Notes for New Constructions of DMPF

tbd[*]
tbd[*]
tbd
tbd
tbd
tbd, tbd, tbd

tbd
tbd
tbd, tbd
tbd

ABSTRACT

tbd.

CCS CONCEPTS

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

KEYWORDS

tbd

ACM Reference Format:
tbd, tbd, and 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, 18 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, where $\hat{\mathbb{G}}$ denotes the description of the group $\mathbb{G}$. A *t-point function* $f_{A,B} : [N] \to \mathbb{G}$ for $A = \{\alpha_1, \cdots \alpha_t\} \subset N$ listed in ascending order and $B = (\beta_1, \cdots, \beta_t) \in \mathbb{G}^t$ evaluates to β_i on input α_i for $1 \leq i \leq t$ and to 0 on all other inputs. 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*.

2.2 Distributed Multi-Point Functions

We begin by defining the notion of distributed point functions (DPF) and distributed multi-point functions (DMPF), that additively and succinctly share (multi-)point functions respectively.

[*]Both authors contributed equally to this research.

Conference acronym 'XX, tbd, tbd

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/tbd

Definition 1 (DPF [8, 15]). *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* $k_0, k_1 \in \{0,1\}^*$*. 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* $\mathsf{Adv}(1^\lambda, \mathcal{A}, i) \leq \nu(\lambda)$ *for all* $\lambda \in \mathbb{N}$.

Definition 2 (DMPF [4, 11]). *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* $k_0, k_1 \in \{0,1\}^*$.
- $\mathsf{Eval}_b(1^\lambda, k_b, x) \to y_b$*: 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 $b' \leftarrow \mathcal{A}(k_b)$.*

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 $A = \{\alpha_1, \ldots, \alpha_t\}$ and $B = \{\beta_1, \ldots, \beta_t\}$. 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. In the remainder of this paper, we'll provide DMPF schemes that has Eval and FullEval time almost independent to t.

2.3 Batch Code

We introduce probabilistic batch code, a batch code permitting small decoding errors, which can be used to construct DMPF (see construction 3).

Definition 3 (Probabilistic Batch Code (PBC)[1, 16, 23]). *An $(N, M, t, m, l, \epsilon)$-PBC over alphabet Σ is given by a tuple of efficient algorithms $(\mathsf{Encode}, \mathsf{Decode})$ with respect to the public randomness r such that:*

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

$$\Pr_r[(C_1, \ldots, C_m) \leftarrow \mathsf{Encode}_r(x), \; x[I] \ne \mathsf{Decode}_r(I, C_1, \ldots, C_m)] \le \epsilon$$

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

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

We will focus on the case $b = 1$ and a special class of batch code called combinatorial batch code (CBC)[1, 16, 20], where each codeword C_i is a subset of x. In this case, Encode_r sends $x[k]$ to the positions defined by $\mathsf{Position}_r(k)$, and Decode_r recovers $x[I]$ by rearranging the symbols it reads, whose positions are defined by Schedule. Note that when $b = 1$, Schedule algorithm implies finding a prefect matching from the size-t subset $I \subseteq [N]$ to the m buckets, in an (N, m)-bipartite graph where $k \in [N]$ is connected to $j \in [m]$ if and only if $x[k]$ is contained in C_j.

A natural way to construct PBC is to define the allocation of symbols in x to the buckets by a random (N, m)-bipartite graph where each left node has degree w for a fixed parameter w. To implement Encode and Decode (or more specifically Position and Schedule), we use the w-way cuckoo hashing algorithm[19] as a concrete and efficient instantiation of PBC as in [1, 12, 23].

w-way cuckoo hashing algorithm. Given t balls, $m = et$ buckets (e is an expansion parameter that is bigger than 1), and w independent random hash functions $h_1, h_2, \cdots, h_w$, each mapping the balls to the buckets, a w-way cuckoo hashing algorithm we describe here aims to allocate t balls to m buckets such that each ball is allocated to one of the buckets output by the w hash functions, and 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, and compute the bucket index $h_i(b)$. If this bucket is empty, then allocate b to this bucket and go to step 1. If this bucket is not empty and filled with ball b', then evict b', allocate b to this bucket, and repeat step 2 with unallocated ball b'.

If the algorithm terminates then it outputs a desired allocation of balls to buckets. To prevent it from running forever, one may set a fixed amount of running time. We call it a *failure* whenver the algorithm fails to output a desired allocation where each bucket contains at most one ball. We'll next summarize known asymptotic and empirical results about the failure probability of cuckoo hashing.

The failure probability of cuckoo hashing. Denote the failure probability of w-way cuckoo hashing to be $\epsilon = 2^{-\lambda_{\mathsf{stat}}}$. The empirical result from [12, Appendix B] shows that when $w = 3$ and $t \geq 4$, the failure probability is computed by

$$\lambda_{\text{stat}} = a_t \cdot e - b_t - \log t$$
$$a_t = 123.5 \cdot \mathsf{CDF}_{\text{Normal}}(x = t, \mu = 6.3, \sigma = 2.3)$$
$$b_t = 120 \cdot \mathsf{CDF}_{\text{Normal}}(x = t, \mu = 6.45, \sigma = 2.18)$$

where $e = m/t$ is the expansion parameter and $\mathsf{CDF}_{\text{Normal}}(x, \mu, \sigma)$ is the cumulative distribution function on input x for the normal distribution with expectation μ and standard deviation σ. Yaxin: More asymptotic and empirical results are listed in Table 5.

There are different ways to generalize cuckoo hashing algorithm in order to achieve negligible failure probability. For instance, one may allow at most $l > 1$ balls to be allocated to each bucket, instead of allowing at most one. One may also add an overflow stash with size s as an additional cache-like bucket [17]. We mostly use cuckoo hashing with $l = 1$ and $s = 0$ to construct PBC and DMPF, but will also point out how general cuckoo hashing may help with these constructions.

Now we display the instantiation of PBC using cuckoo hashing.

Construction 2 (PBC from cuckoo hashing[1]). *Given w-way cuckoo hashing as a sub-procedure allocating t balls to m buckets with failure probability ϵ, an $(N, wN, t, m, 1, \epsilon)$-PBC is as follows:*

- $\mathsf{Encode}_r(x \in \Sigma^N) \to (C_1, \cdots, C_m)$*: Use r to determine w independent random hash functions $h_1, h_2, \cdots h_w$ that maps from $[N]$ to $[m]$. For all $k \in [N]$, replicate $x[k]$ in the positions indicated by* $\mathsf{Position}_r(k)$*. Each output bucket C_j will be $\{x[k] : h_i(k) = j$ for some $i \in [w]\}$, in ascending order of k.*
- $\mathsf{Decode}_r(I, C_1, \cdots, C_m) \to \{x[i]\}_{i \in I}$*: Determine $h_1, \cdots, h_w$ by r as in* Encode*. For each $k \in I$, recover $x[k]$ from the position indicated by* $\mathsf{Schedule}_r(i)$.

with the following sub-processes:

- $\mathsf{Position}_r(k \in [N]) \to C_{h_1(k)}[j_1], \ldots, C_{h_w(k)}[j_w]$*: For $1 \leq l \leq w$, j_l is the order of k in the set $\{k' : \exists t \in [w], h_t(k') = h_l(k)\}$ of all indices mapped to bucket $C_{h_l(k)}$.*
- $\mathsf{Schedule}_r$*: For I of size at most t, run the w-way cuckooo hashing algorithm to allocate the indices in I to the m buckets, such that each bucket contains at most one index in I. For $k \in I$,* $\mathsf{Schedule}_r(k)$ *outputs the position $C_i[j]$ such that k is allocated to bucket C_i, and j is the order of k in bucket C_i (i.e., $x[k]$ will be sent to $C_i[j]$ in the encoding process).*

Note that the incorrectness of the above PBC scheme is equal to the failure probability of $\mathsf{Schedule}_r$ on set I, which equals ϵ.

Computing $\mathsf{Position}_r$ and $\mathsf{Schedule}_r$ requires computing the order of some index $k \in [N]$ in a bucket C_j it is mapped to, which may be inefficient when N is large. [11] address this issue by implementing w hash functions by a single *(pseudo-)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 it is empirically verified that the cuckoo hashing algorithm still succeeds with sufficient probability.

2.4 DMPF Construction from PBC

Next 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[4, 11, 22].

Construction 3 (PBC-based DMPF). *Given* DPF *for any domain of size $\leq N$ and output group $\mathbb{G}$, and an $(N, M, t, m, 1, \epsilon)$-PBC (which is systematic and combinatorial) with alphabet $\Sigma = \mathbb{G}$, we can construct a* DMPF *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 $B = \{\beta_1, \cdots, \beta_t\}$. Suppose the* PBC *encodes to buckets $C_1, \ldots, C_m$. Compute* $\mathsf{PBC.Schedule}(\alpha_k) \to C_{i_k}[j_k]$*, allocating elements in A to positions in the m buckets.*
  For all $1 \leq k \leq t$ For $1 \leq i \leq m$, let $f_i : [|C_i|] \to \mathbb{G}$ be the following:
  - *If there is no such $k \in [t]$ that $i_k = i$, then set f_i to be the all-zero function.*
  - *If there is exactly one $k \in [t]$ that $i_k = i$, then set f_i to be the point function that outputs β_j on j_k and 0 elsewhere.*

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

The scheme is correct with at least $1 - \epsilon$ probability and has distinguishing advantage $O(\epsilon)$.

When instantiating PBC using w-way cuckoo hashing as in Construction 2, the *evaluation time* is the sum of the time for $\mathsf{PBC.Position}(x)$ plus the time for w invocations of DPF.Eval. Similarly, the *full-domain evaluation time* is roughly the time for N invocations of PBC.Position plus w invocations of DPF.FullEval. Therefore, one may expect the evaluation time for the above construction of DMPF to be dependent to w instead of t. We will discuss about its efficiency in later sections.

Remark 4. *As a future direction, it is intriguing to try to use a $(N, M, t, m, l, \epsilon)$-PBC where $l > 1$ (say, instantiated by an w-way cuckoo hashing with bucket size l) to construct a DMPF scheme using l-DMPF with adjustable l, where the primitive l-DMPF can be implemented by any one of the DMPF schemes we discuss about in this paper.*

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 proposed as a primitive for private set intersection (PSI) protocols [14], and improved by a series of works [2, 21].

Definition 5 (Oblivious Key-Value Stores (OKVS)[2, 14, 21]). *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:*

- $\text{Encode}_r(\{(k_1,v_1),(k_2,v_2),\cdots,(k_t,v_t)\}) \rightarrow P$*: 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 $S \in (\mathcal{K}\times\mathcal{V})^t$, $\Pr_{r\leftarrow\{0,1\}^\kappa}[\text{Encode}_r(S) = \perp] \le 2^{-\lambda_{\text{stat}}}$.*
- *For all $S \in (\mathcal{K}\times\mathcal{V})^t$ and $r \in \{0,1\}^\kappa$ such that $\text{Encode}_r(S) \rightarrow P \ne \perp$, it is the case that $\text{Decode}_r(P,k) \rightarrow v$ whenever $(k,v) \in S$.*
- ***Obliviousness:** Given any distinct key sets $\{k_1^0,k_2^0,\cdots,k_t^0\}$ 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.,*

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

If the distinguishing advantage is upperbounded by a negligible function ϵ, then the OKVS scheme is ϵ-oblivious.

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

We evaluate an OKVS scheme by its rate ($\frac{\text{input length } t}{\text{output length } m}$), encoding time and decoding time.

The most naïve OKVS construction is encoding $S = \{(k_i,v_i)\}_{1\le i\le t}$ to a random truth table $TT : \mathcal{K} \rightarrow \mathcal{V}$ such that $TT(k_i) = v_i$ for all $1 \le i \le t$. Note that to ensure obliviousness, for k not appearing in S, the encoding should set $TT(k)$ to a random value. However this naïve construction is very inefficient since it requires the encoding size to be $m = |\mathcal{K}|$, and hence its rate $\frac{t}{|\mathcal{K}|}$ can be tiny.

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*

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

The polynomial OKVS possesses an optimal rate $\frac{t}{m} = 1$, 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) $O(\log^2 t)$.

The work of [2] gives a (linear) OKVS construction that has near optimal rate but much better running time.

Construction 5 (RB-OKVS[2]). *Suppose $\mathcal{V} = \mathbb{G}$ is a group. Let $\text{row}_r(k)$ output a $\{0,1\}^m$ vector consisting of a width-w random band. Formally speaking, $\text{row}_r(k)$ 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 $\text{row}_r(k)$ and leave the rest as 0 entries.*

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

Denote $m = et$ where $e > 1$ is an expansion parameter (the inverse of rate) indicating the blowup to store t pairs. The encoding time is equivalent to solving a linear system whose coefficient matrix is a random band matrix, which can be efficiently done in $O(tw + t\log t)$ time. The decoding time is w additions in $\mathbb{F}$ and the rate $1/e$ can be very close to 1.

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 $w = O(\lambda_{\text{stat}}/(e-1) + \log t)$ ensures the correctness and obliviousness with probability $2^{-\lambda_{\text{stat}}}$ and 2^{-w}, respectively. The empirical results take $e = 1.03, 1.05, 1.07, 1.1$ while w being several hundred to reach the security $\lambda_{\text{stat}} = 40$, with the choice of t varying from 2^{10} to 2^{20}. We will apply this OKVS on a different range of t, such that t can be as small as 1 or 2, which differs with the original application scenarios in [2]. However, since the set of parameters that ensures the successful encoding of t' key-value pairs will also ensure the successful encoding of $t < t'$ key-value pairs, this issue can be natually resolved.

In our later sections, we refer to the RB-OKVS (construction 5) by default when instantiating OKVS. One may switch to other OKVS constructions such as ones in [14, 21], 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 [8]. Each key $k_b (b = 0,1)$ generated by $\text{Gen}(1^\lambda, \hat{f}_{A,B})$ 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 $\{0,1\}^{\le n}$ 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, with each node in the evaluation tree T_b associated with a λ-bit pseudorandom string seed string and an l-bit pseudorandom sign string (l is an adjustable parameter). The evaluation tree T_b is determined by the seed$\|$sign string at its root, and the set of corrections words $\{CW^{(i)}\}_{1\le i\le n}$ at each layer. T_b can be computed recursively as follows: for $i = 1,2,\ldots,n$, given the $(i-1)$th layer of T_b (the 0th layer is the root), for each node v with strings seed_v and sign_v in the $(i-1)$th layer, generate its children's seed and sign strings denoted by $\text{seed}_0\|\text{sign}_0$

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.**

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

5: **procedure** $\textsc{Gen}(1^\lambda, \hat{f}_{A,B})$
6:     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.
7:     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, $A^{(0)} = [\epsilon]$.
8:     For $0 \le i \le n-1$ and $b = 0,1$, initialize empty lists $\mathsf{seed}_b^{(i)}$, $\mathsf{sign}_b^{(i)}$ and $V^{(i)}$.
9:     $\mathsf{Initialize}(\{\mathsf{seed}_b^{(0)}, \mathsf{sign}_b^{(0)}\}_{b=0,1})$.
10:     **for** $i = 1$ to n **do**
11:         **for** $k = 1$ to $|A^{(i-1)}$ **do**
12:             For $c = 0,1$, compute $\Delta\mathsf{seed}^c = G^c_{\mathsf{seed}}(\mathsf{seed}_0^{(i-1)}[k]) \oplus G^c_{\mathsf{seed}}(\mathsf{seed}_1^{(i-1)}[k])$ and $\Delta\mathsf{sign}^c = G^c_{\mathsf{sign}}(\mathsf{seed}_0^{(i-1)}[k]) \oplus G^c_{\mathsf{sign}}(\mathsf{seed}_1^{(i-1)}[k])$.
13:             **if** $A^{(i-1)}[k]\|0 \in A^{(i)}$ and $A^{(i-1)}[k]\|1 \in A^{(i)}$ **then**
14:                 Randomly sample $r \leftarrow \{0,1\}^\lambda$ and append $r\|\Delta\mathsf{sign}^0\|\Delta\mathsf{sign}^1$ to $V^{(i-1)}$.
15:             **else**
16:                 Suppose $A^{(i-1)}[k]\|z \in A^{(i)}$. Append $\Delta\mathsf{seed}^{1-z}\|\Delta\mathsf{sign}^0\|\Delta\mathsf{sign}^1$ to $V^{(i-1)}$.
17:             **end if**
18:         **end for**
19:         $CW^{(i)} \leftarrow \mathsf{GenCW}(i, A, V^{(i-1)})$.
20:         **for** $k = 1$ to $|A^{(i-1)}|$ and $z = 0,1$ **do**
21:             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 $|C_{\mathsf{sign}^0,b}| = |C_{\mathsf{sign}^1,b}| = l$.
22:             **if** $A^{(i-1)}[k]\|z \in A^{(i)}$ **then**
23:                 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 $\mathsf{sign}_b^{(i)}$.
24:             **end if**
25:         **end for**
26:     **end for**
27:     $CW^{(n+1)} \leftarrow \mathsf{GenConvCW}\left(A, B, \left(G_{\mathsf{conv}}(\mathsf{seed}_0^{(n)}[k]) - G_{\mathsf{conv}}(\mathsf{seed}_1^{(n)}[k])\right)_{1\le k\le|A|}, \mathsf{sign}_0^{(n)}, \mathsf{sign}_1^{(n)}\right)$.
28:     Set $k_b \leftarrow (\mathsf{seed}_b^{(0)}, \mathsf{sign}_b^{(0)}, CW^{(1)}, CW^{(2)}, \cdots, CW^{(n+1)})$.
29:     **return** (k_0, k_1).
30: **end procedure**

for the left child and $\mathsf{seed}_1\|\mathsf{sign}_1$ for the right child by first setting $\mathsf{seed}_0\|\mathsf{sign}_0\|\mathsf{seed}_1\|\mathsf{sign}_1 = G(\mathsf{seed}_v)$ where $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2l}$ is a pseudorandom generator, then compute a correction by $C_{\mathsf{seed}}\|C_{\mathsf{sign}^0}\|C_{\mathsf{sign}^1} \leftarrow \mathsf{Correct}(x_1 \ldots x_{i-1}, \mathsf{sign}, CW^{(i)})$ (see line 5), and in the end correct the seed strings seed_0 and seed_1 by adding C_{seed} to both of them, and correct the sign strings sign_0 and sign_1 by adding C_{sign_0} and C_{sign_1} respectively.

We expect the two evaluation trees to satisfy the following important properties:

(1) T_0 and T_1 have identical seed and sign strings on every node not lying on any accepting path.
(2) For a node lying on an accepting path, its seed strings in T_0 and T_1 are pseudorandom and independent, while its sign strings are two correlated pseudorandom strings following

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

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

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

some correlation designed by specific realizations. The correlation is an XOR correlation, meaning the two sign strings should add (by XOR) up to a specific string. [2]

The first property is equivalent to asking T_0 and T_1 to have identical strings on every node exiting an accepting path (meaning the node is not on an accepting path but its parent is): if a parent node is associated with the same strings in T_0 and T_1, then each of its children is associated with the same strings in T_0 and T_1, and so is each of the nodes in the subtree rooted at the parent node. To force the first property, we expect that at each node exiting an accepting path, the correction C_{seed} and C_{sign} for this node eliminates the difference between its original seed‖sign strings generated by the PRG in T_0 and T_1.

To force the second property, we expect that at each node on an accepting path, the correction C_{seed} for this node should preserve the pseudorandomness and independence of the original seed strings in T_0 and T_1, while the correction C_{sign} should force the desired correlation of sign strings in T_0 and T_1.

$\mathsf{Gen}(1^\lambda, \hat{f}_{A,B})$ generates the keys k_0 and k_1, containing seed‖sign string at root and correction words for each layer, which determine T_0 and T_1 respectively. At the ith layer, $\mathsf{Gen}(1^\lambda, \hat{f}_{A,B})$ first records in the list $V^{(i-1)}$ all the strings in the ith layer that need to be corrected, which are the seed strings of the nodes exiting an accepting path, and the sign strings of the nodes whose parent is on an accepting path. Then it utilizes $\mathsf{GenCW}(i, A, V^{(i-1)})$ to generate the hint $CW^{(i)}$ for both parties (line 19), such that at an node on the ith layer, $CW^{(i)}$ along with the node's position and its parent's sign can be processed by the method Correct as the desired corrections C_{seed} and C_{sign} that forces the properties 1 and 2, as discussed before. The detailed implementation of the above process will be discussed in the concrete realizations.

After receiving the key k_b, party b can evaluate the input $x = x_1 \dots x_n$ by calling $\mathsf{Eval}_b(1^\lambda, k_b, x)$. It first parse the key k_b to the seed‖sign string at the root and the correction words $\{CW^{(i)}\}_{i \in [n]}$ for each layer, and then iteratively computes the seed‖sign strings along the path represented by x in T_b. In the ith iteration given the $\mathsf{seed}_{i-1} \| \mathsf{sign}_{i-1}$ string for the node represented by $x_1 \dots x_{i-1}$, the Eval method computes the original seed and sign strings for the node $x_1 \dots x_i$ by invoking PRG on seed_{i-1}, and then correct these strings by invoking Correct on $x_1 \dots x_{i-1}$, sign_{i-1} and $CW^{(i)}$ to obtain the real $\mathsf{seed}_i \| \mathsf{sign}_i$ (see line 5).

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 correction word $CW^{(n+1)}$ 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 $G_{\mathsf{conv}}(\mathsf{seed})$, then adding to $G_{\mathsf{conv}}(\mathsf{seed})$ a correction computed by $\mathsf{ConvCorrect}(x, \mathsf{sign}, CW^{(n+1)})$, and then give a sign $(-1)^b$ depending on the party (see line 8). If the leaf node is not on

[1] The iterative traverse of the evaluation tree is written in a BFS style for better illustration, which will cost a lot of memory in practice. When implementing this scheme the traverse should be done in a DFS style or else.

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

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 27).

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 $\{CW^{(1)}, \dots CW^{(n)}\}$ 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 30 of fig. 2 equals $CW^{(i)}[k]$, the kth entry of the hint $CW^{(i)}$ associated with this layer. According to line 17 in the 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 $B[k]$.

Remark 6. *Note that when $t = 1$, the big-state DMPF scheme is exactly the DPF scheme in [8].*

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

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($\{\text{seed}_b^{(0)}, \text{sign}_b^{(0)}\}_{b=0,1}$)
3: For $b = 0, 1$, let $\text{seed}_b^{(0)} = [r_b]$ where $r_b \xleftarrow{\$} \{0,1\}^\lambda$.
4: For $b = 0, 1$, set $\text{sign}_b^{(0)} = [b \| 0^{t-1}]$.
5: **end procedure**

6: **procedure** GenCW($i, A, V^{(i-1)}$)
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 $V^{(i-1)}[k] = \Delta\text{seed} \| \Delta\text{sign}^0 \| \Delta\text{sign}^1$.
11: **if** $A^{(i-1)}[k] \| z \in A^{(i)}$ holds for both $z = 0, 1$ **then**
12: $d \leftarrow$ the index of $A^{(i-1)}[k] \| 0$ in $A^{(i)}$.
13: $CW[k] \leftarrow \Delta\text{seed} \| (\Delta\text{sign}^0 \oplus e_d) \| (\Delta\text{sign}^1 \oplus e_{d+1})$ where $e_d = 0^{d-1}10^{t-d}$.
14: **else**
15: Suppose $A^{(i-1)}[k] \| z \in A^{(i)}$. Let d be the index of $A^{(i-1)}[k] \| z$ in $A^{(i)}$.
16: $\Delta\text{sign}^z \leftarrow \Delta\text{sign}^z \oplus e_d$.
17: $CW[k] \leftarrow \Delta\text{seed} \| \Delta\text{sign}^0 \| \Delta\text{sign}^1$.
18: **end if**
19: **end for**
20: **return** CW.
21: **end procedure**

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

29: **procedure** Correct($\bar{x}, \text{sign}, CW$)
30: **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_{sign^0} and C_{sign^1} are t-bit.
31: **end procedure**

32: **procedure** ConvCorrect(x, sign, CW)
33: **return** $\sum_{i=1}^t \text{sign}[i] \cdot CW[i]$.
34: **end procedure**

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.

The big-state DMPF is computationally secure when the PRGs G and G_{conv} are secure. We defer the proof to Appendix A.2.

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 are determined such that, 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, recorded by the value list $V^{(i-1)}$. We encode these (node, correction) pairs (there are up to t such pairs) using an OKVS scheme and set the hint $CW^{(i)}$ to 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 31).

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. For the upmost few layers where $2^i < t$, OKVS_i may be realized by the most naïve way of encoding to a random truth table (see Section 2.5), which achieves the optimal rate in this occasion.

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.

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 $\min\{2^{i-1}, t\}$.

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, V)
8: Let $\{A^{(i)}\}_{0\le i\le n}$ be defined as in fig. 1.
9: **for** $k = 1$ to $|A^{(i-1)}|$ **do**
10: Parse $V[k] = \Delta\mathsf{seed}\|\Delta\mathsf{sign}^0\|\Delta\mathsf{sign}^1$.
11: **for** $z = 0,1$ such that $A^{(i-1)}[k]\|z \in A^{(i)}$ **do**
12: $\mathsf{sign}^z \leftarrow \mathsf{sign}^z \oplus 1$.
13: **end for**
14: Update $V[k] \leftarrow \Delta\mathsf{seed}\|\Delta\mathsf{sign}^0\|\Delta\mathsf{sign}^1$.
15: **end for**
16: Copy the list $A^{(i-1)}$ to the list K.
17: **for** $j = |K|+1$ to $\min\{2^{i-1}, t\}$ **do**
18: Set $K[j]$ to be an arbitrary string in $\{0,1\}^{i-1}$ that is different from $K[1\dots j-1]$.
19: Set $V[j]$ to be a random string in $\{0,1\}^{\lambda+2}$.
20: **end for**
21: **return** $\mathsf{OKVS}_i.\mathsf{Encode}(\{(K[j], V[j])\}_{1\le j\le|K|})$.
22: **end procedure**

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

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

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

The OKVS-based DMPF is computationally secure when the PRGs G and G_{conv} are secure and the OKVS schemes $\{\mathsf{OKVS}_i\}_{1\le i\le n}$ and $\mathsf{OKVS}_{\mathsf{conv}}$ are oblivious. We defer the proof to Appendix A.3.

The efficiency of OKVS-based DMPF highly relies 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

Reorganize. 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 w times the number of PRGs than big-state DMPF and OKVS-based DMPF, while big-state DMPF's FullEval time scales with t^2 (since the large-bit-XOR time scales with t^2) and OKVS-based DMPF in addition consumes large field operations (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, and it is likely that the big-state construction performs the best when t is small, while the CBC-based and OKVS-based constructions performs well when t is large.

3.5 Distributed Key Generation

Discussion to be added.

3.6 Distributed Multi-Interval Function

The distributed comparison function (DCF) proposed in [7, 8] generates shares for any comparison function $f_{<\alpha,\beta} : [N] \to \mathbb{G}$ that evaluates to β on any input $x < \alpha$ and 0 elsewhere, for a secret threshold $\alpha \in [N]$ and a secret payload $\beta \in \mathbb{G}$. Previous literatures give efficient and secure construction of DCF that has keysize $\log N(\lambda + 2\log|\mathbb{G}| + 2)$ and evaluation time roughly the same as the evaluation time for DPF in the same domain and output group. A DCF for $f_{<\alpha,\beta}$ can be easily converted to a DCF for other comparisons $f_{\le\alpha,\beta}$, $f_{>\alpha,\beta}$ and $f_{\ge\alpha,\beta}$.

The application of DCF includes distributed public interval containments and spline functions [3, 9], and constructing pseudorandom correlation functions [5]. Yaxin: Haven't got a complete application list. And does the latter one actually ask for a distributed multi-interval function?

The previous construction of DCF is done by extending the PRG-based DPF scheme in the following way: it adds an additional string at each node that can be corrected to be used to compute the shared output of the comparison function. Similarly to this conversion, we argue in this section that any DMPF scheme following the DMPF paradigm in Figure 1 can be converted to a distributed multi-interval function, where a multi-interval function is defined as follows:

Definition 7 (Multi-Interval Function). *Given a domain size $N = 2^n$ and Abelian group $\mathbb{G}$, a k-interval function $f_{I,B} : [N] \to \mathbb{G}$ for $I = ([a_1, b_1], [a_2, b_2], \ldots, [a_k, b_k])$ such that $1 \le a_1 < b_1 < a_2 < b_2 < \cdots < a_k < b_k \le N$ and $B = (\beta_1, \beta_2, \ldots, \beta_k) \in \mathbb{G}^k$ evaluates to β_i on any input x in the interval $[a_i, b_i]$ for $1 \le i \le k$ and to 0 on all other inputs. Call the collection of all k-interval functions for all k the multi-interval functions.*

We define a distributed multi-interval function to be in the similar form as DMPF, while its security requires that each key leaks no information about the intervals I and the payloads B.

Note that a distributed multi-interval function can be constructed by summing up $2k$ DCFs, since $f_{I,B} = \sum_{i=1}^{k} f_{\ge a_i,\beta_i} + \sum_{i=1}^{k} f_{\ge b_i+1,-\beta_i}$. Therefore we can relate the shares of $f_{I,B}$ (denoted as $[f_{I,B}]$) with the shares of the multi-point function $f_{A',B'}$ where $A' = (a_1, b_1 + 1, \ldots, a_k, b_k + 1)$ and $B' = (\beta_1, -\beta_1, \ldots, \beta_k, -\beta_k)$:

$$f_{I,B}(x) = \sum_{a_i \le x} \beta_i - \sum_{b_j < x} \beta_j = \sum_{y \le x} f_{A',B'}(y)$$

Hence the full-domain evaluation of a multi-interval function can be directly computed from the full-domain evaluation of a DMPF, with the same keysize and approximately the same FullEval time.

Next we make additional modifications to convert a DMPF scheme to a distributed multi-interval function, when aiming for the single-input evaluation. For simplicity, in the sequel, we assume the domain is $\{0, 1\}^n$ and the output group $\mathbb{G}$ is the group of g-bit binary strings with the group addition being $\oplus$. In this setting $\beta = -\beta$. One can put a little more effort to generalize the conversion to any group.

For a DMPF scheme for $f_{A',B'}$ following the paradigm in Figure 1, let each node contains an additional g-bit string res, satisfying the following conditions: (1) for each node v on an accepting path of $f_{A',B'}$, let res_0 and res_1 denote its res strings in the evaluation trees T_0 and T_1 respectively, then

$$\mathsf{res}_0 \oplus \mathsf{res}_1 = \sum_{a_i \text{ is } v \text{ or a descendant of } v} \beta_i + \sum_{b_j \text{ is } v \text{ or a descendant of } v} \beta_j$$

(2) for each node v exiting an accepting path, $\mathsf{res}_0 \oplus \mathsf{res}_1 = 0^g$. This, combined with the original invariance 1 for the evaluation tree, indicates that for each node v not on any accepting path, $\mathsf{res}_0 \oplus \mathsf{res}_1 = 0^g$.

Note that the DMPF paradigm in Figure 1 provides the mechanism to control the difference of strings in any node on or exiting an accepting path, by recording the target strings in the list $V^{(i-1)}$ (see lines 14 and 16), generating the correction word CW (line 17), and correcting the target strings by adding corrections generated by CW (line 21). Therefore, one can enforce the above two conditions by additionally recording $\Delta\mathsf{res}^0$ (for the left child) and $\Delta\mathsf{res}^1$ (for the right child) in $V^{(i-1)}$ (corresponds to modifying lines 14 and 16), then generating the proper correction word for res strings using modified GenCW and Correct in order to get the desired (XOR-)correlation.

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 $T_{G_{\mathrm{conv}}}$ as the time for computing $G_{\mathrm{conv}} : \{0,1\}^{\lambda} \to \mathbb{G}$. In the last column, denote OKVS as the OKVS scheme used for the first n layers, and $\mathsf{OKVS}_{\mathrm{conv}}$ as the OKVS scheme used for the convert layer.

	Sum of t DPFs	CBC-based DMPF[1, 4, 11, 22]	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 $+\mathsf{OKVS}_{\mathrm{conv}}$.Codesize
Gen()	$2t\log N \times T_G + 2t \times T_{G_{\mathrm{conv}}}$	$2m\log(wN/m)\times T_G + 2m \times T_{G_{\mathrm{conv}}}$ CBC.Encode + CBC.Decode	$2t\log N \times T_{G^*}$[1] $+t\log N \times (\lambda+t)$-bit-XOR	$2t\log N \times T_G + 2t\times T_{G_{\mathrm{conv}}}$ $+\log N \times$ OKVS.Encode $+\mathsf{OKVS}_{\mathrm{conv}}$.Encode
Eval()	$t\log N \times T_G + t \times T_{G_{\mathrm{conv}}}$	$w\log(wN/m)\times T_G + w\times T_{G_{\mathrm{conv}}}$ Finding all positions the input is mapped to	$\log N \times T_{G^*} + T_{G_{\mathrm{conv}}}$ $+t\log N\times(\lambda+t)$-bit-XOR	$\log N \times T_G$ $+\log N\times$ OKVS.Decode $+\mathsf{OKVS}_{\mathrm{conv}}$.Decode
FullEval()	$tN\times T_G + tN \times T_{G_{\mathrm{conv}}}$	$wN\times T_G + wN\times T_{G_{\mathrm{conv}}}$ Finding the full mapping	$N\times T_{G^*} + N\times T_{G_{\mathrm{conv}}}$ $+2tN\times(\lambda+t)$-bit-XOR	$N\times T_G$ $+N\times$ OKVS.Decode $+N\times \mathsf{OKVS}_{\mathrm{conv}}$.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^*}.

Figure 4: Modifying the Eval method in Figure 1 to the Eval for a distributed multi-interval function.

```
procedure Eval_b(1^λ, k_b, x)
    Parse k_b = (seed, sign, CW^(1), CW^(2), ..., CW^(n)).
    Denote x = x[1...n].
    S = 0^g.
    for i = 1 to n do
        C_seed || C_sign^0 || C_sign^1 || C_res^0 || C_res^1 ← Correct(x[1...(i−1)], sign, CW^(i)).
        Parse G(seed) = seed^0 || sign^0 || res^0 || seed^1 || sign^1 || res^1.
        seed || sign ← seed^{x_i} ⊕ C_seed || sign^{x_i} ⊕ C_sign^{x_i}.
        S ← S ⊕ (res^{1−x_i} ⊕ C_res^{1−x_i}).
    end for
    return S.
end procedure
```

The evaluation on an input $x = x[1\dots n]$ sums up all the res strings on the nodes that takes a left step to exit the path represented by x. The formal description is in Figure 4. We argue that the summation S is a share of $f_{I,B}(x)$. Denote the output of party b as S_b. Suppose $J_1 = \{j \in [n] : x[j] = 1\}$. Then for any $a \in \{0,1\}^n$, $a \le x$ if and only if there exists $j \in J_1$ such that $a[1\dots(j-1)] = x[1\dots(j-1)]$ and $a[j] = 0$. Therefore a is or is a descendant of the node $v_j = x[1\dots(j-1)]\|0$ in the tree. Hence,

$$
\begin{aligned}
S_0 + S_1 &= \sum_{j\in J_1} (\mathsf{res}_{v_j,0} \oplus \mathsf{res}_{v_j,1}) \\
&= \sum_{j\in J_1}\left(\sum_{\substack{a_i \text{ is } v_j \\ \text{or is a descendant of } v_j}} \beta_i + \sum_{\substack{b_i \text{ is } v_j \\ \text{or is a descendant of } v_j}} \beta_i \right) \\
&= \sum_{a_i \le x} \beta_i + \sum_{b_i \le x} \beta_i
\end{aligned}
$$

where $\mathsf{res}_{v,b}$ denotes the res string on node v in tree T_b.

The distributed multi-interval function is secure as long as the DMPF for $f_{A',B'}$ is secure, since the ends of the intervals and the payloads are hidden.

For distributing a k-interval function, we can compare our construction with the naïve construction that adds up $2k$ DCFs, which has the similar evaluation time of $2k$ DPFs and $\times(1+\frac{\log|\mathbb{G}|}{\lambda/2+1})$ the keysize of $2k$ DPFs. Note our conversion from DMPF extends each entry of $V^{(i-1)}$ (the input of GenCW) from $(\lambda+2l)$-bit to $(\lambda+2l+2\log|\mathbb{G}|)$-bit. When the DMPF paradigm is realized by the big-state (where $l = 2k$) or the OKVS-based (where $l = 1$) DMPF scheme, the keysize of the distributed k-interval function is increased to $\times(1+\frac{\log|\mathbb{G}|}{\lambda/2+l})$ the keysize of DMPF. Therefore the comparison between our construction and the naïve construction of distributed k-interval function, should be 'proportional to' the comparison between our construction and the naïve construction of DMPF for $2k$-point functions, which is discussed in Section 3.4 and Section 4.

Remark 8. *Since the PBC-based DMPF does not follow the DMPF paradigm in Figure 1, the conversion does not apply. It is not clear to us how to convert the PBC-based DMPF to a distributed multi-interval function.*

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 $\mathsf{DMPF}_{t,N,\mathbb{G}}$ 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 [6].

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 9 (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*

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

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 [6] 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 $\mathsf{DMPF}_{t^2,2N,\mathbb{Z}_p}$ 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 Yaxin: or $2c^2$ multiplications since $\vec{a}\otimes\vec{a}$ need also be computed? plus $c^2\times$full-domain evaluation time of DMPF.

Remark 10 (From OLE over polynomial ring R to OLE over $\mathbb{Z}_p$). *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[6]. Therefore we mostly consider reducible F and more concretely F a two-power cyclotomic due to its useful properties.*

Amortized expanding time for each OLE correlation over $\mathbb{Z}_p$. The amortized expanding time for each OLE correlation over $\mathbb{Z}_p$ is computed by

$$T_{\text{Amortized}} = c^2 \cdot (2\bar{T}^{\mathsf{MULT}}_N + \bar{T}^{\mathsf{FullEval}}_{t,N})$$

where

$$\bar{T}^{\mathsf{MULT}}_N := \frac{\text{deg} < N \text{ polynomial multiplication}}{N}$$

is the amortized cost for computing one deg$< N$ polynomial multiplication, and

$$\bar{T}^{\mathsf{FullEval}}_{t,N} := \frac{\mathsf{DMPF}_{t^2,2N,\mathbb{Z}_p}.\mathsf{FullEval}}{N}$$

is the amortized cost for computing a share of an entry of $\vec{e_0}\otimes\vec{e_1}$. This cost differs under different $\mathsf{DMPF}_{t^2,2N,\mathbb{Z}_p}$ instantiations: Yaxin: $T_{G_{\text{conv}}}$ is ignored for now.

- Sum of DPFs: $2t^2 \times T_G$.
- CBC-based DMPF ($\mathcal{HW}_t$): $2w \times T_G + 2w \times T_{\text{hash}}$[3].
- Big-state DMPF: $2T_{G^*}$, where G^* maps λ-bit to $(2\lambda + 2t)$-bit.
- OKVS-based DMPF: $2\,(T_G + \mathsf{OKVS.Decode})$[4].

Using the regular noise distribution to split $\mathsf{DMPF}_{t^2,2N,\mathbb{Z}_p}$. A previous optimization in [6], aiming to share the entries of $\vec{e_0}\otimes\vec{e_1}$ through DPFs but with less overhead (in contrast to the $2t^2\times T_G$ cost before), is to substitute $\mathcal{HW}_{R,t}$ with the distribution of random regular weight-t polynomials denoted as $\mathcal{RHW}_{R,t}$. 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, $e_i \cdot f_j$ contributes to a coefficient in the range of degree $[(i+j)\cdot N/t, (i+j+2)\cdot N/t - 2]$. Therefore the deg-$2N$ polynomial $e\cdot f$ can be shared by invoking $\{\mathsf{DMPF}^{(k)} = \mathsf{DMPF}_{\min(k,2t-k),2N/t,\mathbb{Z}_p}\}_{1\leq k\leq 2t-1}$, where $\mathsf{DMPF}^{(k)}$'s domain corresponds to the coefficients in the range of degree $[(k-1)N/t, (k+1)N/t - 2]$ in the resulting polynomial $e\cdot f$. Then each $\mathsf{DMPF}^{(k)}$ in the set $\{\mathsf{DMPF}^{(k)}\}_{1\leq k\leq 2t-1}$ can be instantiated by one of the mentioned schemes: sum of DPFs (used in [6]), CBC-based, big-state, or OKVS-based DMPF. We note that the efficiency (in terms of FullEval time) of all these instantiations are more or less related to the number of nonzero inputs in the targeted DMPF, therefore using $\mathcal{RHW}_{R,t}$ instead to $\mathcal{HW}_{R,t}$ reduces the number of nonzero inputs from t^2 to at most t may be beneficial in some occasions.

An alternative way to split $\mathsf{DMPF}_{t^2,2N,\mathbb{Z}_p}$, basing on the previous observation, is to share the coefficients of $e\cdot f$ by invoking $\{\mathsf{DMPF}'^{(k)} = \mathsf{DMPF}_{2\min(k+1,2t-k)-1,N/t,\mathbb{Z}_p}\}_{0\leq k\leq 2t-1}$, where

[3]The hash function's domain and range are related to t.

[4]The OKVS scheme encodes at most t key-value pairs. OKVS.Decode usually takes a small number of field addition or field multiplication in $\mathbb{F}_{2^\lambda}$, depending on the implementation.

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

$\mathsf{DMPF}'^{(k)}$'s domain corresponds to the coefficients in the range of degree $[kN/t, (k+1)N/t-1]$. Since the number of nonzero coefficients in $[kN/t, (k+1)N/t-1]$ is upperbounded by the sum of number of nonzero coefficients in $[(k-1)N/t, (k+1)N/t-2]$ and in $[kN/t, (k+2)N/t-2]$, $\mathsf{DMPF}'^{(k)}$ has at most $2\min(k+1, 2t-k)-1$ nonzero inputs. The advantage of using $\{\mathsf{DMPF}'^{(k)}\}_{0\le k\le 2t-1}$ is that the previous concatenation through $\{\mathsf{DMPF}^{(k)}\}_{1\le k\le 2t-1}$ creates overlapping ranges, which doubles the number of PRG invocations when realizing by CBC-based, big-state, and OKVS-based DMPF. By using $\{\mathsf{DMPF}'^{(k)}\}_{0\le k\le 2t-1}$, the ranges are not overlapping and therefore maintains the minimal PRG invocations, while also preserving a relatively small number of nonzero inputs (at most $2t-1$) in each $\mathsf{DMPF}'^{(k)}$.

Previously, [6] uses sum of DPFs to instantiate $\{\mathsf{DMPF}^{(k)}\}_{1\le k\le 2t-1}$ in the first optimized design with regular noise distribution. 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 noise distribution $\mathcal{HW}_{R,t}$ and each multiplication of sparse polynomials is shared by $\mathsf{DMPF}_{t^2,2N,\mathbb{Z}_p}$
(2) ith noise distribution $\mathcal{RHW}_{R,t}$ and each multiplication of regular sparse polynomials is shared by

$$\{\mathsf{DMPF}^{(k)} = \mathsf{DMPF}_{\min(k,2t-k),2N/t,\mathbb{Z}_p}\}_{1\le k\le 2t-1}$$

(3) ith noise distribution $\mathcal{RHW}_{R,t}$ and each multiplication of sparse polynomials is shared by

$$\{\mathsf{DMPF}'^{(k)} = \mathsf{DMPF}_{2\min(k+1,2t-k)-1,N/t,\mathbb{Z}_p}\}_{0\le k\le 2t-1}$$

Yaxin: Dec 27: It is also mentioned using more advanced noise distributions to avoid overlapping ranges. For now it remains to give the concrete costs for (1) $\mathsf{DMPF}_{t^2,2N,\mathbb{Z}_p}$; (2)$\mathsf{DMPF}_{1/2/\ldots/t,2N/t,\mathbb{Z}_p}$; (3)$\mathsf{DMPF}_{1/2/\ldots/(2t-1),N/t,\mathbb{Z}_p}$ under sum of DPFs/CBC-based/big-state/OKVS-based instantiations.

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?

Setting parameters (N, c, t) *against best attacks.* Next we plug in concrete parameters and evaluate the performance of different DMPF schemes under different PCG parameter settings.

The parameters (N, c, t) should be set in a way that the corresponding R^c-LPN problem is secure with computational security parameter λ. In [6] the parameters are taken to be $(\lambda, N, c, t) \in \{(128, 2^{20}, 8, 5), (128, 2^{20}, 4, 16), (128, 2^{20}, 2, 76)\}$ against several attacks, with the best among which predicted to be the SD or ISD family. Yaxin: Dec 28: Using the new lowerbound in [18] (if I understood it correctly), the setting of parameters (fixing $\lambda = 128$ and $N = 2^{20}$) becomes $(\lambda, N, c, t) \in \{(128, 2^{20}, 8, 5), (128, 2^{20}, 4, 14), (128, 2^{20}, 2, 70)\}$. In fact, N is independent to other parameters because R is a reducible polynomial ring. The three parameters are comparable in efficiency in [6] because they used the second DMPF instantiation (concatenation of DPFs under $\mathcal{RHW}_t$ distribution) whose cost scales with c^2t. By using the new DMPF instantiations we may see significant difference among the three tuples of parameters.

How to compute (N, c, t)**:** We set the parameters (λ, c, N, t) such that the best attack requires at least 2^λ arithmetic operations over field $\mathbb{F}_p$ of size approximately 2^{128}.

An R^c-LPN instance $a\cdot e$ can be viewed as a (dual-)$\mathsf{LPN}_{cN,N,\mathcal{HW}^{\otimes c}_{N,t}}$ instance $\{H, H\cdot\vec{e}\}$, where $H \in \mathbb{Z}_p^{N\times cN}$ is a concatenation of c circular matrices representing multiplication with a, and $\vec{e} \in \mathbb{Z}_p^{cN}$ with distribution $\mathcal{HW}^{\otimes c}_{N,t}$ represents the concatenation of coefficients of e. The bit security of the R^c-LPN problem is equivalent to the bit security of the described (dual-)LPN problem. As in [6], we consider the bit security of the described (dual-)LPN problem to be the same as the bit security of the standard (dual-)LPN problem, whose error distribution is $\mathcal{HW}_{cN,ct}$, the random weight-ct noises.

According [6], for R from an irreducible F, we lowerbound the number of arithmetic operations by $N\cdot(c\cdot\frac{N}{N-1})^{ct} \approx N\cdot c^{ct}$. Yaxin: Dec 28: [18] seems to indicate a better lowerbound of $N^{2.8}\cdot c^{ct}$. For R from a reducible F, the number of arithmetic operations is lowerbounded by $2^i\cdot c^{w_i}$ Yaxin: Dec 28: or $2^{2.8i}\cdot c^{w_i}$ by [18], where w_i is the expected number of noisy coordinates modulo an 1-sparse deg-2^i polynomial and $i :=$ the smallest integer such that $w_i < 2^i$. Then t is computed by the equation

$$w_i = c\cdot 2^i\left(1-(1-2^{-i})^t\right)$$

Yaxin: Dec 29: In [6] there are two formulas calculating w_i. One is as above, and the other is

$$w_i = ct - 2^ic + ((2^i-1)c + ct)\cdot\left(1-2^{-i}\right)^{t-1}$$

The second one does not make sense to me but is used to compute the concrete results. Nevertheless the two formulas computes similar results so I used the first one which makes more sense to me.

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 6 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.

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 [13]:

- 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.

We'll instantiate DMPF in different ways as listed in table 1. As suggested in [13], 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.

4.3 Security analysis

4.4 Heavy-hitters

private heavy-hitter
or parallel ORAM?

ACKNOWLEDGMENTS

TBD, if any.

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, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. 2020. Function Secret Sharing for Mixed-Mode and Fixed-Point Secure Computation. Cryptology ePrint Archive, Paper 2020/1392. https://eprint.iacr.org/2020/1392 https://eprint.iacr.org/2020/1392.

[4] 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.

[5] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Nicolas Resch, and Peter Scholl. 2022. Correlated Pseudorandomness from Expand-Accumulate Codes. Cryptology ePrint Archive, Paper 2022/1014. https://eprint.iacr.org/2022/1014 https://eprint.iacr.org/2022/1014.

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 6 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 $|Y|$.

[6] 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.

[7] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2015. Function Secret Sharing. In *Advances in Cryptology - EUROCRYPT 2015*, Elisabeth Oswald and Marc Fischlin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–367.

[8] 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.

[9] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2019. Secure Computation with Preprocessing via Function Secret Sharing. Cryptology ePrint Archive, Paper 2019/1095. https://eprint.iacr.org/2019/1095 https://eprint.iacr.org/2019/1095.

[10] 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

[11] 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.

[12] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. 2018. PIR-PSI: Scaling Private Contact Discovery. Cryptology ePrint Archive, Paper 2018/579. https://eprint.iacr.org/2018/579 https://eprint.iacr.org/2018/579.

[13] 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.

[14] 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.

[15] 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.

[16] 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

[17] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. 2009. More Robust Hashing: Cuckoo Hashing with a Stash. *SIAM J. Comput.* 39, 4 (dec 2009), 1543–1561.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] Kevin Yeo. 2023. Cuckoo Hashing in Cryptography: Optimal Parameters, Robustness and Applications. In *Advances in Cryptology – CRYPTO 2023 (Lecture Notes in Computer Science)*, Helena Handschuh and Anna Lysyanskaya (Eds.). Springer Nature Switzerland, Cham, 197–230. https://doi.org/10.1007/978-3-031-38551-3_7

A SECURITY PROOFS OF DMPF SCHEMES

In this section we show that the big-state and OKVS-based DMPF schemes are computationally secure. We prove by first showing that the DMPF paradigm is computationally secure when the methods GenCW and GenConvCW satisfy some security requirements called hiding, and then arguing that the realizations of methods in both the big-state DMPF and the OKVS-based DMPF satisfy hiding.

A.1 The DMPF paradigm is secure under special conditions

We first define the security notion called hiding for GenCW and GenConvCW, and show that the DMPF paradigm in Figure 1 is secure when GenCW and GenConvCW are hiding. For simplicity, assume we're working with multi-point functions with exactly t accepting inputs. If there are less than t accepting inputs, simply pad arbitrary distinct inputs to A and pad B with 0. Denote a the list obtained by applying a function f to a list L of strings in the domain of f as $f(L) = (f(x))_{x \in L}$.

We say that a realization of GenCW is hiding if it gives no information about the multi-point function $f_{A,B}$. It is defined using an indistinguishability-based definition as follows:

Definition 11 (Computationally hiding of GenCW). *GenCW is computationally hiding if* $\forall i \in [n]$, $\forall A \neq A'$,

$$\{CW \leftarrow \mathsf{GenCW}(i, A, V^{(i-1)})\}_{V^{(i-1)} \leftarrow (\{0,1\}^{\lambda+2l})^{\otimes |A^{(i-1)}|}}$$

$$\approx_c \{CW \leftarrow \mathsf{GenCW}(i, A', V'^{(i-1)})\}_{V'^{(i-1)} \leftarrow (\{0,1\}^{\lambda+2l})^{\otimes |A'^{(i-1)}|}}$$

where $A^{(i-1)}$ denotes the set of length-$(i-1)$ prefixes of A.

Similarly we define a realization of GenConvCW to be hiding if it gives no information about $f_{A,B}$:

Definition 12 (Computationally hiding of GenConvCW). GenConvCW *is computationally hiding if* $\forall i \in [n]$, $\forall (A, B) \neq$

$(A', B'),$

$$\{CW \leftarrow \mathsf{GenConvCW}(A, B, \Delta g, \mathsf{sign}_0^{(n)}, \mathsf{sign}_1^{(n)})\}_{\substack{\Delta g \leftarrow \mathbb{G}^t \\ \mathsf{sign}_0 \leftarrow (\{0,1\}^l)^{\otimes t} \\ \mathsf{sign}_1 \leftarrow (\{0,1\}^l)^{\otimes t}}}$$

$$\approx_c \{CW \leftarrow \mathsf{GenConvCW}(A', B', \Delta g, \mathsf{sign}_0^{(n)}, \mathsf{sign}_1^{(n)})\}_{\substack{\Delta g \leftarrow \mathbb{G}^t \\ \mathsf{sign}_0 \leftarrow (\{0,1\}^l)^{\otimes t} \\ \mathsf{sign}_1 \leftarrow (\{0,1\}^l)^{\otimes t}}}$$

Next we show that if GenCW and GenConvCW in the DMPF paradigm Figure 1 are computationally hiding, then the DMPF is computationally secure.

Lemma 13. *Suppose* Initialize *set* $\mathsf{seed}_b^{(0)}$ *to be random strings for* $b = 0, 1$. *Suppose* GenCW *and* GenConvCW *in Figure 1 are computationally hiding with distinguishing advantage* $\epsilon_{\mathsf{GenCW}}$ *and* $\epsilon_{\mathsf{GenConvCW}}$ *respectively. Let* ϵ_G *and* $\epsilon_{G_{\mathsf{conv}}}$ *denote the distinguishing advantage of the PRG* G *and* G_{conv}. *Then the DMPF scheme is* ϵ*-secure against any n.u.p.p.t. adversary, where* $\epsilon = 2tn\epsilon_G + 2t\epsilon_{G_{\mathsf{conv}}} + n\epsilon_{\mathsf{GenCW}} + \epsilon_{\mathsf{GenConvCW}}$.

Proof. Formally, we show that $\forall b \in \{0,1\}, \forall (A,B) \neq (A', B')$,

$$\{k_b \leftarrow \mathsf{Gen}(1^\lambda, \hat{f}_{A,B})\} \approx_c \{k'_b \leftarrow \mathsf{Gen}(1^\lambda, \hat{f}_{A',B'})\}$$

with distinguishing advantage at most $\epsilon = 2tn\epsilon_G + 2t\epsilon_{G_{\mathsf{conv}}} + n\epsilon_{\mathsf{GenCW}} + \epsilon_{\mathsf{GenConvCW}}$. Fix an arbitrary b and (A,B), (A',B') such that $(A,B) \neq (A',B')$. By symmetry we may assume $b = 0$. Parse $k_0 = (\mathsf{seed}_0^{(0)}, \mathsf{sign}_0^{(0)}, CW^{(1)}, \ldots, CW^{(n+1)})$ and $k'_b = (\mathsf{seed}_0'^{(0)}, \mathsf{sign}_0'^{(0)}, CW'^{(1)}, \ldots, CW'^{(n+1)})$. We prove the distributions of k_0 and k'_0 are indistinguishable by a sequence of standard hybrid arguments.

Note that $\{\mathsf{seed}_0^{(0)}, \mathsf{sign}_0^{(0)}\} = \{\mathsf{seed}_0'^{(0)}, \mathsf{sign}_0'^{(0)}\}$ since the Initialize processes are the same. For $i \in [n]$, $\mathsf{GenCW}(1^\lambda, \hat{f}_{A,B})$ having the $\mathsf{seed}_b^{(i-1)}$ and $\mathsf{sign}_b^{(i-1)}$ strings of the previous layer for $b = 0, 1$, computes as the following:

(1) First determine the offset value list V by i, A and a substring of $G(\mathsf{seed}_0^{(i-1)}) \oplus G(\mathsf{seed}_1^{(i-1)})$. We denote the indices of this substring as T.
(2) Then invoke $\mathsf{GenCW}(i, A, V)$ to get $CW^{(i)}$.
(3) For $b = 0, 1$, compute the lists $\mathsf{seed}_b^{(i)}$ and $\mathsf{sign}_b^{(i)}$ by adding a correction (determined by $i, A, CW^{(i)}, \mathsf{sign}^{(i-1)}$) to a substring of $G(\mathsf{seed}_b^{(i-1)})$. Denote the indices of this substring as S (it is the same set of indices for $b = 0, 1$).

We note several crucial points from the above process: First, the computation only dependends on $i, A, \mathsf{seed}_b^{(i-1)}$ and $\mathsf{sign}_b^{(i-1)}$ for $b = 0, 1$. Second, since $CW^{(i)}$ is computed by $\mathsf{GenCW}(i, A, V)$ and according to the construction of V, the correction in the last step is only dependent to $i, A, \mathsf{sign}^{(i-1)}$ and a substring of $G(\mathsf{seed}_0^{(i-1)}) \oplus G(\mathsf{seed}_1^{(i-1)})$ indexed by T. Next, note that the set of seed indices in S does note intersect with T. Therefore, when G is a PRG we have the joint distribution of V and $\mathsf{seed}_b^{(i)}$ is pseudorandom.

Now we construct the following hybrid distributions for each layer $i \in [n]$:

- $\mathsf{Hyb}_i^0 = \{\mathsf{seed}_0'^{(0)}, \mathsf{sign}_0'^{(0)}, (CW'^{(k)})_{1 \le k < i}, (CW^{(k)})_{i \le k \le n+1}\}$ where $\mathsf{seed}_0'^{(0)}, \mathsf{sign}_0'^{(0)}, (CW'^{(k)})_{1 \le k \le i-1}$ are generated by $\mathsf{Gen}(1^\lambda, \hat{f}_{A',B'})$, $\mathsf{seed}_1^{(i-1)}$ is set to truly random with the same length as $A^{(i-1)}$, $\mathsf{sign}_1^{(i-1)}$ is set to satisfy an arbitrary desired correlation with $\mathsf{sign}_0'^{(i-1)}$, and $(CW^{(k)})_{i \le k \le n+1}$ are generated by $\mathsf{Gen}(1^\lambda, \hat{f}_{A,B})$ with the previous state being $\mathsf{seed}_0'^{(0)}, \mathsf{sign}_0'^{(0)}, (CW'^{(k)})_{1 \le k \le i-1}, \mathsf{seed}_1^{(i-1)}, \mathsf{sign}_1^{(i-1)}$.
- $\mathsf{Hyb}_i^1 = \{\mathsf{seed}_0'^{(0)}, \mathsf{sign}_0'^{(0)}, (CW'^{(k)})_{1 \le k < i}, (CW^{(k)})_{i \le k \le n+1}\}$ where $\mathsf{seed}_0'^{(0)}, \mathsf{sign}_0'^{(0)}, (CW'^{(k)})_{1 \le k \le i-1}$ are generated by $\mathsf{Gen}(1^\lambda, \hat{f}_{A',B'})$. To generate the remaining components, $\mathsf{Gen}(1^\lambda, \hat{f}_{A,B})$ sets $\mathsf{sign}_1^{(i-1)}$ to satisfy the arbitrary desired correlation with $\mathsf{sign}_0'^{(i-1)}$, and substitutes the use of $G(\mathsf{seed}_1^{(i-1)})$ with a truly random string. This makes $V^{(i-1)}$ a length-$|A^{(i-1)}|$ list of truly random $(\lambda + 2l)$-bit strings, and $\mathsf{seed}_1^{(i)}$ will be computed by adding a correction to a truly random string s. Then $\mathsf{Gen}(1^\lambda, \hat{f}_{A,B})$ with the previous state being $\mathsf{seed}_0'^{(0)}, \mathsf{sign}_0'^{(0)}, (CW'^{(k)})_{1 \le k \le i-1}, V^{(i-1)}, \mathsf{sign}_1^{(i-1)}, s$.
- $\mathsf{Hyb}_i^2 = \{\mathsf{seed}_0'^{(0)}, \mathsf{sign}_0'^{(0)}, (CW'^{(k)})_{1 \le k \le i}, (CW^{(k)})_{i < k \le n+1}\}$ where $\mathsf{seed}_0'^{(0)}, \mathsf{sign}_0'^{(0)}, (CW'^{(k)})_{1 \le k \le i-1}$ are generated by $\mathsf{Gen}(1^\lambda, \hat{f}_{A',B'})$. To generate the remaining components, set $\mathsf{sign}_1^{(i-1)}$ to satisfy the arbitrary desired correlation with $\mathsf{sign}_0'^{(i-1)}$, sample a length-$|A'^{(i-1)}|$ list $V'^{(i-1)}$ of truly random $(\lambda + 2l)$-bit strings, and generate $CW'^{(i)} \leftarrow \mathsf{GenCW}(i, A', V'^{(i-1)})$. Then invoke Correct to compute $\mathsf{seed}_0^{(i)}$ and $\mathsf{sign}_0^{(i)}$ of length $|A^{(i)}$ in $\mathsf{Gen}(1^\lambda, \hat{f}_{A,B})$. Set $\mathsf{seed}_1^{(i)}$ to be a length-$|A^{(i)}|$ list of truly random strings, $\mathsf{sign}_1^{(i)}$ to satisfy the desired correlation with $\mathsf{sign}_0^{(i)}$, and generate $(CW^{(k)})_{i < k \le n+1}$ by $\mathsf{Gen}(1^\lambda, \hat{f}_{A,B})$ with the previous state being $\mathsf{seed}_0'^{(0)}, \mathsf{sign}_0'^{(0)}, (CW'^{(k)})_{1 \le k \le i}, V^{(i-1)}, \mathsf{seed}_1^{(i-1)}, \mathsf{sign}_1^{(i-1)}$.

For the convert layer, we construct the following hybrid distributions:

- $\mathsf{Hyb}_{n+1}^0 = \{\mathsf{seed}_0'^{(0)}, \mathsf{sign}_0'^{(0)}, (CW'^{(k)})_{1 \le k \le n}, CW^{(n+1)}\}$ where $\mathsf{seed}_0'^{(0)}, \mathsf{sign}_0'^{(0)}, (CW'^{(k)})_{1 \le k \le n}$ are generated by $\mathsf{Gen}(1^\lambda, \hat{f}_{A',B'})$. $\mathsf{seed}_0^{(n)}$ and $\mathsf{sign}_0^{(n)}$ are computed by $\mathsf{Gen}(1^\lambda, \hat{f}_{A,B})$ but with the previous state being $\mathsf{seed}_0'^{(0)}, \mathsf{sign}_0'^{(0)}, (CW'^{(k)})_{1 \le k \le n}$. $\mathsf{seed}_1^{(n)}$ is set to a length-t list with truly random λ-bit strings, $\mathsf{sign}_1^{(n)}$ is set to satisfy the desired correlation with $\mathsf{sign}_0'^{(i-1)}$, and $CW^{(n+1)}$ is generated by $\mathsf{GenConvCW}(A, B, G_{\mathsf{conv}}(\mathsf{seed}_0^{(n)}) - G_{\mathsf{conv}}(\mathsf{seed}_1^{(n)}), \mathsf{sign}_0^{(n)}, \mathsf{sign}_1^{(n)})$.
- $\mathsf{Hyb}_{n+1}^1 = \{\mathsf{seed}_0'^{(0)}, \mathsf{sign}_0'^{(0)}, (CW'^{(k)})_{1 \le k \le n}, CW^{(n+1)}\}$ where $\mathsf{seed}_0'^{(0)}, \mathsf{sign}_0'^{(0)}, (CW'^{(k)})_{1 \le k \le n}$ are generated by $\mathsf{Gen}(1^\lambda, \hat{f}_{A',B'})$. $\mathsf{seed}_0^{(n)}$ and $\mathsf{sign}_0^{(n)}$ are computed by $\mathsf{Gen}(1^\lambda, \hat{f}_{A,B})$ but with the previous state being $\mathsf{seed}_0'^{(0)}$,

$\mathsf{sign}_0'^{(0)}$, $(CW'^{(k)})_{1\le k\le n}$. Sample a length-$t$ list Δg of random $\mathbb{G}$ elements, set $\mathsf{sign}_1^{(n)}$ to satisfy the desired correlation with $\mathsf{sign}_0'^{(i-1)}$, and generate $CW^{(n+1)}$ by $\mathsf{GenConvCW}(A, B, \Delta g, \mathsf{sign}_0^{(n)}, \mathsf{sign}_1^{(n)})$.

- $\mathsf{Hyb}_{n+1}^2 = \{\mathsf{seed}_0'^{(0)}, \mathsf{sign}_0'^{(0)}, (CW'^{(k)})_{1\le k\le n+1}\}$ where $\mathsf{seed}_0'^{(0)}, \mathsf{sign}_0'^{(0)}, (CW'^{(k)})_{1\le k\le n}$ are generated by $\mathsf{Gen}(1^\lambda, \hat{f}_{A',B'})$. $\mathsf{seed}_0^{(n)}$ and $\mathsf{sign}_0^{(n)}$ are computed by $\mathsf{Gen}(1^\lambda, \hat{f}_{A,B})$ but with the previous state being $\mathsf{seed}_0'^{(0)}$, $\mathsf{sign}_0'^{(0)}$, $(CW'^{(k)})_{1\le k\le n}$. Sample a length-$t$ list Δg of random $\mathbb{G}$ elements, set $\mathsf{sign}_1^{(n)}$ to satisfy the desired correlation with $\mathsf{sign}_0'^{(i-1)}$, and generate $CW'^{(n+1)}$ by $\mathsf{GenConvCW}(A', B', \Delta g, \mathsf{sign}_0^{(n)}, \mathsf{sign}_1^{(n)})$.
- $\mathsf{Hyb}_{n+1}^3 = \{\mathsf{seed}_0'^{(0)}, \mathsf{sign}_0'^{(0)}, (CW'^{(k)})_{1\le k\le n+1}\}$ generated by $\mathsf{Gen}(1^\lambda, \hat{f}_{A',B'})$.

Next we argue that $\mathsf{Hyb}_1^0 \approx_c \mathsf{Hyb}_{n+1}^3$. Since Hyb_1^0 is the distribution of k_0 and Hyb_{n+1}^3 is the distribution of k_0', this proves the distributions of k_0 and k_0' are computationally indistinguishable.

For all $i \in [n]$, $\mathsf{Hyb}_i^0 \approx_c \mathsf{Hyb}_i^1$ with distinguishing advantage at most $t\epsilon_G$, since the only difference between the two distributions is the substitution of $G(\mathsf{seed}_1^{(i-1)})$ with truly random strings, which contains $|A^{(i-1)}| < t$ invocations of G. $\mathsf{Hyb}_i^1 \approx_c \mathsf{Hyb}_i^2$ with distinguishing advantage at most $\epsilon_{\mathsf{GenCW}}$ by computational hiding of GenCW, since the only difference between the two distributions is that Hyb_i^2 replaces the output of $\mathsf{GenCW}(i, A, V^{(i-1)})$ by $\mathsf{GenCW}(i, A', V'^{(i-1)})$ where both $V^{(i-1)}$ and $V'^{(i-1)}$ are truly random. In the end $\mathsf{Hyb}_i^2 \approx \mathsf{Hyb}_{i+1}^0$ with distinguishing advantage at most $t\epsilon_G$, since the only difference between the two distributions is the substitution of truly random strings with $G(\mathsf{seed}_1'^{(i-1)})$ (happened in $\mathsf{Gen}(1^\lambda, \hat{f}_{A',B'})$, which is the inverse substitution happened when switching Hyb_i^0 to Hyb_i^1), which contains $|A'^{(i-1)}| < t$ invocations of G. In conclusion, $\mathsf{Hyb}_1^0 \approx_c \mathsf{Hyb}_{n+1}^0$ with distinguishing advantage at most $2tn\epsilon_G + n\epsilon_{\mathsf{GenCW}}$.

Through a similar argument, $\mathsf{Hyb}_{n+1}^0 \approx_c \mathsf{Hyb}_{n+1}^1$, $\mathsf{Hyb}_{n+1}^1 \approx_c \mathsf{Hyb}_{n+1}^2$, and $\mathsf{Hyb}_{n+1}^2 \approx_c \mathsf{Hyb}_{n+1}^3$, with distinguishing advantage at most $t\epsilon_{G_{\mathsf{conv}}}$, $\epsilon_{\mathsf{GenConvCW}}$, and $t\epsilon_{G_{\mathsf{conv}}}$ respectively.

Henceforth, $\mathsf{Hyb}_1^0 \approx_c \mathsf{Hyb}_{n+1}^3$ with distinguishing advantage at most $\epsilon = 2tn\epsilon_G + 2t\epsilon_{G_{\mathsf{conv}}} + n\epsilon_{\mathsf{GenCW}} + \epsilon_{\mathsf{GenConvCW}}$. □

A.2 The big-state DMPF scheme is secure

In this section we show that the big-state realization of DMPF in Figure 2 satisfies the conditions: (1) Initialize set $\mathsf{seed}_0^{(0)}$ and $\mathsf{seed}_1^{(1)}$ to random strings. (2) GenCW is computationally hiding. (3) GenConvCW is computationally hiding. Combined with Lemma 13, it can be concluded that the big-state DMPF scheme is computationally secure.

Condition (1) is direct from the construction.

For condition (2), note that when V is a truly random list, $\mathsf{GenCW}(i, A, V)$ in Figure 2 computes each entry of CW by adding up a random string with one of $0^{\lambda+2t}$, $0^\lambda\|e_d\|0^t$, $0^\lambda\|0^t, e_d$ and $0^\lambda\|e_d\|e_{d+1}$ for some d, determined by i, A. Therefore, the resulting CW is a length-t list of random strings, independent of i, A. Hence $\epsilon_{\mathsf{GenCW}} = 0$.

For condition (3), when Δ_g is a truly random list, $\mathsf{GenConvCW}(A, B, \Delta g, \mathsf{sign}_0^{(n)}, \mathsf{sign}_1^{(n)})$ computes $CW[k] = (-1)^{\mathsf{sign}_0^{(n)}[k][k]}(\Delta g[k] - B[k])$, which is a random group element. Therefore the resulting CW is a length-t list of random group elements, independent of A, B. Hence $\epsilon_{\mathsf{GenConvCW}} = 0$.

The above arguments, combined with Lemma 13, concludes that the big-state DMPF is computationally secure.

Theorem 14 (Big-state DMPF is secure). *Let ϵ_G and $\epsilon_{G_{\mathsf{conv}}}$ denote the distinguishing advantage of the PRG G and G_{conv} respectively. Then the big-state DMPF scheme in Figure 2 is ϵ-secure against any n.u.p.p.t. adversary, where $\epsilon = 2tn\epsilon_G + 2t\epsilon_{G_{\mathsf{conv}}}$.*

A.3 The OKVS-based DMPF scheme is secure

In this section we argue that the OKVS-based realization of DMPF in Figure 3 satisfies the conditions: (1) Initialize set $\mathsf{seed}_0^{(0)}$ and $\mathsf{seed}_1^{(1)}$ to random strings. (2) GenCW is computationally hiding. (3) GenConvCW is computationally hiding. Combined with Lemma 13, it can be concluded that the OKVS-based DMPF scheme is computationally secure.

Condition (1) is direct from the construction.

For condition (2), note that when V is a truly random list, $\mathsf{GenCW}(i, A, V)$ in Figure 3 pads V to be of length t, and updates each entry of V by adding up a random string with one of $0^{\lambda+2}$, $0^\lambda\|10$, $0^\lambda\|01$ and $0^\lambda\|11$ for some d, determined by i, A. Therefore, the resulting V is a length-t list of random strings, independent of i, A. By the obliviousness of the OKVS scheme, CW as an encoding of keys from A and values from V should be computationally indistinguishable from an encoding of keys from A' and values from V. Hence $\epsilon_{\mathsf{GenCW}} = \epsilon_{\mathsf{OKVS}_i}$, the security of the corresponding OKVS_i scheme.

For condition (3), when Δ_g is a truly random list, $\mathsf{GenConvCW}(A, B, \Delta g, \mathsf{sign}_0^{(n)}, \mathsf{sign}_1^{(n)})$ computes a value list V of group elements by $V[k] = (-1)^{\mathsf{sign}_0^{(n)}[k][k]}(\Delta g[k] - B[k])$, which is a random group element. Therefore V is a length-t list of random group elements independent of A, B. Hence $\epsilon_{\mathsf{GenConvCW}} = \epsilon_{\mathsf{OKVS}_{\mathsf{conv}}}$, the security of the corresponding $\mathsf{OKVS}_{\mathsf{conv}}$ scheme.

The above arguments, combined with Lemma 13, concludes that the OKVS-based DMPF is computationally secure.

Theorem 15 (OKVS-based DMPF is secure). *Let ϵ_G and $\epsilon_{G_{\mathsf{conv}}}$ denote the distinguishing advantage of the PRG G and G_{conv} respectively. Suppose in the OKVS-based DMPF scheme in Figure 3, for $1 \le i \le n$ the OKVS_i scheme is $\epsilon_{\mathsf{OKVS}_i}$-oblivious, and the $\mathsf{OKVS}_{\mathsf{conv}}$ scheme is $\epsilon_{\mathsf{OKVS}_{\mathsf{conv}}}$-oblivious. Then the big-state DMPF scheme in Figure 2 is ϵ-secure against any n.u.p.p.t. adversary, where $\epsilon = 2tn\epsilon_G + 2t\epsilon_{G_{\mathsf{conv}}} + \sum_{i=1}^n \epsilon_{\mathsf{OKVS}_i} + \epsilon_{\mathsf{OKVS}_{\mathsf{conv}}}$.*

Construction 6 (RR22[14, 21]). *Suppose $\mathcal{V} = \mathbb{F}$ is a field. Set $\mathsf{row}_r(k) := \mathsf{row}_r^{\mathsf{sparse}}(k)||\mathsf{row}_r^{\mathsf{dense}}(k)$ where $\mathsf{row}_r^{\mathsf{sparse}}(k)$ outputs a uniformly random weight-w vector in $\{0,1\}^{m_1}$, and $\mathsf{row}_r^{\mathsf{dense}}(k)$ outputs a short dense vector in $\mathbb{F}^{m_2}$.*

- *$\mathsf{Encode}_r(\{(k_i, v_i)\}_{1\le i\le t}) \to P$ where P is randomly chosen from the solutions of the system $\{\langle \mathsf{row}_r(k_i), P\rangle = v_i\}_{1\le i\le t}$,* *solved by the triangulation algorithm in [21]. If the system has no solution then output* $\perp$.

- $\mathsf{Decode}_r(P, k) \to \langle \mathsf{row}_r(k), P\rangle$.

We denote $m_1 = et$*, where* e *is an expansion parameter indicating the rough blowup to store* t *pairs. Practically the number of dense columns* m_2 *is 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 $\{\mathsf{row}_r(k_i)\}_{1\le i\le t}$ 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 [21], 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 $\alpha := 0.55\log_2 t + 0.093w^3 - 1.01w^2 + 2.92w - 0.13$.
- $e := e^* + 2^{-\alpha}(\lambda_{\mathsf{stat}} + 9.2)$.
- $\hat{g} := \frac{\lambda_{\mathsf{stat}}}{(w-2)\log_2(et)}$.

Yaxin: Fix t and λ_{stat}, we want to find the best choice of w. The adavantageous choices of w in [21] 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 [21] 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 [14, 21] 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 $m_2 = \hat{g} + \lambda_{\mathsf{stat}}$, as proposed in [14, 21]. As indicated by the empirical results in [21], this binary scheme is usually not as efficient as the original design. Therefore we mostly refer to construction 6.

According to the comparison in [2] of the RR22-OKVS (construction 6) and the RB-OKVS (construction 5) 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?

B OTHER TABLES

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009

Table 5: The relations among the number of balls t, the number of hash functions w, the expansion parameter $e = m/t$ where m denotes the number of buckets and the security parameters λ_{stat} in cuckoo hashing with bucket size 1 and no stash.

	Type	t	λ_{stat}	w	$e = m/t$
[23, Theorem 1]†	Asymptotic			$O(\sqrt{\lambda_{\text{stat}} \log t})$	$O(1)$
[11]	Asymptotic			3	$O(\lambda_{\text{stat}} + \log t)$
[12, Appendix B]	Empirical	$t \geq 4$	$\lambda_{\text{stat}} = a_t \cdot e - b_t - \log t$ $a_t = 123.5 \cdot \mathsf{CDF}_{\mathsf{Normal}}(x = t, \mu = 6.3, \sigma = 2.3)$ $b_t = 120 \cdot \mathsf{CDF}_{\mathsf{Normal}}(x = t, \mu = 6.45, \sigma = 2.18)$	3‡	e
[11] simplifying the above	Empirical	$t \geq 30$*	$\lambda_{\text{stat}} = 123.5e - 120 - \log t$	3	e
[10]**	Empirical	11041	40 ($\lambda_{\text{stat}} = 124.4e - 144.6$)	3	$m = 2^{14}$, $e \approx 1.5$
		5535	40 ($\lambda_{\text{stat}} = 125e - 145$)	3	$m = 2^{13}$, $e \approx 1.5$

† $O(\sqrt{\lambda_{\text{stat}} \log t})$ queries to the hash functions and supposes the hash functions from a $O(t\sqrt{\lambda_{\text{stat}} \log t})$-wise independent hash function family.
‡ Parameters are only slightly different for $w > 3$.
* Should extend to smaller t like $t = 16, 25$.
** It first fixes $m = 2^{13}, 2^{14}$ and then computes the correlation between λ_{stat} and e.