

Pseudorandom Correlation Functions for Multiparty Beaver Triples from Sparse LPN

Sebastian Hasler  and Pascal Reisert 

University of Stuttgart
Stuttgart, Germany

{sebastian.hasler,pascal.reisert}@sec.uni-stuttgart.de

Abstract. We construct a pseudorandom correlation function (PCF) for oblivious linear evaluation (OLE) from sparse LPN over any finite field. The *programmability* property of our PCF implies a PCF for any multiparty degree-two correlation, e.g., Beaver triples. Our PCF is the first PCF for degree-two correlations from a well-established cryptographic assumption, apart from (inefficient) generic PCFs based on homomorphic secret sharing or fully homomorphic encryption. Our PCF outperforms the previously fastest PCF for Beaver triples (Boyle et al., Crypto 2022) by 3.2–28×.

We build on the recent pseudorandom correlation generator (PCG) by Miao et al. (Asiacrypt 2025) and extend it to a PCF using a recursive approach similar to Braun et al. (Asiacrypt 2025). Moreover, we extend these techniques to support *authenticated* degree-two correlations in the important two-party case.

Keywords: Pseudorandom Correlation Functions · Multiparty Computation · Beaver Triples · Oblivious Linear Evaluation · Sparse LPN

1 Introduction

Many secure multiparty computation (MPC) protocols use a two-phase approach, consisting of an input-independent preprocessing phase and an input-dependent online phase. In the preprocessing phase, correlated randomness needs to be generated, e.g. in the form of Beaver triples [Bea92] which are used for multiplications in the online phase. In many protocols, the online phase uses only information-theoretic or symmetric-key primitives and is thus very efficient. For instance, a Boolean circuit can be evaluated with only one bit of communication per party per (non-linear) gate, consuming one Beaver triple from the preprocessing per (non-linear) gate. In contrast, the preprocessing is often much more costly, as it relies on public-key primitives such as oblivious transfer (OT) or homomorphic encryption (HE). Therefore, reducing the cost of preprocessing is the main way to optimize MPC further.

The efficiency of the offline phase can be measured in terms of runtime and communication complexity. While the offline phase is often not time-critical, since it can run well before the actual input data (later used in the online phase) becomes available, the communication complexity is often most critical. Apart from its impact on the overall performance, especially in setups with limited traffic and bandwidth rate restrictions, it might also lead to high financial costs if the MPC protocols are deployed on paid online infrastructure.

For a long time, preprocessing protocols, such as [DPSZ12, KOS16, KPR18], required communication linear in the size of the circuit. This changed with [BCGI18], where the notion of a pseudorandom correlation generator (PCG) was introduced. A PCG generates (small) correlated seeds, one per party, such that each party can locally, i.e., without further interaction, expand her seed into a larger batch of the desired correlation, e.g., Beaver triples, which can later be used in the online phase. In [BCGI18] it is shown that the seeds can be obliviously generated using any MPC protocol, thus obtaining a preprocessing with sublinear communication.

A drawback of PCGs is that the seeds need to be expanded as a one-off operation and the expanded randomness needs to be stored. Furthermore, a PCG provides only an asymptotically polynomial amount of correlated randomness. A natural extension of PCGs are pseudorandom correlation functions (PCFs) [BCG⁺20a]. A PCF consists of a key generation algorithm $\text{PCF.Gen}(1^\lambda)$ that outputs small keys $(k_\sigma^{\text{PCF}})_\sigma$, one per party, and a non-interactive evaluation algorithm $\text{PCF.Eval}_{k_\sigma^{\text{PCF}}}(\cdot)$ that allows the parties to locally and repeatedly query an instance of the target correlation, e.g., shares of a Beaver triple. When the evaluation is fast, this removes the need to precompute and store all of the correlated randomness required for the online phase. Additionally, PCFs usually allow a superpolynomial number of queries.

So far, PCGs for degree-two correlations (such as Beaver triples) have been constructed from a variety of *learning parity with noise* (LPN)-style assumptions, including standard LPN [BCG⁺19b], sparse LPN [MMST25], and structured LPN variants [BCG⁺20b, BCCD23, LXY25]. In contrast, PCFs for degree-two correlations so far only exist based on rather non-standard LPN variants [BCG⁺20a, BCG⁺22] or using a generic approach of evaluating a weak pseudorandom function (PRF) under homomorphic secret sharing (HSS) or fully homomorphic encryption (FHE) [BCG⁺20a, HRK25]. The latter generic approach is usually considered inefficient.

New Pseudorandom Correlation Function. In this work, we present a new PCF for (multiparty) degree-two correlations, such as Beaver triples, from sparse LPN. Our construction is therewith the first (non-generic) PCF for degree-two correlations from a well-established assumption.

Similar to abovementioned prior works on PCGs and PCFs, we mainly target the (two-party) oblivious linear evaluation (OLE) correlation $((x_0, z_0), (x_1, z_1))$ where all entries are uniformly random, except for z_1 which is uniquely determined by $z_1 = x_0 \cdot x_1 - z_0$, implying that (z_0, z_1) are additive shares of $x_0 \cdot x_1$. Our PCF for OLE is *programmable*, which means that the private values x_0, x_1 can be “reused” across multiple PCF instances. This allows us to generate arbitrary multiparty degree-two correlations as a linear combination of OLE instances. For instance, generating M -party Beaver triples requires $M(M-1)$ OLE instances, one for each ordered pair of parties, by [BCG⁺20a, Theorem 10.6].

In the two-party setting, we optimize this by a factor of two, i.e., we provide a PCF that produces Beaver triples directly within a single instance, instead of using two instances of our PCF for OLE. We also show how to extend this PCF to arbitrary *authenticated* two-party degree-two correlations, such as authenticated two-party Beaver triples.

Furthermore, we provide instantiations of our PCFs with concrete parameters and performance evaluation in Section 4.4. When choosing concrete parameters, there is a trade-off between performance and expansion (i.e., the total size of the generated correlated randomness divided by the key size): A larger expansion results in reduced performance and vice versa.

Improved FSS Tensor Operation. Our PCF depends heavily on (two-party) function secret sharing (FSS). In particular, we need to share the tensor products of several sparse vectors. As a side-contribution, we reduce the key sizes of these FSS keys by roughly a factor of two, compared to blindly applying the FSS tensor operation from [BGI16]. Our tensor operation with reduced key size works for arbitrary FSS schemes that conform to our new notion of a *seedable* FSS scheme. In a seedable FSS scheme, the keys consist of a pseudorandom and independently choosable private part, and a common part. This form of the key generation is a very typical property of FSS schemes, including standard distributed point functions (DPFs) and distributed comparison functions (DCFs) [BGI16, BCG⁺21]. Abovementioned key size optimizations were already included in self-contained FSS scheme descriptions (such as the DPF from [BGI16], when viewed as a tensor product of single-bit-input point functions), but

our general notion of seedable FSS schemes, where the optimization can be applied, is useful when applying the tensor operation to existing seedable FSS schemes in a blackbox manner, like we do.

Contributions.

- We construct the first (non-generic) PCF for multiparty (or authenticated two-party) degree-two correlations, such as Beaver triples, from a well-established cryptographic assumption.
- We choose concrete parameters and evaluate the performance of our PCF. Our PCF outperforms the previously fastest PCF for two-party Beaver triples by 3.2–28× (see [Section 1.1](#)).
- We introduce a new notion of *seedable* FSS and reduce the key size in the FSS tensor operation for (the very common case of) seedable FSS schemes by roughly a factor of two.

1.1 Related Work

The pioneer work on PCFs, [\[BCG⁺20a\]](#), presented two instantiations: Firstly, a (conceptually simple but inefficient) generic construction based on evaluating a weak pseudorandom function (WPRF) under HSS (which can be instantiated based on spooky encryption [\[DHRW16\]](#)), and secondly, a PCF based on variable-density LPN (VDLPN). In [\[BCG⁺22\]](#), the efficiency was improved by using a different LPN variant, called expand-accumulate LPN (EALPN), instead. However, both of these assumptions are very novel, as, in fact, they have been first introduced in these works, respectively. In contrast, we base our PCF on a well-established LPN variant.

We show in [Table 1](#) that our PCF is currently the fastest PCF for Beaver triples, with a speedup of 3.2–28× compared to the prior best, [\[BCG⁺22\]](#). At the same time, our PCF has $\approx 9\times$ smaller keys. In that comparison, we target security for $n \approx 2^{30}$ queries, since this is the only output size for which [\[BCG⁺22\]](#) provides an efficiency analysis.

There also exist PCFs for vector OLE (VOLE) [\[OSY21, BCM⁺25\]](#) or (List)OT correlations [\[OSY21, BCM⁺24, CDD⁺24\]](#), but these works so far do not provide a PCF for (standard) OLE or Beaver triples. Our PCF uses techniques similar to [\[BCM⁺25\]](#), but applies them to OLE instead of VOLE, thus obtaining a PCF for degree-two correlations, such as Beaver triples, via the standard pairwise protocol or our two-party optimization.

From PCG to PCF as a Design Philosophy. Our work uses the design philosophy of starting with a PCG and extending it to a PCF. This philosophy has been employed before, although there is not a generic way to do this with any PCG. That is, in general, one cannot instantiate a PCG with superpolynomial output size to obtain a PCF, since we demand from a PCF that it can be evaluated (i.e., produce an instance of the target correlation) efficiently. Therefore, when using a PCG as a starting point, care must be taken that it can be expanded *incrementally*, i.e., any single entry of the output can be computed efficiently.

The PCFs from [\[BCG⁺20a, BCG⁺22\]](#) can be seen as extensions of prior PCGs based on dual LPN [\[BCGI18, BCG⁺19b\]](#). However, these extensions employ non-standard assumptions, in order to allow for efficient incremental evaluation. Namely, dual LPN with its uniformly random matrix is replaced by non-standard LPN variants (VDLPN in [\[BCG⁺20a\]](#) and EALPN in [\[BCG⁺22\]](#)) that use a more structured matrix, allowing to compute any single entry of the matrix-vector product much more efficiently.

Similarly, the PCF from [\[BCM⁺25\]](#) can be regarded as an extension of the PCG based on sparse (primal) LPN from [\[BCGI18\]](#). As noted in [\[BCGI18\]](#), any primal LPN variant has an output size at most quadratic in the seed size. [\[BCM⁺25\]](#) overcomes this limitation by

Table 1: Comparison of PCFs for two-party Beaver triples over \mathbb{F}_2 , targetting security for $n \approx 2^{30}$ queries

	Assumption [#]	Key Sizes (GiB)	Triples/s
[BCG ⁺ 20a]	LWE	χ^*	χ^*
[BCG ⁺ 20a]	VDLPN	χ^*	χ^*
[BCG ⁺ 22]	EALPN	15 [†]	91 [‡]
[HRK25]	Ring-LWE	χ^{\S}	9
Ours (most conservative)	Sparse LPN	1.68	293 [‡]
Ours (optimistic)	Sparse LPN	1.68	2551 [‡]

[#] All PCFs here additionally assume the random oracle model.

^{*} [BCG⁺20a] did not provide an efficiency analysis for Beaver triples.

[†] Key sizes based on their conservative parameters, since the more aggressive parameters were shown to be insecure in [RRT23].

[‡] Runtime calculated based on the number of PRG evaluations, assuming (as in [BCM⁺25]) 20.8 CPU cycles per AES call and 3.8 GHz clock frequency.

[§] [HRK25] did not provide the key size of their PCF, but rough calculations indicate that it's in the Gigabytes.

iterating the construction from [BCGI18], to obtain a larger and larger output size with each iteration. To allow for efficient incremental evaluation, the iteration is then replaced by a recursion that exploits the sparsity of the LPN matrix. In contrast to abovementioned PCFs from EALPN/VDLPN, all of this doesn't require any modification to the underlying sparse LPN assumption.

In this work, we extend the approach by [BCM⁺25], that only provided a PCF for VOLE, to (standard) OLE, Beaver triples, and more. At a single iteration/recursion level, our PCF looks similar to the PCG from [MMST25], in that we compute the component-wise product of two sparse LPN instances. However, [MMST25] achieves further compression of the seeds via dual LPN, while we instead use abovementioned recursive approach, i.e., we compress the seeds by recursively applying our construction, in order to obtain superpolynomial expansion.

1.2 Technical Overview

Recall that a batch of OLE correlations has the form $((\mathbf{x}_0, \mathbf{z}_0), (\mathbf{x}_1, \mathbf{z}_1))$, where $\mathbf{z}_0 + \mathbf{z}_1 = \mathbf{x}_0 \odot \mathbf{x}_1$. In [MMST25], the authors construct a PCG for OLE from sparse LPN for two parties P_σ ($\sigma \in \{0, 1\}$) as follows. Define $\mathbf{x}_\sigma := \mathbf{A}\mathbf{s}_\sigma + \mathbf{e}_\sigma$, where $\mathbf{s}_\sigma \in \mathbb{F}^n$ is a random vector of short length $n \ll m$, $\mathbf{e}_\sigma \in \mathbb{F}^m$ is a sparse vector, and $\mathbf{A} \in \mathbb{F}^{m \times n}$ is a public sparse matrix, i.e., the rows of \mathbf{A} are sparse. Then \mathbf{x}_σ is pseudorandom by the sparse LPN assumption (for suitably chosen parameters), when $\mathbf{s}_\sigma, \mathbf{e}_\sigma$ are kept secret from the other party. Given small PCG seeds containing $(\mathbf{s}_\sigma, \mathbf{e}_\sigma)$ (or a pseudorandom generator (PRG) seed from which $\mathbf{s}_\sigma, \mathbf{e}_\sigma$ are sampled), the party P_σ can locally compute \mathbf{x}_σ . If the PCG seed also contains shares of $\mathbf{s}_0 \otimes \mathbf{s}_1$, $\mathbf{e}_0 \otimes \mathbf{s}_1$, $\mathbf{s}_0 \otimes \mathbf{e}_1$, and $\mathbf{e}_0 \odot \mathbf{e}_1$, P_σ can additionally compute a share of the component-wise product $\mathbf{x}_0 \odot \mathbf{x}_1$. Namely, P_σ computes a share of the i -th entry

$$\begin{aligned} x_{0,i} \cdot x_{1,i} &= (\langle \mathbf{a}_i, \mathbf{s}_0 \rangle + e_{0,i}) \cdot (\langle \mathbf{a}_i, \mathbf{s}_1 \rangle + e_{1,i}) \\ &= \langle \mathbf{a}_i \otimes \mathbf{a}_i, \mathbf{s}_0 \otimes \mathbf{s}_1 \rangle + \langle \mathbf{a}_i, e_{0,i} \cdot \mathbf{s}_1 \rangle + \langle \mathbf{a}_i, \mathbf{s}_0 \cdot e_{1,i} \rangle + e_{0,i} \cdot e_{1,i}, \end{aligned} \quad (1)$$

of $\mathbf{x}_0 \odot \mathbf{x}_1$ by applying the public row vectors \mathbf{a}_i of \mathbf{A} and $\mathbf{a}_i \otimes \mathbf{a}_i$ to their shares of $\mathbf{s}_0 \otimes \mathbf{s}_1$, $\mathbf{e}_{0,i} \cdot \mathbf{s}_1$, $\mathbf{s}_0 \cdot \mathbf{e}_{1,i}$ and adding the shares (including the share of $e_{0,i} \cdot e_{1,i}$). The terms with \mathbf{e}_σ

are sparse (in our work, \mathbf{e}_σ is a concatenation of scaled unit vectors of equal length) and can be shared as a distributed multi-point function (DMPF) (in our case, as a concatenation of several standard DPFs [BGI16]).¹

In [MMST25], the shares of $\mathbf{s}_0 \otimes \mathbf{s}_1$ were further compressed by replacing one of the two \mathbf{s}_σ by a dual LPN instance. Essentially, this can be viewed as generating the shares of $\mathbf{s}_0 \otimes \mathbf{s}_1$ with another (dual LPN-based) PCG. In this work, we use a different approach.

Our Approach. Our first idea is that, using almost the same PCG seeds as above, we can actually compute shares of the *tensor product* $\mathbf{x}_0 \otimes \mathbf{x}_1$, instead of the component-wise product. Observe that the (i, j) -th entry² of that tensor product is

$$\begin{aligned} x_{0,i} \cdot x_{1,j} &= (\langle \mathbf{a}_i, \mathbf{s}_0 \rangle + e_{0,i}) \cdot (\langle \mathbf{a}_j, \mathbf{s}_1 \rangle + e_{1,j}) \\ &= \langle \mathbf{a}_i \otimes \mathbf{a}_j, \mathbf{s}_0 \otimes \mathbf{s}_1 \rangle + \langle \mathbf{a}_j, e_{0,i} \cdot \mathbf{s}_1 \rangle + \langle \mathbf{a}_i, e_{1,j} \cdot \mathbf{s}_0 \rangle + e_{0,i} \cdot e_{1,j}. \end{aligned} \quad (2)$$

Therefore, the only change is that, instead of $\mathbf{e}_0 \odot \mathbf{e}_1$, we need shares of $\mathbf{e}_0 \otimes \mathbf{e}_1$, which can similarly be compressed as a DMPF.

We can use this observation to generate the $\mathbf{s}_0 \otimes \mathbf{s}_1$ from even smaller seeds. Namely, instead of distributing shares of $\mathbf{s}_0 \otimes \mathbf{s}_1$ directly, we can generate them by applying the PCG construction recursively.³ We want to outline briefly how our recursive approach works and how Equation (2) is used.

Let L be a fixed number of levels, $\ell = 1, \dots, L$ the index for the ℓ -th level and $m^{(\ell)}$ the dimension of the level ℓ outputs. Moreover, let $\mathbf{A}^{(\ell)}$ be public matrices of size $m^{(\ell)} \times m^{(\ell-1)}$ for $1 \leq \ell \leq L$.

The parties start at an initial $\mathbf{s}_\sigma^{(0)}$ and compute $\mathbf{s}_\sigma^{(\ell)} := \mathbf{A}^{(\ell)} \mathbf{s}_\sigma^{(\ell-1)} + \mathbf{e}_\sigma^{(\ell)}$ for $\ell \geq 1$ using $\mathbf{e}_\sigma^{(\ell)}$ from their PCG seed. Additionally, for $1 \leq \ell < L$, the parties compute (using the same observation as in Equation (2)) a share of

$$\begin{aligned} \mathbf{s}_0^{(\ell)} \otimes \mathbf{s}_1^{(\ell)} &= (\mathbf{A}^{(\ell)} \mathbf{s}_0^{(\ell-1)} + \mathbf{e}_0^{(\ell)}) \otimes (\mathbf{A}^{(\ell)} \mathbf{s}_1^{(\ell-1)} + \mathbf{e}_1^{(\ell)}) \\ &= \underbrace{(\mathbf{A}^{(\ell)} \mathbf{s}_0^{(\ell-1)}) \otimes (\mathbf{A}^{(\ell)} \mathbf{s}_1^{(\ell-1)})}_{\boxed{1}} \\ &\quad + \underbrace{\mathbf{e}_0^{(\ell)} \otimes (\mathbf{A}^{(\ell)} \mathbf{s}_1^{(\ell-1)})}_{\boxed{2}} + \underbrace{(\mathbf{A}^{(\ell)} \mathbf{s}_0^{(\ell-1)}) \otimes \mathbf{e}_1^{(\ell)}}_{\boxed{3}} + \underbrace{\mathbf{e}_0^{(\ell)} \otimes \mathbf{e}_1^{(\ell)}}_{\boxed{4}}. \end{aligned} \quad (3)$$

Recall that shares of $((\mathbf{A}^{(\ell)} \mathbf{s}_0^{(\ell-1)}) \otimes (\mathbf{A}^{(\ell)} \mathbf{s}_1^{(\ell-1)}))_{i,j} = \langle \mathbf{a}_i^{(\ell)} \otimes \mathbf{a}_j^{(\ell)}, \mathbf{s}_0^{(\ell-1)} \otimes \mathbf{s}_1^{(\ell-1)} \rangle$ can be computed from shares of $\mathbf{s}_0^{(\ell-1)} \otimes \mathbf{s}_1^{(\ell-1)}$, i.e., from the previous level. Moreover, $\boxed{4}$ is a sparse vector, so it can be compressed as a DMPF and shared as part of the PCG seed.

The shares required for $\boxed{2}$ and $\boxed{3}$ are still missing. We outline the construction for $\boxed{2}$, and $\boxed{3}$ can then be treated analogously. First rewrite $(\mathbf{e}_0^{(\ell)} \otimes (\mathbf{A}^{(\ell)} \mathbf{s}_1^{(\ell-1)}))_{i,j} = \langle \mathbf{a}_j^{(\ell)}, e_{0,i}^{(\ell)} \cdot \mathbf{s}_1^{(\ell-1)} \rangle$ as before, i.e., it is enough to compute a share of $\mathbf{e}_0^{(\ell)} \otimes \mathbf{s}_1^{(\ell-1)}$. However, for $1 \leq \ell' < \ell$,

$$\mathbf{e}_0^{(\ell)} \otimes \mathbf{s}_1^{(\ell')} = \mathbf{e}_0^{(\ell)} \otimes (\mathbf{A}^{(\ell')} \mathbf{s}_1^{(\ell'-1)} + \mathbf{e}_1^{(\ell')}) = \mathbf{e}_0^{(\ell)} \otimes (\mathbf{A}^{(\ell')} \mathbf{s}_1^{(\ell'-1)}) + \mathbf{e}_0^{(\ell)} \otimes \mathbf{e}_1^{(\ell')} \quad (4)$$

¹ To regard $\mathbf{e}_\sigma \otimes \mathbf{s}_{1-\sigma}$ as a multi-point function, the input is the \mathbf{e}_σ -index i and the output is a whole vector $e_{\sigma,i} \cdot \mathbf{s}_{1-\sigma}$.

² The (i, j) -th entry is the coefficient of the tensor product of the i -th and j -th basis unit vectors in the corresponding standard basis. In terms of the Kronecker product this corresponds to the $(im + j)$ -th entry.

³ We remark that a similar but simpler and more straightforward recursion has been presented in the recent PCF for VOLE by Braun et al. [BCM⁺25].

shows that a party who possesses a share of $\mathbf{e}_0^{(\ell)} \otimes \mathbf{e}_1^{(\ell')}$ and of $\mathbf{e}_0^{(\ell)} \otimes \mathbf{s}_1^{(\ell'-1)}$ can compute a share of $\mathbf{e}_0^{(\ell)} \otimes \mathbf{s}_1^{(\ell')}$. Thus if P_σ has a share of $\mathbf{e}_0^{(\ell)} \otimes \mathbf{s}_1^{(0)}$ (plus $\mathbf{e}_0^{(\ell)} \otimes \mathbf{e}_1^{(\ell')}$ for all $1 \leq \ell' < \ell$), P_σ can inductively construct a share of $\boxed{2}$. The $\mathbf{e}_0^{(\ell)} \otimes \mathbf{e}_1^{(\ell')}$ can again be shared as DMPFs. The initial term $\mathbf{e}_0^{(\ell)} \otimes \mathbf{s}_1^{(0)}$ can be shared as a DMPF where the input is the $\mathbf{e}_0^{(\ell)}$ -index i and the payload is a whole row $e_{0,i} \cdot \mathbf{s}_1^{(0)}$. Overall, a party can construct a share of $\mathbf{s}_0^{(\ell)} \otimes \mathbf{s}_1^{(\ell)}$ from lower level shares. On the last level $\ell = L$, we compute $\mathbf{s}_0^{(L)} \odot \mathbf{s}_1^{(L)}$ using Equation (1) instead of Equation (2), i.e., we only need shares of $\mathbf{e}_0^{(L)} \odot \mathbf{e}_1^{(L)}$ (instead of $\mathbf{e}_0^{(L)} \otimes \mathbf{e}_1^{(L)}$), which simplifies the DMPF key for the last level. Overall, we see that PCG seeds that contain short $(m^{(0)})^2$ -dimensional seeds plus a sufficient number of DMPF keys are sufficient to produce shares of $m^{(L)}$ pseudorandom OLE correlations. We call the factor by which the output size $m^{(L)} \cdot \log |\mathbb{F}|$ is larger than the (initial) seed size the *expansion* of the PCG.

From PCG to PCF. The PCG construction in the previous paragraph can be instantiated with superpolynomial expansion, i.e., such that $m^{(L)}$ is superpolynomial in the seed size. Unfortunately, the intermediate matrices $\mathbf{A}^{(\ell)}$ will then also become superpolynomial, which leads to storage problems. In particular, our construction so far does not directly yield a PCF, where each entry of the output $(\mathbf{x}_\sigma, \mathbf{z}_\sigma)$ needs to be computed in polynomial time. Fortunately, we can choose the $\mathbf{A}^{(\ell)}$ to be sparse. For sparse $\mathbf{A}^{(\ell)}$ (and suitably chosen parameters), only a polynomial number of matrix rows are actually required to compute any $(x_{\sigma,i}, z_{\sigma,i})$. So evaluating the PCF in polynomial time is possible, if the parties obtain the required rows *on-the-fly*, i.e., without storing the full matrices. In situations where each PCF query uses a different row of a matrix (with independently distributed rows), previous works [BCG⁺20a, BCG⁺22] simply applied the random oracle to the PCF input to sample the row. However, this does not work in our case, because, for $\ell < L$, the set of required row indices of $\mathbf{A}^{(\ell)}$ can overlap across different PCF queries. For instances, when for two separate PCF queries on inputs i, j the rows $\mathbf{a}_i^{(L)}, \mathbf{a}_j^{(L)}$ have a coinciding non-zero index k , then the corresponding row $\mathbf{a}_k^{(L-1)}$ is needed in both PCF queries and thus cannot be sampled independently. Therefore, the challenge is to sample these matrices consistently across all queries. To do so, we employ a strategy of Braun et al. [BCM⁺25], who use a random oracle $\mathbf{H}_A^{(\ell)}$ for each level $1 \leq \ell \leq L$ to sample the rows of $\mathbf{A}^{(\ell)}$ as $\mathbf{a}_i^{(\ell)} \leftarrow \mathbf{H}_A^{(\ell)}(i)$ where $i \in [m^{(\ell)}]$.

Asymptotic Parameters. Presume each row of $\mathbf{A}^{(\ell)}$ has k non-zero entries. Then, to compute one entry of $\mathbf{s}_0^{(\ell)} \otimes \mathbf{s}_1^{(\ell)}$, we need at most k^2 entries of $\mathbf{s}_0^{(\ell-1)} \otimes \mathbf{s}_1^{(\ell-1)}$ (and k entries of both $\mathbf{s}_0^{(\ell-1)} \otimes \mathbf{e}_1^{(\ell)}$ and $\mathbf{e}_0^{(\ell)} \otimes \mathbf{s}_1^{(\ell-1)}$). This amounts to $\mathcal{O}(k^{2L})$ DPF evaluations to compute one entry of \mathbf{z}_σ — a quadratic blowup compared to the VOLE PCF from [BCM⁺25].

Perhaps surprisingly, the number of oracle calls (i.e., the number of accessed matrix rows) doesn't experience the same blowup: If we cache accessed matrix rows across different branches of the recursion tree, then we only need $k^{L-\ell}$ calls to $\mathbf{H}_A^{(\ell)}$ during a single query, which is the same number as in [BCM⁺25]. This is since the non-zero entries of $\mathbf{A}^{(\ell)}$ (for $\ell \geq 2$) determine which rows of $\mathbf{A}^{(\ell-1)}$ are accessed. Note that the total sets of accessed row indices (over the whole width of the recursion tree) on the left and right side of the tensor product are the same, due to symmetry.

As in [BCM⁺25], we use a fixed noise weight $t \in \text{poly}(\lambda)$ and row sparsity $k \in \text{polylog}(\lambda)$ for the asymptotic analysis. We set $n := m^{(0)} \in \text{poly}(\lambda)$ and, for $\ell \geq 1$, $m^{(\ell)} = m^{(\ell-1)}t/\lambda$. These choices are in a regime where sparse LPN is believed to be secure, and these choices are consistent with asymptotic parameters in prior work [BCM⁺25]. One can set $L \in \Theta(\log \lambda / \log \log \lambda)$, in order to have $k^{2L} \in \text{poly}(\lambda)$, i.e., polynomial time complexity for the PCF evaluation. The maximum number of queries then is $m^{(L)} = n(t/\lambda)^L \in \lambda^{\Theta(\log \lambda / \log \log \lambda)}$,

which is superpolynomial. For instance, one can set $L = \log n / (2 \log k)$ to achieve $k^{2L} = n$ and superpolynomial expansion.

Limitations. Our PCF has the following limitations, two of which are shared with the VOLE PCF from [BCM⁺25], which uses a similar recursive strategy.

- The random oracle is inherent to our construction, since the inputs to the random oracles can overlap across different PCF queries. Therefore, we do *not* obtain a *weak* PCF in the standard model, unlike [BCG⁺20a, OSY21, BCG⁺22, BCM⁺24, HRK25].
- Even though our PCF has superpolynomial expansion, the expansion is still asymptotically smaller than the subexponential expansion of previous PCFs for Beaver triples from VDLPN/EALPN [BCG⁺20a, BCG⁺22] (assuming the security of these non-standard LPN variants holds). However, in the asymptotic analysis, when the adversary is polynomially bounded, this subtlety doesn’t matter, since the adversary can only perform a polynomial number of queries, anyway.
- As an additional limitation, the runtime scaling with k^{2L} is an obstacle for the practicality of our PCF, when we want to support a large amount of queries, as we observe in our performance evaluation (Table 2 in Section 4.4). However, the same limitation applies to the previous Beaver triple PCFs from [BCG⁺20a, BCG⁺22], since they similarly experience a quadratic blowup of the number of FSS evaluations, compared to their OT/VOLE variants. As we’ve seen in Table 1, our PCF is faster than these previous PCFs.

1.3 Extensions

Programmability and Multiparty Degree-Two Correlations. Note that the first component x_σ of an output (x_σ, z_σ) from our PCF depends only on $\mathbf{s}_\sigma^{(0)}$ and $\mathbf{e}_\sigma^{(\ell)}$ (for all $1 \leq \ell \leq L$). Therefore, x_σ is *programmable*: We can choose the same $\mathbf{s}_\sigma^{(0)}$ and $\mathbf{e}_\sigma^{(\ell)}$ in two different PCF key generations, in order to make the private values x_σ of a party in both instances coincide, so these can be “reused” for different multiplications, with a potentially different party on the other side. As shown in [BCG⁺20a, Theorem 10.6], this implies a PCF for M -party Beaver triples (i.e., $\llbracket(a, b, ab)\rrbracket$ for uniformly random a and b) by instantiating the OLE PCF between each ordered pair of parties, amounting to $M(M - 1)$ instances in total. The same argument can be extended to arbitrary multiparty degree-two correlations by using the pairwise strategy (i.e., $M(M - 1)$ instances of the OLE PCF) for every multiplication gate in the correlation, while reusing shares appropriately across gates.

Two-Party Optimization. In the two-party setting ($M = 2$), the pairwise construction outlined above yields a Beaver triple PCF from $M(M - 1) = 2$ instances of our OLE PCF. However, our OLE PCF can be adapted to produce two-party Beaver triples directly within a single instance, at a minor efficiency penalty. Namely, instead of revealing $\mathbf{s}_\sigma^{(0)}$ and $\mathbf{e}_\sigma^{(\ell)}$ (for $1 \leq \ell \leq L$) to party P_σ , we instead include *shares* of all of these vectors (i.e., for both $\sigma \in \{0, 1\}$) in the PCF keys. Here, the vectors $\mathbf{e}_\sigma^{(\ell)}$ are shared via DMPFs. The parties can then locally compute shares of $\mathbf{a} := \mathbf{A}^{(L)}(\dots(\mathbf{A}^{(1)}\mathbf{s}_0^{(0)} + \mathbf{e}_0^{(1)})\dots) + \mathbf{e}_0^{(L)}$ and $\mathbf{b} := \mathbf{A}^{(L)}(\dots(\mathbf{A}^{(1)}\mathbf{s}_1^{(0)} + \mathbf{e}_1^{(1)})\dots) + \mathbf{e}_1^{(L)}$. Computing a single entry of \mathbf{a} and \mathbf{b} requires $\mathcal{O}(k^L)$ DPF evaluations, for fixed row sparsity k . Entries of $\mathbf{a} \odot \mathbf{b}$ can be done as before, i.e., with $\mathcal{O}(k^{2L})$ DPF evaluations.

One can extend this construction to arbitrary two-party degree-two correlations by using one instance of the Beaver triple PCF for every multiplication gate in the correlation, while reusing the $\mathbf{s}_\sigma^{(0)}$ and $\mathbf{e}_\sigma^{(\ell)}$ shares appropriately across instances.

Our PCF for Beaver triples has a slightly larger key size and evaluation time than our PCF for OLE. The increase in key size is due to (i.) the inclusion of (shares of) *both* $\mathbf{s}_0^{(0)}, \mathbf{s}_1^{(0)}$

in each party’s key, (ii.) the DMPF keys for $\mathbf{e}_\sigma^{(\ell)}$, and (iii.) the fact that we need to use “normal” DMPFs instead of more efficient *known-index* DMPFs (in order to not leak the non-zero indices of $\mathbf{e}_\sigma^{(\ell)}$ to any party). The increase in evaluation time comes from the additional DMPF evaluations to compute shares of (entries of) \mathbf{a} and \mathbf{b} .

We choose concrete parameters and evaluate the key size and performance for OLE and Beaver triples in Section 4.4. It turns out that abovementioned increase is minor, i.e., our PCF for Beaver triples is only slightly less efficient than our PCF for OLE, both in terms of key size and evaluation time. In other words, our PCF for Beaver triples is about $2\times$ more efficient than using the pairwise strategy from two OLE instances.

Authentication. In the two-party setting, our PCF can be extended to produce *authenticated* shares of any additive degree-two correlation. An authenticated share of some value $x \in \mathbb{F}$ is a share of αx , where $\alpha \in \mathbb{F}$ is a secret random MAC key that is also additively shared between the parties.

We illustrate the extension to authenticated shares by the example of authenticated Beaver triples $[[a, b, ab, \alpha a, \alpha b, \alpha ab]]$. The starting point is our optimized PCF for (unauthenticated) Beaver triples from the previous paragraph, which already produces $[[a, b, ab]]$. In order to produce the missing parts $[[a, b, ab]]$, one can simply add a second instance, where all of the vectors that are secret shared during the PCF key generation (i.e., $\mathbf{s}_\sigma^{(0)}$, $\mathbf{e}_\sigma^{(\ell)}$, and all of the tensor products) are replaced by their product with the MAC key α before being shared. Note that tensor products must only incur one factor of α , e.g., $\mathbf{e}_0^{(\ell)} \otimes \mathbf{e}_1^{(\ell)}$ is replaced by $\alpha \mathbf{e}_0^{(\ell)} \otimes \mathbf{e}_1^{(\ell)}$. Since the (unauthenticated) Beaver triple PCF computes shares of a, b, ab as a linear combination of entries of these secret vectors, it follows that the second instance outputs shares of $\alpha a, \alpha b, \alpha ab$, as desired.

One might wonder whether our PCF for multiparty Beaver triples (i.e., the pairwise construction from $M(M-1)$ OLE instances) can similarly be modified to produce authenticated triples. Unfortunately, this does not work, if there are more than two parties. This is because we would need to compute shares of $\alpha ab = (\sum_i \alpha_i)(\sum_i a_i)(\sum_i b_i)$, which, in the case of three or more parties, contains products like $\alpha_0 a_1 b_2$ that depend on shares from three different parties. We cannot compute shares of such products using two-party FSS, at least not while maintaining security against a dishonest majority, since knowledge of both PCF keys reveals the secret shared function.

Arbitrary Input Domain. The input domain of our PCF as presented above is $[m^{(L)}]$, i.e., for every input $i \in [m^{(L)}]$, we can query one OLE correlation. In order to allow inputs $x \in \mathcal{X}$ from an arbitrary domain \mathcal{X} (such as $\{0, 1\}^\lambda$), one can use a technique from [BCM⁺25], i.e., modify the top-level random oracle $H_A^{(L)}$ to take inputs from \mathcal{X} instead of $[m^{(L)}]$. As long as at most $m^{(L)}$ queries are made, the returned matrix rows still constitute an $m^{(L)} \times m^{(L-1)}$ matrix whose distribution does not depend on the concrete choice of \mathcal{X} . However, when using input $x \in \mathcal{X}$ (and hence $\mathbf{a}_x \leftarrow H_A^{(L)}$), one still somehow needs to map x to some index i in the noise $\mathbf{e}_\sigma^{(L)}$. Braun et al. [BCM⁺25] propose three possibilities to do this, all of which can be immediately applied to our PCF:

- The first and simplest option is to allow for the parties to maintain a small state, in the form of a counter i , between PCF queries. They use the counter to index into the noise and increment it after every PCF query. However, we note that, in scenarios where a counter i is available, the parties could simply use the unmodified input domain and call the PCF on i , invalidating the need for an arbitrary domain \mathcal{X} in the first place.
- The second option is to sample i from another random oracle $H_I : \mathcal{X} \rightarrow [m^{(L)}]$. We expect a collision after about $\sqrt{m^{(L)}}$ queries. In the asymptotic analysis, this is not a problem, if

the adversary is polynomially bounded, since $\sqrt{m^{(L)}}$ is still superpolynomial and, hence, the probability of a collision is negligible. However, in concrete instantiations where $m^{(L)}$ is chosen close to the actual number of queries, collisions are very likely, rendering the PCF insecure.

- The third option is to replace the top-level noise (here denoted \mathbf{e}) by $\mathbf{B}\mathbf{e}$ for a random sparse matrix $\mathbf{B} \in \mathbb{F}^{(m^{(L)}-\rho) \times m^{(L)}}$, that is full rank with high probability, for statistical security parameter ρ . When \mathbf{B} is indeed invertible, then a “collision” (defined as a linear combination of queries where the corresponding linear combination of matrix rows is zero) cannot occur. Formally, this bases security additionally on an adapted version of sparse LPN with “sparse product” noise distribution (as it was called in [BCM⁺25]), and there seems to be no reason why this adaptation should have any negative impact on security. During PCF evaluation, the rows of \mathbf{B} can be sampled from a random oracle H_B as $\mathbf{b}_x \leftarrow H_B(x)$. If at most $m^{(L)} - \rho$ queries are made, then these rows constitute a matrix with the desired distribution. When the rows of \mathbf{B} are k^B -sparse, then this option increases the number of DMPF evaluations for tensor products involving $\mathbf{e}_\sigma^{(L)}$ by the factor k^B . This does not affect performance, since the vast majority of DMPF evaluations happen on the lower levels $\ell < L$.

For simplicity of presentation, we keep the input domain of our PCF as $[m^{(L)}]$ for the remainder of this work.

2 Preliminaries

Notation. Let \mathbb{F} be a finite field throughout the paper. We denote vectors over \mathbb{F} by lower-case bold letters \mathbf{v} and matrices by upper-case bold letters \mathbf{M} . The j -th entry of \mathbf{v} is denoted by v_j and the i -th row of \mathbf{M} is denoted by (lower-case) \mathbf{m}_i . For a natural number m we denote by $[m]$ the set $\{0, \dots, m-1\}$, e.g., the set of indices of a vector \mathbf{v} .

In this work, big \mathcal{O} notations always take a function in the security parameter λ , where the dependence on λ is usually implicit. We also use $\text{poly}(f) := f^{\mathcal{O}(1)}$ and $\text{polylog}(f) := \text{poly}(\log f)$.

2.1 Pseudorandom Correlation Functions

A PCF is built from two keyed functions Eval_{k_σ} (for $\sigma = 0, 1$), such that $(\text{Eval}_{k_0}(x^{(i)}), \text{Eval}_{k_1}(x^{(i)}))$ for distinct (adversarially chosen) inputs $x^{(i)}$ follows a *target correlation* $\mathcal{Y} = (Y_0, Y_1)$.

To ensure security of PCFs in useful applications like MPC, Boyle et al. [BCG⁺20a] require that $(k_\sigma, x^{(i)})$ leaks nothing about $\text{Eval}_{k_{1-\sigma}}(x^{(i)})$, except what can be inferred from the fact that $(\text{Eval}_{k_0}(x^{(i)}), \text{Eval}_{k_1}(x^{(i)}))$ follows \mathcal{Y} . This is captured by demanding (in the *security* part of [Definition 2](#)) that $\text{Eval}_{k_{1-\sigma}}$ is indistinguishable from a distribution which party P_σ could have sampled herself. We recall the concept of a *reverse-sampleable correlation*, to give a formal definition of PCFs in [Definition 2](#).

Definition 1 (Reverse-Sampleable Correlation, [BCG⁺20a, Definition 4.1]). Let $l_0, l_1 \in \text{poly}(\lambda)$ and let $\mathcal{Y}(1^\lambda)$ be a probabilistic algorithm that outputs a pair $(y_0, y_1) \in \{0, 1\}^{l_0(\lambda)} \times \{0, 1\}^{l_1(\lambda)}$. \mathcal{Y} is a reverse-sampleable correlation, if there exists a probabilistic and polynomial time (ppt.) algorithm $\mathcal{Y}.\text{RSample}(1^\lambda, \sigma, y_\sigma)$ such that, for all $\sigma \in \{0, 1\}$, the distribution of $\mathcal{Y}(1^\lambda)$ is statistically close to

$$\left\{ (y_0, y_1) : (y'_0, y'_1) \stackrel{\$}{\leftarrow} \mathcal{Y}(1^\lambda), y_\sigma := y'_\sigma, y_{1-\sigma} \leftarrow \mathcal{Y}.\text{RSample}(1^\lambda, \sigma, y_\sigma) \right\}.$$

Definition 2 (Pseudorandom Correlation Function, [BCG⁺20a, Definition 4.4]).

A pseudorandom correlation function (PCF) with input domain $\mathcal{X}(\lambda)$ for a reverse-sampleable correlation \mathcal{Y} consists of

- a ppt. key generation algorithm $\text{PCF.Gen}(1^\lambda)$ that outputs a pair of private keys $(k_0^{\text{PCF}}, k_1^{\text{PCF}})$ (where λ can be inferred from a key) and
- a polynomial-time deterministic algorithm $\text{PCF.Eval}(\sigma, k_\sigma^{\text{PCF}}, \cdot)$, or $\text{PCF.Eval}_{k_\sigma^{\text{PCF}}}(\cdot)$ for short, with outputs in the range of $(\mathcal{Y}(1^\lambda))_\sigma$.

A PCF $(\text{PCF.Gen}, \text{PCF.Eval})$ is (N, B, ε) -secure if the following two properties hold for the security games presented in [Figure 1](#):

- **Strong Pseudorandom \mathcal{Y} -Correlated Outputs.** For every non-uniform adversary \mathcal{A} of size $B(\lambda)$ that takes an oracle and makes at most $N(\lambda)$ queries to the oracle, it holds for sufficiently large λ that

$$|\Pr[\text{Game}_{\mathcal{A},0}^{\text{pr}}(\lambda) = 1] - \Pr[\text{Game}_{\mathcal{A},1}^{\text{pr}}(\lambda) = 1]| \leq \varepsilon(\lambda).$$

- **Strong Security.** For all $\sigma \in \{0, 1\}$ and every non-uniform adversary \mathcal{A} of size $B(\lambda)$ that takes an oracle and makes at most $N(\lambda)$ queries to the oracle, it holds for sufficiently large λ that

$$|\Pr[\text{Game}_{\mathcal{A},\sigma,0}^{\text{sec}}(\lambda) = 1] - \Pr[\text{Game}_{\mathcal{A},\sigma,1}^{\text{sec}}(\lambda) = 1]| \leq \varepsilon(\lambda).$$

Strong Pseudorandom \mathcal{Y} -Correlated Outputs:		
<u>$\text{Game}_{\mathcal{A},b}^{\text{pr}}(\lambda)$:</u>	<u>$O_0^{\text{pr}}(x \in \mathcal{X}(\lambda))$:</u>	<u>$O_1^{\text{pr}}(x \in \mathcal{X}(\lambda))$:</u>
$(k_0, k_1) \leftarrow \text{PCF.Gen}(1^\lambda)$	If $(x, y_0, y_1) \in Q$:	For $\sigma = 0, 1$:
$Q := \emptyset$	Return (y_0, y_1)	$y_\sigma := \text{PCF.Eval}_{k_\sigma}(x)$
$b' \leftarrow \mathcal{A}^{O_b^{\text{pr}}}(1^\lambda)$	Else:	Return (y_0, y_1)
Return b'	$(y_0, y_1) \leftarrow \mathcal{Y}(1^\lambda)$	
	$Q := Q \cup \{(x, y_0, y_1)\}$	
	Return (y_0, y_1)	
Strong Security:		
<u>$\text{Game}_{\mathcal{A},\sigma,b}^{\text{sec}}(\lambda)$:</u>	<u>$O_{\sigma,0}^{\text{sec}}(x \in \mathcal{X}(\lambda))$:</u>	<u>$O_{\sigma,1}^{\text{sec}}(x \in \mathcal{X}(\lambda))$:</u>
$(k_0, k_1) \leftarrow \text{PCF.Gen}(1^\lambda)$	If $(x, y_{1-\sigma}) \in Q$:	$y_{1-\sigma} := \text{PCF.Eval}_{k_{1-\sigma}}(x)$
$Q := \emptyset$	Return $y_{1-\sigma}$	Return $y_{1-\sigma}$
$b' \leftarrow \mathcal{A}^{O_{\sigma,b}^{\text{sec}}}(1^\lambda, \sigma, k_\sigma)$	Else:	
Return b'	$y_\sigma := \text{PCF.Eval}_{k_\sigma}(x)$	
	$y_{1-\sigma} \leftarrow \mathcal{Y}.\text{RSample}(1^\lambda, \sigma, y_\sigma)$	
	$Q := Q \cup \{(x, y_{1-\sigma})\}$	
	Return $y_{1-\sigma}$	

Fig. 1: Security games for the PCF definition ([Definition 2](#))

2.2 Oblivious Linear Evaluation

Definition 3 (Oblivious Linear Evaluation). The oblivious linear evaluation (OLE) correlation over a field \mathbb{F} is $\text{OLE} := ((x_0, z_0), (x_1, z_1))$ where $x_0, z_0, x_1 \stackrel{\$}{\leftarrow} \mathbb{F}$ and $z_1 = x_0 \cdot x_1 - z_0$.

The OLE correlation is reverse-sampleable via $\text{OLE.RSample}(1^\lambda, \sigma, (x_\sigma, z_\sigma)) = (x_{1-\sigma} \stackrel{\$}{\leftarrow} \mathbb{F}, x_0 \cdot x_1 - z_\sigma)$. We define programmability of a PCF for OLE as in [\[BCG⁺20a, Definition 10.4\]](#), adapted to our notation:

Definition 4 (Programmability). A PCF $(\text{PCF.Gen}, \text{PCF.Eval})$ for OLE is programmable, if there exists a ppt. algorithm $\text{PCF.Gen}_p(1^\lambda, \rho_0, \rho_1)$, that takes additional inputs $\rho_0, \rho_1 \in \{0, 1\}^{n(\lambda)}$ for a function $n \in \text{poly}(\lambda)$, such that the following three properties hold:

– **Indistinguishability.** The two distributions

$$\left\{ (k_0, k_1) \mid (k_0, k_1) \leftarrow \text{PCF.Gen}(1^\lambda) \right\} \text{ and}$$

$$\left\{ (k_0, k_1) \mid \rho_0, \rho_1 \xleftarrow{\$} \{0, 1\}^{n(\lambda)}, (k_0, k_1) \leftarrow \text{PCF.Gen}_p(1^\lambda, \rho_0, \rho_1) \right\}$$

are computationally indistinguishable.

– **Programmability.** There are public efficiently computable functions f_0, f_1 , such that

$$\Pr \left[\begin{array}{l} x_0 = f_0(\rho_0, i) \\ x_1 = f_1(\rho_1, i) \end{array} \mid \begin{array}{l} (\rho_0, \rho_1) \xleftarrow{\$} \{0, 1\}^{n(\lambda)} \\ (k_0, k_1) \leftarrow \text{PCF.Gen}_p(1^\lambda, \rho_0, \rho_1) \\ (x_0, z_0) \leftarrow \text{PCF.Eval}_{k_0}(i) \\ (x_1, z_1) \leftarrow \text{PCF.Eval}_{k_1}(i) \end{array} \right] = 1.$$

– **Security.** For all $\sigma \in \{0, 1\}$, the two distributions

$$\left\{ (k_\sigma, (\rho_\sigma, \rho_{1-\sigma})) \mid \rho_0, \rho_1 \xleftarrow{\$} \{0, 1\}^{n(\lambda)}, (k_0, k_1) \leftarrow \text{PCF.Gen}_p(1^\lambda, \rho_0, \rho_1) \right\} \text{ and}$$

$$\left\{ (k_\sigma, (\rho_\sigma, \tilde{\rho})) \mid \rho_0, \rho_1, \tilde{\rho} \xleftarrow{\$} \{0, 1\}^{n(\lambda)}, (k_0, k_1) \leftarrow \text{PCF.Gen}_p(1^\lambda, \rho_0, \rho_1) \right\}$$

are computationally indistinguishable.

2.3 Learning Parity with Noise

The security of our PCF construction is reduced to the security of the sparse LPN assumption. We borrow some notation from [BCM⁺25]. Let $\text{hw}(\mathbf{v})$ denote the Hamming weight of a vector \mathbf{v} , i.e., the number of non-zero entries of \mathbf{v} . We say \mathbf{v} is $\text{hw}(\mathbf{v})$ -sparse if $\text{hw}(\mathbf{v})$ is small compared to the vector dimension.

Definition 5 (Regular Vector/Distribution). Let $t, m, n \in \mathbb{N}$ with $t \mid n$. A vector $\mathbf{v} \in \mathbb{F}^n$ is said to be t -regular, if $\mathbf{v} \in \{\mathbf{w} \in \mathbb{F}^{n/t} \mid \text{hw}(\mathbf{w}) = 1\}^t$, i.e., \mathbf{v} is a concatenation of t scaled unit vectors of equal length. Further, we define

- $\text{Reg}_{t,n}(\mathbb{F})$ as the uniform distribution on $\{\mathbf{w} \in \mathbb{F}^{n/t} \mid \text{hw}(\mathbf{w}) = 1\}^t$ and
- $\text{RegularCodeGen}(k, n, m, \mathbb{F})$ as the distribution of $(\mathbf{e}_0 \dots \mathbf{e}_{m-1})^T \in \mathbb{F}^{m \times n}$ where each row \mathbf{e}_i^T is sampled as $\mathbf{e}_i \leftarrow \text{Reg}_{k,n}(\mathbb{F})$.

Definition 6 ((Sparse) LPN). Let $n = n(\lambda)$ be the secret size and let $m = m(\lambda)$ be the output size. Let $\{\text{Noise}(n, m, \mathbb{F})\}_{n,m \in \mathbb{N}}$ be a family of probability distributions where $\text{Noise}(n, m, \mathbb{F})$ outputs a vector $\mathbf{e} \in \mathbb{F}^m$. Let $\text{CodeGen}(n, m, \mathbb{F})$ be a probabilistic “code generation” algorithm that outputs a matrix from $\mathbb{F}^{m \times n}$. The $(\text{CodeGen}, \text{Noise}, \mathbb{F})$ -LPN (n, m) assumption states that

$$(\mathbf{A}, \mathbf{A}\mathbf{s} + \mathbf{e}) \approx_c (\mathbf{A}, \mathbf{y}),$$

where $\mathbf{A} \leftarrow \text{CodeGen}(n, m, \mathbb{F})$, $\mathbf{s} \xleftarrow{\$} \mathbb{F}^n$, $\mathbf{e} \leftarrow \text{Noise}_{n,m}(\mathbb{F})$, $\mathbf{y} \xleftarrow{\$} \mathbb{F}^m$, and \approx_c denotes computational indistinguishability. When $\text{CodeGen} = \text{RegularCodeGen}(k, \cdot, \cdot, \cdot)$ and $\text{Noise}(n, m, \cdot) = \text{Reg}_{t,m}$ for some $k = k(\lambda)$ and $t = t(\lambda)$, then we call above assumption sparse LPN and denote it by \mathbb{F} -SparseLPN (k, n, m, t) .

We base the security of our PCF on the assumption that sparse LPN can be securely (against polynomially-bounded distinguishers) instantiated with regular noise of weight $t \in \text{poly}(\lambda)$, row sparsity $k \in \text{polylog}(\lambda)$, secret size $n \in \text{poly}(\lambda)$ and output size $m = nt/\lambda$. This flavor of sparse LPN is consistent with what’s used in [BCM⁺25].

2.4 Function Secret Sharing

Here, we define FSS as in [BGI16, Definitions 2.1 and 2.2], but only for two parties with subtractive reconstruction, and adapted to our notation. The definition is parameterized by a leakage function $\text{Leak} : \{0, 1\} \times \{0, 1\}^* \rightarrow \{0, 1\}^*$, that takes a party index $\sigma \in \{0, 1\}$ and a function description $\hat{f} \in \{0, 1\}^*$ of a function f , and outputs information about \hat{f} that is allowed to be revealed by an FSS key. Typically, one allows at least the input and output size of f to be leaked. We sometimes write f instead of \hat{f} , when it's clear from the context that a description of f is meant.

In contrast to [BGI16], we give the party index σ to Leak , in order to be able to capture asymmetric use-cases, where one party is allowed to learn more than the other party. We use this to formally define *known-index* DPFs below.

Definition 7 (Subtractive FSS). *A subtractive function secret sharing (FSS) scheme consists of*

- a ppt. key generation algorithm $\text{FSS.Gen}(1^\lambda, \hat{f})$, where $\hat{f} \in \{0, 1\}^*$ is a description of a function $f : \{0, 1\}^n \rightarrow \mathbb{G}$ (where \hat{f} explicitly contains the input length 1^n and a description of the Abelian group \mathbb{G}), that outputs a pair of private keys $(k_0^{\text{FSS}}, k_1^{\text{FSS}})$, and
- a polynomial-time deterministic algorithm $\text{FSS.Eval}(\sigma, k_\sigma^{\text{FSS}}, x)$, where $x \in \{0, 1\}^n$, that outputs a group element $y_\sigma \in \mathbb{G}$.

A subtractive FSS scheme $(\text{FSS.Gen}, \text{FSS.Eval})$ for a function family \mathcal{F} and leakage function $\text{Leak} : \{0, 1\} \times \{0, 1\}^* \rightarrow \{0, 1\}^*$ is secure if the following two properties hold:

- **Correctness.** For all $\hat{f} \in \mathcal{F}$ describing $f : \{0, 1\}^n \rightarrow \mathbb{G}$ and all $x \in \{0, 1\}^n$, it holds over $(k_0, k_1) \leftarrow \text{FSS.Gen}(1^\lambda, \hat{f})$ that

$$\Pr[\text{FSS.Eval}(0, k_0, x) - \text{FSS.Eval}(1, k_1, x) = f(x)] = 1.$$

- **Security.** For all $\sigma \in \{0, 1\}$ (corrupted party's index), there exists a ppt. simulator Sim , such that, for all sequences $(\hat{f}^{(\lambda)})_{\lambda=0,1,\dots}$ of function descriptions from \mathcal{F} (where $\hat{f}^{(\lambda)}$ has size in $\text{poly}(\lambda)$), the two distributions

$$\left\{ k_\sigma \mid (k_0, k_1) \leftarrow \text{FSS.Gen}(1^\lambda, \hat{f}^{(\lambda)}) \right\} \quad \text{and} \quad \text{Sim}(1^\lambda, \text{Leak}(\sigma, \hat{f}^{(\lambda)}))$$

are computationally indistinguishable.

For the FSS schemes used in our PCF construction, we additionally require that the outputs of a single party are pseudorandom, i.e., indistinguishable from uniform. This is captured by the following definition, which we adapted from [BCG⁺20a, Definition 5.2].

Definition 8 (FSS with Pseudorandom Outputs). *A subtractive FSS scheme $(\text{FSS.Gen}, \text{FSS.Eval})$ for a function family \mathcal{F} has pseudorandom outputs if, for all $\sigma \in \{0, 1\}$, all $N \in \text{poly}(\lambda)$, all sequences $(\hat{f}^{(\lambda)})_{\lambda=0,1,\dots}$ of function descriptions from \mathcal{F} , and all sequences of input sets $(\{x_i^{(\lambda)}\}_{i \in [N]})_{\lambda=0,1,\dots}$ (where $\hat{f}^{(\lambda)}$ has size in $\text{poly}(\lambda)$, input length $n(\lambda) \in \text{poly}(\lambda)$, range $\mathbb{G}(\lambda)$ with $\log |\mathbb{G}(\lambda)| \in \text{poly}(\lambda)$, and $\forall i \in [N] : x_i^{(\lambda)} \in \{0, 1\}^{n(\lambda)}$), the distribution*

$$\left\{ (y_i)_{i \in [N]} \mid (k_0, k_1) \leftarrow \text{FSS.Gen}(1^\lambda, \hat{f}^{(\lambda)}), y_i \leftarrow \text{FSS.Eval}(\sigma, k_\sigma, x_i^{(\lambda)}) \right\}$$

is computationally indistinguishable from the uniform distribution over $\mathbb{G}(\lambda)$.

Definition 9 (Point Function). *A point function $f_{\alpha, \beta} : [m] \rightarrow \mathbb{G}$ for some $\alpha \in [m]$ and Abelian group $\mathbb{G} \ni \beta$ is defined by $f(\alpha) = \beta$ and $f(x) = 0$ for $x \neq \alpha$.*

Definition 10 (Multi-Point Function). A t -point function for some $t, m \in \mathbb{N}$ is a function $f : [m] \rightarrow \mathbb{G}$ that is non-zero at exactly t points.

Definition 11 (Distributed Point Function, [BGI16, Definition 2.3, adapted]). A distributed point function (DPF) is an FSS scheme for the family of all point functions and leakage $\text{Leak}(\sigma, f_{\alpha, \beta} : [m] \rightarrow \mathbb{G}) = (m, \mathbb{G})$.

In this work, we use sparse LPN with regular t -sparse noise where t is publicly known. In order to succinctly share such noise vectors, we define the notion of a *regular* distributed multi-point function (DMPF). A regular DMPF can be constructed as a concatenation of t DPFs, i.e., DMPF.Gen generates t DPF keys and DMPF.Eval uses exactly one of them.

Definition 12 (Regular DMPF). A regular distributed multi-point function (DMPF) for some $t \in \mathbb{N}$ is an FSS scheme for the function family

$$\mathcal{F} = \{f : [m] \rightarrow \mathbb{G} \mid m \in \mathbb{N}, (f(i))_{i \in [m]} \text{ is } t\text{-regular}\}$$

and leakage $\text{Leak}(\sigma, f) = (m, \mathbb{G}, t)$.

In our PCF construction, we also use *known-index* DPFs that allow one of the parties to learn the index where the point function is non-zero (but not the function value at that index):

Definition 13 (Known-Index DPF). A known-index DPF is defined like a DPF except that, for exactly one $\sigma \in \{0, 1\}$, the leakage is instead $\text{Leak}(\sigma, f_{\alpha, \beta}) = (m, \mathbb{G}, \alpha)$, i.e., party P_σ might learn the index α .

Definition 14 (Regular Known-Index DMPF). A regular known-index DPF is defined like a regular DMPF except that, for exactly one $\sigma \in \{0, 1\}$, the leakage is instead $\text{Leak}(\sigma, f) = (m, \mathbb{G}, t, \{i \mid f(i) \neq 0\})$, i.e., party P_σ might learn the non-zero indices (but not the function values at these indices).

There is a standard construction for *known-index* DPFs based on GGM PPRFs [SGRR19, BCG⁺19a, BCG⁺23]. Like in the “unknown-index” case, a regular known-index DMPF can be constructed as a concatenation of t known-index DPFs.

3 FSS Tensor Operation

In [BGI16], the authors describe a tensor operation for certain two-party FSS schemes. Namely, given a DPF scheme FSS^\bullet for $\mathcal{F}_{\lambda+1}^\bullet$ (where \mathcal{F}_ℓ^\bullet for $\ell \in \mathbb{N}$ denotes the family of point functions with ℓ -bit outputs) and an FSS scheme $\text{FSS}^\mathcal{F}$ for an arbitrary function family \mathcal{F} , they construct an FSS scheme for

$$\begin{aligned} \mathcal{F}_1^\bullet \otimes \mathcal{F} &:= \{f_{\alpha, 1} \otimes f \mid f_{\alpha, 1} \in \mathcal{F}_1^\bullet, f \in \mathcal{F}\}, \text{ where} \\ (f_{\alpha, 1} \otimes f)(x, y) &:= f_{\alpha, 1}(x) \cdot f(y) = \begin{cases} f(y) & \text{if } x = \alpha, \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \quad (5)$$

There are further conditions on $\text{FSS}^\mathcal{F}$, namely, it needs to have pseudorandom keys, subtractive reconstruction, and the output of $\text{FSS}^\mathcal{F}.\text{Eval}(\sigma, k_\sigma^\mathcal{F}, y)$ does not depend on σ . The latter condition ensures that, when both parties use the same key, they obtain the same output, constituting subtractive shares of 0.

On a high level, the construction works as follows. FSS^\bullet is used to generate keys for $f_{\alpha, s|1}$, where $s \stackrel{\$}{\leftarrow} \{0, 1\}^\lambda$ is a secret random seed. To evaluate $(f_{\alpha, 1} \otimes f)(x, y)$, the parties

first evaluate $f_{\alpha, s||1}(x)$ to obtain $s_\sigma || t_\sigma$ where $|s_\sigma| = \lambda$ and $|t_\sigma| = 1$. Assuming $x = \alpha$, it holds that $s_0 \oplus s_1 = s$ and $t_0 \oplus t_1 = 1$. The parties then use a PRG to expand their share s_σ into sufficiently long pseudorandom masks $\text{PRG}(s_\sigma)$. Both parties are also given *both* FSS keys for $\text{FSS}^\mathcal{F}$, but one masked with $\text{PRG}(s_0)$ and one masked with $\text{PRG}(s_1)$, i.e., both parties are given CW_0, CW_1 (the so-called *correction words*) where $\text{CW}_{t_\sigma} := k_\sigma^\mathcal{F} \oplus \text{PRG}(s_\sigma)$. They then use the bit t_σ (from FSS^\bullet 's output) to decide which key to unmask and use for the evaluation with $\text{FSS}^\mathcal{F}$ (on input y). That is, (still in the case $x = \alpha$) party P_σ computes $\text{CW}_{t_\sigma} \oplus \text{PRG}(s_\sigma) = k_\sigma^\mathcal{F}$ and evaluates $\text{FSS}^\mathcal{F}.\text{Eval}(\sigma, k_\sigma^\mathcal{F}, y)$, so the parties obtain shares of $f(y)$. In the other case $x \neq \alpha$, it holds that $s_0 = s_1$ and $t_0 = t_1$, so both parties will perform the same actions (i.e., they choose the same masked key and unmask it using the same randomness), so they will obtain the same outputs in the end, i.e., shares of 0. The resulting key size is $\text{size}^\otimes = \text{size}^\bullet + 2 \cdot \text{size}^\mathcal{F}$.

Moreover, [BGI16] showed how to optimize the key size, if $\text{FSS}^\mathcal{F}$ is a single-bit-input DPF (which is simply a sharing its truth table). In this case, only one correction word is required, and the size of each correction word can be (approximately) halved, hence saving a total factor of ≈ 4 . By iterating the tensor construction (together with this optimization) for each bit of the input, they obtain an FSS scheme for n -bit-input b -bit-output point functions with key size $n(\lambda + 2) + \lambda + b$, which is ≈ 4 times smaller than using the tensor construction without the optimization (as previously done in [BGI15]).

3.1 Improved Tensor Construction and Seedable FSS

In the following, we describe two optimizations for the key size in the tensor construction. These improvements are not restricted to a specific scheme $\text{FSS}^\mathcal{F}$ on the right-hand side, but work for many existing schemes, which we capture by defining a new notion of *seedable* FSS below.

We note that self-contained FSS constructions, such as the DPF from [BGI16] (when viewed as a tensor product of one-bit-input DPFs), already (implicitly) contain these optimizations. However, our result is useful when constructing new FSS schemes as a tensor product of existing FSS schemes in a blackbox manner, as we do in Section 3.2.

1. If $\text{FSS}^\mathcal{F}.\text{Gen}(1^\lambda)$ has outputs of the form $((\rho_0^\mathcal{F}, c^\mathcal{F}), (\rho_1^\mathcal{F}, c^\mathcal{F}))$, i.e., the FSS keys consist of a private part $\rho_\sigma^\mathcal{F}$ and a common part $c^\mathcal{F}$, and furthermore, $(\rho_\sigma^\mathcal{F}, c^\mathcal{F})$ is computationally indistinguishable from $(\hat{k}, c^\mathcal{F})$ for uniformly random \hat{k} , then only the private part $\rho_\sigma^\mathcal{F}$ needs to be masked, i.e., the tensor construction's key consists of $(k_\sigma^\bullet, \text{CW}_0, \text{CW}_1, c^\mathcal{F})$ where $\text{CW}_{t_\sigma} := \rho_\sigma^\mathcal{F} \oplus \text{PRG}(s_\sigma)$. That is, the common part needs to be included only once, saving a factor of 2 for the common part.
2. Under the additional conditions that $\text{FSS}^\mathcal{F}.\text{Gen}$ can be “programmed” to use any uniformly random $\rho_0^\mathcal{F}, \rho_1^\mathcal{F}$, we can get rid of the masks entirely and simply use $\rho_\sigma^\mathcal{F} := \text{PRG}(s_\sigma)$. This corresponds to $\text{CW}_0 = \text{CW}_1 = 0$. Intuitively, the resulting tensor construction is still a secure FSS scheme, because $\text{PRG}(s_\sigma)$ is pseudorandom to the other party, who has no information about s_σ . With this modification, the words CW_0, CW_1 are no longer required (but the common part is), so the final key consists of $(k_\sigma^\bullet, c^\mathcal{F})$. We formally define this construction and rigorously prove its security later in this section.

Many existing FSS schemes, including standard DPFs and DCFs [BGI16, BCG⁺21], have keys where everything is common except for a private λ -bit seed per party, which is chosen independently (from the other party's seed) and uniformly random. With such a scheme on the right-hand side of the tensor operation, both optimizations can be applied, leading to at least a factor 2 improvement in key size (excluding k_σ^\bullet).

In order to formally capture the family of FSS schemes that support both optimizations, we introduce the new notion of a *seedable* FSS scheme. Afterwards, we formally define our improved tensor construction and prove its security.

Definition 15 (Seedable FSS⁴). A subtractive FSS scheme (FSS.Gen, FSS.Eval) for a function family \mathcal{F} is called a seedable, if there exists a ppt. algorithm $\text{FSS.Gen}_s(1^\lambda, \rho_0, \rho_1, \hat{f})$, that takes additional inputs $\rho_0, \rho_1 \in \{0, 1\}^{l(\lambda)}$ for a function $l \in \text{poly}(\lambda)$, such that the following four properties hold for all $\hat{f} \in \mathcal{F}$:

- **Symmetry.** For all $x \in \{0, 1\}^n$ and $\sigma \in \{0, 1\}$, it holds over $(k_0, k_1) \leftarrow \text{FSS.Gen}(1^\lambda, \hat{f})$ that

$$\Pr[\text{FSS.Eval}(0, k_\sigma, x) = \text{FSS.Eval}(1, k_\sigma, x)] = 1.$$

- **Modified Correctness.** Correctness (cf. Definition 7) still holds when replacing (k_0, k_1) by $((\rho_0, c), (\rho_1, c))$ for any $\rho_0, \rho_1 \in \{0, 1\}^{l(\lambda)}$ and $c \leftarrow \text{FSS.Gen}_s(1^\lambda, \rho_0, \rho_1, \hat{f})$.
- **Modified Security.** For all $\sigma \in \{0, 1\}$ (corrupted party's index), there exists a ppt. simulator Sim_s , such that, for all sequences $(\hat{f}^{(\lambda)})_{\lambda=0,1,\dots}$ of function descriptions from \mathcal{F} and all sequences $(\rho_\sigma^{(\lambda)})_{\lambda=0,1,\dots}$ (where $\hat{f}^{(\lambda)}$ has size in $\text{poly}(\lambda)$ and $\rho_\sigma^{(\lambda)} \in \{0, 1\}^{l(\lambda)}$), the two distributions

$$\left\{ c \mid \rho_{1-\sigma}^{(\lambda)} \xleftarrow{\$} \{0, 1\}^{l(\lambda)}, c \leftarrow \text{FSS.Gen}_s(1^\lambda, \rho_0^{(\lambda)}, \rho_1^{(\lambda)}, \hat{f}^{(\lambda)}) \right\} \quad \text{and} \quad \text{Sim}_s(1^\lambda, \text{Leak}(\sigma, \hat{f}^{(\lambda)}))$$

are computationally indistinguishable.

Parameters:

- FSS^\bullet is a DPF scheme for the family $\mathcal{F}_\lambda^\bullet$, i.e., point functions with λ -bit outputs.
- $\text{FSS}^\mathcal{F}$ is a seedable FSS scheme for an arbitrary function family \mathcal{F} .

$\text{FSS}^\otimes.\text{Gen}(1^\lambda, \hat{f}^\otimes)$:

1. Parse \hat{f}^\otimes as a description of $f_{\alpha,1} \otimes f$ for $f_{\alpha,1} \in \mathcal{F}_1^\bullet$ and $f \in \mathcal{F}$.
2. $s \xleftarrow{\$} \{0, 1\}^\lambda$
3. $(k_0^\bullet, k_1^\bullet) \leftarrow \text{FSS}^\bullet.\text{Gen}(1^\lambda, f_{\alpha,s})$
4. **For** $\sigma \in \{0, 1\}$:
 $s_\sigma \leftarrow \text{FSS}^\bullet.\text{Eval}(\sigma, k_\sigma^\bullet, \alpha)$
 $\rho_\sigma^\mathcal{F} := \text{PRG}(s_\sigma)$
5. $c^\mathcal{F} \leftarrow \text{FSS}^\mathcal{F}.\text{Gen}_s(1^\lambda, \rho_0^\mathcal{F}, \rho_1^\mathcal{F}, f)$
6. **Return** $(k_0^\otimes, k_1^\otimes) := ((k_0^\bullet, c^\mathcal{F}), (k_1^\bullet, c^\mathcal{F}))$

$\text{FSS}^\otimes.\text{Eval}(\sigma, k_\sigma^\otimes, (x, y))$:

1. Parse $k_\sigma^\otimes := (k_\sigma^\bullet, c^\mathcal{F})$.
2. $\tilde{s}_\sigma \leftarrow \text{FSS}^\bullet.\text{Eval}(\sigma, k_\sigma^\bullet, x)$
3. $\tilde{\rho}_\sigma^\mathcal{F} := \text{PRG}(\tilde{s}_\sigma)$
4. **Return** $\tilde{y}_\sigma := \text{FSS}^\mathcal{F}.\text{Eval}(\sigma, (\tilde{\rho}_\sigma^\mathcal{F}, c^\mathcal{F}), y)$

Fig. 2: FSS tensor scheme with optimized key size for $\mathcal{F}_1^\bullet \otimes \mathcal{F}$

Theorem 1. Let FSS^\bullet be a secure FSS scheme for the family $\mathcal{F}_\lambda^\bullet$ with key size size^\bullet and leakage Leak^\bullet , where $\text{Leak}^\bullet(\sigma, f_{\alpha,s})$ is independent of s .⁵ Let $\text{FSS}^\mathcal{F}$ be a secure seedable FSS scheme for an arbitrary function family \mathcal{F} with key size $\text{size}^\mathcal{F}$ and leakage $\text{Leak}^\mathcal{F}$. Then FSS^\otimes from Figure 2 is a secure FSS scheme for the function family $\mathcal{F}_1^\bullet \otimes \mathcal{F}$ with key size $\text{size}^\otimes \leq \text{size}^\bullet + \text{size}^\mathcal{F}$ and leakage $\text{Leak}^\otimes(\sigma, f_{\alpha,1} \otimes f) := (\text{Leak}^\bullet(\sigma, f_{\alpha,1}), \text{Leak}^\mathcal{F}(\sigma, f))$.

⁴ Note that our notion of a seedable FSS scheme is *not* related to the notion of a programmable DPF from [BGIK22].

⁵ This captures both DPFs and *known-index* DPFs.

Proof. The key size $\text{size}^\otimes \leq \text{size}^\bullet + \text{size}^\mathcal{F}$ is obvious by the definition of $\text{FSS}^\otimes.\text{Gen}$. It remains to prove *correctness* and *security* from [Definition 7](#):

Correctness. Let $f^\otimes = f_{\alpha,1} \otimes f \in \mathcal{F}_1^\bullet \otimes \mathcal{F}$ and let $(x, y) \in [m] \times \{0, 1\}^*$ be an input to f^\otimes .

- *Case $x \neq \alpha$.* In this case, $f_{\alpha,s}(x) = 0$, so we have $\tilde{s}_0 = \tilde{s}_1$ by the correctness of FSS^\bullet . It follows that $\tilde{k}_0^\mathcal{F} = \text{PRG}(\tilde{s}_0) = \text{PRG}(\tilde{s}_1) = \tilde{k}_1^\mathcal{F}$. By the *symmetry* (cf. [Definition 15](#)) of $\text{FSS}^\mathcal{F}$, it follows that $\tilde{y}_0 = \text{FSS}^\mathcal{F}.\text{Eval}(0, (\tilde{k}_0^\mathcal{F}, c), y) = \text{FSS}^\mathcal{F}.\text{Eval}(1, (\tilde{k}_1^\mathcal{F}, c), y) = \tilde{y}_1$, i.e., the outputs are subtractive shares of $f^\otimes(x, y) = 0$.
- *Case $x = \alpha$.* In this case, $\tilde{s}_\sigma = \text{FSS}^\bullet.\text{Eval}(\sigma, k_\sigma^\bullet, \alpha) = s_\sigma$, hence $\tilde{\rho}_\sigma^\mathcal{F} = \text{PRG}(s_\sigma) = \rho_\sigma^\mathcal{F}$. By the *modified correctness* (cf. [Definition 15](#)) of $\text{FSS}^\mathcal{F}$, it follows that $\tilde{y}_0 - \tilde{y}_1 = f(y) = f^\otimes(x, y)$.

Security. W.l.o.g., let party $\sigma = 0$ be corrupted (since the whole proof works symmetrically for $\sigma = 1$). Let $(\hat{f}^{(\lambda)})_{\lambda=0,1,\dots}$ be a sequence of function descriptions from $\mathcal{F}_1^\bullet \otimes \mathcal{F}$, where $\hat{f}^{(\lambda)}$ has size in $\text{poly}(\lambda)$. By the *security* of FSS^\bullet and *modified security* of $\text{FSS}^\mathcal{F}$, there are simulators Sim^\bullet and $\text{Sim}_s^\mathcal{F}$ with indistinguishability guarantees as in [Definitions 7](#) and [15](#), respectively. Based on these simulators, we define the simulator Sim^\otimes in [Figure 3](#). We prove that $\text{Sim}^\otimes(1^\lambda, \text{Leak}^\otimes(\sigma, \hat{f}^{(\lambda)}))$ is computationally indistinguishable from $\{k_0^\otimes \mid (k_0^\otimes, k_1^\otimes) \leftarrow \text{FSS}^\otimes.\text{Gen}(1^\lambda, \hat{f}^{(\lambda)})\}$ via a sequence of hybrids H_0, \dots, H_4 (also presented in [Figure 3](#)).

- $H_0(\lambda)$ corresponds exactly to $\{k_0^\otimes \mid (k_0^\otimes, k_1^\otimes) \leftarrow \text{FSS}^\otimes.\text{Gen}(1^\lambda, \hat{f}^{(\lambda)})\}$.
- H_1 is distributed identically to H_0 by the *correctness* (cf. [Definition 7](#)) of FSS^\bullet . Now, observe that H_1 no longer accesses k_1^\bullet . Intuitively, this allows us to use the *security* of FSS^\bullet to replace k_0^\bullet by a simulated key in the next step.
- $H_2 \approx_c H_1$ is proven in the following. Note that $H_2 \approx_c H_1$ isn't obvious, because H_1 's output depends not only on k_0^\bullet , but also on s and we cannot delay the sampling of s . To resolve this issue, we use a trick that fixes for each λ a “worst-case” value $s^{(\lambda)}$ of s : Let $H_i^{(s=\tilde{s})}$ denote that, in the definition of H_i , we replace the random choice of s by the deterministic assignment $s := \tilde{s}$. The advantage of a distinguisher \mathcal{D} is

$$\begin{aligned} & \Pr[\mathcal{D}(H_1(\lambda)) = 1] - \Pr[\mathcal{D}(H_2(\lambda)) = 1] \\ &= \sum_{\tilde{s} \in \{0,1\}^\lambda} 2^{-\lambda} \cdot \left(\Pr[\mathcal{D}(H_1^{(s=\tilde{s})}(\lambda)) = 1] - \Pr[\mathcal{D}(H_2^{(s=\tilde{s})}(\lambda)) = 1] \right) \\ &\leq \Pr[\mathcal{D}(H_1^{(s=s^{(\lambda)})}(\lambda)) = 1] - \Pr[\mathcal{D}(H_2^{(s=s^{(\lambda)})}(\lambda)) = 1] \end{aligned}$$

where the latter inequality holds when we define $s^{(\lambda)}$ to be the arg max of the previous sum. This defines an infinite sequence $(f_{\alpha^{(\lambda)}, s^{(\lambda)}})_{\lambda=0,1,\dots}$ as in the *security* part of [Definition 7](#). By the security of FSS^\bullet , it follows that $H_1^{(s=s^{(\lambda)})} \approx_c H_2^{(s=s^{(\lambda)})}$, so the advantage of \mathcal{D} is negligible.

- $H_3 \approx_c H_2$ holds since $\text{PRG}(s_1)$ is a PRG evaluated on uniformly random $s_1 = s_0 \oplus s$. Note that s_1 is uniformly random due to the mask s that is not used elsewhere.
- $H_4 \approx_c H_3$ holds by a similar argument as for $H_2 \approx_c H_1$ above: This time, we take the sum over all possible values of the random coins that influence the choice of $\rho_0^\mathcal{F}$ and define $\rho_0^{(\lambda)}$ as the value of $\rho_0^\mathcal{F}$ induced by the arg max of the sum. Computational indistinguishability then follows by the *modified security* (cf. [Definition 15](#)) of $\text{FSS}^\mathcal{F}$.
- $\text{Sim}^\otimes(1^\lambda, \text{Leak}^\otimes(\sigma, \hat{f}^{(\lambda)}))$ corresponds exactly to $H_4(\lambda)$.

□

In all hybrids, $f_{\alpha,1} \otimes f$ (for $f_{\alpha,1} \in \mathcal{F}_1^\bullet$ and $f \in \mathcal{F}$) denotes the function described by $\hat{f}^{(\lambda)}$.		
$H_0(\lambda)$: $s \xleftarrow{\$} \{0, 1\}^\lambda$ $(k_0^\bullet, k_1^\bullet) \leftarrow \text{FSS}^\bullet.\text{Gen}(1^\lambda, f_{\alpha,s})$ For $\sigma \in \{0, 1\}$: $s_\sigma \leftarrow \text{FSS}^\bullet.\text{Eval}(\sigma, k_\sigma^\bullet, \alpha)$ $\rho_\sigma^\mathcal{F} := \text{PRG}(s_\sigma)$ $c^\mathcal{F} \leftarrow \text{FSS}^\mathcal{F}.\text{Gen}_s(1^\lambda, \rho_0^\mathcal{F}, \rho_1^\mathcal{F}, f)$ Return $k_0^\otimes := (k_0^\bullet, c^\mathcal{F})$	$H_1(\lambda)$: $s \xleftarrow{\$} \{0, 1\}^\lambda$ $(k_0^\bullet, k_1^\bullet) \leftarrow \text{FSS}^\bullet.\text{Gen}(1^\lambda, f_{\alpha,s})$ $s_0 \leftarrow \text{FSS}^\bullet.\text{Eval}(0, k_0^\bullet, \alpha)$ $s_1 := s_0 \oplus s$ For $\sigma \in \{0, 1\}$: $\rho_\sigma^\mathcal{F} := \text{PRG}(s_\sigma)$ $c^\mathcal{F} \leftarrow \text{FSS}^\mathcal{F}.\text{Gen}_s(1^\lambda, \rho_0^\mathcal{F}, \rho_1^\mathcal{F}, f)$ Return $k_0^\otimes := (k_0^\bullet, c^\mathcal{F})$	$H_2(\lambda)$: $s \xleftarrow{\$} \{0, 1\}^\lambda$ $k_0^\bullet \leftarrow \text{Sim}^\bullet(1^\lambda, \text{Leak}^\bullet(0, f_{\alpha,1^\lambda}))$ $s_0 \leftarrow \text{FSS}^\bullet.\text{Eval}(0, k_0^\bullet, \alpha)$ $s_1 := s_0 \oplus s$ For $\sigma \in \{0, 1\}$: $\rho_\sigma^\mathcal{F} := \text{PRG}(s_\sigma)$ $c^\mathcal{F} \leftarrow \text{FSS}^\mathcal{F}.\text{Gen}_s(1^\lambda, \rho_0^\mathcal{F}, \rho_1^\mathcal{F}, f)$ Return $k_0^\otimes := (k_0^\bullet, c^\mathcal{F})$
$H_3(\lambda)$: $k_0^\bullet \leftarrow \text{Sim}^\bullet(1^\lambda, \text{Leak}^\bullet(0, f_{\alpha,1^\lambda}))$ $s_0 \leftarrow \text{FSS}^\bullet.\text{Eval}(0, k_0^\bullet, \alpha)$ $\rho_0^\mathcal{F} := \text{PRG}(s_0)$ $\rho_1^\mathcal{F} \xleftarrow{\$} \{0, 1\}^{l(\lambda)}$ $c^\mathcal{F} \leftarrow \text{FSS}^\mathcal{F}.\text{Gen}_s(1^\lambda, \rho_0^\mathcal{F}, \rho_1^\mathcal{F}, f)$ Return $k_0^\otimes := (k_0^\bullet, c^\mathcal{F})$	$H_4(\lambda)$: $k_0^\bullet \leftarrow \text{Sim}^\bullet(1^\lambda, \text{Leak}^\bullet(0, f_{\alpha,1^\lambda}))$ $c^\mathcal{F} \leftarrow \text{Sim}_s^\mathcal{F}(1^\lambda, \text{Leak}^\mathcal{F}(0, f))$ Return $k_0^\otimes := (k_0^\bullet, c^\mathcal{F})$	$\text{Sim}^\otimes(1^\lambda, \text{leak}^\otimes)$: $\text{Parse leak}^\otimes := (\text{leak}^\bullet, \text{leak}^\mathcal{F})$ $k_0^\bullet \leftarrow \text{Sim}^\bullet(1^\lambda, \text{leak}^\bullet)$ $c^\mathcal{F} \leftarrow \text{Sim}_s^\mathcal{F}(1^\lambda, \text{leak}^\mathcal{F})$ Return $k_0^\otimes := (k_0^\bullet, c^\mathcal{F})$

Fig. 3: Hybrid distributions and simulator for the proof of [Theorem 1](#)

3.2 Application to Tensor Product of Sparse Vectors

In order to succinctly share the tensor product $e_0 \otimes e_1$ of two t_σ -sparse vectors $e_\sigma \in \mathbb{F}^{m_\sigma}$ (over some field \mathbb{F} with bit-size $b := \lceil \log |\mathbb{F}| \rceil$), one can share the function $f^\otimes(i, j) := e_{0,i} \cdot e_{1,j}$ as a sum (or, if the e_σ are regular, a concatenation) of $t_0 t_1$ point functions. This amounts to a key size of $\lambda + t_0 t_1 \cdot (\lceil \log(m_0 m_1) \rceil \cdot (\lambda + 2) + b)$ using DPFs from [\[BGI16\]](#).

However, we can use our improved FSS tensor construction to get a smaller key size (cf. [Theorem 1](#)). Note that, even though $f^\otimes = (i \mapsto e_{0,i}) \otimes (j \mapsto e_{1,j})$ is a tensor product, one cannot immediately apply the FSS tensor construction, since the left-hand side $(i \mapsto e_{0,i})$ is not in \mathcal{F}_1^\bullet . By using the bilinearity of the tensor product we can rewrite

$$f^\otimes = \sum_{i: e_{0,i} \neq 0} f_{i, e_{0,i}} \otimes (j \mapsto e_{1,j}) = \sum_{i: e_{0,i} \neq 0} f_{i,1} \otimes (j \mapsto e_{0,i} e_{1,j})$$

Since $f_{i,1} \in \mathcal{F}_1^\bullet$, the FSS tensor construction can now be applied to each summand individually, leading to a key size of

$$\lambda + t_0 \cdot \underbrace{\left(\lceil \log m_0 \rceil \cdot (\lambda + 2) \right)}_{\text{DPF key for } f_{i,s}} + \underbrace{t_1 \cdot \left(\lceil \log m_1 \rceil \cdot (\lambda + 2) + b \right)}_{\text{DMPF key for } j \mapsto e_{0,i} e_{1,j}}. \quad (6)$$

If we used the original FSS tensor construction from [\[BGI16\]](#), the right-hand side keys would have enjoyed an additional factor of 2.

Compared to the naive sum above (without any tensor construction), the left-hand side keys now enjoy a factor of only t_0 , while the right-hand side keys still enjoy a factor of $t_0 t_1$ (as in the naive sum). For $\log m_0 \approx \log m_1$, this optimization reduces the key size by a factor of ≈ 2 , compared to the naive sum. If $\log m_0 > \log m_1$, the reduction is potentially much larger. In the other case $\log m_0 < \log m_1$, it therefore makes sense to reverse the roles of e_0 and e_1 .

Note that [Equation \(6\)](#) assumes two additional minor optimizations, namely, (i) the private parts for all t_0 DPFs are all generated from a single private λ -bit seed (accounted for by the first summand) and (ii) we don't need to include a final correction word for $f_{i,s}$'s output

of size λ , since the DPF construction from [BGI16], on the special input, natively outputs shares of $s||1$, for some pseudorandom s .

When working with regular t_σ -sparse vectors (i.e., concatenation instead of sum of DPFs), the terms m_σ ($\sigma \in \{0, 1\}$) in Equations (6) and (7) get replaced by m_σ/t_σ .

Known-Index DPFs. In case one party is allowed to know the indices i where $e_{0,i} \neq 0$, we can replace the DPFs for $f_{i,1}$ by *known-index* DPFs (Definition 13). This is the case in our PCF for OLE (where, in fact, one party knows the whole vector \mathbf{e}_0 and not just the indices). Using the standard construction for known-index DPFs [SGRR19, BCG⁺19a, BCG⁺23], this reduces the key size to

$$\begin{aligned} |k_0| &= \lambda + t_0 \cdot \underbrace{\lceil \log(m_0) \rceil \cdot (\lambda + 1)}_{\text{known-index DPF key}} + t_0 t_1 \cdot (\lceil \log m_1 \rceil \cdot (\lambda + 2) + b), \\ |k_1| &= \lambda + t_0 t_1 \cdot (\lceil \log m_1 \rceil \cdot (\lambda + 2) + b). \end{aligned} \tag{7}$$

In our application, it is also the case, that the other party knows the non-zero indices of $j \mapsto e_{1,j}$, which occurs on the right-hand side of the tensor product. Unfortunately, we cannot exploit this, i.e., we cannot use known-index DPFs (at least not the standard construction) on the right-hand side, due to the requirements on $\text{FSS}^{\mathcal{F}}$ demanded by the tensor operation. Namely, known-index DPFs don't have the symmetry property from Definition 15, since $\text{Eval}(\sigma, k_\sigma, \cdot)$ works entirely different for $\sigma = 0$ and $\sigma = 1$.

4 Our Pseudorandom Correlation Functions

We present our programmable PCF for OLE in Section 4.1 and our optimized PCF for two-party Beaver triples in Section 4.2. Each time, we provide formulas for the respective key sizes and evaluation costs. We also conduct an asymptotic analysis in Section 4.3 and provide concrete parameters together with a performance evaluation in Section 4.4. Our PCFs work over any finite field, as long as the corresponding sparse LPN assumptions hold.

4.1 Programmable PCF for OLE

Here, we present our PCF for OLE, denoted PCF^{OLE} . The subprocedures $\text{PCF}^{\text{OLE}}.\text{Gen}$ and $\text{PCF}^{\text{OLE}}.\text{Eval}$ are presented in Figures 4 and 5, respectively. The corresponding security theorem is:

Theorem 2. *Let the parameters be as in Figure 4. If*

- *for all $1 \leq \ell \leq L$, the \mathbb{F} -SparseLPN($k^{(\ell)}, m^{(\ell-1)}, m^{(\ell)}, t$) assumption (cf. Definition 6) holds,*
- *all utilized FSS schemes are secure (cf. Definition 7), and*
- *$\text{DMPF}^{(e_0^{(L)} \odot e_1^{(L)})}$ additionally has pseudorandom outputs (cf. Definition 8),*

then, for $N = m^{(L)}$ queries and adversary size $B \in \text{poly}(\lambda)$, it holds that PCF^{OLE} (from Figures 4 and 5) is an $(N, B, \text{negl}(\lambda))$ -secure PCF for OLE.

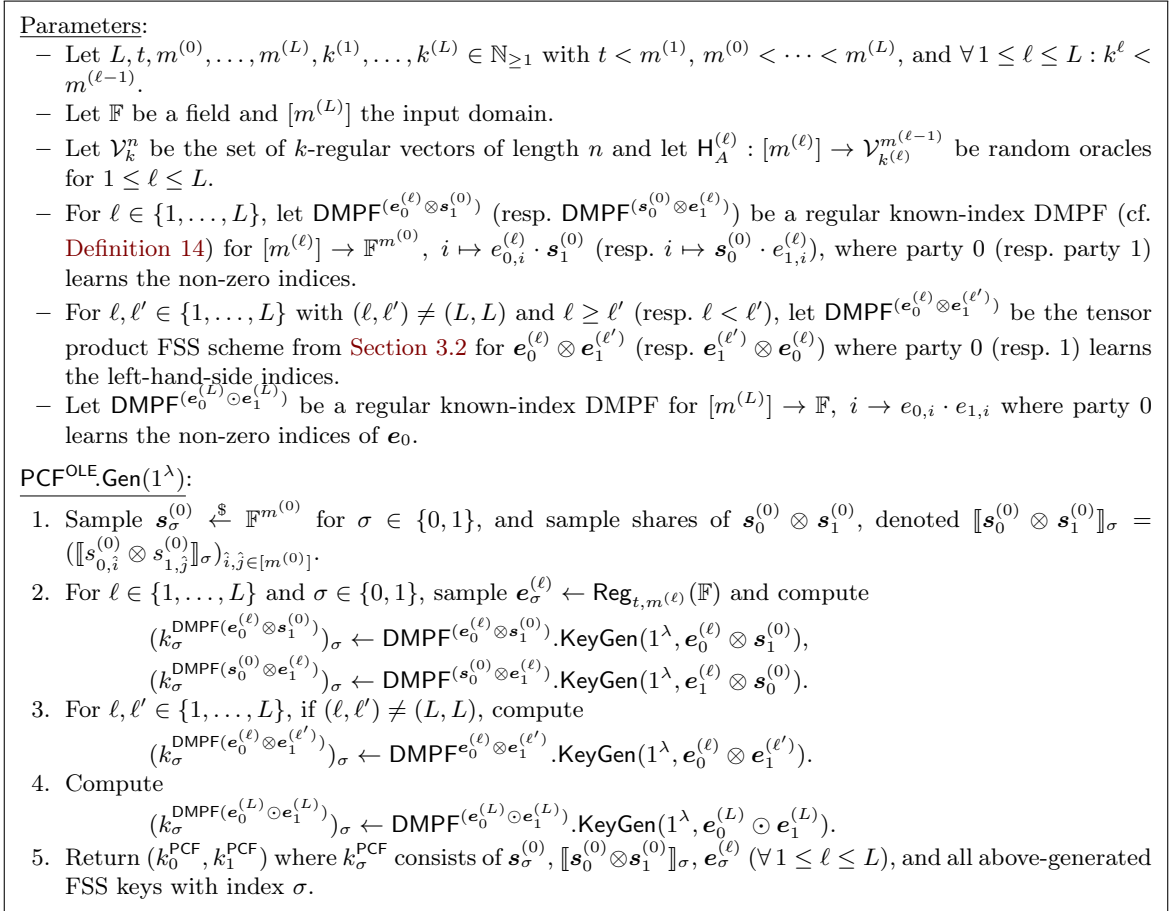


Fig. 4: Key generation of our PCF for OLE

$\text{PCF}^{\text{OLE}}.\text{Eval}(\sigma, k_\sigma^{\text{PCF}}, \hat{i} \in [m^{(L)}]):$

1. Compute $\mathbf{a}_i \leftarrow \mathbf{H}_A^{(L)}(\hat{i})$, $I := \{i \mid a_{x,i} \neq 0\}$.
2. Return $(\text{EvalRecInput}(\sigma, k_\sigma^{\text{PCF}}, L, \hat{i}), z_\sigma)$ where

$$\begin{aligned} z_\sigma &= \sum_{i,j \in I} a_{i,i} \cdot a_{i,j} \cdot \text{EvalRec} \boxed{1}(\sigma, k_\sigma^{\text{PCF}}, L-1, i, j) \\ &+ \sum_{j \in I} a_{i,j} \cdot \text{EvalRec} \boxed{2}(\sigma, k_\sigma^{\text{PCF}}, L, L-1, \hat{i}, j) \\ &+ \sum_{i \in I} a_{i,i} \cdot \text{EvalRec} \boxed{3}(\sigma, k_\sigma^{\text{PCF}}, L-1, L, i, \hat{i}) \\ &+ \text{DMPF}.\text{Eval}(\sigma, k_\sigma^{\text{DMPF}(e_0^{(L)} \odot e_1^{(L)})}, \hat{i}). \end{aligned}$$

$\text{EvalRecInput}(\sigma, k_\sigma^{\text{PCF}}, \ell \in \{0, \dots, L\}, \hat{i} \in [m^{(\ell)}]):$

1. If $\ell = 0$, return $s_{\sigma, \hat{i}}^{(0)}$.
2. Compute $\mathbf{a}_i \leftarrow \mathbf{H}_A^{(\ell)}(\hat{i})$, $I := \{i \mid a_{i,i} \neq 0\}$.
3. Return $\sum_{i \in I} a_{i,i} \cdot \text{EvalRecInput}(\sigma, k_\sigma^{\text{PCF}}, \ell-1, i) + e_{\sigma, \hat{i}}^{(\ell)}$.

$\text{EvalRec} \boxed{1}(\sigma, k_\sigma^{\text{PCF}}, \ell \in \{0, \dots, L-1\}, \hat{i} \in [m^{(\ell)}], \hat{j} \in [m^{(\ell)}]):$

1. If $\ell = 0$, return $\llbracket s_{0, \hat{i}}^{(0)} \cdot s_{1, \hat{j}}^{(0)} \rrbracket_\sigma$.
2. Compute $\mathbf{a}_i \leftarrow \mathbf{H}_A^{(\ell)}(\hat{i})$, $I := \{i \mid a_{i,i} \neq 0\}$,
 $\mathbf{a}_j \leftarrow \mathbf{H}_A^{(\ell)}(\hat{j})$, $J := \{j \mid a_{j,j} \neq 0\}$.
3. Return:

$$\begin{aligned} &\sum_{i \in I, j \in J} a_{i,i} \cdot a_{j,j} \cdot \text{EvalRec} \boxed{1}(\sigma, k_\sigma^{\text{PCF}}, \ell-1, i, j) \\ &+ \sum_{j \in J} a_{j,j} \cdot \text{EvalRec} \boxed{2}(\sigma, k_\sigma^{\text{PCF}}, \ell, \ell-1, \hat{j}, j) \\ &+ \sum_{i \in I} a_{i,i} \cdot \text{EvalRec} \boxed{3}(\sigma, k_\sigma^{\text{PCF}}, \ell-1, \ell, i, \hat{i}) \\ &+ \text{DMPF}.\text{Eval}(\sigma, k_\sigma^{\text{DMPF}(e_0^{(\ell)} \otimes e_1^{(\ell)})}, (\hat{i}, \hat{j})). \end{aligned}$$

$\text{EvalRec} \boxed{2}(\sigma, k_\sigma^{\text{PCF}}, \ell \in \{1, \dots, L\}, \ell' \in \{0, \dots, \ell-1\}, \hat{i} \in [m^{(\ell)}], \hat{j} \in [m^{(\ell')}]):$

1. If $\ell' = 0$, return $(\text{DMPF}.\text{Eval}(\sigma, k_\sigma^{\text{DMPF}(e_0^{(\ell)} \otimes s_1^{(0)})}, \hat{i}))_{\hat{j}}$.
2. Compute $\mathbf{a}_j \leftarrow \mathbf{H}_A^{(\ell')}(\hat{j})$, $J := \{j \mid a_{j,j} \neq 0\}$.
3. Return:

$$\begin{aligned} &\sum_{j \in J} a_{j,j} \cdot \text{EvalRec} \boxed{2}(\sigma, k_\sigma^{\text{PCF}}, \ell, \ell'-1, \hat{i}, j) \\ &+ \text{DMPF}.\text{Eval}(\sigma, k_\sigma^{\text{DMPF}(e_0^{(\ell)} \otimes e_1^{(\ell')})}, (\hat{i}, \hat{j})). \end{aligned}$$

$\text{EvalRec} \boxed{3}(\sigma, k_\sigma^{\text{PCF}}, \ell' \in \{0, \dots, \ell-1\}, \ell \in \{1, \dots, L\}, \hat{i} \in [m^{(\ell)}], \hat{j} \in [m^{(\ell')}]):$

1. If $\ell' = 0$, return $(\text{DMPF}.\text{Eval}(\sigma, k_\sigma^{\text{DMPF}(s_0^{(0)} \otimes e_1^{(\ell)})}, \hat{j}))_{\hat{i}}$.
2. Compute $\mathbf{a}_i \leftarrow \mathbf{H}_A^{(\ell')}(\hat{i})$, $I := \{i \mid a_{i,i} \neq 0\}$.
3. Return:

$$\begin{aligned} &\sum_{i \in I} a_{i,i} \cdot \text{EvalRec} \boxed{3}(\sigma, k_\sigma^{\text{PCF}}, \ell'-1, \ell, i, \hat{j}) \\ &+ \text{DMPF}.\text{Eval}(\sigma, k_\sigma^{\text{DMPF}(e_0^{(\ell')} \otimes e_1^{(\ell)})}, (\hat{i}, \hat{j})). \end{aligned}$$

Fig. 5: Evaluation of our PCF for OLE

Proof. We first show that our PCF always produces “correct” OLE-correlated outputs, i.e., $((x_0, z_0), (x_1, z_1))$ with $z_0 + z_1 = x_0 \cdot x_1$, irrespective of whether they are pseudorandom. This will be useful in our subsequent proofs of the *strong pseudorandom OLE-correlated outputs* and *strong security* properties from Definition 2. The PCF evaluation in Figure 5 uses the following subroutines:

- EvalReInput $(\sigma, k_\sigma^{\text{PCF}}, \ell, \hat{i})$ recursively computes $s_{\sigma, \hat{i}}^{(\ell)} = \langle \mathbf{a}_{\hat{i}}^{(\ell)}, \mathbf{s}_\sigma^{(\ell-1)} \rangle + e_{\sigma, \hat{i}}^{(\ell)}$.
- EvalRec $\boxed{1}(\sigma, k_\sigma^{\text{PCF}}, \ell, \hat{i}, \hat{j})$ computes a share of $s_{0, \hat{i}}^{(\ell)} \cdot s_{1, \hat{j}}^{(\ell)}$ according to Equation (3).
- EvalRec $\boxed{2}(\sigma, k_\sigma^{\text{PCF}}, \ell, \ell', \hat{i}, \hat{j})$ computes a share of $e_{0, \hat{i}}^{(\ell)} \cdot s_{1, \hat{j}}^{(\ell')}$ according to Equation (4).
- EvalRec $\boxed{3}(\sigma, k_\sigma^{\text{PCF}}, \ell', \ell, \hat{i}, \hat{j})$ analogously computes a share of $s_{0, \hat{i}}^{(\ell')} \cdot e_{1, \hat{j}}^{(\ell)}$.
- Finally, PCF^{OLE}.Eval $(\sigma, k_\sigma^{\text{PCF}}, \hat{i})$ returns (x_σ, z_σ) where $x_\sigma := s_{\sigma, \hat{i}}^{(L)}$ and z_σ is a share of $s_{0, \hat{i}}^{(L)} \cdot s_{1, \hat{i}}^{(L)}$ that is computed just like EvalRec $\boxed{1}(\sigma, k_\sigma^{\text{PCF}}, L, \hat{i}, \hat{i})$ would do it, except that we only need a DMPF for the component-wise product $e_0^{(L)} \odot e_1^{(L)}$, instead of the tensor product.

These claims are easy to verify by looking at Figure 5. Hence, the outputs of our PCF are always correct in the sense that $z_0 + z_1 = x_0 \cdot x_1$. We now proceed with the proofs of the two properties from Definition 2.

Strong Pseudorandom OLE-Correlated Outputs. Let \mathcal{A} be a polynomially bounded non-uniform adversary. It suffices to show that $\text{Game}_{\mathcal{A}, 1}^{\text{pr}} \approx_c \text{Game}_{\mathcal{A}, 0}^{\text{pr}}$, as this implies in particular that $|\Pr[\text{Game}_{\mathcal{A}, 0}^{\text{pr}}(\lambda) = 1] - \Pr[\text{Game}_{\mathcal{A}, 1}^{\text{pr}}(\lambda) = 1]| \in \text{negl}(\lambda)$. We do so via the following sequence of hybrids.

- $H_0 := \text{Game}_{\mathcal{A}, 1}^{\text{pr}}$.
- H_1 is identical to H_0 , except that we redefine O_1^{pr} as follows:

$O_1^{\text{pr}}(\hat{i})$:
 $(x_0, z_0) := \text{PCF}^{\text{OLE}}.\text{Eval}_{k_0^{\text{PCF}}}(\hat{i})$
 $x_1 := \text{EvalReInput}(1, k_1^{\text{PCF}}, L, \hat{i})$
 $z_1 := x_0 \cdot x_1 - z_0$
Return $((x_0, z_0), (x_1, z_1))$

Note that this doesn’t change how x_1 is computed. Therefore, and by the correctness ($z_0 + z_1 = x_0 \cdot x_1$) argued above, H_1 is distributed identically to H_0 . Now, observe that H_1 no longer accesses any FSS keys from k_1^{PCF} . In particular, it doesn’t access $k_1^{\text{DMPF}(e_0^{(L)} \odot e_1^{(L)})}$. By the *pseudorandom outputs* property (cf. Definition 8) of $\text{DMPF}(e_0^{(L)} \odot e_1^{(L)})$, we can therefore view $\text{DMPF}.\text{Eval}(0, k_0^{\text{DMPF}(e_0^{(L)} \odot e_1^{(L)})}, \hat{i})$ as a pseudorandom mask on z_0 , i.e., we can replace z_0 by a random value as a next step:

- H_2^0 is identical to H_1 , except that we sample $z_0 \xleftarrow{\$} \mathbb{F}^{m(L)}$ once in the beginning of the game and redefine O_1^{pr} as follows:

$O_1^{\text{pr}}(\hat{i})$:
 $x_0 := \text{EvalReInput}(0, k_0^{\text{PCF}}, L, \hat{i})$
 $z_0 := z_{0, \hat{i}}$
 $x_1 := \text{EvalReInput}(1, k_1^{\text{PCF}}, L, \hat{i})$
 $z_1 := x_0 \cdot x_1 - z_0$
Return $((x_0, z_0), (x_1, z_1))$

H_2^0 is computationally indistinguishable from H_1 as argued above.

- For $\tilde{\ell} \in \{1, \dots, L\}$: $H_2^{\tilde{\ell}}$ is identical to H_2^0 , except that we sample two vectors $\mathbf{s}_0^{(\tilde{\ell})}, \mathbf{s}_1^{(\tilde{\ell})} \leftarrow \mathbb{F}^{m^{(\tilde{\ell})}}$ once in the beginning of the game and, in O_1^{pr} , we redefine `EvalReInput` as follows:

EvalReInput $(\sigma, k_\sigma^{\text{PCF}}, \ell, \hat{i})$:

If $\ell = \tilde{\ell}$, return $s_{\sigma, \hat{i}}^{(\tilde{\ell})}$. Otherwise, proceed as defined before.

Observe that the only difference between $H_2^{\tilde{\ell}-1}$ and $H_2^{\tilde{\ell}}$ is the following:

- In $H_2^{\tilde{\ell}-1}$, the recursion of `EvalReInput` stops at $\ell = \tilde{\ell}-1$, i.e., `EvalReInput` $(\sigma, k_\sigma^{\text{PCF}}, \tilde{\ell}, \hat{i})$ returns the \hat{i} -th component of $\mathbf{A}^{(\tilde{\ell})} \mathbf{s}_\sigma^{(\tilde{\ell}-1)} + \mathbf{e}_\sigma^{(\tilde{\ell})}$, and $\mathbf{s}_\sigma^{(\tilde{\ell}-1)}, \mathbf{e}_\sigma^{(\tilde{\ell})}$ aren't used for anything else during the whole game.
- In $H_2^{\tilde{\ell}}$, the recursion of `EvalReInput` stops at $\ell = \tilde{\ell}$, i.e., `EvalReInput` $(\sigma, k_\sigma^{\text{PCF}}, \tilde{\ell}, \hat{i})$ returns the \hat{i} -th component of $\mathbf{s}_\sigma^{(\tilde{\ell})}$, and $\mathbf{s}_\sigma^{(\tilde{\ell})}$ isn't used for anything else during the whole game.

Since each row of $\mathbf{A}^{(\tilde{\ell})}$ is sampled from $\text{Reg}_{k^{(\tilde{\ell})}, m^{(\tilde{\ell}-1)}}(\mathbb{F})$, the whole matrix $\mathbf{A}^{(\tilde{\ell})}$ is distributed identically to `RegularCodeGen` $(k^{(\tilde{\ell})}, m^{(\tilde{\ell}-1)}, m^{(\tilde{\ell})}, \mathbb{F})$. Furthermore, $\mathbf{e}_\sigma^{(\tilde{\ell})}$ is (in the key generation) sampled from $\text{Reg}_{t, m^{(\tilde{\ell})}}(\mathbb{F})$. Hence, we can apply the \mathbb{F} -SparseLPN $(k^{(\tilde{\ell})}, m^{(\tilde{\ell}-1)}, m^{(\tilde{\ell})}, t)$ assumption, which gives us $H_2^{\tilde{\ell}-1} \approx_c H_2^{\tilde{\ell}}$.

- $H_3 := \text{Game}_{\mathcal{A}, 0}^{\text{pr}}$. Observe that the only difference between H_2^L and H_3 is that H_2^L samples and stores the whole size- $m^{(L)}$ batch of OLEs $((\mathbf{s}_0^{(L)}, \mathbf{z}_0), (\mathbf{s}_1^{(L)}, \mathbf{s}_0^{(L)} \odot \mathbf{s}_1^{(L)} - \mathbf{z}_0))$ in the beginning of the game, while H_3 samples and stores the individual OLEs on-demand (on first access). This doesn't make a difference in the observable output, so H_2^L is distributed identically to $H_3 = \text{Game}_{\mathcal{A}, 0}^{\text{pr}}$.

Strong Security. W.l.o.g., let party $\sigma = 0$ be corrupted (since the whole proof works symmetrically for $\sigma = 1$). Let \mathcal{A} be a polynomially bounded non-uniform adversary. Again, it suffices to show that $\text{Game}_{\mathcal{A}, \sigma, 1}^{\text{sec}} \approx_c \text{Game}_{\mathcal{A}, \sigma, 0}^{\text{sec}}$, as we do via the following sequence of hybrids.

- $H_0 := \text{Game}_{\mathcal{A}, \sigma, 1}^{\text{sec}}$.
- H_1 is identical to H_0 , except that we redefine $O_{\sigma, 1}^{\text{sec}}$ as follows:

$O_{\sigma, 1}^{\text{pr}}(\hat{i})$:

$(x_0, z_0) := \text{PCF}^{\text{OLE}}.\text{Eval}_{k_0^{\text{PCF}}}(\hat{i})$

$x_1 := \text{EvalReInput}(1, k_1^{\text{PCF}}, L, \hat{i})$

$z_1 := x_0 \cdot x_1 - z_0$

Return (x_1, z_1)

Just like in the previous part of the proof, by the correctness $(z_0 + z_1 = x_0 \cdot x_1)$, H_1 is distributed identically to H_0 . Again, observe that H_1 no longer accesses any FSS keys from k_1^{PCF} . Intuitively, this allows us to use the *security* (cf. [Definition 7](#)) of the utilized FSS schemes to replace party 0's FSS keys by simulated keys:

- H_2 is identical to H_1 , except that, in $\text{PCF}^{\text{OLE}}.\text{Gen}(1^\lambda)$, we replace each FSS key generation by a corresponding simulator's output $k_0^{\text{DMPF}} \leftarrow \text{Sim}^{\text{DMPF}}(1^\lambda, \text{Leak}^{\text{DMPF}}(0, f))$. Note that $H_1 \approx_c H_2$ isn't obvious, because the oracle's output depends not only on the FSS keys, but also on the \mathbf{e} and \mathbf{s} vectors from the PCF key generation. However, we can use the same trick as in the security proof of our improved tensor construction ([Theorem 1](#)), i.e., we fix for each λ a "worst-case" value $r^{(\lambda)}$ of the random coins from which the \mathbf{e} and \mathbf{s} vectors are sampled: Let H_i^r denote that, in the definition of H_i , we sample all \mathbf{e} and \mathbf{s} vectors deterministically from coins r . Then the advantage of a distinguisher \mathcal{D} is

$$\Pr[\mathcal{D}(H_1(\lambda)) = 1] - \Pr[\mathcal{D}(H_2(\lambda)) = 1] \leq \Pr[\mathcal{D}(H_1^r(\lambda)) = 1] - \Pr[\mathcal{D}(H_2^r(\lambda)) = 1].$$

The coins $r^{(\lambda)}$ uniquely determine the functions that are shared via FSS during $\text{PCF}^{\text{OLE}}.\text{Gen}(1^\lambda)$. This defines infinite function sequences (one function for every value of λ and every FSS instance) as in the *security* part of [Definition 7](#). By the security of the utilized FSS schemes, it follows that $H_1^{r^{(\lambda)}}(\lambda) \approx_c H_2^{r^{(\lambda)}}(\lambda)$, so the advantage of \mathcal{D} is negligible.

- H_3 is identical to H_2 , except that we sample $\mathbf{x}_1 \xleftarrow{\$} \mathbb{F}^{m^{(L)}}$ once in the beginning of the game and redefine $O_{\sigma,1}^{\text{sec}}$ as follows:

$O_{\sigma,1}^{\text{pr}}(\hat{i})$:
 $(x_0, z_0) := \text{PCF}^{\text{OLE}}.\text{Eval}_{k_0^{\text{PCF}}}(\hat{i})$
 $x_1 := x_{1,\hat{i}}$
 $z_1 := x_0 \cdot x_1 - z_0$
Return (x_1, z_1)

All of the utilized FSS schemes have a leakage function that doesn't output any information about the vectors $\mathbf{s}_1^{(0)}, \mathbf{e}_1^{(1)}, \dots, \mathbf{e}_1^{(L)}$ to party 0 (except for the lengths and domains, which are public anyway). Therefore, in H_2 , these vectors are only used during the computation of $\text{EvalReInput}(1, k_1^{\text{PCF}}, L, \hat{i})$. Hence, we can inductively apply the sparse LPN assumption (similarly as in the previous part of the proof) to see that $H_2 \approx_c H_3$.

- H_4 is identical to H_3 , except that we undo the modification of the key generation, i.e., we go back to using real instead of simulated FSS keys. $H_3 \approx_c H_4$ holds by the same argument as $H_1 \approx_c H_2$.
- $H_5 := \text{Game}_{\mathcal{A},\sigma,0}^{\text{sec}}$. Observe that, in H_4 , the output (x_1, z_1) is distributed identically to $\text{OLE.RSample}(1^\lambda, 0, (x_0, z_0))$. The only difference between H_4 and H_5 is that H_4 samples and stores the whole size- $m^{(L)}$ batch \mathbf{x}_1 in the beginning of the game, while H_5 samples and stores the individual entries on-demand (on first access). This doesn't make a difference in the observable output, so H_4 is distributed identically to $H_5 = \text{Game}_{\mathcal{A},\sigma,0}^{\text{sec}}$.

□

Theorem 3. PCF^{OLE} (from [Figures 4 and 5](#)) is programmable (in the sense of [Definition 4](#)).

Proof. To see that PCF^{OLE} is programmable, give the key generation additional arguments $\rho_0, \rho_1 \in \{0, 1\}^\lambda$ and sample $\mathbf{s}_\sigma^{(0)}$ and $\mathbf{e}_\sigma^{(\ell)}$ (for all $\ell \in \{1, \dots, L\}$) using a PRG with seed ρ_σ . □

Recall that an OLE correlation has the form $((x_0, z_0), (x_1, z_1))$. When generating multiple key pairs, ρ_σ can be reused, causing the corresponding x_σ 's in the PCF output to coincide with the x_σ 's from a *different* PCF instance. This is what allows us to generate multiparty Beaver triples or, more generally, any multiparty degree-two correlation, as already explained in [Section 1.2](#).

Key Size. For the key size analysis of PCF^{OLE} , we assume that the key k_σ^{PCF} contains the λ -bit seed ρ_σ , instead of $\mathbf{s}_\sigma^{(0)}, \mathbf{e}_\sigma^{(1)}, \dots, \mathbf{e}_\sigma^{(L)}$, which can all be efficiently sampled from seed ρ_σ . Apart from that, the key contains shares of $\llbracket \mathbf{s}_0^{(0)} \otimes \mathbf{s}_1^{(0)} \rrbracket_\sigma$ with total size $(m^{(0)})^2 b$ where $b := \lceil \log |\mathbb{F}| \rceil$, and several FSS keys. Since we use regular noise, we can express each $\mathbf{e}_\sigma^{(\ell)} \otimes \mathbf{s}_{1-\sigma}^{(0)}$ as a concatenation of t point functions with range $\mathbb{F}^{m^{(0)}}$. For these, we use standard *known-index* DPFs [[SGRR19](#), [BCG⁺19a](#), [BCG⁺23](#)]. The key size for t of them is

$$\begin{aligned} |k_0^{\text{DMPF}(\mathbf{e}_0^{(\ell)} \otimes \mathbf{s}_1^{(0)})}| &= |k_1^{\text{DMPF}(\mathbf{s}_0^{(0)} \otimes \mathbf{e}_1^{(\ell)})}| = t \cdot (\lceil \log(m^{(\ell)}/t) \rceil \cdot (\lambda + 1) + m^{(0)}b), \\ |k_1^{\text{DMPF}(\mathbf{e}_0^{(\ell)} \otimes \mathbf{s}_1^{(0)})}| &= |k_0^{\text{DMPF}(\mathbf{s}_0^{(0)} \otimes \mathbf{e}_1^{(\ell)})}| = t \cdot \lambda. \end{aligned}$$

For terms of the form $\mathbf{e}_0^{(\ell)} \otimes \mathbf{e}_1^{(\ell')}$, where $\ell \geq \ell'$ and $(\ell, \ell') \neq (L, L)$, we use our tensor product FSS scheme from [Section 3.2](#) with known-index DPFs on the left-hand side and, thus, key size

$$\begin{aligned} |k_0^{\text{DMPF}(\mathbf{e}_0^{(\ell)} \otimes \mathbf{e}_1^{(\ell')})}| &= \lambda + t \cdot \lceil \log(m^{(\ell)}/t) \rceil \cdot (\lambda + 1) \\ &\quad + t^2 \cdot (\lceil \log(m^{(\ell')}/t) \rceil \cdot (\lambda + 2) + b), \\ |k_1^{\text{DMPF}(\mathbf{e}_0^{(\ell)} \otimes \mathbf{e}_1^{(\ell')})}| &= \lambda + t^2 \cdot (\lceil \log(m^{(\ell')}/t) \rceil \cdot (\lambda + 2) + b). \end{aligned}$$

For $\ell < \ell'$ we use the same scheme, but on $\mathbf{e}_1^{(\ell')} \otimes \mathbf{e}_0^{(\ell)}$ (i.e., the operands' order is reversed), to have the smaller vector on the right-hand side, as this yields smaller FSS keys. Finally, for $\mathbf{e}_0^{(L)} \odot \mathbf{e}_1^{(L)}$, we use a concatenation of standard known-index DPFs. We can choose an arbitrary designated party: Say, we choose party 0, then we use the indices i where $e_{0,i}^{(L)} \neq 0$, and the corresponding values $e_{0,i}^{(L)} e_{1,i}^{(L)}$. With this, the key size for $\mathbf{e}_0^{(L)} \odot \mathbf{e}_1^{(L)}$ is

$$\begin{aligned} |k_0^{\text{DMPF}(\mathbf{e}_0^{(L)} \odot \mathbf{e}_1^{(L)})}| &= t \cdot (\lceil \log(m^{(L)}/t) \rceil \cdot (\lambda + 1) + b), \\ |k_1^{\text{DMPF}(\mathbf{e}_0^{(L)} \odot \mathbf{e}_1^{(L)})}| &= t \cdot \lambda. \end{aligned}$$

Evaluation Cost. The bottleneck of evaluating our PCF are the PRG evaluations during DMPF evaluations. Each evaluation of a DPF with domain $[m']$ takes $\lceil \log(m') \rceil$ evaluations of a PRG $G : \{0, 1\}^\lambda \rightarrow \{0, 1\}^{2\lambda+2}$. Evaluation of a regular DMPF with domain $[m']$, implemented as a concatenation of t' DPFs, therefore takes $\lceil \log(m'/t') \rceil$ PRG evaluations. Let $n_{1,\ell}^{\text{PRG}}$, $n_{2,\ell,\ell'}^{\text{PRG}}$, and $n_{3,\ell,\ell'}^{\text{PRG}}$ be the number of PRG evaluations during a call of [EvalRec 1](#), [EvalRec 2](#), and [EvalRec 3](#) (with argument ℓ and, if applicable, ℓ'), respectively. Due to symmetry, we have $n_{2,\ell,\ell'}^{\text{PRG}} = n_{3,\ell',\ell}^{\text{PRG}}$ and, looking at [Figure 5](#), we obtain the recursive formula

$$\begin{aligned} n_{1,\ell}^{\text{PRG}} &= \begin{cases} 2\lceil \log((m^{(\ell)})/t) \rceil + (k^{(\ell)})^2 \cdot n_{1,\ell-1}^{\text{PRG}} + 2 \cdot k^{(\ell)} \cdot n_{2,\ell,\ell-1}^{\text{PRG}} & \ell > 0, \\ 0 & \ell = 0, \end{cases} \\ n_{2,\ell,\ell'}^{\text{PRG}} &= \begin{cases} \lceil \log(m^{(\ell)}/t) \rceil + \lceil \log(m^{(\ell')}/t) \rceil + k^{(\ell')} \cdot n_{2,\ell,\ell'-1}^{\text{PRG}} & \ell' > 0 \\ \lceil \log(m^{(\ell)}/t) \rceil + 1 & \ell' = 0. \end{cases} \end{aligned}$$

In the formula for $n_{2,\ell,0}^{\text{PRG}}$, the final $+1$ is because we need to expand the PRG output at the leaf, since the payload $e_{0,i}^{(\ell)} \cdot s_{1,i}^{(0)}$ is larger than λ bits. We only need one part of the expanded PRG output (as long as $\lceil \log|\mathbb{F}| \rceil \leq \lambda$), since we only access a single entry $e_{0,i}^{(\ell)} \cdot s_{1,j}^{(0)}$ of the payload, so adding $+1$ in the formula is sufficient.

When setting $\ell = L$ in above formula for $n_{1,\ell}^{\text{PRG}}$, we obtain an upper bound for the number of PRG evaluations during a call of $\text{PCF}^{\text{OLE}}.\text{Eval}$, since, on the top-level, we have the (somewhat cheaper) component-wise product $\mathbf{e}_0^{(L)} \odot \mathbf{e}_1^{(L)}$ instead of the tensor product.

4.2 Optimized PCF for Two-Party Beaver Triples

In the two-party setting, we construct a variant of our PCF, denoted $\text{PCF}^{\text{Beaver}}$, that produces two-party Beaver triples directly (instead of relying on $M(M-1) = 2$ instances of our PCF for OLE). We present $\text{PCF}^{\text{Beaver}}$ in [Figures 6](#) and [7](#). We give the following security theorem without proof, since it can be proven analogous to [Theorem 2](#).

Theorem 4. *Let the parameters be as in Figure 6. If*

- *for all $1 \leq \ell \leq L$, the \mathbb{F} -SparseLPN($k^{(\ell)}, m^{(\ell-1)}, m^{(\ell)}, t$) assumption (cf. Definition 6) holds,*
- *all utilized FSS schemes are secure (cf. Definition 7), and*
- *DMPF($e_0^{(L)}$), DMPF($e_1^{(L)}$), and DMPF($e_0^{(L)} \odot e_1^{(L)}$) additionally have pseudorandom outputs (cf. Definition 8),*

then, for $N = m^{(L)}$ queries and adversary size $B \in \text{poly}(\lambda)$, it holds that $\text{PCF}^{\text{Beaver}}$ (from Figures 6 and 7) is an $(N, B, \text{negl}(\lambda))$ -secure PCF for two-party Beaver triples, i.e., for the correlation $((a_0, b_0, c_0), (a_1, b_1, c_1))$, where all entries are uniformly random over \mathbb{F} , except for c_1 , which is uniquely determined by $(a_0 + a_1)(b_0 + b_1) = c_0 + c_1$.

Parameters:

- Let the parameters be as in Figure 4, except that all known-index FSS schemes (i.e., regular known-index DMPFs, and known-index DPFs used in the tensor product FSS scheme) are replaced by their unknown-index counterparts.
- Additionally, for $\ell \in \{1, \dots, L\}$ and $\sigma \in \{0, 1\}$, let DMPF($e_\sigma^{(\ell)}$) be a regular DMPF (cf. Definition 12) for $[m^{(\ell)}] \rightarrow \mathbb{F}$, $i \mapsto e_{\sigma,i}^{(\ell)}$.

PCF^{Beaver}.Gen(1^λ):

1. Sample $\mathbf{s}_\tau^{(0)} \xleftarrow{\$} \mathbb{F}^{m^{(0)}}$ for $\tau \in \{0, 1\}$, and sample shares of $\mathbf{s}_\tau^{(0)}$ and $\mathbf{s}_0^{(0)} \otimes \mathbf{s}_1^{(0)}$, denoted $\llbracket \mathbf{s}_\tau^{(0)} \rrbracket_\sigma = (\llbracket s_{\tau,i}^{(0)} \rrbracket_\sigma)_{i \in [m^{(0)})}$ and $\llbracket \mathbf{s}_0^{(0)} \otimes \mathbf{s}_1^{(0)} \rrbracket_\sigma = (\llbracket s_{0,i}^{(0)} \otimes s_{1,j}^{(0)} \rrbracket_\sigma)_{i,j \in [m^{(0)})}$.
2. For $\ell \in \{1, \dots, L\}$ and $\tau \in \{0, 1\}$, sample $e_\tau^{(\ell)} \leftarrow \text{Reg}_{t, m^{(\ell)}}(\mathbb{F})$ and compute
 - $(k_\sigma^{\text{DMPF}(e_0^{(\ell)})})_\sigma \leftarrow \text{DMPF}(e_0^{(\ell)}).\text{KeyGen}(1^\lambda, e_0^{(\ell)})$,
 - $(k_\sigma^{\text{DMPF}(e_1^{(\ell)})})_\sigma \leftarrow \text{DMPF}(e_1^{(\ell)}).\text{KeyGen}(1^\lambda, e_1^{(\ell)})$,
 - $(k_\sigma^{\text{DMPF}(e_0^{(\ell)} \otimes s_1^{(0)})})_\sigma \leftarrow \text{DMPF}(e_0^{(\ell)} \otimes s_1^{(0)}).\text{KeyGen}(1^\lambda, e_0^{(\ell)} \otimes s_1^{(0)})$,
 - $(k_\sigma^{\text{DMPF}(s_0^{(0)} \otimes e_1^{(\ell)})})_\sigma \leftarrow \text{DMPF}(s_0^{(0)} \otimes e_1^{(\ell)}).\text{KeyGen}(1^\lambda, e_1^{(\ell)} \otimes s_0^{(0)})$.
3. For $\ell, \ell' \in \{1, \dots, L\}$, if $(\ell, \ell') \neq (L, L)$, compute
 - $(k_\sigma^{\text{DMPF}(e_0^{(\ell)} \otimes e_1^{(\ell')})})_\sigma \leftarrow \text{DMPF}(e_0^{(\ell)} \otimes e_1^{(\ell')}).\text{KeyGen}(1^\lambda, e_0^{(\ell)} \otimes e_1^{(\ell')})$.
4. Compute
 - $(k_\sigma^{\text{DMPF}(e_0^{(L)} \odot e_1^{(L)})})_\sigma \leftarrow \text{DMPF}(e_0^{(L)} \odot e_1^{(L)}).\text{KeyGen}(1^\lambda, e_0^{(L)} \odot e_1^{(L)})$.
5. Return $(k_0^{\text{PCF}}, k_1^{\text{PCF}})$ where k_σ^{PCF} consists of $\llbracket \mathbf{s}_0^{(0)} \rrbracket_\sigma$, $\llbracket \mathbf{s}_1^{(0)} \rrbracket_\sigma$, $\llbracket \mathbf{s}_0^{(0)} \otimes \mathbf{s}_1^{(0)} \rrbracket_\sigma$, and all above-generated FSS keys with index σ .

Fig. 6: Key generation of our PCF for two-party Beaver triples

<p>$\text{PCF}^{\text{Beaver}}.\text{Eval}(\sigma, k_\sigma^{\text{PCF}}, \hat{i} \in [m^{(L)}]):$</p> <ol style="list-style-type: none"> 1. Compute $\mathbf{a}_i \leftarrow \text{H}_A^{(L)}(\hat{i})$, $I := \{i \mid a_{x,i} \neq 0\}$. 2. Return $(\text{EvalReInput}(\sigma, k_\sigma^{\text{PCF}}, 0, L, \hat{i}), \text{EvalReInput}(\sigma, k_\sigma^{\text{PCF}}, 1, L, \hat{i}), z_\sigma)$ where z_σ is computed as in Figure 5. <p>$\text{EvalReInput}(\sigma, k_\sigma^{\text{PCF}}, \tau \in \{0, 1\}, \ell \in \{0, \dots, L\}, \hat{i} \in [m^{(\ell)}]):$</p> <ol style="list-style-type: none"> 1. If $\ell = 0$, return $\llbracket s_{\tau, \hat{i}}^{(0)} \rrbracket_\sigma$. 2. Compute $\mathbf{a}_i \leftarrow \text{H}_A^{(\ell)}(\hat{i})$, $I := \{i \mid a_{i,i} \neq 0\}$. 3. Return $\sum_{i \in I} a_{i,i} \cdot \text{EvalReInput}(\sigma, k_\sigma^{\text{PCF}}, \tau, \ell - 1, i) + \text{DMPF.Eval}(\sigma, k_\sigma^{\text{DMPF}(e_\tau^{(\ell)})}, \hat{i})$.

Fig. 7: Evaluation of our PCF for two-party Beaver triples

Key Size. In $\text{PCF}^{\text{Beaver}}$, the key k_σ^{PCF} contains $\llbracket \mathbf{s}_0^{(0)} \rrbracket_\sigma, \llbracket \mathbf{s}_1^{(0)} \rrbracket_\sigma, \llbracket \mathbf{s}_0^{(0)} \otimes \mathbf{s}_1^{(0)} \rrbracket_\sigma$ with total size $2m^{(0)}b + (m^{(0)})^2b$ where $b := \lceil \log |\mathbb{F}| \rceil$, and several FSS keys. As in our PCF for OLE, we express each $\mathbf{e}_\tau^{(\ell)} \otimes \mathbf{s}_{1-\tau}^{(0)}$ as a concatenation of t point functions with range $\mathbb{F}^{m^{(0)}}$. This time, in order to hide the non-zero indices from both parties, we use standard (unknown-index) DPFs [BGI16]. The key size for t of them is

$$|k_\sigma^{\text{DMPF}(e_0^{(\ell)} \otimes \mathbf{s}_1^{(0)})}| = |k_\sigma^{\text{DMPF}(s_0^{(0)} \otimes e_1^{(\ell)})}| = t \cdot (\lceil \log(m^{(\ell)}/t) \rceil \cdot (\lambda + 2) + m^{(0)}b).$$

For terms of the form $\mathbf{e}_0^{(\ell)} \otimes \mathbf{e}_1^{(\ell')}$, where $\ell \geq \ell'$ and $(\ell, \ell') \neq (L, L)$, we use our tensor product FSS scheme from [Section 3.2](#) with key size

$$|k_\sigma^{\text{DMPF}(e_0^{(\ell)} \otimes e_1^{(\ell')})}| = \lambda + t \cdot \lceil \log(m^{(\ell)}/t) \rceil \cdot (\lambda + 2) + t^2 \cdot (\lceil \log(m^{(\ell')}/t) \rceil \cdot (\lambda + 2) + b).$$

For $\ell < \ell'$, we again reverse the order of the operands, to have the smaller vector on the right-hand side, as this yields smaller FSS keys. Finally, for $\mathbf{e}_0^{(L)} \odot \mathbf{e}_1^{(L)}$, we use a concatenation of standard (unknown-index) DPFs. The key size for t of them is

$$|k_\sigma^{\text{DMPF}(e_0^{(L)} \otimes e_1^{(L)})}| = t \cdot (\lceil \log(m^{(L)}/t) \rceil \cdot (\lambda + 2) + b).$$

Evaluation Cost. In the output $(a_\sigma, b_\sigma, c_\sigma)$ of $\text{PCF}^{\text{Beaver}}.\text{Eval}$, the part $c_\sigma = z_\sigma$ is computed like in PCF^{OLE} , i.e., using the same number of PRG evaluations. Computing a_σ and b_σ each takes additional $n_{\text{input}, L}^{\text{PRG}}$ PRG evaluations, where

$$n_{\text{input}, \ell}^{\text{PRG}} = \begin{cases} \lceil \log(m^{(\ell)}/t) \rceil + k^{(\ell)} \cdot n_{\text{input}, \ell-1}^{\text{PRG}} & \ell > 0 \\ 0 & \ell = 0 \end{cases}$$

is the number of PRG evaluations incurred by $\text{EvalReInput}(\sigma, k_\sigma^{\text{PCF}}, \tau, \ell, \hat{i})$.

4.3 Asymptotic Parameters

Based on our asymptotic sparse LPN instantiation from [Section 2.3](#), both of our PCFs (PCF^{OLE} from [Section 4.1](#) and $\text{PCF}^{\text{Beaver}}$ from [Section 4.2](#)) can be instantiated with a fixed noise weight $t \in \text{poly}(\lambda)$, fixed row sparsity $k := k^{(1)} = \dots = k^{(L)} \in \text{polylog}(\lambda)$, $m^{(0)} \in \text{poly}(\lambda)$, and, for $\ell \geq 1$, $m^{(\ell)} = m^{(\ell-1)}t/\lambda$. For these parameters, each PCF evaluation performs $\tilde{O}(k^{2L-1})$ PRG evaluations by [Lemma 4.1](#) below. Therefore, one can choose $L \in \Theta(\log \lambda / \log \log \lambda)$, in order to have $k^{2L-1} \in \text{poly}(\lambda)$, i.e., polynomial time complexity for

the PCF evaluation. The maximum number of queries then is $m^{(L)} = m^{(0)}(t/\lambda)^L \in \lambda^{\Theta(\log \lambda / \log \log \lambda)}$, which is superpolynomial.

For instance, one can set $L = \log n / (2 \log k) + 1/2$ to have $k^{2L-1} = n$ and superpolynomial expansion.

Lemma 4.1. *Let $k := k^{(1)} = \dots = k^{(L)} \in \text{polylog}(\lambda)$, $k \geq 2$, and suppose $\exists p \in \text{poly}(\lambda) : \forall \ell \in \{0, \dots, L\} : m^{(\ell)} \leq p^{\ell+1}$, then $\text{PCF}^{\text{OLE}}.\text{Eval}$ and $\text{PCF}^{\text{Beaver}}.\text{Eval}$ from [Figures 5 and 7](#) both perform $\tilde{O}(k^{2L-1})$ PRG evaluations.*

Proof. Note that, in $\text{PCF}^{\text{Beaver}}.\text{Eval}$, the most expensive part (in terms of the number of PRG evaluations) is the computation of z_σ , which is equally expensive as $\text{PCF}^{\text{OLE}}.\text{Eval}$. Therefore, it suffices to prove the statement for $\text{PCF}^{\text{OLE}}.\text{Eval}$. For $\ell' < \ell$, we have

$$n_{2,\ell,\ell'}^{\text{PRG}} \leq \sum_{i=0}^{\ell'} k^i \cdot \underbrace{\lceil \log(m^{(\ell)} m^{(i)} / t^2) \rceil}_{\leq \log(p^{2\ell+1}) \in \tilde{O}(\ell)} \in \tilde{O}(\ell) \cdot \underbrace{\sum_{i=0}^{\ell'} k^i}_{\in \Theta(k^{\ell'})} = \tilde{O}(\ell k^{\ell'})$$

and similarly

$$\begin{aligned} n_{1,\ell}^{\text{PRG}} &\leq \sum_{i=0}^{\ell-1} k^{2i} \cdot \left(\underbrace{\lceil \log((m^{(\ell-i)})^2 / t^2) \rceil}_{\in \tilde{O}(\ell-i)} + \underbrace{2k \cdot n_{2,\ell-i,\ell-i-1}^{\text{PRG}}}_{\in \tilde{O}((\ell-i)k^{\ell-i})} \right) \\ &\in \tilde{O}(k^\ell) \cdot \underbrace{\sum_{i=0}^{\ell-1} (\ell-i)k^i}_{\in \Theta(k^{\ell-1})} = \tilde{O}(k^{2\ell-1}). \end{aligned}$$

The statement follows since $n_{1,L}^{\text{PRG}}$ is an upper bound for the number of PRG evaluations during a call of $\text{PCF}^{\text{OLE}}.\text{Eval}$. \square

4.4 Concrete Parameters and Performance

For the concrete instantiation of our PCFs over $\mathbb{F} = \mathbb{F}_2$, we use regular noise of fixed weight $t = 1024$ and start with a secret of length $m^{(0)} = 1.5 \cdot 2^{15}$. We use LPN estimator [\[YYW+25\]](#) to obtain largest possible values $m^{(\ell)}$, such that $t \mid m^{(\ell)}$ and the corresponding LPN instance (i.e., $n = m^{(\ell-1)}$, $m = m^{(\ell)}$, and regular noise of weight t) provides $\lambda = 112$ bits of computational security against known attacks, including [\[BJMM12, EKM17, CDMT22, BØ23, LWYY24, ES24, WWY+25\]](#). The expansion factors $m^{(\ell)} / m^{(\ell-1)}$ obtained this way are $(m^{(\ell)} / m^{(\ell-1)})_{\ell=1,\dots,L} \approx (18, 24.9, 34.2, 36.3, 38.7)$, where we stopped at $L = 5$. At higher levels of the recursion, we can have a larger expansion factor for the same security target λ , thus obtaining a better trade-off between key size and expansion, compared to the approach from [\[BCM+25\]](#) of using the lowest-level expansion factor on all levels. In our upcoming performance evaluation, we consider different recursion depths $L \in \{3, 4, 5\}$, where PCF^{OLE} (resp. $\text{PCF}^{\text{Beaver}}$) can produce $m^{(3)} \approx 2^{29.5}$, $m^{(4)} \approx 2^{34.7}$, or $m^{(5)} \approx 2^{40}$ OLE correlations (resp. Beaver triples).

Above security estimation already accounts for the recent hybrid attack from [\[WWY+25\]](#). Note that this hybrid attack downgraded the security of the parameters used in [\[BCM+25\]](#) from originally targeted 128 bits to 107 bits, according to the current version of LPN estimator.

Security Against Linear Tests. Our instantiation additionally relies on the conjecture that sparse LPN is essentially as secure as LPN, as long as the matrix \mathbf{A} has a large-enough *dual distance* $\text{dd}(\mathbf{A})$, which is defined as the smallest weight of a non-zero vector \mathbf{v} such that $\mathbf{v}^T \mathbf{A} = 0$. We refer to [\[BCM+25\]](#) for a detailed discussion of this conjecture. The main

motivation is that most known attacks against LPN variants are instances of the *linear test framework* [BCG⁺20a] and a large-enough dual distance of \mathbf{A} is provably sufficient for an LPN variant to be secure against this class of attacks [CRR21]. Specifically, [CRR21] showed that any attack that fits in that framework requires in the order of $\exp(\Omega(\text{dd}(\mathbf{A}) \cdot t/m))$ iterations. Therefore, the quantity $\text{dd}(\mathbf{A}) \cdot t/m$ should be chosen similar to prior work, in order to achieve a comparable security level. We target $\text{dd}(\mathbf{A}) \cdot t/m \approx 2.5$ for our conservative parameter set, which is the same target as in [BCM⁺25], while [YWL⁺20, WYKW21, YSWW21] target $\text{dd}(\mathbf{A}) \cdot t/m \approx 3$. We also provide an optimistic parameter set with a smaller target, as explained below.

Bounding the Dual Distance. In our case where $\mathbf{A} \in \mathbb{F}_2^{m \times n}$ is a sparse matrix, the expected value of $\text{dd}(\mathbf{A})$ grows with the sparsity parameter k , i.e., the weight of the rows of \mathbf{A} . Braun et al. [BCM⁺25] presented a recursive algorithm to, given matrix dimensions (m, n) and sparsity k , compute the largest bound D , such that $\text{dd}(\mathbf{A}) \geq D$ except with probability $\leq 2^{-\rho}$ (over the random choice of \mathbf{A}), where ρ is a statistical security parameter. We reimplemented this algorithm as a Python script.

In Figure 8, we plot the ratio $\delta := D/n$ for our five different expansion factors m/n and $\rho \in \{40, 64\}$. When the ratio m/n and other parameters are fixed, but n is varied, there’s a minimum value of n , such that the algorithm outputs a non-trivial bound D (i.e., $D > 2$). When increasing n beyond this value, the ratio $\delta = D/n$ very quickly converges. All of our plots start at the value of n where we first obtain a non-trivial bound.

For our conservative parameter sets, we use $D = 2.5 \cdot m/t$, as this implies our target of $\text{dd}(\mathbf{A}) \cdot t/m \geq 2.5$ from above. For a sparsity k to be eligible for the conservative parameters, its plot in Figure 8 must pass through the vertical green line running at $n = m^{(\ell-1)}$ and $\delta \geq 2.5 \cdot m/(t \cdot n) = 2.5 \cdot m^{(\ell)}/(t \cdot m^{(\ell-1)})$.

Similar to [BCM⁺25], we also provide optimistic parameter sets, where we consider every value of k eligible as long as it yields a non-trivial bound D . That is, the plot of k must run through the (orange or green part of the) vertical line at $n = m^{(\ell-1)}$.

Precomputation. When evaluating our PCF, most of the DPF evaluations happen at the lowest level of the recursion tree. The most accessed DMPFs are those for $\mathbf{e}_0^{(1)} \otimes \mathbf{s}_1^{(0)}$ and $\mathbf{s}_0^{(0)} \otimes \mathbf{e}_1^{(1)}$. Similar to [BCM⁺25], we can precompute a full evaluation of these most-accessed DMPFs, in order to speed up subsequent PCF evaluations. For our parameters, the full precomputation of $\llbracket \mathbf{e}_0^{(1)} \otimes \mathbf{s}_1^{(0)} \rrbracket$ and $\llbracket \mathbf{s}_0^{(0)} \otimes \mathbf{e}_1^{(1)} \rrbracket$ together takes ≈ 4.3 seconds and consumes 10.12 GiB of RAM. Below, we therefore present the performance of our PCF without and with precomputation of these two tensor products. We don’t propose precomputation of any of the other tensor products (apart from $\llbracket \mathbf{s}_0^{(0)} \otimes \mathbf{s}_1^{(0)} \rrbracket$, which is part of the key), because the storage requirements would be too high.

Parameter Sets and Performance. In Table 2, we present our parameter sets together with the resulting performance evaluation of our PCF for OLE (cf. Section 4.1), without and with precomputation. We display the key sizes without and with using our tensor product FSS scheme from Section 3.2. We estimate the runtime using the same formula as [BCM⁺25], i.e., based on 20.8 CPU cycles per AES call and a clock frequency of 3.8 GHz. At $L = 3$, we can query up to 2 579 OLEs per second. Unfortunately, at this level, the keys are larger than the output size $m^{(3)}$, which is a common problem with PCFs for degree-two correlations, including the previously best PCF (see our comparison below). Even at $L = 4$, our output size is only marginally larger than the keys. At $L = 5$, we have an expansion of $m^{(5)}/|k_\sigma^{\text{PCF}}| = 22.5$, but at the cost of only being able to query 9.48 OLEs per second, using optimistic sparsity and $\rho = 40$.

Our PCF for two-party Beaver triples (cf. [Section 4.2](#)) can use the same parameters and has almost exactly the same key size and performance characteristics, so we don't provide a separate table. The key size is less than 4% larger and evaluation only up to 1% slower, compared to our PCF for OLE. The previously fastest PCF for Beaver triples, i.e., the PCF based on EALPN from [\[BCG⁺22\]](#), has keys of size ≈ 15 GiB and produces 9 triples per second, when targetting up to 2^{30} triples, which is the only output size for which the authors provided an efficiency analysis. For a similar output size, our PCF has $9\times$ smaller keys and is $3.2\text{--}28\times$ faster, i.e., 293 triples/s using conservative sparsity and $\rho = 64$, or 2551 triples/s using optimistic sparsity and $\rho = 40$.

Table 2: Parameters and performance of our programmable PCF for OLE over \mathbb{F}_2 with security $\lambda = 112$, regular noise weight $t = 1024$, and dimensions $m^{(0)} \approx 2^{15.6}$, $m^{(1)} \approx 2^{19.8}$, $m^{(2)} \approx 2^{24.4}$, $m^{(3)} \approx 2^{29.5}$, $m^{(4)} \approx 2^{34.7}$, and $m^{(5)} \approx 2^{40}$. For each ρ and L , there are two rows: The upper row uses conservative sparsity conforming to $\text{dd}(\mathbf{A}) \cdot t/m \geq 2.5$ and the lower row uses optimistic sparsity.

L	$(k^{(\ell)})_{\ell=1,\dots,L}$	Key Sizes (GiB)		Without Precomp.		With Precomp.	
		unopt. [†]	opt. [‡]	M PRG [§]	ms/OLE	M PRG [§]	ms/OLE
$\rho = 40$							
3	(7, 8, 12)	3.50	1.62	1.96	10.72	0.54	2.95
	(7, 6, 5)			0.21	1.15	0.07	0.39
4	(7, 8, 12, 14)	7.40	3.16	384.45	2 104.35	106.27	581.70
	(7, 6, 5, 4)			3.41	18.64	1.19	6.50
5	(7, 8, 12, 14, 18)	13.39	5.47	124 573.75	681 877.37	34 444.81	188 540.01
	(7, 6, 5, 4, 4)			54.74	299.65	19.26	105.43
$\rho = 64$							
3	(9, 8, 12)	3.50	1.62	2.45	13.38	0.62	3.40
	(9, 7, 6)			0.49	2.68	0.14	0.77
4	(9, 8, 12, 14)	7.40	3.16	480.10	2 627.90	122.44	670.21
	(9, 7, 6, 5)			12.36	67.64	3.63	19.85
5	(9, 8, 12, 14, 18)	13.39	5.47	155 567.28	851 526.18	39 687.22	217 235.29
	(9, 7, 6, 5, 5)			309.62	1 694.78	91.33	499.90

[†] The key size per party when using a naive sum for the tensor product.

[‡] The key size per party when using our tensor product FSS scheme from [Section 3.2](#) instead.

[§] The number of PRG evaluations, in millions, to query one OLE correlation.

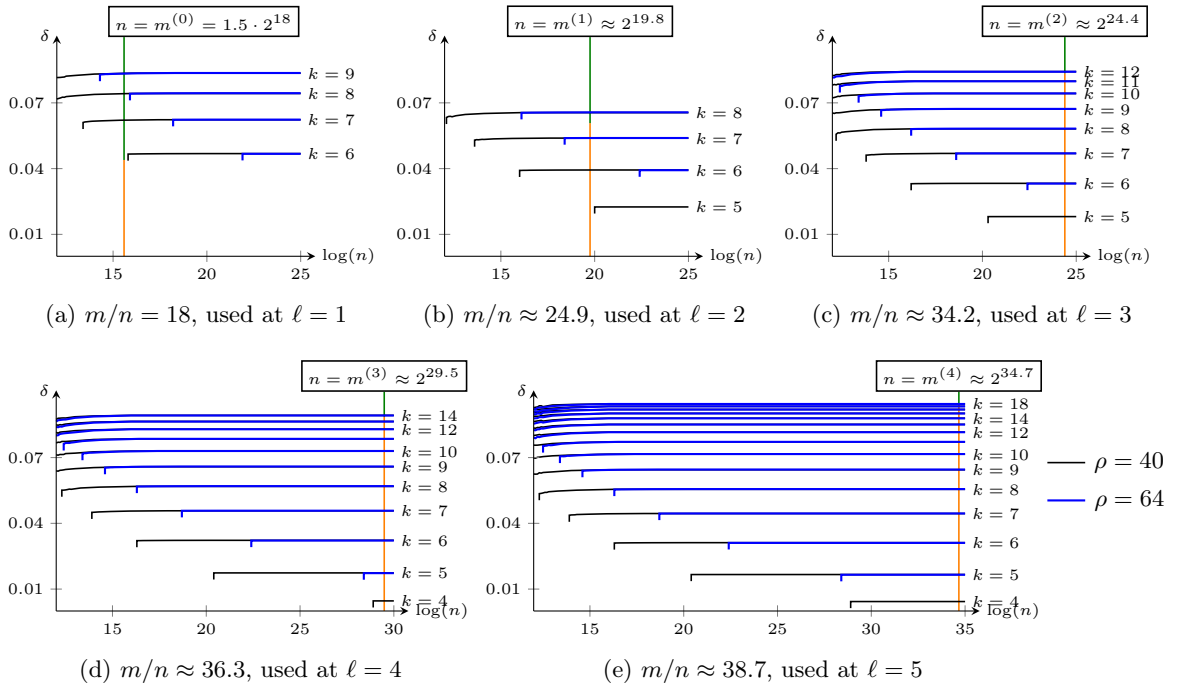


Fig. 8: Values $\delta := D/n$, such that $\Pr[\text{dd}(\mathbf{A}) < D] \leq 2^{-\rho}$ over $\mathbf{A} \leftarrow \text{RegularCodeGen}(k, n, m, \mathbb{F}_2)$, for different expansion factors m/n , different row sparsities k , and statistical security parameter $\rho \in \{40, 64\}$. The expansion factors m/n considered here correspond to $m^{(\ell)}/m^{(\ell-1)}$ in the five levels of our concrete PCF parameters. Values for k whose plot passes through the green (resp. yellow) area can be used in conservative (resp. optimistic) parameters.

Acknowledgments. This research was supported by Advantest as part of the Graduate School “Intelligent Methods for Test and Reliability” (GS-IMTR) at the University of Stuttgart. Additionally, this research was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grants 411720488 and 548713845.

References

- BCCD23. Maxime Bombar, Geoffroy Couteau, Alain Couvreur, and Clément Ducros. Correlated pseudorandomness from the hardness of quasi-abelian decoding. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023, Part IV*, volume 14084 of *Lecture Notes in Computer Science*, pages 567–601, Santa Barbara, CA, USA, August 20–24, 2023. Springer, Cham, Switzerland.
- BCG⁺19a. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 291–308, London, UK, November 11–15, 2019. ACM Press.
- BCG⁺19b. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part III*, volume 11694 of *Lecture Notes in Computer Science*, pages 489–518, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Cham, Switzerland.
- BCG⁺20a. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Correlated pseudorandom functions from variable-density LPN. In *61st Annual Symposium on Foundations of Computer Science*, pages 1069–1080, Durham, NC, USA, November 16–19, 2020. IEEE Computer Society Press.
- BCG⁺20b. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from ring-LPN. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 387–416, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Cham, Switzerland.
- BCG⁺21. Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. Function secret sharing for mixed-mode and fixed-point secure computation. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021, Part II*, volume 12697 of *Lecture Notes in Computer Science*, pages 871–900, Zagreb, Croatia, October 17–21, 2021. Springer, Cham, Switzerland.
- BCG⁺22. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Nicolas Resch, and Peter Scholl. Correlated pseudorandomness from expand-accumulate codes. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022, Part II*, volume 13508 of *Lecture Notes in Computer Science*, pages 603–633, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Cham, Switzerland.
- BCG⁺23. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Nicolas Resch, and Peter Scholl. Oblivious transfer with constant computational overhead. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023, Part I*, volume 14004 of *Lecture Notes in Computer Science*, pages 271–302, Lyon, France, April 23–27, 2023. Springer, Cham, Switzerland.
- BCGI18. Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 896–912, Toronto, ON, Canada, October 15–19, 2018. ACM Press.
- BCM⁺24. Dung Bui, Geoffroy Couteau, Pierre Meyer, Alain Passelègue, and Mahshid Riahinia. Fast public-key silent OT and more from constrained Naor-Reingold. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024, Part VI*, volume 14656 of *Lecture Notes in Computer Science*, pages 88–118, Zurich, Switzerland, May 26–30, 2024. Springer, Cham, Switzerland.
- BCM⁺25. Lennart Braun, Geoffroy Couteau, Kelsey Melissaris, Mahshid Riahinia, and Elahe Sadeghi. Fast pseudorandom correlation functions from sparse LPN. In *Advances in Cryptology – ASIACRYPT 2025*, *Lecture Notes in Computer Science*, Melbourne, Australia, December 2025. Springer.
- Bea92. Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology – CRYPTO’91*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432, Santa Barbara, CA, USA, August 11–15, 1992. Springer Berlin Heidelberg, Germany.

- BGI15. Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 337–367, Sofia, Bulgaria, April 26–30, 2015. Springer Berlin Heidelberg, Germany.
- BGI16. Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 1292–1303, Vienna, Austria, October 24–28, 2016. ACM Press.
- BGIK22. Elette Boyle, Niv Gilboa, Yuval Ishai, and Victor I. Kolobov. Programmable distributed point functions. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022, Part IV*, volume 13510 of *Lecture Notes in Computer Science*, pages 121–151, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Cham, Switzerland.
- BJMM12. Anja Becker, Antoine Joux, Alexander May, and Alexander Meurer. Decoding random binary linear codes in $2^{n/20}$: How $1 + 1 = 0$ improves information set decoding. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 520–536, Cambridge, UK, April 15–19, 2012. Springer Berlin Heidelberg, Germany.
- BØ23. Pierre Briaud and Morten Øygarden. A new algebraic approach to the regular syndrome decoding problem and implications for PCG constructions. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023, Part V*, volume 14008 of *Lecture Notes in Computer Science*, pages 391–422, Lyon, France, April 23–27, 2023. Springer, Cham, Switzerland.
- CDD⁺24. Geoffroy Couteau, Lalita Devadas, Srinivas Devadas, Alexander Koch, and Sacha Servan-Schreiber. QuietOT: Lightweight oblivious transfer with a public-key setup. In Kai-Min Chung and Yu Sasaki, editors, *Advances in Cryptology – ASIACRYPT 2024, Part II*, volume 15485 of *Lecture Notes in Computer Science*, pages 197–231, Kolkata, India, December 9–13, 2024. Springer, Singapore, Singapore.
- CDMT22. Kevin Carrier, Thomas Debris-Alazard, Charles Meyer-Hilfiger, and Jean-Pierre Tillich. Statistical decoding 2.0: Reducing decoding to LPN. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022, Part IV*, volume 13794 of *Lecture Notes in Computer Science*, pages 477–507, Taipei, Taiwan, December 5–9, 2022. Springer, Cham, Switzerland.
- CRR21. Geoffroy Couteau, Peter Rindal, and Srinivasan Raghuraman. Silver: Silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021, Part III*, volume 12827 of *Lecture Notes in Computer Science*, pages 502–534, Virtual Event, August 16–20, 2021. Springer, Cham, Switzerland.
- DHRW16. Yevgeniy Dodis, Shai Halevi, Ron D. Rothblum, and Daniel Wichs. Spooky encryption and its applications. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016, Part III*, volume 9816 of *Lecture Notes in Computer Science*, pages 93–122, Santa Barbara, CA, USA, August 14–18, 2016. Springer Berlin Heidelberg, Germany.
- DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662, Santa Barbara, CA, USA, August 19–23, 2012. Springer Berlin Heidelberg, Germany.
- EKM17. Andre Esser, Robert Kübler, and Alexander May. LPN decoded. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part II*, volume 10402 of *Lecture Notes in Computer Science*, pages 486–514, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Cham, Switzerland.
- ES24. Andre Esser and Paolo Santini. Not just regular decoding: Asymptotics and improvements of regular syndrome decoding attacks. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology – CRYPTO 2024, Part VI*, volume 14925 of *Lecture Notes in Computer Science*, pages 183–217, Santa Barbara, CA, USA, August 18–22, 2024. Springer, Cham, Switzerland.
- HRK25. Sebastian Hasler, Pascal Reisert, and Ralf Küsters. Pseudorandom correlation functions from ring-LWR. In *Advances in Cryptology – ASIACRYPT 2025*, Lecture Notes in Computer Science, Melbourne, Australia, December 2025. Springer.
- KOS16. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 830–842, Vienna, Austria, October 24–28, 2016. ACM Press.
- KPR18. Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*,

- Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 158–189, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Cham, Switzerland.
- LWYY24. Hanlin Liu, Xiao Wang, Kang Yang, and Yu Yu. The hardness of LPN over any integer ring and field for PCG applications. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024, Part VI*, volume 14656 of *Lecture Notes in Computer Science*, pages 149–179, Zurich, Switzerland, May 26–30, 2024. Springer, Cham, Switzerland.
- LXYY25. Zhe Li, Chaoping Xing, Yizhou Yao, and Chen Yuan. Efficient pseudorandom correlation generators for any finite field. In Serge Fehr and Pierre-Alain Fouque, editors, *Advances in Cryptology – EUROCRYPT 2025, Part V*, volume 15605 of *Lecture Notes in Computer Science*, pages 145–175, Madrid, Spain, May 4–8, 2025. Springer, Cham, Switzerland.
- MMST25. Peihan Miao, Alice Murphy, Akshayaram Srinivasan, and Max Tromanhauser. Pseudorandom correlation generators for multiparty beaver triples over \mathbb{F}_2 . In *Advances in Cryptology – ASIACRYPT 2025*, Lecture Notes in Computer Science, Melbourne, Australia, December 2025. Springer.
- OSY21. Claudio Orlandi, Peter Scholl, and Sophia Yakubov. The rise of paillier: Homomorphic secret sharing and public-key silent OT. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 678–708, Zagreb, Croatia, October 17–21, 2021. Springer, Cham, Switzerland.
- RRT23. Srinivasan Raghuraman, Peter Rindal, and Titouan Tanguy. Expand-convolute codes for pseudorandom correlation generators from LPN. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023, Part IV*, volume 14084 of *Lecture Notes in Computer Science*, pages 602–632, Santa Barbara, CA, USA, August 20–24, 2023. Springer, Cham, Switzerland.
- SGRR19. Philipp Schoppmann, Adrià Gascón, Leonie Reichert, and Mariana Raykova. Distributed vector-OLE: Improved constructions and implementation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 1055–1072, London, UK, November 11–15, 2019. ACM Press.
- WWY⁺25. Tianrui Wang, Anyu Wang, Kang Yang, Hanlin Liu, Yu Yu, Jun Zhang, and Xiaoyun Wang. A hybrid algorithm for the regular syndrome decoding problem, December 2025.
- WYKW21. Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *2021 IEEE Symposium on Security and Privacy*, pages 1074–1091, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press.
- YSWW21. Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021: 28th Conference on Computer and Communications Security*, pages 2986–3001, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.
- YWL⁺20. Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 1607–1626, Virtual Event, USA, November 9–13, 2020. ACM Press.
- YYW⁺25. Yu Yu, Kang Yang, Xiao Wang, Anyu Wang, Tianrui Wang, Hanlin Liu, Xinpeng Hao, and Juanru Li. Estimator of LPN problems over any finite fields and power-of-two rings for PCG and MPC applications. <https://lpnestimator.com>, 2025.