

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 email (Conference acronym 'XX)*. ACM, New York, NY, USA, 9 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* $f_{\alpha,\beta} : [N] \rightarrow \mathbb{G}$ 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* $f_{A,B} : [N] \rightarrow \mathbb{G}$ for $A = (\alpha_1, \dots, \alpha_t) \in [N]^t$ and $B = (\beta_1, \dots, \beta_t) \in \mathbb{G}^t$ evaluates to β_i on input α_i for $1 \leq i \leq t$ and to 0 on all other inputs. 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 [4, 8]). A (2-party) distributed point function (DPF) is a triple of algorithms $\Pi = (\text{Gen}, \text{Eval}_0, \text{Eval}_1)$ with the following syntax:

- $\text{Gen}(1^\lambda, \hat{f}_{\alpha,\beta}) \rightarrow (k_0, k_1)$: 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference acronym 'XX, tbd, tbd

© tbd Association for Computing Machinery.

ACM ISBN tbd...\$15.00

<https://doi.org/tbd>

procedures, i.e., $|k_b| \in \text{poly}(\lambda)$. Stress it here or add efficiency requirement? We assume that N and \mathbb{G} are determined by each key.

- $\text{Eval}_b(k_b, x) \rightarrow y_b$: 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 $b = 0, 1$,

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

- **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 $(k_0, k_1) \leftarrow \text{Gen}(1^\lambda, \hat{f}^b)$.
- (3) The adversary outputs a guess $b' \leftarrow \mathcal{A}(k_b)$.

Denote by $\text{Adv}(1^\lambda, \mathcal{A}, i) = \Pr[b = b'] - 1/2$ 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 v such that $\text{Adv}(1^\lambda, \mathcal{A}, i) \leq v(\lambda)$ for all $\lambda \in \mathbb{N}$.

DEFINITION 2 (DMPF). A (2-party) distributed multi-point function (DMPF) is a triple of algorithms $\Pi = (\text{Gen}, \text{Eval}_0, \text{Eval}_1)$ with the following syntax:

- $\text{Gen}(1^\lambda, \hat{f}_{A,B}) \rightarrow (k_0, k_1)$: 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 $|k_i| = \text{poly}(\lambda, t)$.
- $\text{Eval}_b(k_b, x) \rightarrow y_b$: 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 $b = 0, 1$,

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

- **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 $(k_0, k_1) \leftarrow \text{Gen}(1^\lambda, \hat{f}^b)$.
- (3) The adversary outputs a guess $b' \leftarrow \mathcal{A}(k_b)$.

Denote by $\text{Adv}(1^\lambda, \mathcal{A}, i) = \Pr[b = b'] - 1/2$ 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 v such that $\text{Adv}(1^\lambda, \mathcal{A}, i) \leq v(\lambda)$ for all $\lambda \in \mathbb{N}$.

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

2.3 Batch Code

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

DEFINITION 3 (BATCH CODE[9]). An (N, M, t, m) -batch code over alphabet Σ is given by a pair of efficient algorithms $(\text{Encode}, \text{Decode})$ such that:

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

An (N, M, t, m) -batch code can be represented by an (N, m) -bipartite graph $G = (U, V, E)$ where each edge $(u_i, v_j) \in E$ corresponds to Encode assigning $x[i]$ to the bucket C_j , while it is guaranteed that any subset $S \subseteq U$ such that $|S| = t$ has a perfect matching to V . **Yaxin: Add example instantiation (random regular bipartite graph) and explain it is not efficient.**

DEFINITION 4 (PROBABILISTIC BATCH CODE (PBC)[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 $[N]$,

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

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

We mention Cuckoo hashing algorithm[10] as a concrete instantiation of PBC[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, \dots, 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* whenever the algorithm fails to output a proper allocation where each bucket contains at most one ball.

Yaxin: Add asymptotic parameters? Also find evident theorem saying cuckoo hashing is efficient.

The failure probability of cuckoo hashing. Let's denote the failure probability of w -way cuckoo hashing to be $\epsilon = 2^{-\lambda_{\text{stat}}}$. In practice we usually consider the statistical security parameter λ_{stat} to be 30 or 40. The empirical result in [5] shows for $w = 3$, $m = 16384$, $\lambda_{\text{stat}} = 124.4e - 144.6$ where e is the expansion parameter that $m = et$. For $w = 3$, $m = 8192$, $\lambda_{\text{stat}} = 125e - 145$. 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 $\lambda_{\text{stat}} \approx 40$ and claims it follows the analysis from [5], but I don't see how...**

The balls, buckets and hash functions can be represented by a w -regular (t, m) -bipartite graph $G = (U, V, E)$ where each left node has w neighbors, and each edge $(u_i, v_j) \in E$ corresponds to $h_l(i) = j$ for some $1 \leq l \leq w$. In this graph representation the w -way cuckoo hashing essentially computes a perfect matching from U to V . Therefore one can construct a PBC from cuckoo hashing.

CONSTRUCTION 1 (PBC 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) -PBC is as follows:

- $\text{Encode}_r(x \in \Sigma^N) \rightarrow (C_1, \dots, C_m)$: Use r to determine w independent random hash functions h_1, h_2, \dots, h_w that maps from $[N]$ to $[m]$. Initialize C_1, \dots, C_m to be empty. For each $i \in [N]$, append $x[i]$ to $C_{h_j(i)}$ for $1 \leq j \leq w$.
- $\text{Decode}_r(I, C_1, \dots, C_m) \rightarrow \{x[i]\}_{i \in I}$: Determine h_1, \dots, h_w as in Encode . For I of size t , allocate I to $[m]$ using w -way cuckoo hashing. For each $i \in I$, fetch $x[i]$ from C_j where i is allocated to j .

2.4 Oblivious Key-Value Stores

DEFINITION 5 (OBLIVIOUS KEY-VALUE STORES (OKVS)[7, 11]). An Oblivious Key-Value Stores scheme is a pair of randomized algorithms $(\text{Encode}_r, \text{Decode}_r)$ with respect to a statistical security parameter λ_{stat} and a computational security parameter λ , a randomness space $\{0, 1\}^K$, a key space \mathcal{K} , a value space \mathcal{V} , input length t and output length m . The algorithms are of the following syntax:

- $\text{Encode}_r(\{(k_1, v_1), (k_2, v_2), \dots, (k_t, v_t)\}) \rightarrow P$: On input t key-value pairs with distinct keys, the encode algorithm with randomness r in the randomness space outputs an encoding $P \in \mathcal{V}^m \cup \perp$.
- $\text{Decode}_r(P, k) \rightarrow v$: 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 $S \in (\mathcal{K} \times \mathcal{V})^t$, $\Pr_{r \leftarrow \{0, 1\}^K} [\text{Encode}_r(S) = \perp] \leq 2^{-\lambda_{\text{stat}}}$.

- For all $S \in (\mathcal{K} \times \mathcal{V})^t$ and $r \in \{0, 1\}^\kappa$ such that $\text{Encode}_r(S) \rightarrow P \neq \perp$, it is the case that $\text{Decode}_r(P, k) \rightarrow v$ whenever $(k, v) \in S$.
- **Obliviousness:** Given any distinct key sets $\{k_1^0, k_2^0, \dots, k_t^0\}$ and $\{k_1^1, k_2^1, \dots, k_t^1\}$ that are different, if they are paired with random values then their encodings are computationally indistinguishable, i.e.,

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

One can obtain a linear OKVS if in addition require:

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

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

We evaluate an OKVS scheme by its encoding size (output length m), encoding time and decoding time. We stress the following two (linear) OKVS constructions:

CONSTRUCTION 2 (POLYNOMIAL). Suppose $\mathcal{K} = \mathcal{V} = \mathbb{F}$ is a field. Set

- $\text{Encode}(\{(k_i, v_i)\}_{1 \leq i \leq t}) \rightarrow P$ where P is the coefficients of a $(t-1)$ -degree \mathbb{F} -polynomial g_P that $g_P(k_i) = v_i$ for $1 \leq i \leq t$.
- $\text{Decode}(P, k) \rightarrow g_P(k)$.

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) $O(\log^2 t)$.

An alternative construction that has near optimal encoding size but much better running time is as follows.

CONSTRUCTION 3 (3-HASH GARBLED CUCKOO TABLE (3H-GCT) [7, 11]). Suppose $\mathcal{V} = \mathbb{F}$ is a field. Set $\text{row}_r(k) := \text{row}_r^{\text{sparse}}(k) \parallel \text{row}_r^{\text{dense}}(k)$ where $\text{row}_r^{\text{sparse}}$ outputs a uniformly random weight- w vector in $\{0, 1\}^{m_1}$, and $\text{row}_r^{\text{dense}}(k)$ outputs a short dense vector in \mathbb{F}^{m_2} .

- $\text{Encode}(\{(k_i, v_i)\}_{1 \leq i \leq t}) \rightarrow P$ where P is solved from the system $\{\langle \text{row}_r(k_i), P \rangle = v_i\}_{1 \leq i \leq t}$ using the triangulation algorithm in [11].
- $\text{Decode}(P, k) \rightarrow \langle \text{row}_r(k), P \rangle$.

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 a linear encoding time, constant decoding time (the constant relates to w and m_2) while having a linear encoding size.

Yaxin: TBD: Carefully(!) recompute the comparison table for OKVS.

We'll mostly use 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) according to the analysis in [11]: Given w, t and λ_{stat} , the choices of the e and \hat{g} are fixed through the following steps:

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

Yaxin: Fix t and λ_{stat} , we want to find the best choice of w . The advantageous choices of w in [11] 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 [11] 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 [7, 11] 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 also let $\text{row}_r^{\text{dense}}$ output a short dense vector in $\{0, 1\}^{m_2}$, which avoids multiplication of large field elements in the encoding and decoding processes. To achieve same level of security one could simply set $m_2 = \hat{g} + \lambda_{\text{stat}}$, as proposed in [7, 11]. **Yaxin: TBD: mention some connections to cuckoo hashing?**

3 NEW DMPF CONSTRUCTIONS

In this section, we display two new constructions of DMPF that follow the same paradigm shown in fig. 1.

We begin by introducing the DMPF paradigm in fig. 1, which is based on the idea of the DPF construction in [4]. Each key k_b ($b = 0, 1$) generated by $\text{Gen}(\hat{f}_{A,B})$ can span a height- n (n is the input length of $\hat{f}_{A,B}$) complete binary tree T_b (call it the evaluation tree), with which party b can evaluate the input $x = x_1 \cdots x_n$ by starting from the root of this tree, on the i th layer going left if $x_i = 0$ and going right if $x_i = 1$, until reaching a leaf node then computing the result according to this leaf node.

Each node of this tree is associated with a λ -bit seed and a l -bit sign. For a parent node on the i th layer with seed and sign, its children's seeds and signs are generated by $\text{PRG}(\text{seed}) \oplus \text{Correction}$, where the Correction is determined by the parent node's position, its sign and a correction word $CW^{(i)}$ associated with that layer (computed by the method $\text{Correct}()$). On a leaf node on the last layer, its seed will generate a random element in the output group, which will be corrected by adding a Correction determined by the leaf node's position, its sign and the last correction word $CW^{(n+1)}$ (computed by the method $\text{ConvCorrect}()$).

Call any path from the root a leaf corresponding to an input string in A an accepting path. To force the correctness, we maintain the following invariance on the evaluation trees T_0, T_1 of the two parties:

- If a node is not on any accepting path, then T_0 and T_1 assign to it with the same seed and sign.

- If a node is on an accepting path, then T_0 and T_1 assign to it with different signs that controls the corrections on its children (or on the output if the node is on the last layer).

The paradigm contains four methods (GenCW, GenConvCW, Correct, ConvCorrect) and the sign length l to be determined by different constructions. We make the following restrictions on the methods in order to guarantee the invariance on the evaluation trees:

$M(\bar{x}, \text{sign}, CW) = \sum_{i=1}^l \text{sign}[i] \cdot M(\bar{x}, 0^{i-1}10^{l-i}, CW)$ for all $M \in \{\text{Correct}, \text{ConvCorrect}\}$, input \bar{x} and CW .

3.1 Big-State DMPF

Displayed in fig. 2. TBD: explain

3.2 Batch-Code DMPF

We display the construction of DMPF from black-box usage of DPF basing on PBC with appropriate parameters, which has been discussed in previous literature[2, 6].

CONSTRUCTION 4 (DMPF FROM DPF). *Given DPF for any domain of size no larger than N and output group \mathbb{G} , and an (N, M, t, m, ϵ) -PBC 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:*

- $\text{Gen}(1^\lambda, \hat{f}_{A,B}) \rightarrow (k_0, k_1)$: Suppose $A = \{\alpha_1, \dots, \alpha_t\}$ and $B = \{\beta_1, \dots, \beta_t\}$. Let $TT \in \mathbb{G}^N$ be the truth table of $\hat{f}_{A,B}$. Compute $\text{Encode}(TT) \rightarrow (C_1, \dots, C_m)$ according to the PBC. Then run $\text{Decode}(A, C_1, \dots, C_m)$ to determine a perfect matching from A to $\{C_1, \dots, C_m\}$. For $1 \leq i \leq m$, let $f_i : [|C_i|] \rightarrow \mathbb{G}$ 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 l th position of C_i , then set f_i to be the point function that outputs β_j on l and 0 elsewhere.
 For $1 \leq i \leq m$, invoke $\text{DPF.Gen}(1^\lambda, f_i) \rightarrow (k_0^i, k_1^i)$. 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.
- $\text{Eval}_b(k_b, x) \rightarrow y_b$: Follow $\text{Encode}(TT)$ to determine the positions $l_{j_1}, l_{j_2}, \dots, l_{j_s}$ such that the x th entry of TT is sent to the l_{j_i} -th position of C_{j_i} . Compute $y_b = \sum_{i=1}^s \text{DPF.Eval}_b(k_b^{j_i}, l_i)$.

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

Note that if one use batch code instead of PBC then the DMPF scheme perfectly correct and secure. When instantiating PBC from w -way cuckoo hashing, the *key generation time* is roughly the time needed for computing cuckoo hashing algorithm plus the total time of all $\text{DPF.Gen}(1^\lambda, f_i)$. The *evaluation time* is roughly the total time of all $\text{DPF.Eval}_b(k_b^{j_i}, l_i)$. Similarly, the *full-domain evaluation time* is roughly the total time of all $\text{DPF.FullEval}_b(k_b^j)$ for $j = 1, \dots, m$.

3.3 OKVS-based DMPF

Displayed in fig. 3. TBD: explain

Figure 1: The paradigm of our DMPF schemes. We leave the PRG expand 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 $G : \{0, 1\}^\lambda \rightarrow \{0, 1\}^{2\lambda+2l}$. Parse $G(x) = G_0(x) \| G_1(x)$ to the left half and right half.

Suppose there is a public PRG $G_{\text{convert}} : \{0, 1\}^\lambda \rightarrow \mathbb{G}$.

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

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

For $0 \leq i \leq n-1$, let $A^{(i)}$ denote the sorted and deduplicated list of i -bit prefixes of strings in A . Specifically, $A^{(0)} = [\epsilon]$.

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

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

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

$CW^{(i)} \leftarrow \text{GenCW}(i, A, \{\text{seed}_b^{(i-1)}, \text{sign}_b^{(i-1)}\}_{b=0,1})$.

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

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

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

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

end if

end for

end for

$CW^{(n+1)} \leftarrow \text{GenConvCW}(A, B, \{\text{seed}_b^{(n)}, \text{sign}_b^{(n)}\}_{b=0,1})$.

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

return (k_0, k_1) .

end procedure

procedure $\text{EVAL}_b(1^\lambda, k_b, x)$

Parse $k_b = ([\text{seed}], [\text{sign}], CW^{(1)}, CW^{(2)}, \dots, CW^{(n+1)})$.

Denote $x = x_1 x_2 \dots x_n$.

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

$C_{\text{seed}} \| C_{\text{sign}^0} \| C_{\text{sign}^1} \leftarrow \text{Correct}(x_1 \dots x_{i-1}, \text{sign}, CW^{(i)})$.

$\text{seed} \| \text{sign} \leftarrow G_{x_i}(\text{seed})$.

$\text{seed} \| \text{sign} \leftarrow G_{x_i}(\text{seed}) \oplus (C_{\text{seed}} \| C_{\text{sign}^{x_i}})$.

end for

return $(-1)^b \cdot (G_{\text{convert}}(\text{seed}) + \text{ConvCorrect}(x, \text{sign}, CW^{(n+1)}))$.

end procedure

procedure $\text{FULEVAL}_b(1^\lambda, k_b)$

Parse $k_b = (\text{seed}^{(0)}, \text{sign}^{(0)}, CW^{(1)}, CW^{(2)}, \dots, CW^{(n+1)})$.

For $1 \leq i \leq n$, $\text{Path}^{(i)} \leftarrow$ the lexicographical ordered list of $\{0, 1\}^i$. $\text{Path}^{(0)} \leftarrow [\epsilon]$.

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

for $k = 1$ to 2^{i-1} **do**

$C_{\text{seed}} \| C_{\text{sign}^0} \| C_{\text{sign}^1} \leftarrow \text{Correct}(\text{Path}^{(i-1)}[k], \text{sign}^{(i-1)}[k], CW^{(i)})$.

$\text{seed}^{(i)}[2k] \| \text{sign}^{(i)}[2k] \leftarrow G_0(\text{seed}^{(i-1)}[k]) \oplus (C_{\text{seed}} \| C_{\text{sign}^0})$.

$\text{seed}^{(i)}[2k+1] \| \text{sign}^{(i)}[2k+1] \leftarrow G_1(\text{seed}^{(i-1)}[k]) \oplus (C_{\text{seed}} \| C_{\text{sign}^1})$.

end for

end for

for $k = 1$ to 2^n **do**

$\text{Output}[k] \leftarrow (-1)^b \cdot (G_{\text{convert}}(\text{seed}^{(n)}[k]) + \text{ConvCorrect}(\text{Path}[k], \text{sign}^{(n)}[k], CW^{(n+1)}))$.

end for

return Output .

end procedure

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

```

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

procedure GENCW( $i, A, \{\text{seed}_b^{(i-1)}, \text{sign}_b^{(i-1)}\}_{b=0,1}$ )
    Let  $\{A^{(i)}\}_{0 \leq i \leq n}$  be defined as in fig. 1.
    Sample a list  $CW$  of  $t$  random strings from  $\{0, 1\}^{\lambda+2t}$ .
    for  $k = 1$  to  $|A^{(i-1)}|$  do
        Parse  $G(\text{seed}_b^{(i-1)}[k]) = \text{seed}_b^0 \| \text{sign}_b^0 \| \text{seed}_b^1 \| \text{sign}_b^1$ , for
         $b = 0, 1$ ,  $\text{seed}_b^0, \text{seed}_b^1 \in \{0, 1\}^\lambda$  and  $\text{sign}_b^0, \text{sign}_b^1 \in \{0, 1\}^t$ .
        Compute  $\Delta \text{seed}^c = \text{seed}_0^c \oplus \text{seed}_1^c$  and  $\Delta \text{sign}^c = \text{sign}_0^c \oplus \text{sign}_1^c$  for  $c = 0, 1$ .
        Denote  $\text{path} \leftarrow A^{(i-1)}[k]$ .
        if both  $\text{path} \| z$  for  $z = 0, 1$  are in  $A^{(i)}$  then
             $d \leftarrow$  the index of  $\text{path} \| 0$  in  $A^{(i)}$ .
             $CW[d] \leftarrow r \| \Delta \text{sign}^0 \oplus e_d \| \Delta \text{sign}^1 \oplus e_{d+1}$  where  $r \xleftarrow{\$} \{0, 1\}^\lambda$ ,  $e_d = 0^{d-1} 1 0^{t-d}$ .
        else
            Let  $z$  be such that  $\text{path} \| z \in A^{(i)}$ .
             $d \leftarrow$  the index of  $\text{path} \| z$  in  $A^{(i)}$ .
             $CW[d] \leftarrow \begin{cases} \Delta \text{seed}^1 \| \Delta \text{sign}^0 \oplus e_d \| \Delta \text{sign}^1 & z = 0 \\ \Delta \text{seed}^0 \| \Delta \text{sign}^0 \| \Delta \text{sign}^1 \oplus e_d & z = 1 \end{cases}$ .
        end if
    end for
    return  $CW$ .
end procedure

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

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

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

```

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

```

Set  $l \leftarrow 1$ .
For  $1 \leq i \leq n$ , let  $\text{OKVS}_i$  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  $t$ .
let  $\text{OKVS}_{\text{convert}}$  be an OKVS scheme with key space  $\mathcal{K} = \{0, 1\}^n$ ,
value space  $\mathcal{V} = \mathbb{G}$  and input length  $t$ .

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

procedure GENCW( $i, A, \{\text{seed}_b^{(i-1)}, \text{sign}_b^{(i-1)}\}_{b=0,1}$ )
    Let  $\{A^{(i)}\}_{0 \leq i \leq n}$  be defined as in fig. 1.
    Sample a list  $V$  of  $t$  random strings from  $\{0, 1\}^{\lambda+2}$ .
    for  $k = 1$  to  $|A^{(i-1)}|$  do
        Parse  $G(\text{seed}_b^{(i-1)}[k]) = \text{seed}_b^0 \| \text{sign}_b^0 \| \text{seed}_b^1 \| \text{sign}_b^1$ , for
         $b = 0, 1$ ,  $\text{seed}_b^0, \text{seed}_b^1 \in \{0, 1\}^\lambda$  and  $\text{sign}_b^0, \text{sign}_b^1 \in \{0, 1\}$ .
        Compute  $\Delta \text{seed}^c = \text{seed}_0^c \oplus \text{seed}_1^c$  and  $\Delta \text{sign}^c = \text{sign}_0^c \oplus \text{sign}_1^c$  for  $c = 0, 1$ .
        Denote  $\text{path} \leftarrow A^{(i-1)}[k]$ .
        if both  $\text{path} \| z$  for  $z = 0, 1$  are in  $A^{(i)}$  then
             $V[k] \leftarrow r \| \Delta \text{sign}^0 \oplus 1 \| \Delta \text{sign}^1 \oplus 1$ , where  $r \xleftarrow{\$} \{0, 1\}^\lambda$ .
        else
            Let  $z$  be such that  $\text{path} \| z \in A^{(i)}$ .
             $V[k] \leftarrow \Delta \text{seed}^1 \| \Delta \text{sign}^0 \oplus (1 - z) \| \Delta \text{sign}^1 \oplus z$ .
        end if
    end for
    return  $\text{OKVS}_i.\text{Encode}(\{A^{(i-1)}[k], V[k]\}_{1 \leq k \leq |A^{(i-1)}|})$ .
end procedure

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

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

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

```


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

3.4 Comparison

Table 1 displays the keysize, running time of Gen, Eval and FullEval for different DMPF schemes, computed in terms of costs of abstract tools such as PRG, batch code and OKVS. Yaxin: We can plug in the actual costs of these tools after carrying out a complete experiment. Yaxin: Note that the PRG in big-state DMPF maps from $\{0, 1\}^\lambda$ to $\{0, 1\}^{2\lambda+2t}$.

Yaxin: Take PCG as a potential application. We care about FullEval time which is related to PCG seed expanding time. In this aspect, the batch code DMPF consumes $d \times$ 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

4 APPLICATIONS

Yaxin: This section is highly incomplete... Each subsection contains some to do list and I'll do more research.

For convenience of discussion DMPF applications, 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} .

4.1 PCG for OLE from Ring-LPN

We begin by briefly introducing the protocol of PCG for OLE from Ring-LPN assumption, proposed in [3].

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 $\mathcal{H}\mathcal{W}_{R,t}$ 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 $\vec{a}^{(1)}, \dots, \vec{a}^{(m)} \leftarrow R^{c-1}$, $\vec{u}^{(1)}, \dots, \vec{u}^{(m)} \leftarrow R$, $\vec{s} \leftarrow \mathcal{H}\mathcal{W}_{R,t}^{c-1}$, and $\vec{e}^{(1)}, \dots, \vec{e}^{(m)} \leftarrow \mathcal{H}\mathcal{W}_{R,t}$.

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 $\vec{e}' \leftarrow \mathcal{H}\mathcal{W}_{R,t}^c$.

The PCG protocol in [3] generates seed for the OLE correlation $(x_0, x_1, z_0, z_1) \in R^4$ such that $x_0 + x_1 = z_0 \cdot z_1$. 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 $\vec{e}_b \leftarrow \mathcal{H}\mathcal{W}_{R,t}^c$). 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 $c^2 N$ multiplications in R plus $c^2 \times$ full-domain evaluation time of DMPF.

REMARK 7. Note that the above PCG protocol generates seed for OLE correlation over 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 . Therefore we mostly consider reducible F of such form.

A previous optimization is to substitute $\mathcal{H}\mathcal{W}_{R,t}$ with regular weight- t polynomials denoted as regular- $\mathcal{H}\mathcal{W}_{R,t}$. 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 $j = 0, \dots, t-1$. When multiplying two regular weight- t polynomials e and f , $e_i \cdot f_j$ contributes to a coefficient in the range of degree $[(i+j) \cdot (2N/t), (i+$

Table 1: Keysize and running time comparison for different DMPF constructions for domain size N , t accepting points and computational security parameter λ . The convert layer is ignored for now. 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 used 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).

	Sum of t DPFs	Batch code DMPF[1, 2, 6, 12]	Big-state DMPF	OKVS-based DMPF
Keysize	$t(\lambda + 2) \log N$	$m\lambda \log(wN/m)$	$t(\lambda + 2t) \log N$	$\log N \times \text{OKVS code size}$
$Gen()$	$2t \log N \times \text{PRG}$	$2m \log(wN/m) \times \text{PRG}$ Finding a matching of t inputs to m buckets	$2t \log N \times \text{PRG}$	$2t \log N \times \text{PRG}$, $\log N \times \text{OKVS.Encode}$
$Eval()$	$t \log N \times \text{PRG}$	$w \log(wN/m) \times \text{PRG}$ Finding all buckets an input is mapped to	$\log N \times \text{PRG}$	$\log N \times \text{PRG}$, $\log N \times \text{OKVS.Decode}$
$FullEval()$	$tN \times \text{PRG}$	$wN \times \text{PRG}$ Finding the input sequence in every bucket	$N \times \text{PRG}$	$N \times \text{PRG}$, $N \times \text{OKVS.Decode}$

Table 2: Concrete applications of DMPF.

Concrete application	Cost in terms of DMPF per correlation/execution	Typical DMPF parameters
PCG for OLE from Ring-LPN	seedsize $\propto \text{DMPF.keysizesize}$ expand time $\propto \text{DMPF.FullEval}()$ communication $\propto \text{DMPF.keysizesize}$	$t = 5^2, 16^2, 76^2$ $N = 2^{20}$
PSI-WCA	client computation $\propto \text{DMPF.Gen}()$ server computation $\propto \text{DMPF.Eval}()$	$t = \text{any}$ $N = 2^{128}$

$j + 2) \cdot (2N/t) - 2]$. Therefore the deg- $2N$ polynomial $e \cdot f$ can be shared by invoking $\{\text{DMPF}_{k,2N/t,\mathbb{Z}_p}\}_{k=1,2,\dots,t-1,t,t-1,\dots,2,1}$, 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 $\{\text{DMPF}_{k,2N/t,\mathbb{Z}_p}\}_{1 \leq k \leq t-1}$ and $\text{DMPF}_{t,2N/t,\mathbb{Z}_p}$;
- (2) with nonregular noise and each multiplication of sparse polynomials is shared by $\text{DMPF}_{t^2,2N,\mathbb{Z}_p}$.

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 batch-code / 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 batch-code 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 $\text{regular-}\mathcal{HW}_{R_2,t_2}$ 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 $N_1 = N_2/e$ 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 $\times e$ speedup.

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 [3], for R from irreducible F , we lowerbound the number of arithmetic operations by $N \cdot (c \cdot \frac{N}{N-1})^W \approx N \cdot c^W$. For R from reducible F , we lowerbound the number of arithmetic operations by $2^i \cdot c^{W_i}$ **Yaxin: (to be checked)**, where $i := \arg \min_{1 \leq i \leq \log N} \left(\left(\frac{c \cdot 2^i}{(c-1) \cdot 2^i - 1} \right)^{W_i} \cdot 2^i \right)$

and $W_i := c(c-1) \cdot 2^i \left(1 - \left(1 - \frac{1}{(c-1)2^i} \right)^{W/c} \right)$. **Yaxin: (to be checked)** 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 $t \geq 8$ gives at least $\times 2$ acceleration on expand time over other choices with sacrifice on the keysize. There is a tradeoff between the batch-code 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 batch-code-DMPF.

4.2 PSI-WCA

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

Yaxin: TBD:

- plug in new DMPF and analyze advantage interval
- plug in distributed gen

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 batch-code construction. The keysize (\propto communication complexity) of our choice is usually smaller than the batch-code 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] 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>.
- [3] 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>.
- [4] 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>.
- [5] 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>

- [6] 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>.
- [7] 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>.
- [8] 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.
- [9] 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>
- [10] 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.
- [11] Srinivasan Raghuraman and Peter Rindal. 2022. Blazing Fast PSI from Improved OKVS and Subfield VOLE. Cryptology ePrint Archive, Paper 2022/320. <https://eprint.iacr.org/2022/320> <https://eprint.iacr.org/2022/320>.
- [12] Philipp 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>.

A SECURITY PROOFS