Security: *For any corrupted party* $b = 0, 1$, *there exists a probabilistic polynomial-time simulator* Sim *such that for every* $\hat{f} = \hat{f}_{A,B} = (N, \hat{\mathbb{G}}, t, A, B)$, *the outputs of the following experiments* Real *and* Ideal *are computationally indistinguishable:* [1]

- Real : $(k_0, k_1) \leftarrow \mathsf{Gen}(1^\lambda, \hat{f})$, *output* k_b.
- Ideal : *Output* $\mathsf{Sim}(1^\lambda, N, \hat{\mathbb{G}}, t)$.

Concretely, if for any distinguisher running in time T, *its distinguishing advantage is upper bounded by* ϵ, *we say that the DMPF is* (T, ϵ)*-secure.*

2.3. Probabilistic Batch Codes

We provide a description of probabilistic batch codes (PBC), on which prior constructions of DMPF are based. A batch code encodes a length-N string $x = x_1 x_2 \dots x_N$ to m codewords (or buckets) $C_1, C_2, \dots, C_m$ of total length M, such that decoding any subset $I \subset [N]$ of size t requires querying at most one symbol in each codeword. An (N, M, t, m, ϵ)-PBC ([11], [21], [22]) is a relaxed variant of batch codes permitting a decoding error with probability ϵ over the random coins of the encoder.

Definition 2 (Probabilistic Batch Code (PBC)[11], [21], [22])**.** *An* $(N, M, t, m, l, \epsilon)$-PBC *over alphabet* Σ *is given by a pair of efficient algorithms* $(\mathsf{Encode}, \mathsf{Decode})$ *with public randomness* r *such that:*

- $\mathsf{Encode}_r(x \in \Sigma^N) \to (C_1, C_2, \dots, C_m)$*: Any string* $x \in \Sigma^N$ *is encoded into* m *codewords (or 'buckets')* $C_1, C_2, \cdots C_m \in \Sigma^*$ *of total length* M.
- $\mathsf{Decode}_r(I, C_1, C_2, \dots, C_m) \to x[I]$*: On input a set* $I \subseteq [N]$ *of* $\leq t$ *distinct elements in* $[N]$ *and* m *codewords, recover the subset* $x[I]$ *of* x *indexed by* I, *while querying at most* l *positions in each codeword.*
- ***Correctness****: for any string* x *and any set* I *of* t *distinct indices in* $[N]$,

$$\Pr_r[(C_1, \dots, C_m) \leftarrow \mathsf{Encode}_r(x),\\ x[I] \neq \mathsf{Decode}_r(I, C_1, \dots, C_m)] \leq \epsilon$$

By default, we assume the batch code to be combinatorial *and* systematic*, which means that each symbol of* x *is copied to some fixed positions in the buckets. This is formalized by the two sub-processes of* Encode_r *and* Decode_r *respectively:*

- $\mathsf{Position}_r(k \in [N]) \to C_{i_1}[j_1], C_{i_2}[j_2], \dots$*: On input an index* $k \in [N]$, *output the sequence of positions in buckets that* $x[k]$ *is copied to.*
- $\mathsf{Schedule}_r(I \subseteq [N], k \in I) \to C_{i_1}[j_1], C_{i_2}[j_2], \dots$*: For any* $I \subseteq [N]$ *such that* $|I| \leq t$, *and* $k \in I$, $\mathsf{Schedule}_r(I, k)$ *outputs a set of positions in buckets relevant to* k *that* Decode_r *reads when decoding to* x_k *for* $k \in I$. *For all* $i \in [m]$, $|C_i \cap \bigcup_{k\in I} \mathsf{Schedule}_r(I, k)| \leq l$.

In a combinatorial and systematic PBC, $\mathsf{Encode}_r(x)$ *encodes* x *by copying each* $x_k (k \in [N])$ *to the positions* $\mathsf{Position}_r(k)$, *while* $\mathsf{Decode}_r(I, C_1, \dots, C_m)$ *decodes each* $x_k (k \in I)$ *by reading the letters at the positions* $\mathsf{Schedule}_r(I, k)$.

We will focus combinatorial and systematic PBC [11], [21], [23] with $l = 1$, which has been implemented in the literature [11], [6], [22] by w-way cuckoo hashing algorithm [24].

PBC from w-way cuckoo hashing [11], [6], [22]. Cuckoo hashing [24] is a hashing technique that uses w independent random hash functions $h_1, h_2, \cdots, h_w : \{0,1\}^* \to [m]$ to allocate t balls to $m = et (e > 1)$ buckets and guarantees that each ball b is allocated to one of the buckets $h_1(b), h_2(b), \dots, h_w(b)$, while each bucket contains at most one ball. At a high level, the w-way cuckoo hashing algorithm repeats the following process until termination or some upperbound on the number of iterations is achieved:

1. Choose an arbitrary unallocated ball b. If there is no unallocated ball, output the current allocation.
2. Choose a random $i \in [w]$, and compute the bucket index $h_i(b)$. If this bucket is empty, then allocate b to this bucket and go to step 1. If this bucket is not empty and filled with ball b', then evict b', allocate b to this bucket, and repeat step 2 with unallocated ball b'.

We call it a *failure* whenever the algorithm fails to output a desired allocation. We'll choose cuckoo hashing parameters that the failure probability is at most $\epsilon = 2^{-\lambda_{\mathsf{stat}}}$. According to the empirical result from [6, Appendix B], when $w = 3$ and $t \geq 4$, the failure probability can be computed by

$$\begin{aligned}\lambda_{\mathsf{stat}} &= a_t \cdot e - b_t - \log t\\ a_t &= 123.5 \cdot \mathsf{CDF}_{\mathsf{Normal}}(x = t, \mu = 6.3, \sigma = 2.3)\\ b_t &= 130 \cdot \mathsf{CDF}_{\mathsf{Normal}}(x = t, \mu = 6.45, \sigma = 2.18)\end{aligned}$$

where $e = m/t$ is the expansion parameter and $\mathsf{CDF}_{\mathsf{Normal}}(x, \mu, \sigma)$ is the cumulative distribution function on input x for the normal distribution with expectation μ and standard deviation σ. By default, we set the statistical security parameter $\lambda_{\mathsf{stat}} = 40$. When $t \geq 12$, the $\mathsf{CDF}_{\mathsf{Normal}}$ values are ≥ 0.99 and we can compute $e = \frac{170+\log t}{123.5}$ and $m = e \cdot t$ according to the concrete value of t.

Next we present the instantiation of PBC using cuckoo hashing.

Construction 1 (PBC from cuckoo hashing[11], [14])**.** *Given* w*-way cuckoo hashing as a sub-procedure allocating* t *balls to* m *buckets with failure probability* ϵ, *an* $(N, wN, t, m, 1, \epsilon)$-PBC *is as follows:*

1. In a scenario where the corrupted party can adaptively choose inputs $S = \{x_1, x_2, \dots, x_m\} \subseteq [N]$ on which the two parties publish their evaluations, our security definition guarantees that there exists a probabilistic polynomial-time simulator capable of simulating the view of the corrupted party, by running Sim to get k_b and simulating the honest party's evaluation by computing $f_{A,B}(x_i) - \mathsf{Eval}_b(k_b, x_i)$ for each query x_i.

- $\mathsf{Encode}_r(x \in \Sigma^N) \to (C_1, \cdots, C_m)$*: For all* $k \in [N]$*, copy* $x[k]$ *to the positions indicated by* $\mathsf{Position}_r(k)$.
- $\mathsf{Decode}_r(I, C_1, \cdots, C_m) \to \{x[i]\}_{i \in I}$*: For each* $k \in I$*, obtain* $x[k]$ *from the position indicated by* $\mathsf{Schedule}_r(I, i)$.

with the following sub-processes:

- $\mathsf{Position}_r(k \in [N]) \to C_{i_1}[j_1], \ldots, C_{i_w}[j_w]$*: Interpret* r *as a pseudorandom permutation* $P : [w] \times [N] \to [m] \times [B]$ *where* $B = wN/m$*, such that for each* $l \in [w]$ *and* $k \in [N]$*,* $P(l, k) = (i, j)$ *indicating the position* $C_i[j]$*. Then* $\mathsf{Position}_r(k)$ *outputs* $\{C_{i_l}[j_l] : (i_l, j_l) \leftarrow P(l, k)\}_{l \in w}$.
- $\mathsf{Schedule}_r(I, k)$*: Interpret* r *as a pseudorandom permutation* $P : [w] \times [N] \to [m] \times [B]$*. For* $l \in [w]$*, let* $h_l : [N] \to [m]$ *be* $h_l(k) = i$ *where* $P(l, k) = (i, j)$*. This gives* w *hash functions* $h_1, h_2, \cdots h_w$ *that maps from* $[N]$ *to* $[m]$*. For* I *of size at most* t*, run the* w*-way cuckoo hashing algorithm to allocate the indices in* I *to* m *buckets, such that* $k \in I$ *is allocated to the bucket* $h_{l_k}(k)$*, and each bucket contains at most one index in* I*. In the end, for* $k \in I$*,* $\mathsf{Schedule}_r(I, k)$ *outputs the position* $C_i[j]$ *such that* $P(l_k, k) = (i, j)$.

Remark 3. *By using a pseudorandom permutation* $P : [w] \times [N] \to [m] \times [B]$*, it is direct to obtain the positions in buckets where any* $k \in [N]$ *is encoded to, however the hash functions* $h_1, \ldots, h_w$ *are not completely independent. Nevertheless, it is empirically verified in [14] that the cuckoo hashing algorithm still succeeds with sufficient probability.*

The probability of the above PBC scheme being incorrect is equal to the failure probability of $\mathsf{Schedule}_r$ on set I, which equals the cuckoo hashing failure probability ϵ.

2.4. Oblivious Key-Value Stores

One of our DMPF constructions makes use of Oblivious key-value stores (OKVS). OKVS was proposed as a primitive for private set intersection (PSI) protocols [16], and improved by a series of works [17], [18].

Definition 4 (OKVS [16], [17], [18])**.** *An Oblivious Key-Value Stores scheme is a pair of randomized algorithms* $(\mathsf{Encode}_r, \mathsf{Decode}_r)$ *with respect to a statistical security parameter* λ_{stat}*, a computational security parameter* λ*, a randomness space* $\{0,1\}^\kappa$*, a key space* $\mathcal{K}$*, a value space* $\mathcal{V}$*, input length* t *and output length (code size)* m*. The algorithms have the following syntax:*

1) $\mathsf{Encode}_r(\{(k_1, v_1), (k_2, v_2), \cdots, (k_t, v_t)\}) \to P$*: On input* t *key-value pairs with distinct keys, the encode algorithm with randomness* r *outputs an encoding* $P \in \mathcal{V}^m \cup \perp$.
2) $\mathsf{Decode}_r(P, k) \to v$*: On input an encoding from* $\mathcal{V}^m$ *and a key* $k \in \mathcal{K}$*, output a value* v.

We require the scheme to satisfy

Correctness: *For all* $S \in (\mathcal{K} \times \mathcal{V})^t$*,* $\Pr[r \leftarrow \{0,1\}^\kappa, \mathsf{Encode}_r(S) = \perp] \leq \epsilon$*. For all* $S \in (\mathcal{K} \times \mathcal{V})^t$ *and* $r \in \{0,1\}^\kappa$ *such that* $\mathsf{Encode}_r(S) \to P \neq \perp$*,* $\mathsf{Decode}_r(P, k)$ *outputs* v *whenever* $(k, v) \in S$*. By default we require* $\epsilon = 2^{-\lambda_{\mathsf{stat}}}$.

Obliviousness: *Given any distinct sets* $\{k_1^0, k_2^0, \cdots, k_t^0\}$ *and* $\{k_1^1, k_2^1, \cdots, k_t^1\}$ *that are different, if they are paired with random values then their encodings are computationally indistinguishable, i.e.,*

$$\{r, \mathsf{Encode}_r(\{(k_1^0, v_1), \cdots, (k_t^0, v_t)\})\}_{\substack{v_1, \cdots, v_t \leftarrow \mathcal{V}, \\ r \leftarrow \{0,1\}^\kappa}}$$
$$\approx_c \{r, \mathsf{Encode}_r(\{(k_1^1, v_1), \cdots, (k_t^1, v_t)\})\}_{\substack{v_1, \cdots, v_t \leftarrow \mathcal{V}, \\ r \leftarrow \{0,1\}^\kappa}}$$

Concretely, if for any distinguisher running in time T*, the distinguishing advantage is upper bounded by* ϵ*, then the OKVS scheme is* (T, ϵ)-oblivious.

A linear OKVS *has an additional requirement:*

- ***Linearity:*** *There exists a function family* $\{\mathsf{row}_r : \mathcal{K} \to \mathcal{V}^m\}_{r \in \{0,1\}^\kappa}$ *such that* $\mathsf{Decode}_r(P, k) = \langle \mathsf{row}_r(k), P \rangle$.

The Encode process for a linear OKVS is the process of sampling a random P from the set of solutions of the linear equation system $\{\langle \mathsf{row}_r(k_i), P \rangle = v_i\}_{1 \leq i \leq t}$.

The work of [18] gives a state-of-the-art (linear) OKVS construction (RB-OKVS) using a random band matrix as the coefficient matrix of $\{\langle \mathsf{row}_r(k_i), P \rangle = v_i\}_{1 \leq i \leq t}$ where each row has a width-w random binary band. The encoding can be efficiently done in $O(tw + t \log t)$ time leveraging the structure of the random bands. The decoding time is typically dominated by w additions in $\mathbb{F}$. To guarantee the success of Encode, denoting $m = (1 + e)t$, [18] suggests taking $\lambda_{\mathsf{stat}} = 2.751\delta w + g(\delta, n)$ (where $g(\delta, n)$ is computed empirically), and typically setting $\delta \in \{3\%, 5\%, 7\%, 10\%\}$ while w lying roughly between 200 and 600, with the choice of t varying from 2^{10} to 2^{24}.

2.5. Previous DMPF Constructions

In this section, we present existing DMPF constructions as baselines for comparison with our own constructions.

Naïve DMPF. The distributed point function (DPF [20], [2]) is a special case of DMPF where $t = 1$. One can construct a DMPF scheme for t-point functions by simply summing t DPFs, which is the most widely used construction. We refer to this construction as the naïve DMPF. When the DPF scheme is correct and secure, the naïve DMPF is also correct and secure. We note that the keysize and running time of Gen, Eval and FullEval of the naïve DMPF equals $t\times$ those of DPF, serving as our baseline for comparison.

PBC-based DMPF. Another known construction of DMPF is based on Probabilistic (combinatorial) Batch Codes (PBC) [12], [13], [14], [5]. The PBC-based DMPF for a t-point function $f_{A,B}$ with domain size N uses an (N, M, t, m, ϵ)-PBC and m DPF instances with domain size $|C_1|, |C_2|, \ldots, |C_m|$ respectively. It uses PBC to encode the truth table of $f_{A,B}$ to codewords $(C_1, C_2, \ldots, C_m)$, and by the property of PBC, the nonzero points of $f_{A,B}$ can be distributed to a collection of entries with one entry in each codeword, identified by $\mathsf{Schedule}(A)$. Therefore, the desired

DMPF can be converted to m DPF's for $C_1, C_2, \ldots, C_m$ respectively.

Construction 2 (PBC-based DMPF)**.** *Given a family* $\{\mathsf{DPF}_{N',\mathbb{G}}\}_{N'}$ *for any domain of size* $N' \leq N$ *and output group* $\mathbb{G}$*, and an* (N, M, t, m, ϵ)-PBC*, we can construct a* DMPF *for* t*-point functions with domain size* N *and output group* $\mathbb{G}$ *as follows:*

1) $\mathsf{Gen}(1^\lambda, \hat{f}_{A,B}) \to (k_0, k_1)$*: Let* PBC *encode* $[N]$ *to codewords* $C_1, \ldots, C_m$*. Invoke* PBC.Schedule *on* A *and all entries of* A *to get the ordered list of entries* $(C_{i_k}[j_k])_{k\in[t]}$.
   For all $1 \leq k \leq t$ *For* $1 \leq i \leq m$*, let* $f_i : [|C_i|] \to \mathbb{G}$ *be the following:*
   - *If there is exactly one* $k \in [t]$ *that* $i_k = i$*, then set* f_i *to be the point function that outputs* $B[k]$ *on* j_k *and* $0_\mathbb{G}$ *elsewhere.*
   - *Otherwise set* f_i *to be the all-zero function.*

   For $1 \leq i \leq m$*, invoke* $\mathsf{DPF.Gen}(1^\lambda, \hat{f}_i) \to (k_0^i, k_1^i)$*. Set* $(k_0, k_1) = (\{k_0^i\}_{i\in[m]}, \{k_1^i\}_{i\in[m]})$.
2) $\mathsf{Eval}_b(k_b, x) \to y_b$*: Invoke* $\mathsf{PBC.Position}(x)$ *to obtain all occurrences of* x *in codewords* $C_{i_1}[j_1], \ldots, C_{i_s}[j_s]$*. Compute* $y_b = \sum_{l=1}^{s} \mathsf{DPF.Eval}_b(k_b^{i_l}, j_l)$.
3) $\mathsf{FullEval}_b(k_b) \to Y_b$*: For* $1 \leq i \leq m$*, compute* $Y_b^i = \mathsf{DPF.FullEval}_b(k_b^i)$*. For all* $x \in [N]$*, invoke* $\mathsf{PBC.Position}(x) \to C_{i_1}[j_1], \ldots, C_{i_s}[j_s]$*, and set* $Y_b[x] \leftarrow \sum_{l=1}^{s} Y_b^{i_l}[j_l]$.

The scheme is correct with at least $1 - \epsilon$ probability. When instantiating PBC using w-way cuckoo hashing as in Construction 1, the running time of Eval is the sum of the time for $\mathsf{PBC.Position}(x)$ plus the time for w invocations of DPF.Eval. Similarly, the running time of FullEval is the time for N invocations of PBC.Position plus w invocations of DPF.FullEval. Therefore, one may expect the evaluation time for the above construction of DMPF to be dependent to the small constant parameter w, independent of t. We will discuss about its efficiency in later sections.

3. New DMPF constructions

We present two new constructions of DMPF: big-state DMPF in Section 3.2 and OKVS-based DPF in Section 3.3. Both constructions will share a common skeletal template, which we describe in Section 3.1.

3.1. DMPF template

We begin by introducing the DMPF template in Figure 1, which is inspired by the tree-structured construction of DPF in [2]. In the template, the length parameter l and methods Initialize, GenCW, GenConvCW, Correct, ConvCorrect are left to be determined by specific constructions in the later sections.

High-level overview. Each key k_b $(b = 0, 1)$ generated by $\mathsf{Gen}(1^\lambda, \hat{f}_{A,B})$ spans a depth-n (n is the input length of $\hat{f}_{A,B}$) complete binary tree T_b, referred to as the 'evaluation tree'. Each node in T_b is approached by a path starting from the root, which corresponds to a string in $\{0,1\}^{\leq n}$ where 0 stands for going left and 1 stands for going right. We call a path that corresponds to any nonzero input $a \in A$ an 'accepting path'.

Every node in the evaluation tree T_b is associated with a λ-bit pseudorandom seed string and an l-bit pseudorandom sign string (l is an adjustable parameter). All strings in evaluation tree T_b are determined by the seed$\|$sign string at its root, and the set of corrections words $\{\mathsf{CW}^{(i)}\}_{1\leq i\leq n}$ at each layer, computed layer by layer by the following steps:

1) For each node v with strings seed_v and sign_v in the $(i-1)$th layer, generate its children's seed and sign strings ($\mathsf{seed}_0\|\mathsf{sign}_0$ for the left child and $\mathsf{seed}_1\|\mathsf{sign}_1$ for the right child) by first setting $\mathsf{seed}_0\|\mathsf{sign}_0\|\mathsf{seed}_1\|\mathsf{sign}_1 = G(\mathsf{seed}_v)$ where $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2l}$ is a pseudorandom generator.
2) Compute a correction by $C_{\mathsf{seed}}\|C_{\mathsf{sign}^0}\|C_{\mathsf{sign}^1} \leftarrow \mathsf{Correct}(x_1 \ldots x_{i-1}, \mathsf{sign}, \mathsf{CW}^{(i)})$.
3) Add C_{seed} to seed_0 and seed_1, and add $C_{\mathsf{sign}_0}, C_{\mathsf{sign}_1}$ to $\mathsf{sign}_0, \mathsf{sign}_1$ respectively in order to correct them to satisfy desired correlations.

We expect the strings in T_0 and T_1 satisfy the following correlations, which are the core of this template:

Definition 5 (Correlations satisfied by evaluation trees)**.**

1) T_0 *and* T_1 *have identical* seed *and* sign *strings on every node not lying on any accepting path.*
2) *For a node lying on an accepting path, its* seed *strings in* T_0 *and* T_1 *are pseudorandom and independent, while its* sign *strings are two correlated pseudorandom strings following some correlation designed by specific realizations. The correlation is an XOR correlation, meaning the two* sign *strings should add (by XOR) up to a specific string.* [2]

The first correlation says that T_0 and T_1 to have identical strings on every node exiting an accepting path (i.e., the node that is not on an accepting path but its parent is): if a parent node is associated with the same strings in T_0 and T_1, then each of its children is associated with the same strings in T_0 and T_1, and so is each of the nodes in the subtree rooted at the parent node. To force the first correlation, we expect that at each node exiting an accepting path, the correction C_{seed} and C_{sign} for this node eliminates the difference between its original seed$\|$sign strings generated by the PRG in T_0 and T_1. To force the second correlation, we expect that at each node on an accepting path, the correction C_{seed} for this node should preserve the pseudorandomness and independence of the original seed strings in T_0 and T_1,

2. In the big-state realization in Figure 2 the two sign strings add up to a unit vector indicating which accepting path the node is on, and in the OKVS-based realization in Figure 3 the two sign bits add up to 1 if and only if the node is on an accepting path.

Figure 1: The template of our DMPF schemes. We leave the sign string length l, methods Initialize, GenCW, GenConvCW, Correct, ConvCorrect to be determined by specific constructions.

Public parameters:

The t-point function family $\{f_{A,B}\}$ with t an upper bound on the number of nonzero points, input domain $[N] = \{0,1\}^n$ and the output group $\mathbb{G}$.

Suppose there is a public PRG $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2l}$. Parse $G(x) = G_0(x)\|G_1(x)$ to the left half and right half of the output. Moreover, for simplicity, for $b = 0, 1$ define $G_b^{\mathsf{seed}} : \{0,1\}^\lambda \to \{0,1\}^\lambda$ to be $G_b^{\mathsf{seed}}(x) = G_b(x)[1 \dots \lambda]$, the first λ bits. Similarly, define $G_b^{\mathsf{sign}} : \{0,1\}^\lambda \to \{0,1\}^l$ to be $G_b^{\mathsf{sign}}(x) = G_b(x)[(\lambda+1) \dots (\lambda+l)]$, the last l bits of G_b. Denote $G^{\mathsf{sign}} : \{0,1\}^\lambda \to \{0,1\}^{2l}$ to be $G^{\mathsf{sign}}(x) = G_0^{\mathsf{sign}}(x)\|G_1^{\mathsf{sign}}(x)$.

Suppose there is a public PRG $G_{\mathsf{conv}} : \{0,1\}^\lambda \to \mathbb{G}$.

procedure $\textsc{Gen}(1^\lambda, \hat{f}_{A,B})$

Denote $A = (\alpha_1, \cdots, \alpha_t) \in (\{0,1\}^n)^t$ in lexicographic order, $B = (\beta_1, \cdots, \beta_t) \in \mathbb{G}^t$. If $|A| < t$, extend A to size-t with arbitrary $\{0,1\}^n$ strings and B with 0's.

For $0 \le i \le n-1$, let $A^{(i)}$ denote the sorted list of i-bit prefixes of strings in A, without duplicates. Specifically, $A^{(0)}$ is the empty string $A^{(0)} = [\epsilon]$.

For $0 \le i \le n-1$ and $b = 0, 1$, initialize empty lists $\mathsf{seed}_b^{(i)}$, $\mathsf{sign}_b^{(i)}$ and $V^{(i)}$.

$\mathsf{Initialize}(\{\mathsf{seed}_b^{(0)}, \mathsf{sign}_b^{(0)}\}_{b=0,1})$.

for $i = 1$ to n **do**

  for $k = 1$ to $|A^{(i-1)}|$ **do**

    For $c = 0, 1$, compute $\Delta\mathsf{seed}^c = G_c^{\mathsf{seed}}(\mathsf{seed}_0^{(i-1)}[k]) \oplus G_c^{\mathsf{seed}}(\mathsf{seed}_1^{(i-1)}[k])$ and $\Delta\mathsf{sign}^c = G_c^{\mathsf{sign}}(\mathsf{seed}_0^{(i-1)}[k]) \oplus G_c^{\mathsf{sign}}(\mathsf{seed}_1^{(i-1)}[k])$.

    if $A^{(i-1)}[k]\|0 \in A^{(i)}$ and $A^{(i-1)}[k]\|1 \in A^{(i)}$ **then**

      Randomly sample $r \leftarrow \{0,1\}^\lambda$ and append $r\|\Delta\mathsf{sign}^0\|\Delta\mathsf{sign}^1$ to $V^{(i-1)}$.

    else

      Suppose $A^{(i-1)}[k]\|z \in A^{(i)}$. Append $\Delta\mathsf{seed}^{1-z}\|\Delta\mathsf{sign}^0\|\Delta\mathsf{sign}^1$ to $V^{(i-1)}$.

    end if

  end for

  $\mathsf{CW}^{(i)} \leftarrow \mathsf{GenCW}(i, A, V^{(i-1)})$.

  for $k = 1$ to $|A^{(i-1)}|$ and $z = 0, 1$ **do**

    Compute $C_{\mathsf{seed},b}\|C_{\mathsf{sign}^0,b}\|C_{\mathsf{sign}^1,b} \leftarrow \mathsf{Correct}(A^{(i-1)}[k], \mathsf{sign}_b^{(i-1)}[k], \mathsf{CW}^{(i)})$ for $b = 0, 1$, where $|C_{\mathsf{seed},b}| = \lambda$ and $|C_{\mathsf{sign}^0,b}| = |C_{\mathsf{sign}^1,b}| = l$.

    if $A^{(i-1)}[k]\|z \in A^{(i)}$ **then**

      Append the first λ bits of $G_z(\mathsf{seed}_b^{(i-1)}[k]) \oplus (C_{\mathsf{seed},b}\|C_{\mathsf{sign}^z,b})$ to $\mathsf{seed}_b^{(i)}$ and the rest l bits to $\mathsf{sign}_b^{(i)}$.

    end if

  end for

end for

$\mathsf{CW}^{(n+1)} \leftarrow \mathsf{GenConvCW}(A, B, \left(G_{\mathsf{conv}}(\mathsf{seed}_0^{(n)}[k]) - G_{\mathsf{conv}}(\mathsf{seed}_1^{(n)}[k])\right)_{1 \le k \le |A|}, \mathsf{sign}_0^{(n)}, \mathsf{sign}_1^{(n)})$.

Set $k_b \leftarrow (\mathsf{seed}_b^{(0)}, \mathsf{sign}_b^{(0)}, \mathsf{CW}^{(1)}, \mathsf{CW}^{(2)}, \cdots, \mathsf{CW}^{(n+1)})$.

return (k_0, k_1).

end procedure

and meanwhile the correction C_{sign} should force the desired correlation of sign strings in T_0 and T_1.

$\mathsf{Gen}(1^\lambda, \hat{f}_{A,B})$ generates the keys k_0 and k_1, containing $\mathsf{seed}\|\mathsf{sign}$ string at root and correction words for each layer, which determine T_0 and T_1 respectively. At the ith layer, $\mathsf{Gen}(1^\lambda, \hat{f}_{A,B})$ first records in the list $V^{(i-1)}$ all the strings in the ith layer that need to be corrected, which are the seed strings of the nodes exiting an accepting path, and the sign strings of the nodes whose parent is on an accepting path. Then it runs the method $\mathsf{GenCW}(i, A, V^{(i-1)})$ to generate $\mathsf{CW}^{(i)}$ for both parties (line 19), such that at a node v on the ith layer, the method $\mathsf{Correct}(v, \text{sign of } v\text{'s parent}, \mathsf{CW}^{(i)})$ outputs the desired corrections C_{seed} and C_{sign} for v.

After receiving the key k_b, party b can evaluate the input $x = x_1 \dots x_n$ by calling $\mathsf{Eval}_b(1^\lambda, k_b, x)$. It first parses the key k_b to the $\mathsf{seed}\|\mathsf{sign}$ string at the root and the correction words $\{\mathsf{CW}^{(i)}\}_{i \in [n]}$ for each layer, then computes the $\mathsf{seed}\|\mathsf{sign}$ strings along the path represented by x in T_b layer by layer. On the ith layer, given the $\mathsf{seed}_{i-1}\|\mathsf{sign}_{i-1}$ string for the node $x_1 \dots x_{i-1}$, the Eval method computes $G(\mathsf{seed}_{i-1})$ and the correction $\mathsf{Correct}(x_1 \dots x_{i-1}, \mathsf{sign}_{i-1}, \mathsf{CW}^{(i)})$ to obtain $\mathsf{seed}_i\|\mathsf{sign}_i$

Figure 1 cont'd: The template of our DMPF schemes, continued.

1: **procedure** $\text{EVAL}_b(1^\lambda, k_b, x)$
2: Parse $k_b = ([\mathsf{seed}], [\mathsf{sign}], \mathsf{CW}^{(1)}, \mathsf{CW}^{(2)}, \cdots, \mathsf{CW}^{(n+1)})$.
3: Denote $x = x_1 x_2 \cdots x_n$.
4: **for** $i = 1$ to n **do**
5: $C_{\mathsf{seed}} \| C_{\mathsf{sign}^0} \| C_{\mathsf{sign}^1} \leftarrow \mathsf{Correct}(x_1 \cdots x_{i-1}, \mathsf{sign}, \mathsf{CW}^{(i)})$, where $|C_{\mathsf{seed}}| = \lambda$ and $|C_{\mathsf{sign}^0}| = |C_{\mathsf{sign}^1}| = l$.
6: $\mathsf{seed} \| \mathsf{sign} \leftarrow G_{x_i}(\mathsf{seed}) \oplus (C_{\mathsf{seed}} \| C_{\mathsf{sign}^{x_i}})$, where $|\mathsf{seed}| = \lambda$ and $|\mathsf{sign}| = l$.
7: **end for**
8: **return** $(-1)^b \cdot \big(G_{\mathsf{conv}}(\mathsf{seed}) + \mathsf{ConvCorrect}(x, \mathsf{sign}, \mathsf{CW}^{(n+1)})\big)$.
9: **end procedure**

10: **procedure** $\text{FULLEVAL}_b(1^\lambda, k_b)$
11: Parse $k_b = (\mathsf{seed}^{(0)}, \mathsf{sign}^{(0)}, \mathsf{CW}^{(1)}, \mathsf{CW}^{(2)}, \cdots, \mathsf{CW}^{(n+1)})$.
12: For $1 \le i \le n$, $\mathsf{Path}^{(i)} \leftarrow$ the lexicographical ordered list of $\{0,1\}^i$. $\mathsf{Path}^{(0)} \leftarrow [\epsilon]$.
13: **for** $i = 1$ to n **do**
14: **for** $k = 1$ to 2^{i-1} **do**
15: $C_{\mathsf{seed}} \| C_{\mathsf{sign}^0} \| C_{\mathsf{sign}^1} \leftarrow \mathsf{Correct}(\mathsf{Path}^{(i-1)}[k], \mathsf{sign}^{(i-1)}[k], \mathsf{CW}^{(i)})$, where $|C_{\mathsf{seed}}| = \lambda$ and $|C_{\mathsf{sign}^0}| = |C_{\mathsf{sign}^1}| = l$.
16: $\mathsf{seed}^{(i)}[2k] \| \mathsf{sign}^{(i)}[2k] \leftarrow G_0(\mathsf{seed}^{(i-1)}[k]) \oplus (C_{\mathsf{seed}} \| C_{\mathsf{sign}^0})$, where $|\mathsf{seed}^{(i)}[2k]| = \lambda$ and $|\mathsf{sign}^{(i)}[2k]| = l$.
17: $\mathsf{seed}^{(i)}[2k+1] \| \mathsf{sign}^{(i)}[2k+1] \leftarrow G_1(\mathsf{seed}^{(i-1)}[k]) \oplus (C_{\mathsf{seed}} \| C_{\mathsf{sign}^1})$, where $|\mathsf{seed}^{(i)}[2k+1]| = \lambda$ and $|\mathsf{sign}^{(i)}[2k+1]| = l$.
18: **end for**
19: **end for**
20: **for** $k = 1$ to 2^n **do**
21: $\mathsf{Output}[k] \leftarrow (-1)^b \cdot \big(G_{\mathsf{conv}}(\mathsf{seed}^{(n)}[k]) + \mathsf{ConvCorrect}(\mathsf{Path}[k], \mathsf{sign}^{(n)}[k], \mathsf{CW}^{(n+1)})\big)$.
22: **end for**
23: **return** Output.
24: **end procedure**

(see line 21).

Converting strings to $\mathbb{G}$ elements. After the nth layer of the evaluation tree, the template adds a convert layer associated with $\mathsf{CW}^{(n+1)}$ to convert the strings at a leaf node to an element in the output group $\mathbb{G}$ of $f_{A,B}$. At a leaf node x with string $\mathsf{seed}\|\mathsf{sign}$, the final output is computed by invoking $G_{\mathsf{conv}}(\mathsf{seed})$ to get a pseudorandom $\mathbb{G}$-element and invoking $\mathsf{ConvCorrect}(x, \mathsf{sign}, \mathsf{CW}^{(n+1)})$ to get a correction on $G_{\mathsf{conv}}(\mathsf{seed})$ (see line 8). $\mathsf{CW}^{(n+1)}$ generated in Gen (see line 27) will force the following correlations of outputs in T_0 and T_1 , similar to the correlations in the first n layers as in Definition 5: If the leaf node is not on any accepting path, then the outputs in T_0 and T_1 at this node should add up to $0_{\mathbb{G}}$. On the other hand, if the leaf node is on any accepting path, then the corrected outputs in T_0 and T_1 should add up to the corresponding element in B.

In a nutshell, we provide the key generation Gen, single-input evaluation Eval and full-domain evaluation FullEval in the template in Figure 1. The methods will be realized by big-state DMPF and OKVS-based DMPF differently in the next sections.

Remark 6 (Early termination optimization)**.** *When the output group $\mathbb{G}$ of $f_{A,B}$ has small size, we can apply the early termination optimization to reduce the $\log(\lambda/\log|\mathbb{G}|)$ layers of the evaluation tree, as proposed in [2] for optimizing DPF in the same situation.*

3.2. Big-State DMPF

In this section we present our first instantiation of DMPF in Figure 2 referred to as the big-state DMPF, based on the template of DMPF in Figure 1.

High-level overview. In the big-state DMPF we set the length l of the sign string to be t, the number of accepting inputs indicated in $\hat{f}_{A,B}$. The evaluation trees T_0 and T_1 satisfy correlations in Definition 5, in a way that the sign string at a node stores a share of the unit vector indicating which accepting path this node is on: for a node lying on the kth accepting path in the depth-i layer, its sign strings in T_0 and T_1 should add up (by bit-wise XOR) to $e_k = 0^{k-1}10^{t-k}$. The desired correlations are ensured by the design of GenCW and Correct in Figure 2. On the ith layer, GenCW generates a list $\mathsf{CW}^{(i)}$ of $(\lambda+2t)$-bit strings, where $\mathsf{CW}^{(i)}[k]$ is the intended correction for children of the node on the kth accepting path on the $(i-1)$th layer. Correct derives the intended correction using the sign string through a straightforward inner-product.

For the convert layer, GenConvCW set $\mathsf{CW}^{(n+1)}[k]$ to be the correction that makes the kth accepting leaf's outputs in T_0 and T_1 to add up to $B[k]$, which can be extracted by ConvCorrect using the sign string and again a straightforward inner-product.

Figure 2: The parameter l and methods' setting that turns the template of DMPF in Figure 1 into the big-state DMPF.

Set $l \leftarrow t$, the upperbound of $|A|$.
procedure INITIALIZE($\{\mathsf{seed}_b^{(0)}, \mathsf{sign}_b^{(0)}\}_{b=0,1}$)
  For $b = 0, 1$, let $\mathsf{seed}_b^{(0)} = [r_b]$ where $r_b \xleftarrow{\$} \{0,1\}^\lambda$.
  For $b = 0, 1$, set $\mathsf{sign}_b^{(0)} = [b\|0^{t-1}]$.
end procedure

procedure GENCW($i, A, V^{(i-1)}$)
  Let $\{A^{(i)}\}_{0\le i\le n}$ be defined as in Figure 1.
  Sample a list CW of t random strings from $\{0,1\}^{\lambda+2t}$.
  for $k = 1$ to $|A^{(i-1)}|$ **do**
    Parse $V^{(i-1)}[k] = \Delta\mathsf{seed}\|\Delta\mathsf{sign}^0\|\Delta\mathsf{sign}^1$.
    if $A^{(i-1)}[k]\|z \in A^{(i)}$ holds for both $z = 0, 1$ **then**
      $d \leftarrow$ the index of $A^{(i-1)}[k]\|0$ in $A^{(i)}$.
      $\mathsf{CW}[k] \leftarrow \Delta\mathsf{seed}\|(\Delta\mathsf{sign}^0 \oplus e_d)\|(\Delta\mathsf{sign}^1 \oplus e_{d+1})$ where $e_d = 0^{d-1}10^{t-d}$.
    else
      Suppose $A^{(i-1)}[k]\|z \in A^{(i)}$. Let d be the index of $A^{(i-1)}[k]\|z$ in $A^{(i)}$.
      $\Delta\mathsf{sign}^z \leftarrow \Delta\mathsf{sign}^z \oplus e_d$.
      $\mathsf{CW}[k] \leftarrow \Delta\mathsf{seed}\|\Delta\mathsf{sign}^0\|\Delta\mathsf{sign}^1$.
    end if
  end for
  return CW.
end procedure

procedure GENCONVCW($A, B, \Delta g, \mathsf{sign}_0^{(n)}, \mathsf{sign}_1^{(n)}$)
  Sample a list CW of t random $\mathbb{G}$-elements.
  for $k = 1$ to $|A|$ **do**
    $\mathsf{CW}[k] \leftarrow (-1)^{\mathsf{sign}_0^{(n)}[k][k]}(\Delta g[k] - B[k])$.
  end for
  return CW.
end procedure

procedure CORRECT($\bar{x}$, sign, CW)
  return $C_{\mathsf{seed}}\|C_{\mathsf{sign}^0}\|C_{\mathsf{sign}^1} \leftarrow \sum_{i=1}^{t} \mathsf{sign}[i] \cdot \mathsf{CW}[i]$, where C_{sign^0} and C_{sign^1} are t-bit.
end procedure

procedure CONVCORRECT(x, sign, CW)
  return $\sum_{i=1}^{t} \mathsf{sign}[i] \cdot \mathsf{CW}[i]$.
end procedure

Remark 7. *Note that when $t = 1$, the big-state DMPF scheme is exactly the DPF scheme in [2].*

We informally argue that the correctness of the big-state DMPF holds since the desired correlations in Definition 5 of T_0 and T_1 are ensured, which in turn gives the correct shares of outputs in the end of evaluation. The security holds since in the view of party b, for each $i \in [n]$, on the ith layer, each entry of $\mathsf{CW}^{(i)}$ is either a truly random string, or in the form of $(x\|y\|w) \oplus \left(G_{1-z}^{\mathsf{seed}}(\mathsf{seed}_{\bar{b}})\|G_0^{\mathsf{sign}}(\mathsf{seed}_{\bar{b}})\|G_1^{\mathsf{sign}}(\mathsf{seed}_{\bar{b}})\right)$, for $\mathsf{seed}_{\bar{b}}$ a pseudorandom seed string held by party $\bar{b}$ and values of z, x, y, w independent of $\mathsf{seed}_{\bar{b}}$. In consequence, $\mathsf{CW}^{(i)}$ in party b's view can be simulated by a truly random string, assuming the security of the underlying PRG. A similar argument holds for simulating $\mathsf{CW}^{(n+1)}$ in the convert layer. We defer the formal security proof to our full version.

Theorem 8 (Big-state DMPF is correct and secure)**.** *There is a polynomial p such that the following holds. Assume G and G_{conv} are (T, ϵ_G)-secure and $(T, \epsilon_{G_{\mathsf{conv}}})$-secure respectively. Then the big-state DMPF scheme in Figure 2 is $(T - p(\lambda), \epsilon)$-secure where $\epsilon = tn\epsilon_G + t\epsilon_{G_{\mathsf{conv}}}$.*

3.3. OKVS-based DMPF

Next we display our second instantiation of DMPF in Figure 3 referred to as the OKVS-based DMPF, based on the template of DMPF in Figure 1. We call this instantiation the OKVS-based DMPF, since we utilize the primitive OKVS (see Section 2.4 for an introduction).

High-level overview. In the OKVS-based DMPF, we set the length l of the sign string to be 1. The evaluation trees T_0 and T_1 satisfy correlations in Definition 5, in a way that the sign string at a node stores a share of 1 if this node is on an accepting path, a share of and 0 if this node is not on any accepting path. In order to ensure desired correlations, at a parent node on an accepting path, the corrections $C_{\mathsf{seed}}, C_{\mathsf{sign}^0}$ and C_{sign^1} for the strings at its children are determined by the following if-else sentences: if one of its children exits the accepting path, then the seed correction C_{seed} should zero out this child's seed strings in T_0 and T_1. Otherwise C_{seed} will be a random correction. The sign corrections C_{sign^0} and C_{sign^1} will force the sign strings at each child to be a share of 0 if this child exits the accepting path, or to be a share of 1 if it remains on an accepting path.

Remark 9. *Note that when $t = 1$, the evaluation trees T_0 and T_1 are identical as the trees in [2]. However the* CW *strings are generated differently, as is demonstrated below.*

In Figure 3, GenCW generates $\{\mathsf{CW}^{(i)}\}$ which can derive the intended corrections by running Correct. They utilize the OKVS primitive that can encode key-value pairs to a data structure, which can be later decoded with any stored key to its corresponding value. On the $(i-1)$th layer, we define the key space to be the set of all nodes on this layer and the value space to be $\{0,1\}^{\lambda+2}$. Each node on this layer that is also on an accepting path needs a $(\lambda+2)$-bit correction for its children, recorded by the value list $V^{(i-1)}$. GenCW sets $\mathsf{CW}^{(i)}$ to be the encoding of these (node, correction) pairs (there are up to t such pairs) using OKVS_i. Given $\mathsf{CW}^{(i)}$ and any parent node on the $(i-1)$th layer, Correct obtains the intended corrections by decoding $\mathsf{CW}^{(i)}$ using OKVS_i

For the convert layer, GenConvCW set $\mathsf{CW}^{(n+1)}$ to be the encoding of (leaf node, output correction) pairs where

Figure 3: The parameter l and methods' setting that turns the template of DMPF in Figure 1 into the OKVS-based DMPF.

Set $l \leftarrow 1$.
For $1 \leq i \leq n$, let OKVS_i be an OKVS scheme (Definition 4) with key space $\mathcal{K} = \{0,1\}^{i-1}$, value space $\mathcal{V} = \{0,1\}^{\lambda+2}$ and input length t.
let $\mathsf{OKVS}_{\mathsf{conv}}$ be an OKVS scheme with key space $\mathcal{K} = \{0,1\}^n$, value space $\mathcal{V} = \mathbb{G}$ and input length $\min\{2^{i-1}, t\}$.

procedure INITIALIZE($\{\mathsf{seed}_b^{(0)}, \mathsf{sign}_b^{(0)}\}_{b=0,1}$)
    For $b = 0, 1$, let $\mathsf{seed}_b^{(0)} = [r_b \xleftarrow{\$} \{0,1\}^\lambda]$ and $\mathsf{sign}_b^{(0)} = [b]$.
end procedure

procedure GENCW(i, A, V)
    Let $\{A^{(i)}\}_{0 \leq i \leq n}$ be defined as in Figure 1.
    for $k = 1$ to $|A^{(i-1)}|$ **do**
        Parse $V[k] = \Delta\mathsf{seed} \| \Delta\mathsf{sign}^0 \| \Delta\mathsf{sign}^1$.
        for $z = 0, 1$ such that $A^{(i-1)}[k] \| z \in A^{(i)}$ **do**
            $\mathsf{sign}^z \leftarrow \mathsf{sign}^z \oplus 1$.
        end for
        Update $V[k] \leftarrow \Delta\mathsf{seed} \| \Delta\mathsf{sign}^0 \| \Delta\mathsf{sign}^1$.
    end for
    Copy the list $A^{(i-1)}$ to the list K.
    for $j = |K| + 1$ to $\min\{2^{i-1}, t\}$ **do**
        Set $K[j]$ to be an arbitrary string in $\{0,1\}^{i-1}$ that is different from $K[1 \ldots j-1]$.
        Set $V[j]$ to be a random string in $\{0,1\}^{\lambda+2}$.
    end for
    return $\mathsf{OKVS}_i.\mathsf{Encode}(\{(K[j], V[j])\}_{1 \leq j \leq |K|})$.
end procedure

procedure GENCONVCW($A, B, \Delta g, \mathsf{sign}_0^{(n)}, \mathsf{sign}_1^{(n)}$)
    Sample a list V of t random $\mathbb{G}$-elements.
    for $k = 1$ to $|A|$ **do**
        $V[k] \leftarrow (-1)^{\mathsf{sign}_0^{(n)}[k][k]}(\Delta g[k] - B[k])$.
    end for
    return $\mathsf{OKVS}_{\mathsf{conv}}(\{(A[k], V[k])\}_{1 \leq k \leq t})$.
end procedure

procedure CORRECT($\bar{x}, \mathsf{sign}, \mathsf{CW}$)
    return $C_{\mathsf{seed}} \| C_{\mathsf{sign}^0} \| C_{\mathsf{sign}^1} \leftarrow \mathsf{sign} \cdot \mathsf{OKVS}_i.\mathsf{Decode}(\mathsf{CW}, \bar{x})$, where C_{sign^0} and C_{sign^1} are bits.
end procedure

procedure CONVCORRECT($x, \mathsf{sign}, \mathsf{CW}$)
    return $\mathsf{sign} \cdot \mathsf{OKVS}_{\mathsf{conv}}.\mathsf{Decode}(\mathsf{CW}, x)$.
end procedure

each output correction associated with a leaf node makes the leaf's outputs in T_0 and T_1 add up to the corresponding element in B.

Note that in Figure 3 the OKVS scheme OKVS_i we use for the ith layer has key space of size 2^i and value space $\{0,1\}^{\lambda+2}$. For uniformity we may extend the key space of OKVS_i to size 2^n, and realize $\{\mathsf{OKVS}_i\}_{i \in [n]}$ using the same OKVS scheme. For the upmost few layers where $2^i < t$, OKVS_i may be realized by the most naïve way of encoding to a random truth table.

We informally argue that armed with an OKVS scheme that fails with negligible probability, the correctness of the OKVS-based DMPF holds with overwhelming probability since the desired correlations of T_0 and T_1 in Definition 5 are ensured, which in turn gives correct shares of outputs in the end of evaluation. The security holds since in the view of party b, for each $i \in [n]$, on the ith layer, $\mathsf{CW}^{(i)}$ is the OKVS_i encoding of the following value list $V^{(i-1)}$: each entry of $V^{(i-1)}$ is either a truly random string, or in the form of $(x \| y \| w) \oplus \left(G_{1-z}^{\mathsf{seed}}(\mathsf{seed}_{\bar{b}}) \| G_0^{\mathsf{sign}}(\mathsf{seed}_{\bar{b}}) \| G_1^{\mathsf{sign}}(\mathsf{seed}_{\bar{b}})\right)$, for $\mathsf{seed}_{\bar{b}}$ a pseudorandom seed string held by party $\bar{b}$ and values of z, x, y, w independent of $\mathsf{seed}_{\bar{b}}$. In consequence, $V^{(i-1)}$ in party b's view can be simulated by a truly random string, assuming the security of the underlying PRG. And by the obliviousness of OKVS_i, $\mathsf{CW}^{(i-1)}$ can be simulated without knowing A and $V^{(i-1)}$. A similar argument holds for simulating $\mathsf{CW}^{(n+1)}$ in the convert layer. We defer the formal security proof to our full version.

Theorem 10 (OKVS-based DMPF is correct and secure)**.** *There is a polynomial p such that the following holds. Assume G and G_{conv} are (T, ϵ_G)-secure and $(T, \epsilon_{G_{\mathsf{conv}}})$-secure respectively. Assume in the OKVS-based DMPF scheme in Figure 3, for $1 \leq i \leq n$ the OKVS_i scheme is $(T, \epsilon_{\mathsf{OKVS}_i})$-oblivious, and the $\mathsf{OKVS}_{\mathsf{conv}}$ scheme is $(T, \epsilon_{\mathsf{OKVS}_{\mathsf{conv}}})$-oblivious. Then the OKVS-based DMPF scheme in Figure 2 is $(T - p(\lambda), \epsilon)$-secure where $\epsilon = tn\epsilon_G + t\epsilon_{G_{\mathsf{conv}}} + \sum_{i=1}^{n} \epsilon_{\mathsf{OKVS}_i} + \epsilon_{\mathsf{OKVS}_{\mathsf{conv}}}$.*

3.4. Comparison

We summarize the efficiency of the DMPF instantiations we've mentioned and constructed so far in Table 1. We provide the keysize and running time of Gen,Eval and FullEval for different DMPF constructions for domain size N, t accepting points, output group $\mathbb{G}$, statistical security parameter λ_{stat}, and computational security parameter λ. In Table 1, T_G is the time for computing $G : \{0,1\}^{\lambda+1} \to \{0,1\}^{2\lambda+2}$, and $T_{G_{\mathsf{conv}}}$ is the time for computing $G_{\mathsf{conv}} : \{0,1\}^\lambda \to \mathbb{G}$. We leave this table with the abstraction of PBC in the second column and the abstraction of OKVS in the last column. In the second column, PBC is the PBC scheme, where m stands for the number of codewords, w stands for the number of codewords that each input coordinate is copied to, and the (sub-)processes Encode, Position, Schedule are defined in Definition 2. In the last column, OKVS is the OKVS scheme used for the first n layers, and $\mathsf{OKVS}_{\mathsf{conv}}$ is the OKVS scheme used for the convert layer.

TABLE 1: Keysize and running time comparison for different DMPF constructions.

	Naïve DMPF	PBC-based DMPF	Big-state DMPF	OKVS-based DMPF
Keysize	$t(\lambda+2)\log N + t\log\mathbb{G}$	$m(\lambda+2)\log(wN/m) + m\log\mathbb{G}$	$t(\lambda+2t)\log N + t\log\mathbb{G}$	$\log N \times \mathsf{OKVS.Codesize}$ $+\mathsf{OKVS}_{\mathsf{conv}}.\mathsf{Codesize}$
Gen	$2t\log N \times T_G + 2t \times T_{G_{\mathsf{conv}}}$	$2m\log(wN/m) \times T_G + 2m \times T_{G_{\mathsf{conv}}}$ $+\mathsf{PBC.Encode} + \mathsf{PBC.Schedule}$	$2t\log N \times T_{G^*}$ [1] $+t^2\log N \times (\lambda+t)$-bit-XOR	$2t\log N \times T_G + 2t \times T_{G_{\mathsf{conv}}}$ $+\log N \times \mathsf{OKVS.Encode}$ $+t\log N \times \mathsf{OKVS.Decode}$ $+\mathsf{OKVS}_{\mathsf{conv}}.\mathsf{Encode}$ $+\mathsf{OKVS}_{\mathsf{conv}}.\mathsf{Decode}$
Eval	$t\log N \times T_G + t \times T_{G_{\mathsf{conv}}}$	$w\log(wN/m) \times T_G + w \times T_{G_{\mathsf{conv}}}$ $+\mathsf{PBC.Position}$	$\log N \times T_{G^*} + T_{G_{\mathsf{conv}}}$ $+t\log N \times (\lambda+t)$-bit-XOR	$\log N \times T_G$ $+\log N \times \mathsf{OKVS.Decode}$ $+\mathsf{OKVS}_{\mathsf{conv}}.\mathsf{Decode}$
FullEval	$tN \times T_G + tN \times T_{G_{\mathsf{conv}}}$	$wN \times T_G + wN \times T_{G_{\mathsf{conv}}}$ $+N \times \mathsf{PBC.Position}$	$N \times T_{G^*} + N \times T_{G_{\mathsf{conv}}}$ $+2tN \times (\lambda+t)$-bit-XOR	$N \times T_G + N \times T_{G_{\mathsf{conv}}}$ $+N \times \mathsf{OKVS.Decode}$ $+N \times \mathsf{OKVS}_{\mathsf{conv}}.\mathsf{Decode}$

[1] The PRG used in big-state DMPF maps from $\{0,1\}^\lambda$ to $\{0,1\}^{2\lambda+2t}$ whose computation time grows with t. We mark this PRG as G^* and its computation time as T_{G^*}.

We expect the PBC-based, big-state and OKVS-based DMPF to have slightly larger keysize than the naïve DMPF, but much faster evaluation time (for both Eval and FullEval). The evaluation time of the naïve construction is dominated by the total PRG invocation time, which scales linearly with t. In contrast, the evaluation times for the PBC-based DMPF, big-state DMPF, and OKVS-based DMPF involve PRG invocations that are $\frac{w}{t}$, $\frac{1}{t}$, and $\frac{1}{t}$ times the number of PRG invocations required by the naïve construction, respectively. However, these DMPF constructions come with varying additional overheads as compensation:

- The PBC-based DMPF requires an invocation of PBC.Position (a sub-procedure of PBC defined in Definition 2) in Eval and N invocations of PBC.Position in FullEval.
- For the big-state DMPF, the PRG's output length is dependent to t and so is the total PRG invocation time. Implementing PRG with AES, the overall PRG invocation time in big-state DMPF is roughly $\frac{\lambda+t}{t\lambda+t}\times$ the overall PRG invocation time of naïve DMPF. In addition, big-state DMPF has extra cost of long string XOR's scaling with t^2. Therefore we expect big-state DMPF to perform well when t is small, and become much less competitive when t is large.
- For the OKVS-based DMPF, in addition to having the shortest PRG invocation time among all DMPF constructions, it requires $\log N$ invocations of OKVS.Decode in Eval and N invocations of OKVS.Decode in FullEval.

Generally speaking, we expect different DMPF schemes to be the top choice for different regimes of N and t. Their concrete performance will be discussed later in Section 4.

3.5. Distributed Key Generation

The key generation of a DMPF scheme requires the existence of a trusted third party. To drop this assumption, we consider distributed key generation which turns out to benefit many applications such as PCG. In a brute-force way, the distributed key generation of DMPF can be done by a generic secure 2-party computation (2PC). We propose more efficient constructions for distributed key generation for any DMPF scheme following the template in Figure 1 with feasible domain size to save more online communication cost, by extending the key generation protocol for (weight-1) DPF by Doerner and shelat [8] that pushes all PRG evaluations to offline and local computation.

In Gen of Figure 1, for each $i \in [n]$, $\mathsf{CW}^{(i)}$ can be viewed as computed by the following steps:

(1) Compute the list $\Delta^{(i-1)} := G(\mathsf{seed}_0^{(i-1)}) \oplus G(\mathsf{seed}_1^{(i-1)})$, a length-$2^{i-1}$ list of $(2\lambda+2l)$-bit strings.
(2) Extract the list of nonzero entries $\Delta^{(i-1)}|_{A^{(i-1)}} := \{\Delta^{(i-1)}[x] : x \in A^{(i-1)}\}$ on the accepting paths. [3]
(3) Convert $\Delta^{(i-1)}|_{A^{(i-1)}}$ to the value list $V^{(i-1)}$.
(4) Compute $\mathsf{CW}^{(i)} \leftarrow \mathsf{GenCW}(i, A, V^{(i-1)})$.
(5) Use $\mathsf{CW}^{(i)}$ and $\mathsf{sign}_b^{(i-1)}$ to correct $G(\mathsf{seed}_b^{(i-1)})$ and obtain $\mathsf{seed}_b^{(i)}, \mathsf{sign}_b^{(i)}$, for $b = 0, 1$.

The distributed key generation protocol pushes step (1) that contains all PRG evaluations to local computation, where party b computes list $\Delta_b^{(i-1)} := G(\mathsf{seed}_b^{(i-1)})$, a share of $\Delta^{(i-1)}$. Then the protocol computes step (2), (3), and (4) with online communication, after which the parties switch to another offline phase to locally compute $\mathsf{seed}_b^{(i)}, \mathsf{sign}_b^{(i)}$ for the next layer.

We introduce two approaches to achieve online computation of step (2) with sublinear communication in the size of $|\Delta^{(i-1)}|$, while step (1) and (5) are clear from the algorithm and step (3) and (4) can be achieved by generic secure 2PC for relatively simple circuits. We provide the first approach requiring secure arithmetic operations in our full version due to space limit, and focus on a concrete and more practical approach only for big-state DMPF.

A distributed key generation protocol for big-state DMPF scheme. For a better illustration, let's suppose the t-point function is f_A such that $f_A(x) = 1$ if and only if $x \in A$, and neglect the convert layer. We present the

3. Note that by the construction of T_0 and T_1, the entries corresponding to nodes not on any accepting path are zero strings.

Figure 4: The distributed key generation protocol $\Pi_{\mathsf{big\text{-}state\text{-}DKG}}$ and its ideal functionality $\mathcal{F}_{\mathsf{big\text{-}state\text{-}DKG}}$ for big-state DMPF in Figure 2. Two parities $b \in \{0,1\}$ generate the key for the t-point function $f_A : \{0,1\}^n \to \{0,1\}$ such that $f_A(x) = 1$ if and only if $x \in A$. Other notations are as in Figure 1. Party b has input $[\![A]\!]_b$ which is an additive share of A. The functionality interacts with a party b and an adversary Adv.

$\mathcal{F}_{\mathsf{big\text{-}state\text{-}DKG}}(1^\lambda)$:
- Get $[\![A]\!]_b$ from party b and $[\![A]\!]_{\bar{b}}$ from Adv, and reconstruct $A = [\![A]\!]_0 + [\![A]\!]_1$.
- Invoke $(k_0, k_1) \leftarrow \mathsf{Gen}(1^\lambda, f_A)$.
- Output k_b to party b and $k_{\bar{b}}$ to Adv.

$\Pi_{\mathsf{big\text{-}state\text{-}DKG}}(1^\lambda, [\![A]\!]_b, n')$ for party $b \in \{0,1\}$:
- Create empty lists $\mathsf{seed}_b^{(i)}$, $\mathsf{sign}_b^{(i)}$ and $V^{(i)}$ for $0 \le i \le n-1$. Initialize $\mathsf{seed}_b^{(0)} = [r_b]$ where $r_b \xleftarrow{\$}$ and $\mathsf{sign}_b^{(0)} = [b \| 0^{t-1}]$.
- **for** $i = 1$ to n **do**
  - Compute the list $\Delta_b^{(i-1)} := G(\mathsf{seed}_b^{(i-1)})$.
  - Initialize a length-t list L_b of $(2\lambda + 2t)$-bit strings.
  - **for** $r = 1$ to t and $l = 1$ to $2\lambda + 2t$ **do**
    - **if** $i \le n - n'$ **then**
      - Invoke secure 2PC with party $\bar{b}$ to compute
$$L_b[r][l] \leftarrow \left[\!\!\left[(\mathsf{sign}^{(i-1)}[k][r])_{k\in\{0,1\}^{i-1}} \cdot (\Delta^{(i-1)}[k][l])_{k\in\{0,1\}^{i-1}} \right]\!\!\right]_b$$
    - **else**
      - Invoke $\Pi_{\mathsf{SharedIPD}}$ with party $\bar{b}$ to get
$$L_b[r][l] \leftarrow \Pi_{\mathsf{SharedIPD}} \left((\mathsf{sign}_b^{(i-1)}[k][r])_{k\in\{0,1\}^{i-1}}, (\Delta_b^{(i-1)}[k][l])_{k\in\{0,1\}^{i-1}} \right).$$
    - **end if**
  - **end for**
  - Invoke Π_{CW} with party $\bar{b}$ to get $\mathsf{CW}^{(i)} \leftarrow \Pi_{\mathsf{CW}}(i, [\![A]\!]_b, L_b)$.
  - Compute $C_{\mathsf{seed},b} \| C_{\mathsf{sign}^0,b} \| C_{\mathsf{sign}^1,b} \leftarrow \mathsf{Correct}(k, \mathsf{sign}_b^{(i-1)}[k], \mathsf{CW}^{(i)})$ for $k \in \{0,1\}^{i-1}$.
  - For $z = 0, 1$, store the first λ bits of $G_z(\mathsf{seed}_b^{(i-1)}[k]) \oplus (C_{\mathsf{seed},b} \| C_{\mathsf{sign}^z,b})$ in entry $\mathsf{seed}_b^{(i)}[k\|z]$ and the remaining l bits in entry $\mathsf{sign}_b^{(i)}[k\|z]$.
- **end for**
- Output k_b.

distributed key generation protocol for big-state DMPF (Figure 4) in this section along with two sub-protocols $\Pi_{\mathsf{SharedIPD}}$ (Figure 10) and Π_{CW} (Figure 9) in Section A. $\Pi_{\mathsf{big\text{-}state\text{-}DKG}}$ computes the list $\Delta^{(i-1)}|_{A^{(i-1)}}$ by utilizing the fact that for each $1 \le i \le n$ and each $1 \le r \le t$, $\mathsf{sign}_0^{(i-1)}[k][r] \oplus \mathsf{sign}_1^{(i-1)}[k][r] = 1$ if and only if k is the rth entry in $A^{(i-1)}$, and equivalently, if and only if $\Delta^{(i-1)}[k]$ is the rth entry in $\Delta^{(i-1)}|_{A^{(i-1)}}$. With the help of the sub-protocol $\Pi_{\mathsf{SharedIPD}}$ which computes additive shares of the inner-product of a shared unit vector and a shared sparse vector of hamming weight $\le t$, $\Pi_{\mathsf{big\text{-}state\text{-}DKG}}$ is able to extract $\Delta^{(i-1)}|_{A^{(i-1)}}$, after which it invokes the sub-protocol Π_{CW} to compute the value list $V^{(i-1)}$ and the desired $\mathsf{CW}^{(i)}$ by generic secure 2PC.

All of our realizations are secure in the semi-honest setting and in the OT-hybrid model, and we provide the security proofs in our full version, due to page limit.

Theorem 11. *Set the security parameters s and s' in $\Pi_{\mathsf{SharedIPD}}$ such that $\max\{(1-1/\mathrm{e})^s, s2^{-s'}\} \cdot n'(2\lambda+2t) \le \varepsilon / \log_2(\mathrm{e})$, where n' denotes the number of layers where $\Pi_{\mathsf{SharedIPD}}$ is applied. The construction $\Pi_{\mathsf{big\text{-}state\text{-}DKG}}$ realizes the ideal functionality $\mathcal{F}_{\mathsf{big\text{-}state\text{-}DKG}}$ in Figure 4 with perfect security and $1-\varepsilon$ correctness against semi-honest adversaries in the OT-hybrid model.*

We analyze the concrete efficiency of the protocol $\Pi_{\mathsf{big\text{-}state\text{-}DKG}}$ in terms of local operations and online communication secure 2PC for AND gates, under the parameter setting $\lambda = 128$, $n = 27$, $t = 27$ and $\varepsilon = 0.3$ from the application to PCG (see Section 5.1). [4] Setting $n' = 13$, $s = 22$ and $s' = 19$, we get the local and online costs of $\Pi_{\mathsf{big\text{-}state\text{-}DKG}}$ as follows:

- The local operations for each party are dominated by 2^{27} PRG evaluations, each mapping a λ-bit string to a $(2\lambda + 2t)$-bit string.
- The online communication cost is dominated by $t(2\lambda + 2t)$ invocations of generic secure 2PC for inner-product

4. The correctness error ε can be converted to security loss by adding a verification procedure in the end of $\Pi_{\mathsf{big\text{-}state\text{-}DKG}}$, say, via the low-weight sketching in [25]. We're able to consider constant loss of security because this leakage is acceptable when applied to PCG, as argued in [5].

for the first 14 layers, followed by $t(2\lambda+2t)$ invocations of $\Pi_{\mathsf{SharedIPD}}$ for the last 13 layers. In total, a circuit of size $2^{30.8}$ AND gates is computed by secure 2PC, requiring $2^{30.8}$ bits of communication.

The brute-force realization of distributed key generation for big-state DMPF (Figure 2) requires 2^{27} PRG evaluations plus at least $t(2\lambda+2t)N = 2^{40}$ AND gates in the online circuit computed by 2PC. Our optimization completely eliminates online PRG evaluations and reduces communication from over 2^{40} bits to $2^{30.8}$ bits. Additionally, we note that the gap widens rapidly as N increases; for example, when $N = 2^{30}$, the comparison is 2^{43} bits for the brute-force realization versus 2^{31} bits for our realization.

3.6. Distributed Multi-Interval Function

In this section we showed that our DMPF template in Figure 1 can be easily converted to a template for distributed multi-interval function (DMIF), following the same idea of converting DPF to distributed comparison function (DCF) in [20], [2]. Both DMIF and its special-case DCF have various applications including distributed public interval containments, distributed spline functions [26], [27], and constructing pseudorandom correlation functions [28]. [5]

We start with defining the multi-interval function, the collection of all k-interval functions.

Definition 12 (k-Interval Function)**.** *Given a domain size $N = 2^n$ and Abelian group $\mathbb{G}$, a k-interval function $f_{I,B} : [N] \to \mathbb{G}$ for $I = ([a_1, b_1], [a_2, b_2], \ldots, [a_k, b_k])$ such that $1 \le a_1 < b_1 < a_2 < b_2 < \cdots < a_k < b_k \le N$ and $B = (\beta_1, \beta_2, \ldots, \beta_k) \in \mathbb{G}^k$ evaluates to β_i on any input x in the interval $[a_i, b_i]$ for $1 \le i \le k$ and to $0_{\mathbb{G}}$ elsewhere.*

We define a distributed multi-interval function to be an additive and succinct secret sharing scheme for the truth table of a multi-interval function containing procedures $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{FullEval})$, that leaks no information about I and B except for an upperbound on $|I|$.

DMIF can be constructed naïvely by summing up k DCFs, resulting in roughly the same cost as naïve DMPF according to the state-of-the-art DCF in [2]. In addition we outline two approaches to convert DMPF to DMIF without too much overhead. We first present a generic conversion from the FullEval of DMPF to the FullEval of DMIF, for any underlying DMPF scheme. Then we sketch a construction of all algorithms $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{FullEval})$ of DMIF from our new DMPF constructions following the DMPF template (Figure 1). Note that the existing naïve and PBC-based DMPF do not fall into the second category.

A generic conversion from DMPF.FullEval to DMIF.FullEval. Let $f_{A',B'}$ be a $(2k)$-point function where $A' = (a_1, b_1 + 1, \ldots, a_k, b_k + 1)$ and $B' = (\beta_1, -\beta_1, \ldots, \beta_k, -\beta_k)$. Then

$$f_{I,B}(x) = \sum_{i\in[k], a_i \le x} \beta_i - \sum_{j\in[k], b_j < x} \beta_j = \sum_{y \le x} f_{A',B'}(y)$$

Therefore DMIF.FullEval for a k-interval function $f_{I,B}$ can be directly computed DMPF.FullEval for a $(2k)$-point function $f_{A',B'}$, with keys generated by the same DMPF.Gen.

DMIF.Eval by modifying DMPF template. When DMIF.Eval is needed, the previous conversion won't be efficient. We briefly sketch how to modify the DMPF template for $(2k)$-point function in Figure 1 to obtain a DMIF scheme for a k-interval function with an Eval method, following the same idea in [2] that converts DPF to DCF by adding an additional string besides seed and sign at each node to store the shared output of the k-interval function. For simplicity, in the sequel, we assume the domain is $\{0,1\}^n$ and the output group $\mathbb{G} = (\{0,1\}^g, \oplus)$. We may provide the generalization of $\mathbb{G}$ in our full version.

Let's again use the notation $f_{I,B}$ for the k-interval function and $f_{A',B'}$ for the $(2k)$-point function as before, and we make modification on the DMPF scheme for $f_{A',B'}$. In addition to the seed_b and sign_b strings at each node in the evaluation tree T_b, our modification adds an additional g-bit string res_b for the shared output of the k-interval function, satisfying the following correlations in T_0 and T_1:

(1) For each node v not on any accepting path of $f_{A',B'}$, $\mathsf{res}_0 \oplus \mathsf{res}_1 = 0^g$.
(2) For each node v on an accepting path, $\mathsf{res}_0 \oplus \mathsf{res}_1 = \bigoplus_{a_i \text{ is in the subtree of } v} \beta_i \oplus \bigoplus_{(b_i+1) \text{ is in the subtree of }} \beta_i$

Note that by switching to a PRG of output length $(2\lambda+2l+2g)$ and modifying the generation of $V^{(i-1)}$ (lines 14 and 16) and the methods GenCW, Correct in the DMPF template in Figure 1, we'll be able to enforce the desired correlations for res strings.

The Eval algorithm on an input $x = x_1 \ldots x_n$ sums up all the res strings on the nodes that takes a left step to exit the path represented by x, as described in Figure 5.

We argue that the output S_b for party b is a share of $f_{I,B}(x)$. Suppose $J_1 = \{j \in [n] : x_j = 1\}$. Then for any $a \in \{0,1\}^n$, $a \le x$ if and only if there exists $j \in J_1$ such

Figure 5: $\mathsf{DMIF.Eval}_b(1^\lambda, k_b, x_1 \ldots x_n)$.

1: Parse $k_b = (\mathsf{seed}, \mathsf{sign}, \mathsf{CW}^{(1)}, \mathsf{CW}^{(2)}, \cdots, \mathsf{CW}^{(n)})$.
2: Set $S = 0^g$.
3: **for** $i = 1$ to n **do**
4: $\quad C_{\mathsf{seed}} \| C_{\mathsf{sign}^0} \| C_{\mathsf{sign}^1} \| C_{\mathsf{res}^0} \| C_{\mathsf{res}^1} \leftarrow \mathsf{Correct}(x_1 \ldots x_{i-1}, \mathsf{sign}, \mathsf{CW}^{(i)})$.
5: $\quad \mathsf{seed}^0 \| \mathsf{sign}^0 \| \mathsf{res}^0 \| \mathsf{seed}^1 \| \mathsf{sign}^1 \| \mathsf{res}^1 \leftarrow G(\mathsf{seed})$.
6: $\quad \mathsf{seed} \| \mathsf{sign} \leftarrow \mathsf{seed}^{x_i} \oplus C_{\mathsf{seed}} \| \mathsf{sign}^{x_i} \oplus C_{\mathsf{sign}^{x_i}}$.
7: $\quad$ If $x_i = 1$, $S \leftarrow S \oplus (\mathsf{res}^0 \oplus C_{\mathsf{res}^0})$.
8: **end for**
9: **return** S.

5. Although previous literature only applied DCF to corresponding applications, we note that the DCF's usually have overlapping domain, and one can potentially switch from summing many DCF's to a single DMIF.

that $a_1 \dots a_{j-1} = x_1 \dots x_{j-1}$ and $a_j = 0$, and then a is in the subtree of node $v(j) = x_1 \dots x_{j-1} \| 0$. Then,

$$
\begin{aligned}
& S_0 \oplus S_1 \\
&= \bigoplus_{j \in J_1} \left(\bigoplus_{a_i \text{ is in the subtree of } v(j)} \beta_i \oplus \bigoplus_{(b_i+1) \text{ is in the subtree of } v(j)} \beta_i \right) \\
&= \bigoplus_{a_i \le x} \beta_i \oplus \bigoplus_{b_i < x} \beta_i = f_{I,B}(x)
\end{aligned}
$$

In turn, we can obtain a DMIF scheme for a k-interval function from the DMPF template for $(2k)$-point function while preserving the security. Moreover, the time and space costs for the resulting DMIF are similar to the DMPF, up to some overhead relevant to the additional strings.

4. Implementation and Evaluation

In this section we discuss implementation, benchmarks and optimizations of all DMPF constructions presented in this paper and compare them against each other as well as to the naïve DMPF. From the perspective of practical applications, we focus on analyzing and optimizing FullEval and Eval time, while preserving a reasonable keysize and Gen time for each DMPF construction.

We present different parameter ranges that are optimal for each DMPF construction and point out some optimizations we employed to each construction. Later, in Section 5, we discuss implications of our benchmark results within two specific applications.

4.1. Benchmarking DMPF Constructions

We implemented all DMPF constructions using Rust and benchmarked their performance on an AWS `t2.large` instance across a wide range of non-zero number of points and domain sizes. Our code is also publicly available.[6]

We instantiate the PBC by cuckoo hashing with parameters set for failure probability 2^{-40} (see "Failure probability of cuckoo hashing" discussion in Section 2.3), with hash functions defined by pseudorandom permutations to enable efficient "reverse lookup." The choice (and overhead) of PRP is affected greatly by the relevant domain size; further discussion below. For the OKVS, we use the "random bands" RB-OKVS construction of [18]; see further detail in Optimizations discussion below.

Optimizations. Our implementations includes a number of optimizations. Most notable are the following:

- **Early Termination.** Mentioned in [2], this optimization reduces the bottom $\log(\lambda / \log|\mathbb{G}|)$ levels in the DPF tree. In our implementation this reduces the tree by 7 levels, reducing the total work of EvalAll by a factor of $\times 128$.
- **Larger Last-Level-PRG Output.** By employing a length-quadrupling PRG in the last level's PRG we are able to skip the penultimate layer and save 25% of the overall computation (by reduced memory accesses) at the cost of doubling the size of the last correction word. This optimization can be generalized to skip the memory accesses of the k-before-bottom layers by multiplying the size of the last correction word by 2^k using a PRG with an expansion of 2^{k+1}.
- **Preprocessing Big-State and OKVS Outputs.** Zooming in to the Eval and FullEval algorithms of OKVS and big-state DMPF, at each level of the evaluation tree, a sum of a subset of the correction word elements is computed, where the subset is determined by a binary vector generated during the evaluation process. This results in an excessive memory and computation overhead, especially for the big-state DMPF where this summation becomes the bottleneck as the number of non-zero points t increases. Our optimization takes chunks of small size B (say 4 or 8) and pre-computes all subset-sums of each chunk. This reduces computation by a factor of B at the cost of increasing memory consumption by a factor of 2^B. In our benchmarks we use $B = 4$. Notice that this optimization can be done without increasing the keysize since these subset sums can be generated directly from the key.
- **Reducing OKVS Decode Costs.** We use the "random bands" RB-OKVS construction of [18]. The encoding of n keys results in a vector of size $n \cdot (1+\epsilon)$ for some overhead parameter ϵ. The OKVS decode operation is an inner product between a length-w binary vector and a length-w sub-band of the OKVS key. This offers a trade-off between the overhead ϵ and the decoding time (proportional to) w.
  In [18] a concrete analysis is presented of the parameters necessary to achieve different encoding overheads $\epsilon \in \{3\%, 5\%, 7\%, 10\%\}$, where w lies roughly between 200 and 600. With our goal of minimizing evaluation time, we have conducted a separate analysis for $\epsilon = 100\%$ for which $w = 58$ is needed to achieve 40 bits of statistical security for the range of $t \le 2^{18}$ and $w = 49$ is needed for $t \le 2^{10}$. This resulted in an almost $\times 4$ improvement at the cost of doubling the OKVS key size.

Benchmark and results for full-domain evaluation. In Figure 6 we provide a comparison of the FullEval operation of the four DMPF constructions for varying numbers of non-zero points t between 1 and $10,240$ over domain size $N = 2^{27}$ with bit-sized outputs. From the analysis from Table 1, the costs of FullEval of all DMPF constructions should scale linearly with the domain size, and we note that our benchmark results confirm this analysis. In particular, FullEval cost for other values of N can be approximated given the $N = 2^{27}$ values provided, and the relation between different DMPF approaches remains the same.

Basing on our experimental results, we derive the optimal DMPF construction in terms of minimal FullEval time for different ranges of t as follows:

- For $t \le 2$ the naïve DMPF performs best.
- For $3 \le t \le 70$ big-state DMPF dominates.

6. https://github.com/MatanHamilis/dmpf

- For $70 < t \leq 10{,}240$ OKVS-based DMPF dominates.
- By extrapolating our measurements we estimate that starting from $t \approx 21{,}000$ PBC-based DMPF is expected to have the fastest FullEval operation.

We emphasize that while for small values of t big-state DMPF is optimal, for larger values of t it becomes impractical, due to the key size and Gen time scaling quadratically with t.

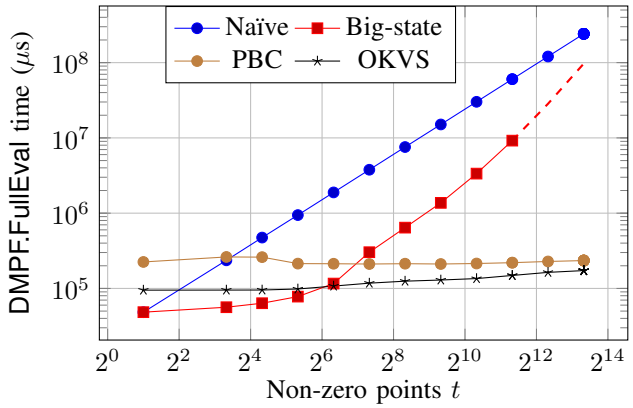

Figure 6: Performance comparison for the FullEval operation of the four DMPF constructions with varying number of non-zero points over domain size $N = 2^{27}$ with $\mathbb{F}_2$ outputs. Big-state DMPF measurements are extrapolated for $t > 2560$ due to excessive memory usage.

Benchmark and results for single evaluation. In Figure 7 we examine the setting of $N = 2^{60}$ and measure the performance of the four DMPF constructions' Eval algorithm for a varying number of non-zero points t.

First, we stress a significant downside of the PBC-based DMPF. The results of the PBC-based DMPF are somewhat overly optimistic, due to the fact that all cuckoo-hashing based constructions require a pseudorandom permutation (PRP) of an appropriate domain size ([12], [13], [14], [5]). This is needed in order to compute the "reverse-lookup" mapping to identify which positios in each bucket are associated with a desired vector index $i \in [n]$. Our implementation realizes this PRP naïvely by expanding a short seed to a pseudorandom permutation, using time linear in the domain size. Such a realization for PRP is practical in FullEval with a feasible domain size (e.g. $N = 2^{27}$) in the previous paragraph, and it takes an insignificant fraction of the overall FullEval time. However the realization for PRP becomes impractical over exponentially-large domains (e.g. $N = 2^{60}$). Moreover, even for a feasible domain size, our realization asks Eval on a single input to expand the whole PRP, which is clearly sub-optimal. As an alternative, one may consider Eval for feasible domain size and Format Preserving Encryption (FPE) [29] as means to instantiate PRPs efficiently, but existing secure and standardized constructions are limited both in their byte-sized granularity and in their input domain size ($> 10^6$). We also mention that to the best of our knowledge, existing state-of-the-art small-domain PRP constructions are either considered impractical ([30]) or require setup-time that is linear in the domain size ([31]). The above showcases the greatest advantage of the OKVS-based DMPF over existing PBC-based DMPF: It can be easily adapted to domains of arbitrarily small and large sizes.

In our benchmark for Eval, the results of the PBC-based construction are made of a sum of the $w = 3$ Eval and PRP invocations using an open source implementation of FPE (for domain size 2^{64}).[7] We conclude from the results that for $t \geq 4$ the performance of OKVS-based DMPF is roughly $\times 4$ better compared to the PBC-based DMPF.

We also report a somewhat unexpected phenomenon in the results: For a fixed domain size N, both the OKVS and the PBC-based DMPF perform *better* when increasing the number of non-zero points t. For the OKVS-based DMPF we explain this by the fact that the first $\log t$ layers of the evaluation tree can be given explicitly to the parties in an uncompressed way while keeping the key size linear in t. For the PBC-based construction we explain this by the fact that as the value of t gets bigger, the size of each codeword becomes smaller, which makes the corresponding underlying DPFs have a lower depth.

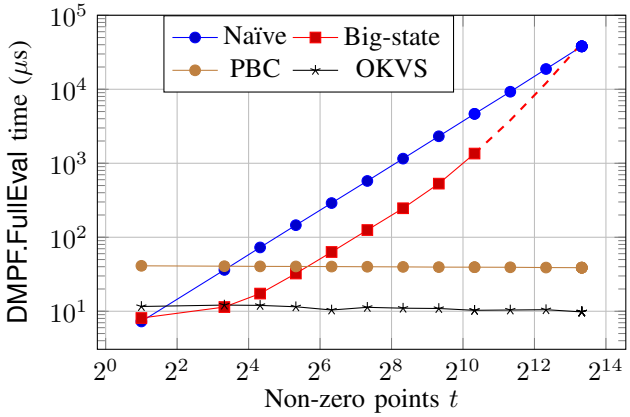

Figure 7: Performance comparison for the Eval operation of the four DMPF constructions as a function of the number of non-zero points, over an infeasible domain size of $N = 2^{60}$ with $\mathbb{F}_2$ outputs. For big-state construction evaluation is extrapolated for $t > 1280$ due to excessive execution times.

Benchmark and results for key generation. In Figure 8 we give measurement results of DMPF.Gen for a large domain size of $N = 2^{60}$. For this large domain size, the PRP for the PBC approach can be implemented by using FPE as well. The performance of the PBC-based approach are lower-bound using $m \times$ DPF.Eval of the appropriate size and $t \times w \times$ PRP.Eval where m is the number of PBC buckets and w is the number of cuckoo-hashing hash functions. Following our expectations, the naive approach is optimal for all parameter ranges, with the PBC approach running $\times 5$ slower. The big-state construction shows poor performance due to the quadratic dependency on t, making it practical

7. https://crates.io/crates/fpe

and useful only for small numbers of non-zero points. The OKVS scales well but incurs significant overhead of about $\times 20$ compares to the naive approach due to the OKVS encodings and decodings.

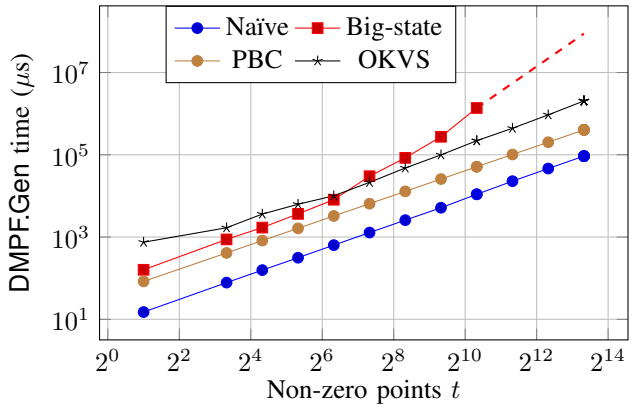

Figure 8: Performance comparison for the Gen operation of the four DMPF constructions as a function of the number of non-zero points, over domain size $N = 2^{60}$ with $\mathbb{F}_2$ outputs. For big-state construction evaluation is extrapolated for $t > 1280$ due to excessive execution times.

5. Applications

In this section, we discuss implications of our benchmark results within two sample application settings: (1) Pseudorandom correlation generators (PCG) for efficiently generating many instances of useful types of correlated randomness, including Oblivious Transfer correlations, and (2) a form of 2-server unbalanced private set intersection (with weighted cardinality) (PSI-WCA).

We remark that these applications have complementary requirements for their underlying DMPF. The PCG application typically demands an efficient DMPF.FullEval for fast seed expansion, whereas the unbalanced PSI-WCA application requires efficient DMPF.Gen and DMPF.Eval, along with a short key size. Consequently, different DMPF constructions exhibit varying performance across these applications, as we will demonstrate in the next section.

For convenience of discussion, we use $\mathsf{DMPF}_{t,N,\mathbb{G}}$ to denote a DMPF scheme for t-point functions with domain $[N]$ and output group $\mathbb{G}$, which can be instantiated by any of the DMPF constructions we mentioned earlier.

5.1. Use Case: Faster PCG for OLE

A pseudorandom correlation generator (PCG) for OLE consists of $(\mathsf{PCG.Gen}, \mathsf{PCG.Expand})$ where PCG.Gen generates a pair of short seeds, and PCG.Expand expands both seeds to a pair of strings satisfying the OLE correlation. Prior works [4], [5] introduce and construct PCGs for the Oblivious Linear Evaluation (OLE) correlation over various fields from Learning Parity with Noise (LPN)-type cryptographic hardness assumptions, using DMPF as a primitive.

FOLEAGE [5] PCG. As one particularly motivated special case, we focus on the construction of [5], which constructs PCG for N random OLE correlations over $\mathbb{F}_4$, which in turn can be locally converted to binary multiplication triples. Their construction is based on the Quasi-Abelian Syndrome Decoding (QA-SD) assumption, parameterized by the total degree N, an LPN-like compression factor c and a weight parameter t such that $c \cdot t \approx O(\lambda)$.

The PCG.Gen algorithm in [5] is comprised of $c^2t \times \mathsf{DMPF}_{t,N/t,\mathbb{F}_4}.\mathsf{Gen}$ executions and some additional operations. The PCG.Expand algorithm contains three major steps:

(1) $\mathsf{DMPF}_{t,N/t,\mathbb{F}_4}.\mathsf{FullEval}$ for all c^2t DMPF keys.
(2) Transpose the results of FullEval and view them as $c(c+1)/2$ polynomials in the multi-variate polynomial ring $\mathcal{R} = \mathbb{F}_4[X_1, \ldots, X_n]/(X_1^3 - 1, \ldots, X_n^3 - 1)$, where $N = 3^n$.
(3) Compute $c(c+1)/2$ FFT's over $\mathcal{R}$ and sum the results.

In [5] each $\mathsf{DMPF}_{t,N/t,\mathbb{F}_4}$ is instantiated using the naïve DMPF, with an additional statement that step (1) of evaluating these naïve DMPFs is the bottleneck of their overall implementation. This choice is due to the simple structure, as well as its efficient distributed key generation procedure. They also mentioned potential optimization by switching from naïve DMPF to PBC-based DMPF for a better asymptotic evaluation cost (but without implementation).

Improved PCG from big-state DMPF. To begin, we modified the code of [5] to collect information regarding the performance of each of the three steps above, and confirmed that about 70% of the running time is spent on step (1).

Under their chosen parameters of $N = 2^{27}$ (adjusted from 3^{17}),[8] $c = 3, t = 27$, our benchmarks show that by directly switching from the naïve DMPF to the big-state DMPF, we achieve a $\times 9.5$ improvement in step (1), which translates to a $\times 2.68$ speedup of the complete PCG.Expand algorithm, at the expense of $\times 1.4$ blowup on the seed size.[9]

As a consequence, by plugging in our implementation to silently generate Beaver triples as in [5], we achieve a throughput of 22.1 million Beaver triples per second on a single core machine, while the implementation of [5] generates 8.2 million Beaver triples per second. This outperforms even the best "non-silent" OT extension protocol [19], which generates 16 million beaver triples per second on the same machine.

5.2. Use Case: Unbalanced PSI-WCA

DMPF is also useful when constructing a non-colluding 2-server-1-client unbalanced private set intersection with weighted cardinality (PSI-WCA) protocol, as proposed in [7], which can be utilized to privacy preserving applications

8. Note that in [5] $N = 3^n$ is used; we adjust to a close (larger) power-of-2 domain size for our constructions.

9. We mention that the FullEval of big-state DMPF is still $\times 1.8$ better than FullEval of the suggested PBC-based DMPF, based on our optimized implementation. This translates to $\sim \times 1.45$ overall throughput while still offering a reasonable secure protocol for distributed key generation.

such as contact tracing [32]. In an unbalanced PSI-WCA protocol, there are two servers each holding the same large set of keywords $X \subseteq [N]$ of size n_X, and a client holding a small set of keywords $Y \subseteq [N]$ of size n_Y. The client learns the weighted cardinality of $X \cap Y$, without the protocol revealing any information about Y to the servers (except an upperbound on n_Y) or any additional information about X to the client.

The work of [7] designs an unbalanced PSI-WCA using DMPF in the following manner:

(1) Client invokes $\mathsf{DMPF}_{n_Y,N,\mathbb{G}}.\mathsf{Gen}(1^\lambda, \hat{f}_{Y,W(Y)})$ to get a pair of keys (k_0, k_1), where $W(Y)$ is the set of weights (from group $\mathbb{G}$) of elements in Y. Then the client sends k_0 to server 0 and k_1 to server 1.
(2) Server b computes $\sum_{x \in X} \mathsf{DMPF}_{n_Y,N,\mathbb{G}}.\mathsf{Eval}_b(1^\lambda, k_b, x)$ and sends the result sum_b to the client.
(3) Client computes $W(X \cap Y) = \mathsf{sum}_0 + \mathsf{sum}_1$.

We plug our new DMPF constructions in the above protocol, to obtain a significantly better server-respond time than the original realization in [7] which instantiated $\mathsf{DMPF}_{n_Y,N,\mathbb{G}}$ by naïve DMPF.

More concretely, we consider a typical setting of $N = 2^{60}$, and let the client's set size n_Y vary. As the server's set size n_X is usually a large number, the performance of this scheme becomes closely attached to the performance of the underlying $\mathsf{DMPF.Eval}$ algorithm. Figure 7 compares the four DMPF constructions' $\mathsf{DMPF.Eval}$ time over the domain size N. We conclude that our OKVS-based construction offers the best performance compared to all other constructions with a $\times 4$ better performance compared to the PBC-based approach, while allowing for arbitrary choices of N that does not depend on limitations imposed by PRPs. On the other hand, if highest priority is in minimizing the (already small) client runtime, PBC-based DMPF is a competitive alternative, as its (centralized) Gen procedure is about $\times 4$ faster than OKVS.

Acknowledgements. Research done in part while the last three authors were visiting NTT Research. E. Boyle was supported by AFOSR Award FA9550-21-1-0046 and ERC Project HSS (852952). M. Hamilis was supported by ERC Project HSS (852952). Y. Ishai was supported by ERC grant NTSC (742754), BSF grant 2022370, ISF grant 2774/20, and ISF-NSFC grant 3127/23. N. Gilboa was supported by ISF grant 2951/20 and MOST grant 8109282. Y. Tu was supported by ERC Project HSS (852952) while at Reichman University.

References

[1] N. Gilboa and Y. Ishai, "Distributed point functions and their applications," in *Advances in Cryptology – EUROCRYPT 2014*, 2014.

[2] E. Boyle, N. Gilboa, and Y. Ishai, "Function secret sharing: Improvements and extensions," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[3] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl, "Efficient pseudorandom correlation generators: Silent ot extension and more," in *Advances in Cryptology – CRYPTO 2019*, 2019.

[4] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl, "Efficient pseudorandom correlation generators from Ring-LPN," in *Advances in Cryptology – CRYPTO 2020*, 2020.

[5] M. Bombar, D. Bui, G. Couteau, A. Couvreur, C. Ducros, and S. Servan-Schreiber, "FOLEAGE: $\mathbb{F}_4$ ole-based multi-party computation for boolean circuits," *Cryptology ePrint Archive*, 2024.

[6] D. Demmler, P. Rindal, M. Rosulek, and N. Trieu, "PIR-PSI: scaling private contact discovery," *Proc. Priv. Enhancing Technol.*, 2018.

[7] S. Dittmer, Y. Ishai, S. Lu, R. Ostrovsky, M. Elsabagh, N. Kiourtis, B. Schulte, and A. Stavrou, "Streaming and unbalanced PSI from function secret sharing," in *SCN 2022*.

[8] J. Doerner and A. Shelat, "Scaling ORAM for secure computation," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[9] W. Zhang, X. Guo, K. Yang, R. Zhu, Y. Yu, and X. Wang, "Efficient actively secure DPF and RAM-based 2PC with one-bit leakage," Cryptology ePrint Archive, Paper 2024/426, 2024, https://eprint.iacr.org/2024/426. [Online]. Available: https://eprint.iacr.org/2024/426

[10] H. Corrigan-Gibbs and D. Boneh, "Prio: Private, robust, and scalable computation of aggregate statistics," in *USENIX 2017*, 2017.

[11] S. Angel, H. Chen, K. Laine, and S. T. V. Setty, "PIR with compressed queries and amortized query processing," in *SP 2018*, 2018.

[12] E. Boyle, G. Couteau, N. Gilboa, and Y. Ishai, "Compressing vector OLE," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[13] P. Schoppmann, A. Gascón, L. Reichert, and M. Raykova, "Distributed vector-ole: Improved constructions and implementation," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[14] L. de Castro and A. Polychroniadou, "Lightweight, maliciously secure verifiable function secret sharing," in *Advances in cryptology – EUROCRYPT 2022*, 2022.

[15] E. Külaots, T. Krips, H. Eerikson, and P. Pullonen-Raudvere, "SLAMP-FSS: two-party multi-point function secret sharing from simple linear algebra," *IACR Cryptol. ePrint Arch.*, p. 1394, 2024.

[16] G. Garimella, B. Pinkas, M. Rosulek, N. Trieu, and A. Yanai, "Oblivious key-value stores and amplification for private set intersection," in *Advances in Cryptology – CRYPTO 2021*, 2021.

[17] S. Raghuraman and P. Rindal, "Blazing fast PSI from improved OKVS and subfield VOLE," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[18] A. Bienstock, S. Patel, J. Y. Seo, and K. Yeo, "Near-Optimal oblivious Key-Value stores for efficient PSI, PSU and Volume-Hiding Multi-Maps," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[19] L. Roy, "Softspokenot: Quieter ot extension from small-field silent vole in the minicrypt model," in *Advances in Cryptology – CRYPTO 2022*, Y. Dodis and T. Shrimpton, Eds. Cham: Springer Nature Switzerland, 2022, pp. 657–687.

[20] E. Boyle, N. Gilboa, and Y. Ishai, "Function secret sharing," in *Advances in Cryptology - EUROCRYPT 2015*, 2015.

[21] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai, "Batch codes and their applications," in *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, 2004.

[22] K. Yeo, "Cuckoo hashing in cryptography: Optimal parameters, robustness and applications," in *Advances in Cryptology – CRYPTO 2023*, H. Handschuh and A. Lysyanskaya, Eds., 2023.

[23] M. B. Paterson, D. R. Stinson, and R. Wei, "Combinatorial batch codes," *Adv. Math. Commun.*, 2009.

[24] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *Algorithms – ESA 2001*, 2001.

[25] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai, "Arithmetic sketching," in *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part I*, 2023.

[26] E. Boyle, N. Chandran, N. Gilboa, D. Gupta, Y. Ishai, N. Kumar, and M. Rathee, "Function secret sharing for mixed-mode and fixed-point secure computation," in *Advances in Cryptology – EUROCRYPT 2021*, 2021.

[27] E. Boyle, N. Gilboa, and Y. Ishai, "Secure computation with preprocessing via function secret sharing," in *Theory of Cryptography*, 2019.

[28] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, N. Resch, and P. Scholl, "Correlated pseudorandomness from expand-accumulate codes," in *Advances in Cryptology – CRYPTO 2022*, 2022.

[29] J. Black and P. Rogaway, "Ciphers with arbitrary finite domains," in *Topics in Cryptology — CT-RSA 2002*, 2002.

[30] L. Granboulan and T. Pornin, "Perfect block ciphers with small blocks," in *Fast Software Encryption*, 2007.

[31] E. Stefanov and E. Shi, "FastPRP: Fast pseudo-random permutations for small domains," *IACR Cryptol. ePrint Arch.*, 2012.

[32] K. T. Eames and M. J. Keeling, "Contact tracing and disease control," *Proceedings of the Royal Society of London. Series B: Biological Sciences*, 2003.

Appendix A.
Sub-protocols $\Pi_{\mathsf{SharedIPD}}, \Pi_{\mathsf{CW}}$ of $\Pi_{\mathsf{big\text{-}state\text{-}DKG}}$

Figure 9: The ideal functionality $\mathcal{F}_{\mathsf{CW}}$ in $\Pi_{\mathsf{big\text{-}state\text{-}DKG}}$. Party b has input a number $i \in [n]$, a share $[\![A]\!]_b$ of the accepting points, and a share $[\![L]\!]_b$ of a length-t list L of $(2\lambda + 2t)$-bit strings. The functionality interacts with a party b and an adversary Adv.

$\mathcal{F}_{\mathsf{CW}}$:
- Get i, $[\![A]\!]_b, [\![L]\!]_b$ from party b and $[\![A]\!]_{\bar{b}}, [\![L]\!]_{\bar{b}}$ from Adv.
- Reconstruct $A = [\![A]\!]_0 + [\![A]\!]_1$ and $L = [\![L]\!]_0 \oplus [\![L]\!]_1$.
- Initialize an empty list $V^{(i-1)}$
- **for** $k = 1$ to $|A^{(i-1)}|$ **do**
  - Parse $L[k] = \Delta\mathsf{seed}^0 \| \Delta\mathsf{sign}^0 \| \Delta\mathsf{seed}^1 \| \Delta\mathsf{sign}^1$ where $|\mathsf{seed}^0| = |\mathsf{seed}^1| = \lambda$ and $|\mathsf{sign}^0| = |\mathsf{sign}^1| = t$.
  - Append entries to $V^{(i-1)}$ as is done in line 13 to line 17 in Figure 1.
- **end for**
- Invoke $\mathsf{CW} \leftarrow \mathsf{GenCW}(i, A, V^{(i-1)})$ in Figure 2.
- Output CW to both party b and Adv.

We briefly explain here the intuition of how $\Pi_{\mathsf{SharedIPD}}$ realizes its functionality, which is the core of the security of $\Pi_{\mathsf{big\text{-}state\text{-}DKG}}$. The formal security proofs for these protocols are provided in our full version.

Opening up the shared inner-product functionality, it can be directly realized by generic secure 2PC using $|\Delta^{(i-1)}|$ single-bit AND gates and $|\Delta^{(i-1)}|$ XOR gates. When $|\Delta^{(i-1)}|$ is relatively small, i.e., for the first several layers, this is efficient enough. To further reduce the online communication for larger i, we use a hashing-based protocol

Figure 10: $\Pi_{\mathsf{SharedIPD}}$ and its ideal functionality $\mathcal{F}_{\mathsf{SharedIPD}}$ in $\Pi_{\mathsf{big\text{-}state\text{-}DKG}}$. Fix two security parameters s, s'. Fix a length parameter l and a weight parameter t which is the same t as in $\Pi_{\mathsf{big\text{-}state\text{-}DKG}}$ (l might be smaller than t). Denote $T = \log_2(\mathrm{e})\mathrm{t}$. Party b has input $[\![e]\!]_b$ and $[\![v]\!]_b$, where e is a unit vector and v is of hamming weight at most t in $\{0,1\}^l$. The functionality interacts with a party b and an adversary Adv.

$\mathcal{F}_{\mathsf{SharedIPD}}$:
- Get $[\![e]\!]_b, [\![v]\!]_b$ from party b and $[\![e]\!]_{\bar{b}}, [\![v]\!]_{\bar{b}}$ from Adv.
- Reconstruct $e = [\![e]\!]_0 \oplus [\![e]\!]_1$ and $v = [\![v]\!]_0 \oplus [\![v]\!]_1$.
- Compute additive shares of the inner-product $e \cdot v$.
- Generate random shares $\{[\![0]\!]_b\}_{b=0,1}$ and $\{[\![1]\!]_b\}_{b=0,1}$.
- **if** $e \cdot v = 0$ **then**
  - With probability $1 - \left(1 - (1 - 1/T)^{t-1}\right)^s \left(1 - 2^{-s'}\right)^s$ output $[\![0]\!]_b$ and $[\![0]\!]_{\bar{b}}$ to party b and Adv, and otherwise output $[\![1]\!]_b$ and $[\![1]\!]_{\bar{b}}$ to party b and Adv.
- **else**
  - With probability $p_1 = \left(1 - 2^{-s'}\right)^s$ output $[\![1]\!]_b$ and $[\![1]\!]_{\bar{b}}$ to party b and Adv, and otherwise output $[\![0]\!]_b$ and $[\![0]\!]_{\bar{b}}$ to party b and Adv.
- **end if**

$\Pi_{\mathsf{SharedIPD}}([\![e]\!]_b, [\![v]\!]_b)$ for party $b \in \{0,1\}$:
- Let $P_1, P_2, \ldots, P_s$ be s public random partitions of $[l]$ such that in P_i, each element in $[l]$ randomly falls into one of the T blocks $P_i^1, P_i^2, \ldots, P_i^T$.
- For $i \in [s]$ and $k \in [T]$, let all $R_i^k \xleftarrow{\$} \{0,1\}^{s' \times |P_i^k|}$ be independently sampled public random matrices.
- For $i \in [s]$ and $k \in [T]$, compute $[\![\mu_i^k]\!]_b := \bigoplus_{r \in P_i^k} [\![e]\!]_b[r]$ and $[\![\nu_i^k]\!]_b := R_i^k \cdot [\![v]\!]_b|_{P_i^k}$, where $[\![v]\!]_b|_{P_i^k}$ denotes the restriction of $[\![v]\!]_b$ to the indices contained in P_i^k.
- Invoke a generic secure 2-party computation with party $\bar{b}$ to compute a share $[\![\varphi]\!]_b$ of the Boolean formula $\varphi := \bigvee_{\substack{i \in [s] \\ k \in [T]}} \left(\mu_i^k \wedge \mathbf{1}_{\nu_i^k = 0^{s'}}\right)$.
- Output $b \oplus [\![\varphi]\!]_b$.

$\Pi_{\mathsf{SharedIPD}}$ in Figure 10 that randomly and simultaneously partitions the unit vector $\vec{e}$ (with the jth entry being 1) and the sparse vector $\vec{v}$ into $T = \log_2(\mathrm{e})\mathrm{t}$ blocks, then checks whether $\vec{v}$ is a zero vector restricting to the block containing the jth entry. For a *good* partition where the mentioned block does not contain any nonzero entries of $\vec{v}$ except for the jth entry, $\vec{v}$ is a zero vector restricting to this block if and only if $\vec{e} \cdot \vec{v} = 0$. A partition is good with at least $(1 - \frac{1}{T})^{t-1} = \Omega(1)$, so with overwhelming probability a good partition can appear and be detected within some repetitions. The size of circuit to compute the above function is $O_s(t)$ where s denotes a security parameter, and is sublinear (in fact independent) to the length of vectors.

Appendix B.
Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This paper introduces two new constructions for Distributed Multi-Point Functions (DMPF). Given sets $A = \{\alpha_1, \ldots, \alpha_t\}$ and $B = \{\beta_1, \ldots, \beta_t\}$, the multi-point function $f_{A,B}$ is defined as follows: $f_{A,B}(\alpha_i) = \beta_i$ for $i \leq t$ and $f_{A,B}(x) = 0$ for $x \notin A$. A distributed multi-point function (DMPF) is a cryptographic primitive allowing one to efficiently generate shares of the entire function $f_{A,B}$ across two parties enabling each party to locally generate additive shares of $f_{A,B}(x)$ for any possible input x. The first DMPF construction directly generalizes the indicator bit in the standard GGM tree-based DPF construction to an indicator string, to identify multiple nonzero entries of the shared vector. The second construction replaces the correction word at each layer of the tree with an oblivious key-value store (OKVS) that holds t correction words at each layer of the tree. The parties look up the appropriate correction word in the OKVS at each layer.

B.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) The paper provides a valuable step forward in an established field. The paper provides two novel DMPF constructions.
2) The paper creates a new tool to enable future science. The author(s) implement their DMPF constructions and have made their source code available. The paper empirically demonstrate applications of their DMPF constructions to pseudorandom correlation generators (PCFs) and two-server private set intersection (PSI).

B.4. Noteworthy Concerns

The original submission of the paper did not include proofs for any of the three theorems in the paper. The reviewers found value in the paper without the proofs, but encouraged the authors to include the proofs in a full version. Because the proofs were not made available until late in the review cycle they have not been checked in detail by the PC.