Notes for New Constructions of DMPF

tbd

ABSTRACT

tbd.

CCS CONCEPTS

• **Theory of computation → Cryptographic primitives**.

KEYWORDS

tbd

ACM Reference Format:

tbd. tbd. Notes for New Constructions of DMPF. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 12 pages. https://doi.org/tbd

1 INTRODUCTION

tbd

2 PRELIMINARY

2.1 Basic Notations

Point and multi-point functions. 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. A *t-point function* $f_{A,B} : [N] \to \mathbb{G}$ for $A = (\alpha_1, \cdots \alpha_t) \in [N]^t$ and $B = (\beta_1, \cdots, \beta_t) \in \mathbb{G}^t$ evaluates to β_i on input α_i for $1 \le i \le t$ and to 0 on all other inputs. Denote $\hat{f}_{A,B}(N, \hat{\mathbb{G}}, t, A, B)$ the representation of such a t-point function. Call the collection of all t-point functions for all t *multi-point functions.*

Enote: MPF. Also representation of groups.

2.2 Distributed Multi-Point Functions

Enote: should directly adapt to multi-point function case

We begin by defining a slightly generalized notion of distributed point functions (DPFs), which accounts for the extra parameter $\mathbb{G}'$. Yaxin: What is $\mathbb{G}'$?

Definition 1 (DPF [5, 10]). *A (2-party)* distributed point function (DPF) *is a triple of algorithms* $\Pi = (\mathsf{Gen}, \mathsf{Eval}_0, \mathsf{Eval}_1)$ *with the following syntax:*

- $\mathsf{Gen}(1^\lambda, \hat{f}_{\alpha,\beta}) \to (k_0, k_1)$*: On input security parameter* $\lambda \in \mathbb{N}$ *and point function description* $\hat{f}_{\alpha,\beta} = (N, \hat{\mathbb{G}}, \alpha, \beta)$*, the (randomized) key generation algorithm* Gen *returns a pair of keys* $k_0, k_1 \in \{0,1\}^*$. *Yaxin: Matan points out: we want efficient procedures, i.e.,* $|k_b| \in \mathsf{poly}(\lambda)$. *Stress it here or add efficiency requirement?We assume that* N *and* $\mathbb{G}$ *are determined by each key.*
- $\mathsf{Eval}_b(k_b, x) \to y_b$*: On input key* $k_b \in \{0,1\}^*$ *and input* $x \in [N]$ *the (deterministic) evaluation algorithm of server* b*,* Eval_b *returns* $y_b \in \mathbb{G}$.

We require Π *to satisfy the following requirements:*

- ***Correctness:*** *For every* λ, $\hat{f} = \hat{f}_{\alpha,\beta} = (N, \hat{\mathbb{G}}, \alpha, \beta)$ *such that* $\beta \in \mathbb{G}$*, and* $x \in [N]$*, for* $b = 0, 1$,

$$\Pr\left[(k_0, k_1) \leftarrow \mathsf{Gen}(1^\lambda, \hat{f}), \sum_{i=0}^{1} \mathsf{Eval}_b(k_b, x) = f_{\alpha,\beta}(x)\right] = 1$$

- ***Security:*** *Consider the following semantic security challenge experiment for corrupted server* $b \in \{0,1\}$*:*
  1. *The adversary produces two point function descriptions* $(\hat{f}^0 = (N, \hat{\mathbb{G}}, \alpha_0, \beta_0), \hat{f}^1 = (N, \hat{\mathbb{G}}, \alpha_1, \beta_1)) \leftarrow \mathcal{A}(1^\lambda)$*, where* $\alpha_b \in [N]$ *and* $\beta_b \in \mathbb{G}$.
  2. *The challenger samples* $b \leftarrow \{0,1\}$ *and* $(k_0, k_1) \leftarrow \mathsf{Gen}(1^\lambda, \hat{f}^b)$.
  3. *The adversary outputs a guess* $b' \leftarrow \mathcal{A}(k_b)$.

  Denote by $\mathsf{Adv}(1^\lambda, \mathcal{A}, i) = \Pr[b = b'] - 1/2$ *the advantage of* $\mathcal{A}$ *in guessing* b *in the above experiment. For every non-uniform polynomial time adversary* $\mathcal{A}$ *there exists a negligible function* ν *such that* $\mathsf{Adv}(1^\lambda, \mathcal{A}, i) \le \nu(\lambda)$ *for all* $\lambda \in \mathbb{N}$.

Definition 2 (DMPF). *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:*

- $\mathsf{Gen}(1^\lambda, \hat{f}_{A,B}) \to (k_0, k_1)$*: 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* $k_0, k_1 \in \{0,1\}^*$. *Yaxin: On Matan's behalf: same comment as well. Maybe* $|k_i| = \mathsf{poly}(\lambda, t)$.
- $\mathsf{Eval}_b(1^\lambda, k_b, x) \to y_b$*: On input key* $k_b \in \{0,1\}^*$ *and input* $x \in [N]$ *the (deterministic) evaluation algorithm of server* b*,* Eval_b *returns* $y_b \in \mathbb{G}$.

We require Π *to satisfy the following requirements:*

- ***Correctness:*** *For every* λ, $\hat{f} = \hat{f}_{A,B} = (N, \hat{\mathbb{G}}, t, A, B)$ *such that* $B \in \mathbb{G}^t$*, and* $x \in [N]$*, for* $b = 0, 1$,

$$\Pr\left[(k_0, k_1) \leftarrow \mathsf{Gen}(1^\lambda, \hat{f}), \sum_{i=0}^{1} \mathsf{Eval}_b(k_b, x) = f_{A,B}(x)\right] = 1$$

- ***Security:*** *Consider the following semantic security challenge experiment for corrupted server* $b \in \{0,1\}$*:*
  1. *The adversary produces two* t*-point function descriptions* $(\hat{f}^0 = (N, \hat{\mathbb{G}}, t, A_0, B_0), \hat{f}^1 = (N, \hat{\mathbb{G}}, t, A_1, B_1)) \leftarrow \mathcal{A}(1^\lambda)$*, where* $\alpha_b \in [N]$ *and* $\beta_b \in \mathbb{G}$.
  2. *The challenger samples* $b \leftarrow \{0,1\}$ *and* $(k_0, k_1) \leftarrow \mathsf{Gen}(1^\lambda, \hat{f}^b)$.
  3. *The adversary outputs a guess* $b' \leftarrow \mathcal{A}(k_b)$.

Conference acronym 'XX, tbd, tbd

ACM ISBN tbd…$15.00

https://doi.org/tbd

Denote by $\mathsf{Adv}(1^\lambda, \mathcal{A}, i) = \Pr[b = b'] - 1/2$ *the advantage of* $\mathcal{A}$ *in guessing* b *in the above experiment. For every non-uniform polynomial time adversary* $\mathcal{A}$ *there exists a negligible function* ν *such that* $\mathsf{Adv}(1^\lambda, \mathcal{A}, i) \le \nu(\lambda)$ *for all* $\lambda \in \mathbb{N}$.

We will also be interested in applying the evaluation algorithm on *all* inputs. Given a DMPF $(\mathsf{Gen}, \mathsf{Eval}_0, \mathsf{Eval}_1)$, we denote by $\mathsf{FullEval}_b$ an algorithm which computes Eval_b on every input x. Hence, $\mathsf{FullEval}_b$ receives only a key k_b as input.

One can construct a DMPF scheme for t-point functions by simply summing t DPFs. We denote this DMPF scheme as the naïve construction.

Construction 1 (Naïve construction of DMPF). *Given DPF for domain of size* N *and output group* $\mathbb{G}$*, we can construct a DMPF scheme for* t*-point functions with domain size* N *and output group* $\mathbb{G}$ *as follows:*

- $\mathsf{Gen}(1^\lambda, \hat{f}_{A,B}) \to (k_0, k_1)$*: Suppose* $A = \{\alpha_1, \dots, \alpha_t\}$ *and* $B = \{\beta_1, \dots, \beta_t\}$*. For* $1 \le i \le t$*, invoke* $DPF.\mathsf{Gen}(1^\lambda, \hat{f}_{\alpha_i,\beta_i}) \to (k_0^i, k_1^i)$*. Set* $(k_0, k_1) = (\{k_0^i\}_{i\in[t]}, \{k_1^i\}_{i\in[t]})$.
- $\mathsf{Eval}_b(k_b, x) \to y_b$*: Compute* $y_b = \sum_{i\in[t]} DPF.\mathsf{Eval}_b(k_b^i, x)$.
- $\mathsf{FullEval}_b(k_b) \to Y_b$*: Compute* $Y_b = \sum_{i\in[t]} DPF.\mathsf{FullEval}_b(k_b^i, x)$.

When the DPF scheme is correct and secure, the naïve construction of DMPF is also correct and secure. We note that the keysize and running time of Gen, Eval and FullEval of the naïve construction of DMPF equals $t\times$ the keysize and $t\times$ the running time of Gen, Eval and FullEval of DPF, respectively. We aim to provide DMPF schemes that has Eval and FullEval time almost independent to t.

2.3 Batch Code

We introduce batch code and probabilistic batch code, which can be used to construct DMPF (see construction 3).

Definition 3 (Batch Code[11]). *An* (N, M, t, m)*-batch code over alphabet* Σ *is given by a pair of efficient algorithms* (Encode, Decode) *such that:*

- $\mathsf{Encode}(x \in \Sigma^N) \to (C_1, C_2, \cdots, C_m)$*: Any string* $x \in \Sigma^N$ *is encoded into an* m*-tuple of codewords* $C_1, C_2, \cdots C_m \in \Sigma^*$ *of total length* M.
- $\mathsf{Decode}(I, C_1, C_2, \cdots, C_m) \to \{x[i]\}_{i\in I}$*: On input a set* I *of* t *distinct indices in* $[N]$ *and* m *codewords, recover* t *coordinates of* x *indexed by* I *by reading at most one coordinate from each of the* m *codeword.*

We will focus on a special class of batch code called combinatorial batch code (CBC)[11, 14], where each codeword C_i is a subset of x and during decoding the elements read from m codewords should cover the desired set $\{x[i]\}_{i\in I}$. In this case, the encoding algorithm is equivalent to replicating and allocating the indices in $[N]$ to m buckets, and the decoding algorithm is equivalent to finding a prefect matching from the size-t subset $I \subseteq [N]$ to the m buckets, while indicating which position in each bucket should be read.

Yaxin: Add example instantiation (random regular bipartite graph) and explain it is not efficient?

It is useful to relax the definition of batch code, allowing failure probability when decoding.

Definition 4 (Probabilistic Batch Code [1]). *An* (N, M, t, m, ϵ)*-probabilistic batch code over alphabet* Σ *is a randomized* (N, M, t, m)*-batch code with* ***public randomness*** r *(and possibly private randomness for each sub-procedure) such that for any string* x *and any set* I *of* t *distinct indices in* $[N]$,

$$\Pr[\mathsf{Decode}_r(I, \mathsf{Encode}_r(x)) \to \{x[i]\}_{i\in I}] = 1 - \epsilon$$

where the probability is taken over the public randomness r *and private randomness of* Encode *and* Decode *algorithms.*

Similarly, a probabilistic CBC (PCBC) will just be a CBC with failure probability when decoding. We mention Cuckoo hashing algorithm[13] as a concrete instantiation of PCBC[1].

w-way cuckoo hashing. Given t balls, $m = et$ buckets (e is some expansion parameter that is bigger than 1), and w independent hash functions $h_1, h_2, \cdots, h_w$ randomly mapping every ball to a bucket, allocates all balls to the buckets such that each bucket contains at most one ball through the following process:

1. Choose an arbitrary unallocated ball b. If there is no unallocated ball, output the allocation.
2. Choose a random hash function h_i 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 set b' the current unallocated ball, and repeat step 2.

If the algorithm terminates then its output is an allocation of balls to buckets such that each bucket contains at most one ball. However there is no guarantee that the algorithm will terminate - it may end up in a loop and keeps running forever. To fixed this problem, the algorithm should be given a fixed amount of time to run, or equipped with a loop detection process to guarantee termination. We call it a *failure* whenver the algorithm fails to output a proper allocation where each bucket contains at most one ball.

Yaxin: Asymptotic parameters: [17, Theorem 1] gives w-way cuckoo hashing scheme for t balls, with $w = O(\sqrt{\lambda_{\mathsf{stat}} \log t})$ and $m = O(t)$, making $O(\sqrt{\lambda_{\mathsf{stat}} \log t})$ queries to the hash functions. It supposes the hash functions from a $O(t\sqrt{\lambda_{\mathsf{stat}} \log t})$-wise independent hash function family

The failure probability of cuckoo hashing. Let's denote the failure probability of w-way cuckoo hashing to be $\epsilon = 2^{-\lambda_{\mathsf{stat}}}$. In practice we usually consider the statistical security parameter λ_{stat} to be 30 or 40. The empirical result in [6] shows for $w = 3$, $m = 16384$, $\lambda_{\mathsf{stat}} = 124.4e - 144.6$ where e is the expansion parameter that $m = et$. For $w = 3$, $m = 8192$, $\lambda_{\mathsf{stat}} = 125e - 145$. However we use cuckoo hashing to construct DMPF for t-point functions, in which case we'd also care about t being small, say 2, 3 or 100, and m should not be too large. In this sense the previous empirical results are not complete. Yaxin: [1] uses $w = 3, e = 1.5, t > 200$ and $\lambda_{\mathsf{stat}} \approx 40$ and claims it follows the analysis from [6], but I don't see how...Check [17] for concrete parameters with different N and t.

With the t balls replicated and allocated to m buckets, the cuckoo hashing algorithm essentially finds a perfect matching from t balls to m buckets, which coincides with the form of (probabilistic) CBC decoding. Therefore a PCBC follows directly from a cuckoo hashing scheme:

Construction 2 (PCBC from cuckoo hashing). *Given w-way cuckoo hashing as a sub-procedure allocating t balls to m buckets with failure probability ϵ, an (N, wN, t, m, ϵ)-PCBC is as follows:*

- $\mathsf{Encode}_r(x \in \Sigma^N) \to (C_1, \cdots, C_m)$*: Use r to determine w independent random hash functions $h_1, h_2, \cdots h_w$ that maps from $[N]$ to $[m]$. Let C_j be $\{x[i] : h_l(i) = j \text{ for some } l \in [w]\}$, in ascending order of i.*
- $\mathsf{Decode}_r(I, C_1, \cdots, C_m) \to \{x[i]\}_{i\in I}$*: Determine $h_1, \cdots, h_w$ as in* Encode*. For I of size t, find a perfect matching from I to $[m]$ using a w-way cuckoo hashing scheme. For each $i \in I$, fetch $x[i]$ from C_j where i and j are matched in the perfect matching. Note that $x[i]$ can be found in the kth position of C_j where i is the kth smallest index of $\{i : h_l(i) = j \text{ for some } l \in [w]\}$.*

An ambiguous point in Decode_r is how to find the index of $x[i]$ in C_j it is mapped to. We display two solutions to this index finding problem:

(1) When N is a feasible number, one can directly compute the entire hash tables derived by $h_1, \ldots, h_w$ and compute the index of $x[i]$ in C_j.
(2) One can implement w hash functions by a single *random permutation* P mapping from $[w] \times [N]$ to $[m] \times [B]$, where $B = wN/m$. Invocation of $h_i(j)$ is done by computing $P(i, j)$, which outputs the bucket number in $[m]$ and the index in $[B]$. Note that in this case $h_1, \ldots, h_w$ are not independent random hash functions, but as long as they Yaxin: satisfy some sufficient independence property. To be clarified. This solution is noted in [7] where P is realized by a PRP.

2.4 DMPF Construction from CBC

We display the construction of DMPF from black-box usage of DPF basing on PCBC with appropriate parameters, which has been discussed in previous literature[3, 7]. As discussed before, we assume that the PCBC encoding and decoding are oblivious of the content of the input string.

Construction 3 (DMPF from DPF basing on PCBC). *Given DPF for any domain of size no larger than N and output group $\mathbb{G}$, and an (N, M, t, m, ϵ)-PCBC with alphabet $\Sigma = \mathbb{G}$, we can construct a DMPF scheme for t-point functions with domain size N and output group $\mathbb{G}$ as follows:*

- $\mathsf{Gen}(1^\lambda, \hat{f}_{A,B}) \to (k_0, k_1)$*: Suppose $A = \{\alpha_1, \cdots, \alpha_t\}$ and $B = \{\beta_1, \cdots, \beta_t\}$. Compute* $\mathsf{Encode}([N]) \to (C_1, \cdots, C_m)$ *according to the PCBC. Then run* $\mathsf{Decode}(A, C_1, \cdots, C_m)$ *to determine a perfect matching from A to $\{C_1, \cdots, C_m\}$. For $1 \le i \le m$, let $f_i : [|C_i|] \to \mathbb{G}$ be the following:*
  - *If C_i is assigned none of A by the perfect matching, then set f_i to be the all-zero function.*
  - *If exactly one α_j of A is assigned to the lth position of C_i, then set f_i to be the point function that outputs β_j on l and 0 elsewhere.*

  For $1 \le i \le m$, invoke DPF.$\mathsf{Gen}(1^\lambda, f_i) \to (k_0^i, k_1^i)$*. Set $(k_0, k_1) = (\{k_0^i\}_{i\in[m]}, \{k_1^i\}_{i\in[m]})$. If* Decode *fails then run* Encode *and* Decode *again with fresh randomness.*
- $\mathsf{Eval}_b(k_b, x) \to y_b$*: Follow* $\mathsf{Encode}([N])$ *to determine the positions $l_{j_1}, l_{j_2}, \cdots, l_{j_s}$ such that the x is sent to the l_{j_i}-th position of C_{j_i} (since the allocation of indices is oblivious of the content of TT). Compute $y_b = \sum_{i=1}^{s}$ DPF.*$\mathsf{Eval}_b(k_b^{j_i}, l_i)$.
- $\mathsf{FullEval}_b(k_b) \to Y_b$*: Compute $Y_b^i =$ DPF.*$\mathsf{FullEval}_b(k_b^i)$ *for $1 \le i \le m$. For each input $x \in [N]$, follow* $\mathsf{Encode}([N])$ *to choose a position l_x in bucket C_{j_x} that x is sent to. Let $Y_b[x] \leftarrow Y_b^{j_x}[l_x]$.*

The scheme is correct with overwhelming probability and has distinguish advantage $< 2\epsilon$.

Note that if one use CBC instead of PCBC then the DMPF scheme is perfectly correct and secure.

When instantiating PCBC using w-way cuckoo hashing, the *key generation time* is roughly the time for computing cuckoo hashing algorithm, the time for finding t indices for t elements in the buckets, plus the total time of all $\mathsf{DPF.Gen}(1^\lambda, f_i)$. The *evaluation time* is roughly the time for finding w indices for one element in the buckets plus the total time of all $\mathsf{DPF.Eval}_b(k_b^{j_i}, l_i)$. Similarly, the *full-domain evaluation time* is roughly the time for finding N indices for N elements in the buckets plus the total time of all $\mathsf{DPF.FullEval}_b(k_b^j)$ for $j = 1, \ldots, m$.

2.5 Oblivious Key-Value Stores

We introduce the notion of Oblivious key-value stores (OKVS) which can be used to construct DMPF. OKVS was originally proposed as a primitive for private set intersection (PSI) protocols (see [9, 15]).

Definition 5 (Oblivious Key-Value Stores (OKVS)[9, 15]). *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} and 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 m. The algorithms are of the following syntax:*

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

- *For all $S \in (\mathcal{K} \times \mathcal{V})^t$, $\Pr_{r\leftarrow\{0,1\}^\kappa}[\mathsf{Encode}_r(S) = \perp] \le 2^{-\lambda_{\mathsf{stat}}}$.*
- *For all $S \in (\mathcal{K}\times\mathcal{V})^t$ and $r \in \{0,1\}^\kappa$ such that $\mathsf{Encode}_r(S) \to P \neq \perp$, it is the case that $\mathsf{Decode}_r(P, k) \to v$ whenever $(k, v) \in S$.*
- ***Obliviousness:** Given any distinct key 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.,*

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

One can obtain a linear OKVS *if in addition require:*

- ***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 system $\{\langle \mathrm{row}_r(k_i), P\rangle = v_i\}_{1\le i\le t}$.

We evaluate an OKVS scheme by its rate ($\frac{\text{input length } t}{\text{output length } m}$), encoding time and decoding time. A well-known, optimal-rate OKVS construction is encoding t key-value pairs using a deg-t polynomial:

Construction 4 (Polynomial). *Suppose $\mathcal{K} = \mathcal{V} = \mathbb{F}$ is a field. Set*

- $\mathrm{Encode}(\{(k_i, v_i)\}_{1\le i\le t}) \to P$ *where P is the coeffients of a $(t-1)$-degree $\mathbb{F}$-polynomial g_P that $g_P(k_i) = v_i$ for $1 \le i \le t$.*
- $\mathrm{Decode}(P, k) \to g_P(k)$.

The polynomial OKVS possesses an optimal encoding size $m = n$, but the Encode process is a polynomial interpolation which is only known to be achieved in time $O(t \log^2 t)$. The time for a single decoding is $O(t)$ and that for batched decodings is (amortized) $O(\log^2 t)$.

In the sequel we stress two alternative (linear) OKVS constructions that has near optimal encoding size but much better running time.

Construction 5 (RR22[9, 15]). *Suppose $\mathcal{V} = \mathbb{F}$ is a field. Set $\mathrm{row}_r(k) := \mathrm{row}_r^{\mathrm{sparse}}(k) || \mathrm{row}_r^{\mathrm{dense}}(k)$ where $\mathrm{row}_r^{\mathrm{sparse}}(k)$ outputs a uniformly random weight-w vector in $\{0,1\}^{m_1}$, and $\mathrm{row}_r^{\mathrm{dense}}(k)$ outputs a short dense vector in $\mathbb{F}^{m_2}$.*

- $\mathrm{Encode}_r(\{(k_i, v_i)\}_{1\le i\le t}) \to P$ *where P is randomly chosen from the solutions of the system $\{\langle \mathrm{row}_r(k_i), P\rangle = v_i\}_{1\le i\le t}$, solved by the triangulation algorithm in [15]. If the system has no solution then output $\perp$.*
- $\mathrm{Decode}_r(P, k) \to \langle \mathrm{row}_r(k), P\rangle$.

We denote $m_1 = et$, where e is an expansion parameter indicating the rough blowup to store t pairs. In practice the number of dense columns m_2 is usually set to a small constant.

This OKVS construction features an efficient encoding process, constant decoding time ($(w + m_2)$ additions and m_2 multiplications in $\mathbb{F}$) while having a linear encoding size.

Encode may output $\perp$ if the matrix formed by $\{\mathrm{row}_r(k_i)\}_{1\le i\le t}$ is not full-rank. Therefore we need to adjust the parameters $m_1 = et$ and m_2 to ensure negligible error probability (represented by the statistical security parameter λ_{stat}). The expansion parameter e and the number of dense columns $m_2 := \hat{g}$ (where $\hat{g}$ is a parameter relating to the equation system solving process) are given by the analysis in [15], with the range of N from 2^6 to 2^{18}: Given w, t and λ_{stat}, the choices of the e and $\hat{g}$ are fixed through the following steps:

- Set $e^* = \begin{cases} 1.223 & w = 3 \\ 1.293 & w = 4 \\ 0.1485w + 0.6845 & w \ge 5 \end{cases}$.
- Compute $\alpha := 0.55 \log_2 t + 0.093w^3 - 1.01w^2 + 2.92w - 0.13$.
- $e := e^* + 2^{-\alpha}(\lambda_{\mathrm{stat}} + 9.2)$.
- $\hat{g} := \frac{\lambda_{\mathrm{stat}}}{(w-2)\log_2(et)}$.

Yaxin: Fix t and λ_{stat}, we want to find the best choice of w. The adavantageous choices of w in [15] are $w = 3$ and $w = 5$. From the first sight when w is smaller e can be smaller but $\hat{g}$ will be larger. Since $w + \hat{g}$ stands for number of $\mathbb{F}$-ADD's and $\hat{g}$ stands for number of $\mathbb{F}$-MULT's in decoding, previously I thought $\hat{g}$ is the dominating factor of Decode running time. However table 1 in [15] suggests that $w = 3$ outruns nearly all of other choices of w while $w = 5$ is almost 3 times slower in decoding time. This may suggest there are some other heavy computations other than $\mathbb{F}$-MULT that need to be considered when evaluating running time.

The range of t previous literature [9, 15] have considered in their empirical results are also limited, which will be one of our problems. We want to cover small t, say $t < 100$, while previous literature aiming for constructing PSI protocols usually consider very large t.

One may let row_r^{dense} output a short dense vector in $\{0,1\}^{m_2}$ to avoid multiplication of large field elements in the encoding and decoding processes. To achieve same level of security one could simply set $m_2 = \hat{g} + \lambda_{\mathrm{stat}}$, as proposed in [9, 15]. As indicated by the empirical results in [15], this binary scheme is usually not as efficient as the original design. Therefore we mostly refer to construction 5.

Construction 6 (RB-OKVS[2]). *Suppose $\mathcal{V} = \mathbb{G}$ is a group. Let $\mathrm{row}_r(k)$ output a $\{0,1\}^m$ vector consisting of a width-w random band. Formally speaking, $\mathrm{row}_r(k)$ first determine a starting point $1 \le i \le m - w + 1$ for the band, and then determine random w-bit string to fill in the positions $[i, i + w - 1]$ of $\mathrm{row}_r(k)$ and leave the rest as 0 entries.*

- $\mathrm{Encode}_r(\{(k_i, v_i)\}_{1\le i\le t}) \to P$ *where P is randomly chosen from the random band matrix system $\{\langle \mathrm{row}_r(k_i), P\rangle = v_i\}_{1\le i\le t}$. If the system has no solution then output $\perp$.*
- $\mathrm{Decode}_r(P, k) \to \langle \mathrm{row}_r(k), P\rangle$.

Denote $m = et$ where $e > 1$ is an expansion parameter indicating the blowup to store t pairs.

The encoding time is equivalent to solving a random band matrix system, which can be efficiently done in $O(Nw + n \log n)$ time [2]. The decoding time is w additions in $\mathbb{F}$ and the rate can be very close to 1.

Again, to guarantee the success of Encode, the random band matrix must be full-rank with overwhelming probability. According to [2], fixing $e > 1$ and taking $w = O(\lambda_{\mathrm{stat}}/(e-1) + \log N)$ ensures the correctness and obliviousness with probability $2^{-\lambda_{\mathrm{stat}}}$ and 2^{-w}, respectively. Practically, $e = 1.03, 1.05, 1.07, 1.1$ are taken while w being several hundred to reach the security $\lambda_{\mathrm{stat}} = 40$, with the choice of N varying from 2^{10} to 2^{20}.

According to the comparison in [2] of the RR22-OKVS (construction 5) and the RB-OKVS (construction 6) with the choices of $N = 2^{16}, 2^{20}, 2^{24}$, the RB-OKVS has a better rate and features a tradeoff between rate and encoding/decoding time (one can choose to have better rate with longer encoding/decoding time). The RB-OVS has better encoding time while the RR22-OKVS has better decoding time.

Yaxin: Maybe (and how to) put a (quantitative) summarizing table of OKVS efficiency here?

In our later sections, we will give the decoding efficiency of the OKVS the most priority. To this end, we refer to the RR22-OKVS (construction 5) when instantiating OKVS. One may switch to other OKVS constructions depending on different needs in practice.

3 NEW DMPF CONSTRUCTIONS

In this section, we display two new constructions of DMPF in section 3.2 and section 3.3 respectively, that follow the same paradigm introduced in section 3.1.

3.1 DMPF paradigm

We begin by introducing the DMPF paradigm in fig. 1, which is based on the idea of the DPF construction in [5]. Each key $k_b (b = 0, 1)$ generated by $\mathsf{Gen}(1^\lambda, \hat{f}_{A,B})$ can span a depth-n (n is the input length of $\hat{f}_{A,B}$) complete binary tree T_b. Each node in either tree 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 correponds to any nonzero input $a \in A$ an accepting path.

We call the trees T_0, T_1 the evaluation trees. Each node in the evaluation tree T_b is associated with a $(\lambda + l)$-bit pseudorandom string seed‖sign (the λ-bit seed and l-bit sign are defined in line 28). The two evaluation trees satisfies the following important properties:

(1) T_0 and T_1 have identical strings on every node except for the nodes lying on accepting paths.
(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 pseudorandom and follow some correlation (the correlation is designed by specific instantiations).

Party b can evaluate the input $x = x_1 \cdots x_n$ by calling $\mathsf{Eval}_b(1^\lambda, k_b, x)$, which first parse the key k_b to the seed‖sign string at the root together with n hints $\{CW^{(i)}\}_{i\in[n]}$, for the depth-i layer ($1 \leq i \leq n$) respectively. $\mathsf{Eval}_b(1^\lambda, k_b, x)$ traverses T_b along the path indicated by x, starting from the root, and at a depth-$(i-1)$ node with string seed‖sign generates its children's strings by first computing the $(2\lambda+2l)$-bit pseudorandom string $G(\text{seed})$ where $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2l}$ is a pseudorandom generator, then adding to $G(\text{seed})$ a correction computed by $\mathsf{Correct}(x_1 \ldots x_{i-1}, \text{sign}, CW^{(i)})$ (see line 27), and then assign the left $(\lambda + l)$-bit string to its left child and the rest to its right child. In particular, the additive correction for the seed strings of two children nodes are the same (C_{seed} in line 27) , but those for the sign strings of two children nodes are different (C_{sign^0} for the left child and C_{sign^1} for the right child) in order to force the desired correlation of sign strings.

It is $\mathsf{Gen}(1^\lambda, \hat{f}_{A,B})$'s job to generate appropriate strings for roots of T_0 and T_1 and hints $\{CW^{(i)}\}$ for all layers that maintains the properties 1 and 2. At the depth-i layer, $\mathsf{Gen}(1^\lambda, \hat{f}_{A,B})$ utilizes $\mathsf{GenCW}(i, A, \{\text{seed}_b^{(i-1)}, \text{sign}_b^{(i-1)}\}_{b=0,1})$ to generate the hint $CW^{(i)}$ for both parties (line 11), where $\text{seed}_b^{(i-1)}$ records the seed strings in T_b at the nodes on the accepting paths in the previous layer, and so on. To force the properties 1 and 2 of the evaluation tree, the hint $CW^{(i)}$ should satisfy the following principles:

(1) If a depth-$(i-1)$ parent node is on an accepting path and it has a child node exiting this accepting path, then the corrections for this child node (computed by line 27) should force the strings at this node in T_0 and T_1 to be the same.
(2) For every depth-$(i-1)$ parent node on an accepting path, the sign corrections for its child that is still on an accepting path should force the sign strings at this node in T_0 and T_1 to follow the desired correlation.

The detailed realization of these principles will be discussed in concrete instantiations. We note that forcing the same strings at each node that exits an accepting suffices for achieving property 1: According to the computation of Eval_b, 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.

The paradigm add a convert layer after the last layer of the evaluation tree to convert the strings at the leaf nodes to an element in the output group $\mathbb{G}$ of $f_{A,B}$. A hint $CW^{(n+1)}$ is associated with the convert layer. The output at a leaf node x with string seed‖sign is generated by first computing a pseudorandom $\mathbb{G}$-element $G_{\text{conv}}(\text{seed})$, then adding to $G_{\text{conv}}(\text{seed})$ a correction computed by $\mathsf{ConvCorrect}(x, \text{sign}, CW^{(n+1)})$, and then give a sign $(-1)^b$ depending on the party (see line 30). If the leaf node is not on any accepting path, then $G_{\text{conv}}(\text{seed})$ and the correction should be the same in T_0 and T_1, which means 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 hint $CW^{(n+1)}$ given by $\mathsf{Gen}(1^\lambda, \hat{f}_{A,B})$ should yield corrections that force the outputs in T_0 and T_1 to add up to the corresponding element in B. Such $CW^{(n+1)}$ is correctly generated by GenConvCW (see line 19).

To sum up, we provide the key generation Gen, single-input evaluation Eval and full-domain evaluation FullEval in the paradigm in fig. 1. The computation involves the following methods which will be realized in the next sections:

- Initialize defines the strings at the roots of T_0, T_1.
- GenCW computes hints $\{CW^{(1)}, \cdots CW^{(n)}\}$ associated with n layers that help generate corrections for the strings at the nodes. Two parties use the same set of correction words.
- GenConvCW computes the hint $CW^{(n+1)}$ associated with the convert layer that help genereate corrections for the final output. Two parties use the same set of correction words.
- Correct given a depth-$(i-1)$ parent node, its sign string and the hint $CW^{(i)}$, outputs an (additive) correction for its children's strings.
- ConvCorrect given a leaf node, its sign string and the hint $CW^{(n+1)}$, outputs a correction for the final output in the output group $\mathbb{G}$.

Yaxin: Mention early termination?

3.2 Big-State DMPF

We display our first instantiation of DMPF in fig. 2, basing on the paradigm of DMPF in fig. 1. 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 satisfies properties 1 and 2, such 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 $e_k = 0^{k-1}10^{t-k}$. Then, the (additive) corrections for computing strings at its children generated by line 33 of fig. 2 equals $CW^{(i)}[k]$, the kth entry of the hint $CW^{(i)}$ associated with this layer. According to line 19 in the

Figure 1: The paradigm 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 upperbound of 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.
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 lexicographically order, $B = (\beta_1, \cdots, \beta_t)$. 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 and deduplicated list of i-bit prefixes of strings in A. Specifically, $A^{(0)} = [\epsilon]$.
  For $0 \le i \le n-1$ and $b = 0, 1$, initialize empty lists $\mathsf{seed}_b^{(i)}$ and $\mathsf{sign}_b^{(i)}$.
  $\mathsf{Initialize}(\{\mathsf{seed}_b^{(0)}, \mathsf{sign}_b^{(0)}\}_{b=0,1})$.
  for $i = 1$ to n **do**
    $CW^{(i)} \leftarrow \mathsf{GenCW}(i, A, \{\mathsf{seed}_b^{(i-1)}, \mathsf{sign}_b^{(i-1)}\}_{b=0,1})$.
    for $k = 1$ to $|A^{(i-1)}|$ and $z = 0, 1$ **do**
      Compute $C_{\mathsf{seed},b}\|C_{\mathsf{sign}^0,b}\|C_{\mathsf{sign}^1,b} \leftarrow \mathsf{Correct}(A^{(i-1)}[k], \mathsf{sign}_b^{(i-1)}[k], CW^{(i)})$ 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
  $CW^{(n+1)} \leftarrow \mathsf{GenConvCW}(A, B, \{\mathsf{seed}_b^{(n)}, \mathsf{sign}_b^{(n)}\}_{b=0,1})$.
  Set $k_b \leftarrow (\mathsf{seed}_b^{(0)}, \mathsf{sign}_b^{(0)}, CW^{(1)}, CW^{(2)}, \cdots, CW^{(n+1)})$.
  return (k_0, k_1).
end procedure

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

procedure $\textsc{FullEval}_b(1^\lambda, k_b)$
  Parse $k_b = (\mathsf{seed}^{(0)}, \mathsf{sign}^{(0)}, CW^{(1)}, CW^{(2)}, \cdots, CW^{(n+1)})$.
  For $1 \le i \le n$, $\mathsf{Path}^{(i)} \leftarrow$ the lexicographical ordered list of $\{0,1\}^i$. $\mathsf{Path}^{(0)} \leftarrow [\epsilon]$.
  Yaxin: The evaluation is BFS-style, which costs a lot of memory to store lists $\mathsf{seed}^{(i)}, \mathsf{sign}^{(i)}$. Need a DFS version for large N to reduce memory use? Write in the clear or explain by words?
  for $i = 1$ to n **do**
    for $k = 1$ to 2^{i-1} **do**
      $C_{\mathsf{seed}}\|C_{\mathsf{sign}^0}\|C_{\mathsf{sign}^1} \leftarrow \mathsf{Correct}(\mathsf{Path}^{(i-1)}[k], \mathsf{sign}^{(i-1)}[k], CW^{(i)})$, where $|C_{\mathsf{seed}}| = \lambda$ and $|C_{\mathsf{sign}^0}| = |C_{\mathsf{sign}^1}| = l$.
      $\mathsf{seed}^{(i)}[2k]\|\mathsf{sign}^{(i)}[2k] \leftarrow G_0(\mathsf{seed}^{(i-1)}[k]) \oplus (C_{\mathsf{seed}}\|C_{\mathsf{sign}^0})$, where $|\mathsf{seed}^{(i)}[2k]| = \lambda$ and $|\mathsf{sign}^{(i)}[2k]| = l$.
      $\mathsf{seed}^{(i)}[2k+1]\|\mathsf{sign}^{(i)}[2k+1] \leftarrow G_1(\mathsf{seed}^{(i-1)}[k]) \oplus (C_{\mathsf{seed}}\|C_{\mathsf{sign}^1})$, where $|\mathsf{seed}^{(i)}[2k+1]| = \lambda$ and $|\mathsf{sign}^{(i)}[2k+1]| = l$.
    end for
  end for
  for $k = 1$ to 2^n **do**
    $\mathsf{Output}[k] \leftarrow (-1)^b \cdot \big(G_{\mathsf{conv}}(\mathsf{seed}^{(n)}[k]) + \mathsf{ConvCorrect}(\mathsf{Path}[k], \mathsf{sign}^{(n)}[k], CW^{(n+1)})\big)$.
  end for
  return Output.
end procedure

construction of GenCW, if one of the children exits the accepting path, the seed correction C_{seed} will zero out the difference of 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^t if this child exits the accepting path, or to be a unit vector indicating the index of the accepting path in the next layer this child lies on.

For the convert layer, GenConvCW set $CW^{(n+1)}[k]$ to be the correction that makes the kth accepting leaf's outputs in T_0 and T_1 to add up to $B[k]$.

We informally argue that the correctness of the big-state DMPF holds since properties 1 and 2 of T_0 and T_1 are ensured, which in turn gives correct shares of outputs in the end of evaluation. The security holds since (1) the seed||sign string at the root of T_b is independent of A and B, and (2) each hint $CW^{(i)}$ is masked by the pseudorandom value determined by the other party's key, which is indistinguishable with a truly random hint.

In the end of this section we briefly discuss about the efficiency of the big-state DMPF, which will be discussed in more details in section 4. Set the naïve solution of DMPF that is a sum of t DPFs as a primary benchmark. The ratio of keysize of the big-state DMPF over the naïve solution is roughly $(\lambda+2t)/(\lambda+2) > 1$, which is close to 1 if $t \ll \lambda$. Gen, Eval and FullEval all traverse one evaluation tree while the naïve solution traverse t evaluation trees. However, the PRG used in the big-state DMPF have output length $2\lambda + 2t$, which means the running time still grows with t. In short, the big-state DMPF is faster than the naïve solution with the sacrifice of larger keysize. When $t \ll \lambda$, compared to the naïve solution, the big-state DMPF has similar keysize and almost $\times t$ speedup in running time.

3.3 OKVS-based DMPF

Next we display our second instantiation of DMPF in fig. 3, basing on the paradigm of DMPF in fig. 1. We call this instantiation the OKVS-based DMPF, since we utilize primitive OKVS (see section 2.5 for introduction).

In the OKVS-based DMPF, we set the length l of the sign string to be 1. The sign strings at the same node in T_0 and T_1 will obey the following correlation: they are shares of 1 if this node is on an accepting path and 0 if this node is not on any accepting path. In order to ensure properties 1 and 2, for a parent node on an accepting path, the additive correction $C_{\mathsf{seed}}, C_{\mathsf{sign}^0}$ and C_{sign^1} for the strings at its children can be determined (see the right hand side in line 15 and 18): 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.

To generate hints $\{CW^{(i)}\}$ to yield the corrections, we 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 correponding value. On the depth-i layer, we define the key space to be the set of all depth-i nodes 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. We encode these (node, correction) pairs (there are up to t such pairs) using an OKVS scheme and set the hint $CW^{(i)}$ to

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

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

6: **procedure** GenCW($i, A, \{\mathsf{seed}_b^{(i-1)}, \mathsf{sign}_b^{(i-1)}\}_{b=0,1}$)
7: Let $\{A^{(i)}\}_{0 \le i \le n}$ be defined as in fig. 1.
8: Sample a list CW of t random strings from $\{0,1\}^{\lambda+2t}$.
9: **for** $k = 1$ to $|A^{(i-1)}|$ **do**
10: Parse $G(\mathsf{seed}_b^{(i-1)}[k]) = \mathsf{seed}_b^0 \| \mathsf{sign}_b^0 \| \mathsf{seed}_b^1 \| \mathsf{sign}_b^1$, for $b = 0, 1$, $\mathsf{seed}_b^0, \mathsf{seed}_b^1 \in \{0,1\}^\lambda$ and $\mathsf{sign}_b^0, \mathsf{sign}_b^1 \in \{0,1\}^t$.
11: Compute $\Delta\mathsf{seed}^c = \mathsf{seed}_0^c \oplus \mathsf{seed}_1^c$ and $\Delta\mathsf{sign}^c = \mathsf{sign}_0^c \oplus \mathsf{sign}_1^c$ for $c = 0, 1$.
12: Denote $\mathsf{path} \leftarrow A^{(i-1)}[k]$.
13: **if** both $\mathsf{path} \| z$ for $z = 0, 1$ are in $A^{(i)}$ **then**
14: $d \leftarrow$ the index of $\mathsf{path} \| 0$ in $A^{(i)}$.
15: $CW[k] \leftarrow r \| (\Delta\mathsf{sign}^0 \oplus e_d) \| (\Delta\mathsf{sign}^1 \oplus e_{d+1})$ where $r \xleftarrow{\$} \{0,1\}^\lambda$, $e_d = 0^{d-1}10^{t-d}$.
16: **else**
17: Let z be such that $\mathsf{path} \| z \in A^{(i)}$.
18: $d \leftarrow$ the index of $\mathsf{path} \| z$ in $A^{(i)}$.
19: $CW[k] \leftarrow \begin{cases} \Delta\mathsf{seed}^1 \| (\Delta\mathsf{sign}^0 \oplus e_d) \| \Delta\mathsf{sign}^1 & z = 0 \\ \Delta\mathsf{seed}^0 \| \Delta\mathsf{sign}^0 \| (\Delta\mathsf{sign}^1 \oplus e_d) & z = 1 \end{cases}$.
20: **end if**
21: **end for**
22: **return** CW.
23: **end procedure**

24: **procedure** GenConvCW($A, B, \{\mathsf{seed}_b^{(n)}, \mathsf{sign}_b^{(n)}\}$)
25: Sample a list CW of t random $\mathbb{G}$-elements.
26: **for** $k = 1$ to $|A|$ **do**
27: $\Delta g \leftarrow G_{\mathsf{conv}}(\mathsf{seed}_0^{(n)}[k]) - G_{\mathsf{conv}}(\mathsf{seed}_1^{(n)}[k])$.
28: $CW[k] \leftarrow (-1)^{\mathsf{sign}_0^{(n)}[k][k]}(\Delta g - B[k])$.
29: **end for**
30: **return** CW.
31: **end procedure**

32: **procedure** Correct($\vec{x}, \mathsf{sign}, CW$)
33: **return** $C_{\mathsf{seed}} \| C_{\mathsf{sign}^0} \| C_{\mathsf{sign}^1} \leftarrow \sum_{i=1}^t \mathsf{sign}[i] \cdot CW[i]$, where C_{sign^0} and C_{sign^1} are t-bit.
34: **end procedure**

35: **procedure** ConvCorrect(x, sign, CW)
36: **return** $\sum_{i=1}^t \mathsf{sign}[i] \cdot CW[i]$.
37: **end procedure**

be the encoding (see line 21). When evaluating, we decode $CW^{(i)}$ using the same OKVS scheme to obtain the correction with regard to any node (see line 32).

For the convert layer, GenConvCW set $CW^{(n+1)}$ to be the encoding of (leaf node, output correction) pairs where 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 fig. 3 the OKVS scheme OKVS_i we use for the depth-i layer has key space of size 2^i and value space $\{0,1\}^\lambda$. For simplicity 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.

Yaxin: One point: the row matrix of the current layer contains the row matrix of the previous layers, which might be useful for speedup.

We informally argue that armed with an OKVS scheme that fails with negligible probabllity, the correctness of the OKVS-based DMPF holds with overwhelming probability since properties 1 and 2 are ensured, which in turn gives correct shares of outputs in the end of evaluation. The security holds as long as the OKVS scheme is oblivious. Since the corrections are pseudorandom strings that are masked by pseudorandom values determined by the other party's key, the OKVS scheme won't leak any information about the accepting paths due to its obliviousness.

The efficiency of OKVS-based DMPF relies highly on the efficiency of the OKVS scheme it uses. Setting the naïve solution as a benchmark, the ratio of keysize of the naïve solution over the OKVS-based DMPF is roughly the rate of the OKVS scheme. Similar to the advantage of the big-state DMPF, the OKVS-based DMPF also only traverse one evaluation tree (as opposed to traversing t evaluation trees in the naïve solution). However Gen consumes an OKVS encoding time per layer, and Eval and FullEval consume an OKVS decoding/batch decodings per layer. Therefore with an OKVS scheme that has high rate, fast encoding and decoding will result in an OKVS-based DMPF scheme that has small keysize, fast Gen and Eval/FullEval, respectively.

3.4 Comparison

In this section we summarize the efficiency of the DMPF instantiations we've mentioned and constructed so far. We display the keysize and running time of Gen,Eval and FullEval of different DMPF schemes, computed in terms of costs of abstract tools such as PRG, batch code and OKVS. The concrete efficiency will be discussed later in application scenarios in section 4.

Take PCG as a potential application. We care about FullEval time which is related to PCG seed expanding time. In this aspect, the CBC-based DMPF consumes d×PRGs than big-state DMPF and OKVS-based DMPF, while big-state DMPF's FullEval time scales with t and OKVS-based DMPF in addition consumes large field multiplications (in OKVS decoding, and maybe more than this). Therefore we expect different DMPF schemes to be the top choice in different choices of t and depending on the computing time of PRG and large field multiplication.

3.5 Distributed Key Generation

3.6 Distributed Multi-Inverval

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

1: Set $l \leftarrow 1$.
2: For $1 \le i \le n$, let OKVS_i be an OKVS scheme (definition 5) with key space $\mathcal{K} = \{0,1\}^{i-1}$, value space $\mathcal{V} = \{0,1\}^{\lambda+2}$ and input length t.
3: 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 t.

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

7: **procedure** GenCW($i, A, \{\mathsf{seed}_b^{(i-1)}, \mathsf{sign}_b^{(i-1)}\}_{b=0,1}$)
8: Let $\{A^{(i)}\}_{0\le i\le n}$ be defined as in fig. 1.
9: Sample a list V of t random strings from $\{0,1\}^{\lambda+2}$.
10: **for** $k = 1$ to $|A^{(i-1)}|$ **do**
11: Parse $G(\mathsf{seed}_b^{(i-1)}[k]) = \mathsf{seed}_b^0 \| \mathsf{sign}_b^0 \| \mathsf{seed}_b^1 \| \mathsf{sign}_b^1$, for $b = 0, 1$, $\mathsf{seed}_b^0, \mathsf{seed}_b^1 \in \{0,1\}^\lambda$ and $\mathsf{sign}_b^0, \mathsf{sign}_b^1 \in \{0,1\}$.
12: Compute $\Delta\mathsf{seed}^c = \mathsf{seed}_0^c \oplus \mathsf{seed}_1^c$ and $\Delta\mathsf{sign}^c = \mathsf{sign}_0^c \oplus \mathsf{sign}_1^c$ for $c = 0, 1$.
13: Denote path $\leftarrow A^{(i-1)}[k]$.
14: **if** both path$\|z$ for $z = 0, 1$ are in $A^{(i)}$ **then**
15: $V[k] \leftarrow r \| (\Delta\mathsf{sign}^0 \oplus 1) \| (\Delta\mathsf{sign}^1 \oplus 1)$, where $r \xleftarrow{\$} \{0,1\}^\lambda$.
16: **else**
17: Let z be such that path$\|z \in A^{(i)}$.
18: $V[k] \leftarrow \Delta\mathsf{seed}^1 \| \left(\Delta\mathsf{sign}^0 \oplus (1-z)\right) \| (\Delta\mathsf{sign}^1 \oplus z)$.
19: **end if**
20: **end for**
21: **return** OKVS_i.Encode($\{A^{(i-1)}[k], V[k]\}_{1\le k\le|A^{(i-1)}|}$).
22: **end procedure**

23: **procedure** GenConvCW($A, B, \{\mathsf{seed}_b^{(n)}, \mathsf{sign}_b^{(n)}\}$)
24: Sample a list V of t random $\mathbb{G}$-elements.
25: **for** $k = 1$ to $|A|$ **do**
26: $\Delta g \leftarrow G_{\mathsf{conv}}(\mathsf{seed}_0^{(n)}[k]) - G_{\mathsf{conv}}(\mathsf{seed}_1^{(n)}[k])$.
27: $V[k] \leftarrow (-1)^{\mathsf{sign}_0^{(n)}[k][k]}(\Delta g - B[k])$.
28: **end for**
29: **return** $\mathsf{OKVS}_{\mathsf{conv}}(\{A[k], V[k]\}_{1\le k\le t})$.
30: **end procedure**

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

34: **procedure** ConvCorrect(x, sign, CW)
35: **return** $\mathsf{sign} \cdot \mathsf{OKVS}_{\mathsf{conv}}$.Decode($CW, x$).
36: **end procedure**

Table 1: Keysize and running time comparison for different DMPF constructions for domain size N, t accepting points, output group $\mathbb{G}$ and computational security parameter λ. We leave this table with the abstraction of (probabilistic) batch code in the second column and the abstraction of OKVS in the last column, and plug in concrete instantiations later. m in the second column stands for the number of buckets in batch code, and w stands for the number of buckets that each input coordinate is mapped to (we only consider regular degree because this is the case in most instantiations). Yaxin: Denote T_G as the time for computing $G : \{0,1\}^{\lambda+1} \to \{0,1\}^{2\lambda+2}$, and $T_{G_{\text{conv}}}$ as the time for computing $G_{\text{conv}} : \{0,1\}^{\lambda} \to \mathbb{G}$. In the last column, denote OKVS as the OKVS scheme used for the first n layers, and $\text{OKVS}_{\text{conv}}$ as the OKVS scheme used for the convert layer.

	Sum of t DPFs	CBC-based DMPF[1, 3, 7, 16]	Big-state DMPF	OKVS-based DMPF
Keysize	$t(\lambda+2)\log N + t\log\mathbb{G}$	$m(\lambda+2)\log(wN/m) + m\log\mathbb{G}$	$t(\lambda+2t)\log N + t\log\mathbb{G}$	$\log N \times$ OKVS.Codesize $+\text{OKVS}_{\text{conv}}$.Codesize
$Gen()$	$2t\log N \times T_G + 2t \times T_{G_{\text{conv}}}$	$2m\log(wN/m) \times T_G + 2m \times T_{G_{\text{conv}}}$ CBC.Encode + CBC.Decode	$2t\log N \times T_{G^*}$ [1]	$2t\log N \times T_G + 2t \times T_{G_{\text{conv}}}$ $+\log N \times$ OKVS.Encode $+\text{OKVS}_{\text{conv}}$.Encode
$Eval()$	$t\log N \times T_G + t \times T_{G_{\text{conv}}}$	$w\log(wN/m) \times T_G + w \times T_{G_{\text{conv}}}$ Finding all positions the input is mapped to	$\log N \times T_{G^*} + T_{G_{\text{conv}}}$	$\log N \times T_G$ $+\log N \times$ OKVS.Decode $+\text{OKVS}_{\text{conv}}$.Decode
$FullEval()$	$tN \times T_G + tN \times T_{G_{\text{conv}}}$	$wN \times T_G + wN \times T_{G_{\text{conv}}}$ Finding positions that cover the full domain	$N \times T_{G^*} + N \times T_{G_{\text{conv}}}$	$N \times T_G$ $+N \times$ OKVS.Decode $+N \times \text{OKVS}_{\text{conv}}$.Decode

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

4 APPLICATIONS

In this section we compare and discuss about the efficiency of different DMPF schemes in concrete application scenarios, namely when used for constructing pseudorandom correlation generator (PCG) and unbalanced private set intersection protocol (unbalanced PSI). For convenience of discussion, we use $\text{DMPF}_{t,N,\mathbb{G}}$ to denote a DMPF scheme for t-point functions with domain $[N]$ and output group $\mathbb{G}$.

To give a rough impression, we list the number of accepting points t, the domain size N and the output group $\mathbb{G}$ of DMPF usually used in generating the PCG or unbalanced PSI protocol in table 2.

4.1 PCG for OLE from Ring-LPN

In this section we discuss the efficiency of different DMPF schemes in the PCG application. We begin by briefly introducing the protocol of PCG for OLE from Ring-LPN assumption, proposed in [4].

The PCG protocol for OLE correlation. The hardness assumption we will make use of is a variant of Ring-LPN, called module-LPN assumption.

Definition 6 (Module-LPN). *Let $c \geq 2$ be an integer, $R = \mathbb{Z}_p[X]/F(X)$ for a prime p and a deg-N polynomial $F(X) \in \mathbb{Z}_p[X]$, and $\mathcal{HW}_{R,t}$ be the uniform distribution over weight-t polynomials in R whose degree is less than N and has at most t nonzero coefficients. For $R = R(\lambda)$, $t = t(\lambda)$ and $m = m(\lambda)$, we say that the module-LPN problem R^c-LPN is hard if for every nonuniform polynomial-time probabilistic distinguisher $\mathcal{A}$, it holds that*

$$|\Pr[\mathcal{A}(\{\vec{a}^{(i)}, \langle\vec{a}^{(i)}, \vec{s}\rangle + \vec{e}^{(i)}\}_{i\in[m]})] - \Pr[\mathcal{A}(\{\vec{a}^{(i)}, \vec{u}^{(i)}\}_{i\in[m]})]| \leq \text{negl}(\lambda)$$

where the probabilities are taken over the randomness of $\mathcal{A}$, random samples $\vec{a}^{(1)}, \cdots, \vec{a}^{(m)} \leftarrow R^{c-1}$, $\vec{u}^{(1)}, \cdots, \vec{u}^{(m)} \leftarrow R$, $\vec{s} \leftarrow \mathcal{HW}^{c-1}_{R,t}$, and $\vec{e}^{(1)}, \cdots, \vec{e}^{(m)} \leftarrow \mathcal{HW}_{R,t}$.

When we only consider $m = 1$, each R^c-LPN instance $\langle\vec{a}, \vec{s}\rangle + \vec{e}$ can be restated as $\langle\vec{a'}, \vec{e'}\rangle$ where $\vec{a'} = 1||\vec{a}$ and $\vec{e'} \leftarrow \mathcal{HW}^c_{R,t}$.

The PCG protocol in [4] generates seed for the OLE correlation $(x_0, x_1, z_0, z_1) \in R^4$ such that $x_0 + x_1 = z_0 \cdot z_1$, where $R = \mathbb{Z}_p[X]/F(X)$ for a prime p and a deg-N polynomial $F(X) \in \mathbb{Z}_p[X]$. The idea is to first set $z_b = \langle\vec{a}, \vec{e}_b\rangle$ (an R^c-LPN instance with public $\vec{a}$ and $\vec{e}_b \leftarrow \mathcal{HW}^c_{R,t}$). Basing on the fact that $\langle\vec{a}, \vec{e}_0\rangle \cdot \langle\vec{a}, \vec{e}_1\rangle = \langle\vec{a}\otimes\vec{a}, \vec{e}_0\otimes\vec{e}_1\rangle$, the next step is to additively share the tensor product $\vec{e}_0 \otimes \vec{e}_1$ and each party can compute an additive share of $z_0 \cdot z_1$. Note that the tensor product $\vec{e}_0 \otimes \vec{e}_1$ consists of c^2 entries, each being an deg-$2N$ polynomial with at most t^2 nonzero coefficients. Therefore it can be shared by invoking $\text{DMPF}_{t^2,2N,\mathbb{Z}_p}$ for c^2 times.

One can compute the seed size and expanding time of this PCG protocol as follows:

- The seed size is $ct(\log N + \log p)$ bits for specifying $\vec{e}_b$ plus the $c^2\times$keysize of DMPF.
- The expanding time is c^2 multiplications in the deg-$2N$ polynomial ring plus $c^2\times$full-domain evaluation time of DMPF.

Remark 7. *Note that the above PCG protocol generates seed for OLE correlation over deg-N polynomial ring R. One can immediately convert an OLE correlation over ring R to N OLE correlations over $\mathbb{Z}_p$ if the polynomial $F(X)$ splits into N distinct linear factors modulo p[4]. Therefore we mostly consider reducible F of such form.*

A previous optimization is to substitute $\mathcal{HW}_{R,t}$ with regular weight-t polynomials denoted as regular-$\mathcal{HW}_{R,t}$. Each regular weight-t polynomial e contains exactly one nonzero coefficient e_j in the range of degree $[j\cdot(N/t), (j+1)\cdot(N/t)-1]$ for $j = 0, \cdots t-1$. When multiplying two regular weight-t polynomials e and f, $e_i \cdot f_j$

Table 2: Parameters of DMPF in concrete applications.

Concrete application	Cost in terms of DMPF per correlation/execution	Typical DMPF parameters
PCG for OLE from Ring-LPN	seedsize $\propto$ DMPF.*keysize* expand time $\propto$ DMPF.*FullEval*()	Number of accepting points: $5^2, 16^2, 76^2$ Domain size: 2^{20} Output group: $\mathbb{Z}_p$ where $\log p = 128$
PSI-WCA	communication $\propto$ DMPF.*keysize* client computation $\propto$ DMPF.*Gen*() server computation $\propto$ DMPF.*Eval*()	Number of accepting points: any Domain size: 2^{128} Output group: any

contributes to a coefficient in the range of degree $[(i+j)\cdot(2N/t), (i+j+2)\cdot(2N/t)-2]$. Therefore the deg-$2N$ polynomial $e\cdot f$ can be shared by invoking $\{\mathsf{DMPF}_{k,2N/t,\mathbb{Z}_p}\}_{k=1,2,\cdots,t-1,t,t-1,\cdots,2,1}$, which cuts down the domain size compared to the original design.

The previous literature uses sum of DPFs to achieve DMPF in either the original design with nonregular noise or the optimized design with regular noise. It indicates using batch code to achieve DMPF as another optimization but not in the clear. We'll analyze the cost of this PCG protocol under the following settings:

(1) with regular noise and each multiplication of sparse polynomials is shared by 2 sets of $\{\mathsf{DMPF}_{k,2N/t,\mathbb{Z}_p}\}_{1\le k\le t-1}$ and $\mathsf{DMPF}_{t,2N/t,\mathbb{Z}_p}$;
(2) with nonregular noise and each multiplication of sparse polynomials is shared by $\mathsf{DMPF}_{t^2,2N,\mathbb{Z}_p}$.

We'll instantiate DMPF in different ways as listed in table 1. The costs of PCG protocols under different settings are listed in table 3.

Yaxin: One caveat: can CBC-based / OKVS-based DMPF fit into the regular design, while it requires shares of 1, 2, 3-point functions?

From table 3 we can see that if we allow small blowup of the seed size of PCG, then we can gain much faster seed expansion by using nonregular noise distribution and either CBC-based DMPF or big-state DMPF or OKVS-based DMPF.

Comparing the entropy of regular and nonregular noise. Moreover, let's consider using the entropy of the noise distribution as the security parameter (which means that we consider the adversary's attack being randomly guessing the secret). Since the entropy of the nonregular noise distribution $\mathcal{HW}_{R_1,t_1}$ is $\log(\frac{N_1^{t_1}}{t_1!})$ while the entropy of the regular noise distribution regular-$\mathcal{HW}_{R_2,t_2}$ is $\log(\frac{N_2^{t_2}}{t_2!})$, to settle for the same level of security we only need $t_1 = t_2$ and $N_1 = N_2/\mathrm{e}$ where e is the natural logarithm. Placing this back to table 3, the total FullEval time (i.e., the seed expanding time of PCG) can have an extra $\times\mathrm{e}$ speedup.

Yaxin: Double check i the cost of practical attacks against R^c-LPN scales with entropy.

Next we plug in concrete parameters and evaluate the performance of different DMPF schemes under different PCG parameter settings.

Define the number of noisy coordinate $W := ct$. We set (λ, c, N, W) such that the best attack requires at least 2^λ arithmetic operations over field $\mathbb{F}_p$ of size approximately 2^{128}. According to [4], for R from an irreducible F, we lowerbound the number of arithmetic operations by $N\cdot(c\cdot\frac{N}{N-1})^W \approx N\cdot c^W$. Yaxin: TBD: Check [12] for latest update about this cost. For R from a reducible F, we lowerbound the number of arithmetic operations by $2^i\cdot c^{W_i}$, where $i := \underset{1\le i\le\log N}{\arg\min}\left(2^i\cdot c^{W_i}\right)$ and $W_i := W-cn+(c(n-1)+W)\cdot\left(1-\frac{1}{n}\right)^{W/c-1}$. In the end we round W such that $t = W/c$ is an integer.

Yaxin: Previous calculation as a reference: choosing the big-state DMPF for $t < 8$ and the OKVS-DMPF for $t \ge 8$ gives at least $\times 2$ acceleration on expand time over other choices with sacrifice on the keysize. There is a tradeoff between the CBC-based and OKVS-DMPF in that the OKVS-DMPF always provides a $\sim\times 2$ acceleration on expand time, but a loss in seed size that when t is large it may blow up the seed size to $\sim\times 2$ that of the CBC-based-DMPF.

4.2 Unbalanced PSI-WCA

A private set intersection (PSI) protocol allows two parties with input X, Y being two sets to learn about their intersection $X\cap Y$ without revealing additional information of X or Y. We denote by PSI-WCA (weighted cardinality) a variant of PSI that computes the weighted cardinality of elements in $X\cap Y$ where the weights are determined by a pre-fixed function $w(\cdot)$.

We will be interested in *unbalanced* PSI-WCA where $|X| \gg |Y|$ and the output should be received by the party holding Y. In this problem we call the party holding X as the server, and the party holding Y as the client. If further the big set X is held by two non-colluding servers, then such an unbalanced PSI-WCA protocol can be constructed from DMPF, as suggested in [8]:

- The client invokes $\mathsf{DMPF.Gen}(1^\lambda, \hat{f}_{Y,w(Y)}) \to (k_0, k_1)$, where $w(Y)$ is the set of weights of elements in Y. Then the client send k_0 to server 0 and k_1 to server 1.
- Server b computes $s_b = \sum_{x\in X}\mathsf{DMPF.Eval}_b(1^\lambda, k_b, x)$ and send it back to the client.
- The client computes $s_0 + s_1$, which will be the weighted cardinality of $X\cap Y$.

One caveat is that this protocol reveals information about Y that is leaked by DMPF. Plugging in any DMPF instantiations we have mentioned, the size of $|Y|$ will be leaked to the servers.

The cost of our unbalanced PSI-WCA can be computed as follows:

- The communication cost equals the keysize of DMPF.
- The client computation time equals the key generation time of DMPF.
- The server computation time equals $|X|\times$the evaluation time of DMPF.

Table 3: Seed size and expanding time of PCG protocols for the same (λ, N, c, t) with different choices of noise distributions in module-LPN assumption, and with different DMPF instantiations. We use construction 5 as an instantiation of OKVS. The seed size is represented by total DMPF share size and the expanding time is represented by total DMPF.FullEval time. The PRG evaluations in the first $\log(2N)$ layers and in the convert layer are both regarded as the same PRG. $e = m/t$ in the second row represents the expansion parameter for PBC where m is the number of buckets, and e' in the last row represents the expansion parameter (the inverse of rate) for OKVS.

DMPF instantiation	Noise type	Total share size	Total FullEval time (only listed PRG and OKVS)
Sum of DPFs	regular	$c^2t^2\lambda\log(2N/t) + c^2t^2\log p$	$4c^2tN\times$PRG
	nonregular	$c^2t^2\lambda\log(2N) + c^2t^2\log p$	$4c^2t^2N\times$PRG
Batch-code DMPF	regular	$ec^2t^2\lambda\log(\frac{wN}{et}) + ec^2t^2\log p$	$8c^2wN\times$PRG
	nonregular	$ec^2t^2\lambda\log(\frac{2wN}{et^2}) + ec^2t^2\log p$	$4c^2wN\times$PRG
Big-state DMPF	regular	$c^2t^2(\lambda+\frac{4}{3}t)\log(2N) + c^2t^2\log p$	$8c^2N\times$PRG*[1]
	nonregular	$c^2t^2(\lambda+2t)\log(2N) + c^2t^2\log p$	$4c^2N\times$PRG*
OKVS-based DMPF	regular	$e'c^2t^2\lambda\log(2N/t) + e'c^2t^2\log p$	$8c^2N\times$PRG+$8c^2N\times$ OKVS.Decode
	nonregular	$e'c^2t^2\lambda\log(2N) + e'c^2t^2\log p$	$4c^2N\times$PRG+$4c^2N\times$ OKVS.Decode

[1] The PRG used in big-state DMPF maps from $\{0,1\}^\lambda$ to $\{0,1\}^{2\lambda+2t^2}$ whose computation time should grow with t^2.

We'll instantiate DMPF in different ways as listed in table 1. As suggested in [8], we take an infeasibly large domain for the sets X and Y to locate, whose size is $N = 2^{128}$. The set sizes $|X|$ and $|Y|$ can vary depending on application scenarios. Since $|Y|$ is the crucial factor that distinguishes different DMPF instantiations, we will only consider the change of $|Y|$. The costs of PSI-WCA protocols under different settings of $|Y|$ are listed in table 4.

Yaxin: Previous calculation as a reference: A short conclusion is using big-state DMPF for $t < 64$ and the OKVS-DMPF for $t \geq 64$ gives at $\sim \times 2$ faster Eval() time and faster Gen() time compared to the naive and CBC-based construction. The keysize ($\propto$ communication complexity) of our choice is usually smaller than the CBC-based DMPF and slightly larger than the naive construction.

4.3 Security analysis

4.4 Heavy-hitters

private heavy-hitter
or parallel ORAM?

5 ACKNOWLEDGMENTS

tbd

REFERENCES

[1] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. 2017. PIR with compressed queries and amortized query processing. Cryptology ePrint Archive, Paper 2017/1142. https://eprint.iacr.org/2017/1142 https://eprint.iacr.org/2017/1142.

[2] Alexander Bienstock, Sarvar Patel, Joon Young Seo, and Kevin Yeo. 2023. Near-Optimal Oblivious Key-Value Stores for Efficient PSI, PSU and Volume-Hiding Multi-Maps. Cryptology ePrint Archive, Paper 2023/903. https://eprint.iacr.org/2023/903 https://eprint.iacr.org/2023/903.

[3] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. 2019. Compressing Vector OLE. Cryptology ePrint Archive, Paper 2019/273. https://doi.org/10.1145/3243734.3243868 https://eprint.iacr.org/2019/273.

[4] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. 2022. Efficient Pseudorandom Correlation Generators from Ring-LPN. Cryptology ePrint Archive, Paper 2022/1035. https://doi.org/10.1007/978-3-030-56880-1_14 https://eprint.iacr.org/2022/1035.

[5] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2018. Function Secret Sharing: Improvements and Extensions. Cryptology ePrint Archive, Paper 2018/707. https://eprint.iacr.org/2018/707 https://eprint.iacr.org/2018/707.

[6] Hao Chen, Kim Laine, and Peter Rindal. 2017. Fast Private Set Intersection from Homomorphic Encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1243–1255. https://doi.org/10.1145/3133956.3134061

[7] Leo de Castro and Antigoni Polychroniadou. 2021. Lightweight, Maliciously Secure Verifiable Function Secret Sharing. Cryptology ePrint Archive, Paper 2021/580. https://eprint.iacr.org/2021/580 https://eprint.iacr.org/2021/580.

[8] Samuel Dittmer, Yuval Ishai, Steve Lu, Rafail Ostrovsky, Mohamed Elsabagh, Nikolaos Kiourtis, Brian Schulte, and Angelos Stavrou. 2020. Function Secret Sharing for PSI-CA: With Applications to Private Contact Tracing. Cryptology ePrint Archive, Paper 2020/1599. https://eprint.iacr.org/2020/1599 https://eprint.iacr.org/2020/1599.

[9] Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 2021. Oblivious Key-Value Stores and Amplification for Private Set Intersection. Cryptology ePrint Archive, Paper 2021/883. https://eprint.iacr.org/2021/883 https://eprint.iacr.org/2021/883.

[10] Niv Gilboa and Yuval Ishai. 2014. Distributed Point Functions and Their Applications. In *Advances in Cryptology – EUROCRYPT 2014*, Phong Q. Nguyen and Elisabeth Oswald (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 640–658.

[11] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2004. Batch Codes and Their Applications. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing* (Chicago, IL, USA) *(STOC '04)*. Association for Computing Machinery, New York, NY, USA, 262–271. https://doi.org/10.1145/1007352.1007396

[12] Hanlin Liu, Xiao Wang, Kang Yang, and Yu Yu. 2022. The Hardness of LPN over Any Integer Ring and Field for PCG Applications. Cryptology ePrint Archive, Paper 2022/712. https://eprint.iacr.org/2022/712 https://eprint.iacr.org/2022/712.

[13] Rasmus Pagh and Flemming Friche Rodler. 2001. Cuckoo Hashing. In *Algorithms — ESA 2001*, Friedhelm Meyer auf der Heide (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 121–133.

[14] M. B. Paterson, D. R. Stinson, and R. Wei. 2008. Combinatorial batch codes. Cryptology ePrint Archive, Paper 2008/306. https://eprint.iacr.org/2008/306 https://eprint.iacr.org/2008/306.

[15] Srinivasan Raghuraman and Peter Rindal. 2022. Blazing Fast PSI from Improved OKVS and Subfield VOLE. Cryptology ePrint Archive, Paper 2022/320. https:

Table 4: Communication cost, client and server computation time of the PSI-WCA protocol for domain size $N = 2^{128}$, weight group $\mathbb{G}$, and different choices of client's set size $|Y|$. We use construction 5 as an instantiation of OKVS. The PRG evaluations in the first $\log N$ layers and in the convert layer are both regarded as the same PRG. e in the second row represents the expansion parameter for PBC, and e' in the last row represents the expansion parameter for OKVS.

DMPF instantiation	Communication cost	Client computation time	Server computation time
Sum of DPFs	$\|Y\|\lambda \log N + \|Y\| \log \|\mathbb{G}\|$	$2\|Y\| \log N \times$PRG	$\|X\| \cdot \|Y\| \log N \times$PRG
Batch-code DMPF	$e\|Y\|\lambda \log(\frac{wN}{e\|Y\|}) + e\|Y\| \log \|\mathbb{G}\|$	$2e\|Y\| \log(\frac{wN}{e\|Y\|}) \times$PRG	$w\|X\| \log(\frac{wN}{e\|Y\|}) \times$PRG
Big-state DMPF	$\|Y\|(\lambda + 2\|Y\|) \log N + \|Y\| \log \|\mathbb{G}\|$	$2\|Y\| \log N \times$PRG^{*1}	$\|X\| \log N \times$PRG*
OKVS-based DMPF	$e'\|Y\|\lambda \log N + e'\|Y\| \log \|\mathbb{G}\|$	$2\|Y\| \log N \times$PRG+$\log N \times$ OKVS.Encode	$\|X\|(\log N \times$PRG+$\log N \times$ OKVS.Decode)

[1] The PRG used in big-state DMPF maps from $\{0,1\}^\lambda$ to $\{0,1\}^{2\lambda+2|Y|}$ whose computation time should grow with $|Y|$.

//eprint.iacr.org/2022/320 https://eprint.iacr.org/2022/320.

[16] Phillipp Schoppmann, Adrià Gascón, Leonie Reichert, and Mariana Raykova. 2019. Distributed Vector-OLE: Improved Constructions and Implementation. Cryptology ePrint Archive, Paper 2019/1084. https://doi.org/10.1145/3319535.3363228 https://eprint.iacr.org/2019/1084.

[17] Kevin Yeo. 2022. Cuckoo Hashing in Cryptography: Optimal Parameters, Robustness and Applications. Cryptology ePrint Archive, Paper 2022/1455. https://eprint.iacr.org/2022/1455 https://eprint.iacr.org/2022/1455.

A SECURITY PROOFS