Low-Communication Multiparty Triple Generation for SPDZ from Ring-LPN[⋆]

Damiano Abram and Peter Scholl

Aarhus University, Aarhus, Denmark
{damiano.abram, peter.scholl}@cs.au.dk

Abstract. The SPDZ protocol for multi-party computation relies on a correlated randomness setup consisting of authenticated, multiplication triples. A recent line of work by Boyle et al. (Crypto 2019, Crypto 2020) has investigated the possibility of producing this correlated randomness in a *silent preprocessing* phase, which involves a "small" setup protocol with less communication than the total size of the triples being produced. These works do this using a tool called a *pseudorandom correlation generator* (PCG), which allows a large batch of correlated randomness to be compressed into a set of smaller, correlated seeds. However, existing methods for compressing SPDZ triples only apply to the 2-party setting. In this work, we construct a PCG for producing SPDZ triples over large prime fields in the multi-party setting. The security of our PCG is based on the ring-LPN assumption over fields, similar to the work of Boyle et al. (Crypto 2020) in the 2-party setting. We also present a corresponding, actively secure setup protocol, which can be used to generate the PCG seeds and instantiate SPDZ with a silent preprocessing phase. As a building block, which may be of independent interest, we construct a new type of 3-party distributed point function supporting outputs over arbitrary groups (including large prime order), as well as an efficient protocol for setting up our DPF keys with active security.

1 Introduction

Multi-party computation (MPC) allows a set of parties to securely compute on private inputs, while learning nothing but the desired result of the computation. Modern MPC protocols often use a source of secret, *correlated randomness*, which can be distributed to the parties ahead of time, and used to help improve efficiency of the protocol. This is especially important in the dishonest majority setting, where up to $n-1$ out of n parties may be corrupted, since these types of protocols rely on expensive, 'public key'-type cryptographic primitives.

For instance, the SPDZ family of protocols [DPSZ12,DKL+13], which achieves active security with a dishonest majority, uses preprocessed, authenticated multiplication triples, to achieve a very fast online phase where the computation takes place. Multiplication triples, coming from the work of Beaver [Bea92], are

[⋆] Supported by the Aarhus University Research Foundation and Independent Research Fund Denmark (DFF) under project number 0165-00107B (C3PO).

triples of random, secret-sharings of values a, b, c over some ring, where $c = a \cdot b$, and allow protocols to offload the heavy work of MPC multiplication to the preprocessing phase. Unfortunately, producing these triples, although it can be done ahead of time, is still an expensive process in terms of computation, communication and storage costs, since typically a very large number of triples is required, for any reasonably complex computation.

Most current techniques for triple generation are either based on homomorphic encryption [DPSZ12, DKL$^+$13, KPR18] or oblivious transfer [KOS16]. Homomorphic encryption is computationally expensive and also incurs moderately high communication costs (especially due to the use of zero-knowledge proofs for active security), while oblivious transfer is much cheaper computationally, but requires a large amount of bandwidth.

More recently, Boyle et al. [BCG$^+$19b] proposed using *pseudorandom correlation generators* to produce a large amount of correlated randomness without interaction, starting from only a short set of correlated seeds. More concretely, a PCG consists of a seed-generation algorithm, Gen, which outputs a set of correlated seeds $\kappa_0, \ldots, \kappa_{n-1}$, one given to each party. There is then an Expand algorithm, which deterministically expands κ_i into a large amount of correlated randomness R_i. The security requirements are that the expanded outputs $(R_i)_i$ should be indistinguishable from a sample from the target correlation, and furthermore, knowing a subset of the keys should not reveal any information about the missing outputs (beyond what can be deduced from their evaluation). This paradigm offers the potential to greatly reduce communication in the preprocessing of MPC protocols, while also reducing storage costs for the necessary correlated randomness, since the PCG seeds need only be expanded "on-demand".

The first construction of a PCG for authenticated triples [BCG$^+$19b] was based on homomorphic encryption, and not so efficient in practice. However, more recently, the authors proposed another construction [BCG$^+$20] based on a variant of the *ring learning parity with noise* (ring-LPN) assumption. By using distributed point functions [GI14, BGI15] to compress secret-shared, sparse vectors, this construction achieves much better concrete efficiency, as well as a good compression rate.

Unfortunately, both of these PCGs for authenticated, SPDZ-style triples are restricted to the 2-party setting. Note that *unauthenticated* triples, as used in passively secure protocols, can be generated with a PCG in the multi-party setting, with a transformation from [BCG$^+$19b], however, this does not apply to the more complex task of authenticated sharings.

1.1 Our Contributions

In this work, we investigate the possibility of constructing PCGs for SPDZ-style, authenticated triples in the multi-party setting. As our main contributions, we construct such a PCG based on the ring-LPN assumption over large prime fields, and design an actively secure protocol for distributing the PCG seeds among the parties. Our PCG allows expanding short, correlated seeds of size $O(n^3\sqrt{N})$ into

N SPDZ triples for n parties. Meanwhile, our actively secure setup protocol produces N SPDZ triples for n parties with $O(n^4\sqrt{N})$ communication. Compared with previous protocols for SPDZ [KPR18, KOS16], which use $O(n^2N)$ communication, our protocol scales sublinearly in the number of triples, but is less suitable for a large number of parties. (In the above, we ignore asymptotic factors that only depend on the security parameter.)

Below, we expand on our results in a little more detail.

Background: Construction of [BCG+20]. We first briefly recall the 2-party PCG for authenticated triples from [BCG+20]. Their construction relies on a variant of the ring-LPN (or module-LPN) assumption, which works over the polynomial ring $R = \mathbb{F}[X]/(F(X))$, for some finite field $\mathbb{F}$ and fixed polynomial $F(X)$. The assumption, for noise weight t and dimension c, states that the distribution

$$\{\boldsymbol{a}, \langle \boldsymbol{a}, \boldsymbol{e}\rangle \,|\boldsymbol{a} \leftarrow R^c, \boldsymbol{e} \leftarrow R^c \text{ s.t. } \mathrm{wt}(e_i) = t\}$$

is indistinguishable from random, when each $e_i \in R$ is a sparse polynomial of degree $< N$, with up to t non-zero coefficients. In typical parameters, N will be very large, while c is a small constant, and t the order of the security parameter.

The goal will be to produce a PCG that outputs 2-party, additive shares of a random tuple $(x, y, z, \alpha x, \alpha y, \alpha z)$, where $\alpha \in \mathbb{F}$, $x, y \leftarrow R$ and $z = x \cdot y \in R$. When R is chosen appropriately, such an authenticated triple over R can be locally converted into a large batch of N triples over $\mathbb{F}$.

To obtain a PCG, the construction picks vectors of sparse polynomials $\boldsymbol{u}, \boldsymbol{v} \in R^c$, and computes the tensor product $\boldsymbol{u} \otimes \boldsymbol{v} \in R^{c^2}$. Each of these polynomial products is still somewhat sparse, having at most t^2 non-zero coordinates. The idea is that the sparse $\boldsymbol{u}, \boldsymbol{v}$, as well as their products, can be secret-shared using distributed point functions (DPFs) [GI14, BGI15, BGI16], which provide a way to share sparse vectors in a succinct manner.

Given shares of these values, the parties can locally compute inner products with the public vector $\boldsymbol{a}$, to transform the sparse vectors $\boldsymbol{u}, \boldsymbol{v}$ into pseudorandom polynomials $x = \langle \boldsymbol{a}, \boldsymbol{u}\rangle$ and $y = \langle \boldsymbol{a}, \boldsymbol{v}\rangle$. Similarly, the shares of $\boldsymbol{u} \otimes \boldsymbol{v}$ can be locally transformed into shares of xy, due to bilinearity of the tensor product.

The above blueprint gives additive shares of the (x, y, z) components of the triple. This easily extends to obtain shares of $(\alpha x, \alpha y, \alpha z)$, since multiplying each sparse vector by $\alpha \in \mathbb{F}$ preserves its sparsity, so these can be distributed in the same way.

Using 3-Party Distributed Point Functions. A natural approach to extend the above to more than two parties, is to simply use multi-party DPFs. Unfortunately, existing n-party DPFs [BGI15] scale badly, with a key size growing exponentially in the number of parties. Instead, in the full version of [BCG+20], Boyle et al. sketched an approach using *3-party DPFs*, based on the observation that the product αxy can be broken down into a sum of $\alpha_i x_j y_k$, over parties $i, j, k \in [n]$. This means that each of these terms only needs to be shared between 3 parties, so 3-party DPFs suffice.

However, it turns out this approach is not so straightforward. An immediate challenge is that existing 3-party DPFs only output shares that are XOR-sharings, or shares over $\mathbb{Z}_p$ for small primes p; this excludes the important case of $\mathbb{F}_p$ where p is a large prime, as often used in protocols like SPDZ. Therefore, our first contribution is to construct a 3-party DPF suitable for this setting, by modifying the DPF of Bunn et al. [BKKO20] to work with outputs over any abelian group. Our modification introduces some leakage into the construction: when two specific parties are corrupted, they now learn some information about the secret index of the point function that is being hidden. Fortunately, it turns out that for our application to SPDZ, this leakage is harmless, since it translates to corrupt parties $\{P_j, P_k\}$ learning information on the product $x_j y_k$, which P_j and P_k already know if they collude.

An additional benefit of our DPF, beyond supporting more general outputs, is that our key sizes are smaller than the 3-party DPF of [BKKO20] by around a factor of 3.

PCG for Authenticated Triples. Given our 3-party DPF, we give the full construction of a multi-party PCG for authenticated triples over a large field $\mathbb{F}$. The basic construction for producing N triples with n parties has seeds of size $O(n^3 t^2 \sqrt{N} \lambda)$ bits, where λ is the security parameter and t is roughly λ, although this can be optimized slightly with a more aggressive assumption. Compared with the 2-party PCG of [BCG$^+$20], we incur some extra costs moving to the multi-party setting, since theirs scales with $O(\log N)$ and not $O(\sqrt{N})$. This is due to the $O(\sqrt{N})$ seed size in our DPF, which is also inherited from previous 3-party DPFs [BGI15, BKKO20][1].

Efficient, Actively Secure Distributed Setup for 3-Party DPF. To obtain our triple generation protocol, we need a way of securely setting up the PCG seeds among the parties. The main necessary ingredient is a protocol for distributing the keys in our 3-party DPF. Previously, Bunn et al. [BKKO20] gave a secure protocol for setting up their 3-party DPF keys; however, as well as being very complex, their protocol only has passive security and tolerates 1 out of 3 corruptions. We therefore set out to design an *actively secure* protocol for our 3-party DPF, tolerating any number of corruptions, while only introducing a minimal communication overhead relative to the size of the underlying DPF keys. Our starting point is a lightweight, passively secure setup protocol based on OT and 2-party DPFs, which we combine with a recursive step to generate the necessary "correction word" in the DPF keys. Using recursion here helps to keep the communication overhead down in our protocol. We add active security, by first replacing OT with *authenticated OT*, whereby the receiver's choice bits

[1] In the 2-party setting, there are efficient DPFs with logarithmic key size [BGI15, BGI16].

are authenticated using MACs. We then apply several consistency checks on the DPF keys, including one inspired by a recent OT extension protocol [YWL$^+$20], to prove that the parties behaved honestly. Here, we exploit the fact that the OT choice bits were authenticated, which allows us to reliably perform linear tests on these bits as part of our checks.

Our final setup protocol is very lightweight, and only communicates a small constant factor (2–3x) more information than the size of the DPF keys. On top of the inherent leakage in our DPF, the protocol introduces a small amount of leakage, in the form of allowing the adversary to try and guess some information about the secret point function. This is similar to leakage from other PCG setup protocols based on LPN [BCG$^+$19a, YWL$^+$20], and essentially only translates into an average of one bit of leakage on the (ring)-LPN secret.

Concrete Efficiency. We analyse the concrete efficiency of our actively secure protocol for setting up the PCG seeds, and producing authenticated triples. The main bottleneck is the distributed execution of the 3-party DPF, the only part of the protocol with $\Omega(\sqrt{N})$ complexity. We measure the efficiency of the construction by considering its "stretch", the ratio between the size of the produced triples and the total communication. We observed that the stretch becomes greater than 1 when N is above 2^{24}, meaning producing more than 16 million multiplication triples. When N increases, the stretch improves, reaching values close to 8 for $N = 2^{28}$. This comes, however, at a greater computational cost as the latter scales as $O(N \log(N))$. On the other hand, even for $N = 2^{20}$, our construction performs significantly better than alternative approaches such as Overdrive [KPR18], improving the communication complexity by at least a factor of 10. In this parameter regime, the 2-party PCG of [BCG$^+$20] has practical computational cost, and although we have not implemented our construction, we believe the same will hold since it uses similar building blocks.

2 Notation and Preliminaries

We denote the multiplicative group of a finite field by $\mathbb{F}^\times$. The ideal generated by a polynomial $F(X) \in \mathbb{F}[X]$ is $\big(F(X)\big)$.

When dealing with bit sequences, with an abuse of notation, we identify the sets $\{0,1\}^k$, $\mathbb{F}_2^k$ and $\mathbb{F}_{2^k}$ as different representations of the finite field with 2^k elements. For this reason, when multiplying two elements $a, b \in \{0,1\}^k$, we mean multiplication in $\mathbb{F}_{2^k}$.

Throughout the paper, we will deal with protocols between an ordered set of n parties, $P_1, \ldots, P_n$. We let $\mathcal{H}$ be the set of indices of honest parties, and $\mathcal{C}$ the set of indices of corrupt ones.

The symbol $[m]$ indicates the set $\{0, 1, 2, \ldots, m-1\}$ and $\lfloor\cdot\rfloor$ denotes the integral part of a real number. We represent vectors using bold font, the j-th entry of a vector $\boldsymbol{v}$ is denoted by v_j or $v[j]$. We indicate the scalar product by

$\langle \cdot, \cdot \rangle$ The function $\delta_y(\cdot)$ denotes the Kronecker delta function, that is,

$$\delta_y(x) := \begin{cases} 1 & \text{if } x = y, \\ 0 & \text{otherwise.} \end{cases}$$

Given two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ of dimensions l and m respectively, we denote their outer product by $\boldsymbol{u} \otimes \boldsymbol{v}$. Observe that this is an ml-dimensional vector whose $(im + j)$-th entry is $u_i \cdot v_j$. In a similar way, we define their outer sum $\boldsymbol{u} \boxplus \boldsymbol{v}$ as the ml-dimensional vector whose $(im + j)$-th entry is $u_i + v_j$.

We write $a \xleftarrow{\$} S$, where S is a set, to mean that a is randomly sampled from S. Finally, λ denotes the security parameter and $\mathbb{P}$ represents a probability measure.

Polynomial Rings. Let p be prime and N a positive integer. We will work with the ring $R := \mathbb{F}_p[X]/\big(F(X)\big)$, where $F(X)$ is an irreducible, degree-N polynomial in $\mathbb{Z}[X]$. Similarly to the case of homomorphic encryption [SV14], we will be interested in the case where $F(X)$ factors completely modulo p into a product of distinct, linear terms. In this case, we say that R is fully splittable, and have the isomorphism $R \cong \mathbb{F}_p^N$. This can be ensured, for instance, by choosing N to be a power of 2 and the cyclotomic polynomial $F(X) = X^N + 1$, with $p = 1 \bmod (2N)$.

2.1 Module-LPN

The security of our triple generation protocol relies on the Module-LPN assumption with static leakage, a generalisation of Ring-LPN that was recently studied by Boyle et al. [BCG+20]. We recap here its definition.

Definition 1 (Module-LPN with static leakage). *Let $R := \mathbb{F}_p[X]/\big(F(X)\big)$, for a prime p and $F(X)$ of degree N. Let t and c be two positive integers with $c \geq 2$. Let $\mathcal{HW}_t$ be the distribution that samples t noise positions $\omega[i] \xleftarrow{\$} [N]$ and t payloads $\beta[i] \xleftarrow{\$} \mathbb{F}_p$, outputting the polynomial*

$$e(X) := \sum_{i \in [t]} \beta[i] \cdot X^{\omega[i]}$$

embedded in the ring R. Let $\mathcal{A}$ be a PPT adversary and consider the game $\mathcal{G}^{\text{Module-LPN}}_{R,t,c,\mathcal{A}}(\lambda)$ described in Figure 1. We say that the R^c-LPN$_t$ problem with static leakage is hard if, for PPT adversary $\mathcal{A}$, the advantage

$$Adv^{\text{Module-LPN}}_{R,t,c,\mathcal{A}}(\lambda) := \left| \mathbb{P}\Big(\mathcal{G}^{\text{Module-LPN}}_{R,t,c,\mathcal{A}}(\lambda) = 1\Big) - \frac{1}{2} \right|$$

is negligible in the security parameter λ.

Clearly, in the definition, we assume that the ring R and the values c and t depends on the security parameter λ. Observe that the greater c and t are, the harder the distinguishability becomes. A thorough analysis of the assumption

$\mathcal{G}^{\text{Module-LPN}}_{R,t,c,\mathcal{A}}(\lambda)$

Initialisation. The challenger activates $\mathcal{A}$ with $\mathbb{1}^\lambda$ and samples a random bit $b \xleftarrow{\$} \{0,1\}$. Then, it samples c elements of the ring $e_0, e_1, \ldots, e_{c-1} \leftarrow \mathcal{HW}_t$. Let the j-th noise positions of e_i be $\omega_i[j]$.

Query. The adversary is allowed to adaptively issue a polynomial number of queries of the form (i,j,I) where $i \in [c]$, $j \in [t]$ and $I \subseteq [N]$. If $\omega_i[j] \in I$, the challenger answers with SUCCESS, otherwise, it sends ABORT and halts.

Challenge. After the Query phase, for every $i \in [c-1]$, the challenger samples $a_i \xleftarrow{\$} R$ and sets

$$u_1 \leftarrow \sum_{i=0}^{c-2} a_i \cdot e_i + e_{c-1}.$$

Moreover, it samples $u_0 \xleftarrow{\$} R$. Finally, it gives $(a_0, a_1, \ldots, a_{c-2}, u_b)$ to the $\mathcal{A}$. The adversary replies with a bit b'. The final output of the game is 1 if and only if $b = b'$.

Fig. 1. The Module-LPN game.

can be found in [BCG$^+$20], including for the case when the polynomial $F(X)$ splits completely into linear factors over $\mathbb{F}$, i.e. when $R \cong \mathbb{F}^N$.

Regarding the leakage, note that in Fig. 1, the adversary's guesses are restricted to *before* it learns the ring-LPN challenge; thus, even though there may be many queries, the resulting leakage is very small: just 1 bit of information on the secret (that is, the fact that all guesses were correct).

Choice of Error Distribution. The basic module-LPN definition assumes each error polynomial is chosen uniformly, subject to having t non-zero coefficients. We can also improve efficiency with more structured errors, such as *regular errors*, where the non-zero coordinates are more evenly spaced out, so that each is guaranteed to lie in a unique interval of size N/t. We use this variant in our efficiency estimates to improve parameters. Note that it has also been used previously [BCG$^+$19b, BCG$^+$20, YWL$^+$20], and is conjectured to have essentially the same security as the standard assumption.

2.2 Pseudorandom Correlation Generators

To obtain a low communication complexity, our protocol uses pseudorandom correlation generators (PCGs) [BCG$^+$19a, BCG$^+$19b, BCG$^+$20]. An n-party PCG is a pair of algorithms, the first of which outputs n correlated seeds of relatively small size. These can be, later on, locally expanded by the parties to obtain a large amount of desired correlated randomness. Since the expansion phase does not require any communication between the parties and the seed size is small compared to the output, the hope is to design low-communication protocols that securely generate and distribute the seeds to the parties. This allows the secure generation of large amounts of correlated randomness with low communication complexity.

The syntax of a PCG is given firstly by the algorithm Gen, which on input the security parameter, outputs n correlated keys κ_i, for $i \in [n]$. Secondly, the Expand algorithm takes as input (i, κ_i), and produces an expanded output R_i. The formal definition of a PCG, shown in the full-version of this work [AS21], requires both a correctness property and a security property.

Essentially, correctness requires that the joint distribution of the parties' outputs $(R_1, \dots, R_n)$ is indistinguishable from the target correlation $\mathcal{C}_{\text{corr}}$. The security property states that the knowledge of a subset of the seeds leaks no information about the other outputs, that could not already be inferred from the knowledge of the expansion of the given seeds.

2.3 Distributed Point Functions

In [GI14], Gilboa and Ishai introduced distributed point functions (DPFs). A point function is a function f whose support (i.e. the elements which have non-zero image) contains at most one element. Therefore, if the domain has size N, we can regard f as an N-dimensional vector with at most one non-zero entry, whose i-th entry, for $i \in [N]$, corresponds to the evaluation $f(i)$. We call such vector a unit vector, and often refer to the index of the non-zero entry as the *special position* and its value as the *non-zero element*.

An n-party DPF consists of a pair of algorithms, the first of which takes as input the description of a point function f and outputs n succinct keys. These can be, later on, locally evaluated by the parties on input x to obtain a secret-sharing of $f(x)$. DPFs and PCGs have some similarity, in that in both cases, we have an initial phase in which correlated, succinct keys are generated, followed by an evaluation phase that locally produces the desired output. The analogy between the two notions is the reason why DPFs are often a key building block of PCGs. Our protocol is no exception.

Definition 2 (DPF with leakage). *Let $(\mathbb{G}, +)$ be an abelian group and let N be a positive integer. An n-party distributed point function (DPF) for $(N, \mathbb{G})$ with leakage function* Leak *is a pair of PPT algorithms* $(\mathsf{DPF}^n_N.\mathsf{Gen}, \mathsf{DPF}^n_N.\mathsf{Eval})$ *with the following syntax:*

- *On input $\mathbb{1}^\lambda$, $\omega \in [N]$ and $\beta \in \mathbb{G}$,* $\mathsf{DPF}^n_N.\mathsf{Gen}$ *outputs n keys $\kappa_0, \kappa_1, \dots, \kappa_{n-1}$.*
- *On input (i, κ_i, x) for $i \in [n]$ and $x \in [N]$,* $\mathsf{DPF}^n_N.\mathsf{Eval}$ *outputs a value $v_i \in \mathbb{G}$.*

Moreover, the following properties are satisfied

- ***(Correctness)**. For every $x, \omega \in [N]$ and $\beta \in \mathbb{G}$,*

$$\mathbb{P}\left(\sum_{i=0}^{n-1} v_i = \beta \cdot \delta_\omega(x) \;\middle|\; \begin{array}{l} (\kappa_0, \kappa_1, \dots, \kappa_{n-1}) \leftarrow \mathsf{DPF}^n_N.\mathsf{Gen}(\mathbb{1}^\lambda, \omega, \beta) \\ v_i \leftarrow \mathsf{DPF}^n_N.\mathsf{Eval}(i, \kappa_i, x) \quad \forall i \in [n] \end{array}\right) = 1.$$

– ***(Security)**. There exists a PPT simulator* Sim *such that for every $T \subsetneq [n]$, $\omega \in [N]$ and $\beta \in \mathbb{G}$, the following distributions are computationally indistinguishable*

$$\Big\{(\kappa_i)_{i \in T} | (\kappa_0, \kappa_1, \ldots, \kappa_{n-1}) \leftarrow \mathsf{DPF}^n_N.\mathsf{Gen}(\mathbb{1}^\lambda, \omega, \beta)\Big\} \equiv_C$$
$$\Big\{(\kappa_i)_{i \in T} \leftarrow \mathsf{Sim}\big(\mathbb{1}^\lambda, T, \mathsf{Leak}(T, \omega, \beta)\big)\Big\}.$$

Essentially, correctness requires that the evaluation of the keys on x is a secret-sharing of β if $x = \omega$, or of 0 otherwise. Security instead states that the information inferable from a subset of the keys is bounded by the leakage function Leak, which takes as input the special position ω, the non-zero value β and the set of corrupted parties T. In most cases, Leak just outputs the domain size N and the codomain $\mathbb{G}$ of the point function. However, this will not happen in the DPF on which our protocol relies.

We write $\mathsf{DPF}^n_N.\mathsf{FullEval}(i, \kappa_i)$ to mean the result of calling Eval on the entire domain of the function, obtaining a secret-sharing of the full length-N unit vector.

State-of-the-art. Actually, very little is known about DPFs. In [BGI15], the authors presented a 2-party DPF with $O\big(\log(N)\big)$ key size and an n-party construction with $O(\sqrt{N})$ key size. In both cases, the only leakage is N and $\mathbb{G}$, however, while the 2-party construction allows outputs in any group $\mathbb{G}$, the multiparty DPF essentially works only when $\mathbb{G} = (\{0,1\}^l, \oplus)$, or when $\mathbb{G}$ has polynomial order. In [BKKO20], Bunn et al. presented an improved version of the second algorithm for the 3-party case, however, obtaining again $O(\sqrt{N})$ key size. As we will show in Section 3, this construction is also limited to outputs in $\mathbb{G} = (\{0,1\}^l, \oplus)$, and does not extend to e.g. $\mathbb{F}_p$ for a large prime p.

Distributed Sum of Point Functions We use a simple extension of DPFs to sums of point functions, as also done in [BCG+20]. A DSPF scheme $\mathsf{DSPF}^n_{N,t}$ consists of algorithms $(\mathsf{DSPF}^n_{N,t}.\mathsf{Gen}, \mathsf{DSPF}^n_{N,t}.\mathsf{Eval})$, just as a DPF, except now Gen takes as input a pair of length-t vectors $\boldsymbol{\omega}, \boldsymbol{\beta} \in [N]^t \times \mathbb{G}^t$, which define the sum of point functions

$$f_{\boldsymbol{\omega},\boldsymbol{\beta}}(x) = \sum_{i \in [t]} \boldsymbol{\beta}[i] \cdot \delta_{\boldsymbol{\omega}[i]}(x)$$

Observe that $f_{\boldsymbol{\omega},\boldsymbol{\beta}}$ can be represented as a sum of unit vectors. We will refer to the latter as a multi-point vector.

The correctness property of a DSPF is then the same as a DPF, except we require that $\sum_{i \in [n]} v_i = f_{\boldsymbol{\omega},\boldsymbol{\beta}}(x)$, where $v_i = \mathsf{DSPF}^n_{N,t}.\mathsf{Eval}(i, \kappa_i, x)$. The security property is defined the same way as in a DPF.

Given a DPF, constructing a t-point DSPF can be done in the natural way, using one DPF instance for each of the t points, and summing up the t outputs of $\mathsf{DPF}^n_N.\mathsf{Eval}$ to evaluate the DSPF.

3 Generalisation of the 3-party DPF to Prime Fields

In this section, we first recap the 3-party DPF of [BKKO20], and then describe our extension of this to support outputs modulo p for any prime p.

High-level description of [BKKO20]. The scheme assumes N, the domain size, is a perfect square, and the codomain is $\mathbb{F}_{2^l}$. It uses a PRG $G : \{0,1\}^\lambda \longrightarrow \mathbb{F}_{2^l}^{\sqrt{N}}$. DPF keys in their construction do not leak anything beyond the domain and codomain, namely, the leakage function is given by $\mathsf{Leak}(T, \omega, \beta) = (N, \mathbb{F}_{2^l})$ for every subset of parties $T \subsetneq [3]$, special position $\omega \in [N]$ and non-zero value $\beta \in \mathbb{F}_{2^l}$.

During key generation, the unit vector representing the point function is rearranged into a $\sqrt{N} \times \sqrt{N}$ matrix M. If we rewrite $x \in [N]$ as $x'\sqrt{N} + x''$ with $0 \le x', x'' < \sqrt{N}$, the x-th element of the unit vector is moved to the x'-th row and x''-th column of the matrix M. We call the row containing β the special row.

$$M := \begin{array}{cc} \begin{bmatrix} 0 & 0 & \cdots & \cdots & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & 0 & \cdots & \cdots & 0 & \cdots & \cdots & \cdots & 0 \\ \vdots & \vdots & & & \vdots & & & & \vdots \\ 0 & 0 & & & 0 & & & & 0 \\ 0 & 0 & \cdots & 0 & \beta & 0 & \cdots & \cdots & 0 \\ 0 & 0 & & & 0 & & & & 0 \\ \vdots & \vdots & & & \vdots & & & & \vdots \\ 0 & 0 & \cdots & \cdots & 0 & \cdots & \cdots & \cdots & 0 \end{bmatrix} & \begin{array}{c} \\ \\ \\ \\ \leftarrow \omega' \\ \\ \\ \\ \end{array} \\ \begin{array}{c} \uparrow \\ \omega'' \end{array} & \end{array}$$

The algorithm is essentially based on the observation that it is possible to compress a 3-party secret-sharing of a row of zeros. Indeed, it suffices to sample 3 random PRG seeds a_j, b_j, c_j for every row j and give $\{a_j, b_j\}$ to P_0, $\{b_j, c_j\}$ to P_1 and $\{c_j, a_j\}$ to P_2. To decompress, each party just has to evaluate the seeds and XOR the results. We obtain a secret-sharing of zero since

$$\Big(G(a_j) \oplus G(b_j)\Big) \oplus \Big(G(b_j) \oplus G(c_j)\Big) \oplus \Big(G(c_j) \oplus G(a_j)\Big) = \mathbf{0}.$$

In order to not leak ω', the parties need to obtain similar seeds for the special row too. Observe that for every row, an adversary controlling two parties sees two sets of seeds with only one element in common, therefore, security requires that the property to hold for the special row too. For this reason, the algorithm samples 4 PRG seeds $a_{\omega'}, b_{\omega'}, c_{\omega'}, d_{\omega'}$ and gives $\{a_{\omega'}, d_{\omega'}\}$ to P_0, $\{b_{\omega'}, d_{\omega'}\}$ to P_1 and $\{c_{\omega'}, d_{\omega'}\}$ to P_2. Observe how the property is still satisfied.

Although security is guaranteed, the seeds $a_{\omega'}, b_{\omega'}, c_{\omega'}, d_{\omega'}$ are not a compression of the special row. Indeed, by expanding them ad XORing the results as for the other rows, we obtain a secret-sharing of a random vector

$$\boldsymbol{r} := G(a_{\omega'}) \oplus G(b_{\omega'}) \oplus G(c_{\omega'}) \oplus G(d_{\omega'}).$$

Observe that when there exists at least one honest party, one of the seeds remains unknown to the adversary, therefore, $\boldsymbol{r}$ is always indistinguishable from random. The DPF exploits this fact to include the correction word

$$\boldsymbol{CW} := \boldsymbol{r} \oplus \underbrace{(\overbrace{0, 0, \dots, 0}^{\omega''}, \beta, 0, 0, \dots, 0)}_{\sqrt{N} \text{ elements}}$$

to the key of every party. By adding the correction word to the expansion of the seeds of the special row, we obtain exactly what we desire, however, we must find a way to perform this operation without leaking the position ω'. The algorithm solves the problem by including in the keys a secret-sharing $[[\boldsymbol{y}]]_2$ of the unit vector having 1 in the special position ω'. Let $y_i[x']$ denote the x'-th bit of P_i's share of $\boldsymbol{y}$. By summing $y_i[x'] \cdot \boldsymbol{CW}$ to the expansion of the seeds, we add the correction word only to the special row. To summarise, the evaluation algorithm retrieves the row corresponding to the point that has to be evaluated, expands the associated seeds and obliviously adds the correction word when necessary.

Prime field generalisation. In order to generate multiplication triples over large prime fields $\mathbb{F}$ following the blueprint described in the introduction, we needed a 3-party DPF with codomain $\mathbb{F}$. Therefore, the first necessary step was to generalise the construction of [BKKO20]. As we have already mentioned, our modification requires weakening security, by introducing additional leakage.

The issue. The main cause of problems is that large prime fields have characteristic different from 2 and therefore addition and subtraction are different operations. Referring to the roadmap in the previous section, we can still compress a secret-sharing of zero by sampling 3 PRG seeds a_j, b_j, c_j and giving $\{a_j, b_j\}$ to P_0, $\{b_j, c_j\}$ to P_1 and $\{c_j, a_j\}$ to P_2. However, the decompression requires attention, indeed, when two parties have a seed in common, one of them has to add its expansion to its secret-sharing, the other one has to subtract it. It is therefore necessary to associate every seed in the keys with a bit, which will be set if and only if the expansion of the seed has to be added. Whenever two parties have a seed in common, the associated bits will be opposites.

This property has to be satisfied by the seeds of the special row too. One possibility would be of course to do exactly the same as for the normal rows, obtaining a secret-sharing of zero. However, that would not allow us to use the correction word $\boldsymbol{CW}$ as it would leak the non-zero value β. The only other possibility would be to sample 4 PRG seeds $a_{\omega'}, b_{\omega'}, c_{\omega'}, d_{\omega'}$ as before and give $\{a_{\omega'}, d_{\omega'}\}$ to P_0, $\{b_{\omega'}, d_{\omega'}\}$ to P_1 and $\{c_{\omega'}, d_{\omega'}\}$ to P_2. Clearly, we have to associate every seed with a bit expressing whether its expansion has to be added or subtracted, but whatever way we do it for $d_{\omega'}$, there will always be two parties with the same bit. If those two parties are corrupted, the value of ω' is leaked to them, compromising security of the DPF. On the other hand, this leakage turned out not to be problematic for our application.

Our solution. We decided to generate the sign bits so that ω' is leaked when the last two parties are corrupted. Since these bits do not need to be random, it is enough to ensure that the first party always subtracts the expansion of its seeds, the second party always adds them and the last party always adds the expansion of the first seed and subtracts the expansion of the second one. This means that the seeds of the second and the third party now have to be ordered. For instance, when $j \neq \omega'$, we can give $\{a_j, b_j\}$ to P_0, (b_j, c_j) to P_1 and (a_j, c_j) to P_2. When instead $j = \omega'$, we can give $\{a_j, d_j\}$ to P_0, (d_j, b_j) to P_1 and (d_j, c_j) to P_2. The construction is secure as long as the seeds in common with the first party are always in the first position of P_1 and P_2's pairs (which are ordered). On the other hand, it is crucial that the seeds of P_0 are an unordered set, otherwise ω' would be leaked to the adversary when P_0 and P_1, or P_0 and P_2 are both corrupted.

The fact that we do not need to protect ω' from an adversary corrupting the last two parties allows us to further improve the efficiency of the construction. For instance, we can secret-share $\boldsymbol{y}$ only between the second and the third party and remove the correction word from the key of the first party. Actually, since $\boldsymbol{y}$ is a unit vector, we can further compress the secret-sharing using the 2-party DPF of [BGI15], which has logarithmic key size.

Also, the seeds (a_j, b_j, c_j) can be somewhat compressed. If we consider the last seeds of the second and the third party, we observe that they coincide for every $j \neq \omega'$. When instead $j = \omega'$, the two seeds are independent. Essentially, they form a secret-sharing over $\mathbb{F}_{2^\lambda}$ of a $\sqrt{N}$-dimensional unit vector having special position ω' and random non-zero element. Such a secret-sharing can again be compressed using a 2-party DPF, such as from [BGI15, BGI16].

As a final optimization, it turns out the remaining seeds can also be compressed by roughly a factor of 2. This technique relies on the fact that we can generate the missing seeds using Random-OT tuples[2], which can themselves be compressed using a PCG based on the LPN assumption with *logarithmic* overhead [BCG+19b]. We omit the details here for ease of presentation, but the technique is used in our 3-party DPF protocol in Section 5.

Construction and Concrete Efficiency. Our 3-party DPF following the above ideas is given in Figure 2. The construction assumes the domain size is a perfect square and has a prime field $\mathbb{F}$ as codomain. The size of the key κ_0 is $\sqrt{N} \cdot 2\lambda$ bits, while the size of κ_1 and κ_2 is dominated by $\sqrt{N} \cdot (\lambda + \log|\mathbb{F}|) + O(\log(N) \cdot \lambda)$ bits. When $|\mathbb{F}| \approx 2^\lambda$, this gives a total of around $6\sqrt{N}\lambda$ bits for all three keys. If we additionally apply the optimization mentioned above, and compress the seeds using random OT and LPN, the total key size falls to $3\sqrt{N}\lambda$ bits (ignoring small $\log N$ terms), which is around 3x smaller than that of [BKKO20] (which only works in groups of small characteristic, but on the other hand, does not leak any information on ω).

[2] Tuples $\big((X_0, X_1), (b, X_b)\big)$ where $X_0, X_1 \xleftarrow{\$} \{0,1\}^\lambda$ and $b \xleftarrow{\$} \{0,1\}$.

Prime field 3-party DPF

Let N be a perfect square and suppose that $\mathbb{G} = \mathbb{F}$. Let $G : \{0,1\}^\lambda \longrightarrow \mathbb{F}^{\sqrt{N}}$ be a PRG and let $\mathsf{DPF}^2_{\sqrt{N}}$ denote a 2-party DPF with domain size $\sqrt{N}$.

DPF.Gen. On input $1\!\!1^\lambda$, $\omega \in [N]$ and $\beta \in \mathbb{F}$, perform the following operations:

1. Rewrite ω as $\omega' \cdot \sqrt{N} + \omega''$ where $0 \le \omega', \omega'' < \sqrt{N}$.
2. Sample $\Delta \xleftarrow{\$} \mathbb{F}_{2^\lambda}$ and compute
$$(\widehat{\kappa}^1_1, \widehat{\kappa}^2_1) \leftarrow \mathsf{DPF}^2_{\sqrt{N}}.\mathsf{Gen}(1\!\!1^\lambda, \omega', 1), \qquad (\widehat{\kappa}^1_2, \widehat{\kappa}^2_2) \leftarrow \mathsf{DPF}^2_{\sqrt{N}}.\mathsf{Gen}(1\!\!1^\lambda, \omega', \Delta).$$
3. For every $j \in [\sqrt{N}]$ with $j \neq \omega'$, sample $a_j, b_j \xleftarrow{\$} \{0,1\}^\lambda$ and set
$$S^0_j \leftarrow \{a_j, b_j\}, \qquad S^1_j \leftarrow b_j, \qquad S^2_j \leftarrow a_j.$$
4. Sample $a_{\omega'}, d_{\omega'} \xleftarrow{\$} \{0,1\}^\lambda$ and set
$$S^0_{\omega'} \leftarrow \{a_{\omega'}, d_{\omega'}\}, \qquad S^1_{\omega'} \leftarrow d_{\omega'}, \qquad S^2_{\omega'} \leftarrow d_{\omega'}.$$
5. Compute
$$b_{\omega'} \leftarrow \mathsf{DPF}^2_{\sqrt{N}}.\mathsf{Eval}(1, \widehat{\kappa}^1_2, \omega'), \qquad c_{\omega'} \leftarrow \mathsf{DPF}^2_{\sqrt{N}}.\mathsf{Eval}(2, \widehat{\kappa}^2_2, \omega'),$$
$$\boldsymbol{CW} \leftarrow G(a_{\omega'}) - G(b_{\omega'}) + G(c_{\omega'}) - G(d_{\omega'}) + \underbrace{(\overbrace{0,0,\ldots,0}^{\omega''},\beta,0,0,\ldots,0)}_{\sqrt{N}\text{ elements}}.$$
6. Output $(\kappa_0, \kappa_1, \kappa_2)$ where
$$\kappa_0 := (S^0_j)_{j\in[\sqrt{N}]}, \qquad \kappa_i := \left(\widehat{\kappa}^i_1, \widehat{\kappa}^i_2, (S^i_j)_{j\in[\sqrt{N}]}, \boldsymbol{CW}\right) \quad \text{if } i \in \{1,2\}.$$

DPF.Eval. On input $i \in [3]$, the key κ_i and a point $x \in [N]$, perform the following operations:

1. Rewrite x as $x' \cdot \sqrt{N} + x''$ where $0 \le x', x'' < \sqrt{N}$.
2. If $i = 0$, pick an arbitrary ordering and rewrite $S^0_{x'}$ as (w^0_1, w^0_2).
3. If $i \in \{1,2\}$, set $w^i_1 \leftarrow S^i_{x'}$ and compute
$$y_i[x'] \leftarrow \mathsf{DPF}^2_{\sqrt{N}}.\mathsf{Eval}(i, \widehat{\kappa}^i_1, x'), \qquad w^i_2 \leftarrow \mathsf{DPF}^2_{\sqrt{N}}.\mathsf{Eval}(i, \widehat{\kappa}^i_2, x').$$
4. Compute
$$\boldsymbol{v^i_{x'}} \leftarrow \begin{cases} -G(w^0_1) - G(w^0_2) & \text{if } i = 0, \\ y_i[x'] \cdot \boldsymbol{CW} + G(w^1_1) + G(w^1_2) & \text{if } i = 1, \\ y_i[x'] \cdot \boldsymbol{CW} + G(w^2_1) - G(w^2_2) & \text{if } i = 2. \end{cases}$$
5. Output $v^i_{x'}[x'']$.

Fig. 2. The prime field 3-party DPF

Theorem 1. *The construction described in Figure 2 is a 3-party DPF for* $(N, \mathbb{F})$ *with leakage*

$$\mathsf{Leak}(T, \omega, \beta) = \begin{cases} (N, \mathbb{F}) & \text{if } T \neq \{1, 2\}, \\ \left(N, \mathbb{F}, \lfloor \omega/\sqrt{N} \rfloor\right) & \text{if } T = \{1, 2\}. \end{cases}$$

The proof of Theorem 1 can be found in [AS21, Appendix A].

Extension to Distributed Sum of Point Functions. In later sections, we will use a distributed *sum of* point functions, built on top of our 3-party DPF in the naive way, as described in Section 2.3. Here, the leakage function is extended to output $\lfloor \omega_i/\sqrt{N} \rfloor$, for each special position ω_i, for $i \in [t]$, when the set of corruptions is $T = \{1, 2\}$.

4 Multiparty PCG for Triple Generation

In this section, we show how to use our 3-party DPF to construct a multi-party, pseudorandom correlation generator for authenticated triple generation.

Authenticated secret-sharing. We produce additively secret-shared values with information-theoretic MACs, as used in SPDZ [DPSZ12, DKL+13]. Here, an n-party secret-sharing of $x \in \mathbb{F}$ is given by a tuple

$$[\![x]\!] := (\alpha_i, x_i, m_{x,i})_{i \in [n]}$$

where (α_i, x_i, m_{x_i}) are known to the i-th party. Each $\alpha_i \in \mathbb{F}$ is fixed for every sharing x, and is a share of the global MAC key $\alpha = \sum_i \alpha_i$. The shares $x_i \in \mathbb{F}$ and MAC shares $m_{x,i} \in \mathbb{F}$ then satisfy

$$\sum_i x_i = x, \quad \sum_i m_{x,i} = \alpha \cdot x$$

We construct a PCG for the correlation which samples a random triple $([\![x]\!], [\![y]\!], [\![z]\!])$, where x, y are random elements of the ring $R = \mathbb{F}[X]/F(X)$, and $z = x \cdot y$ (while the MAC key α is a scalar in $\mathbb{F}$). As discussed in Section 2, when p is a suitable prime and $F(X)$ is e.g. a cyclotomic polynomial of degree N, this is equivalent to a batch of N triples over $\mathbb{F}$, thanks to the CRT isomorphism $R \cong \mathbb{F}^N$.

Note that it is easy to see that this correlation satisfies the reverse-samplable requirement.

4.1 Construction

Our construction is given in Fig. 3 and Fig. 4. We combine 3-party DPFs with the ring-LPN assumption, following the outline in the introduction (also sketched in [BCG+20]).

In more detail, we will compress the x, y terms of the triple using sparse, random polynomials $u^r(X), v^r(X) \in R$, for $r \in [c]$. Recall that if $\boldsymbol{a} \in R^c$ is a public, random vector over R, then

$$x = \langle \boldsymbol{a}, \boldsymbol{u} \rangle, \quad y = \langle \boldsymbol{a}, \boldsymbol{v} \rangle$$

are computationally indistinguishable from random R elements, under the module-LPN assumption.

We sample the u^r, v^r polynomials by first picking sparse u_i^r, v_i^r for each party, and summing up these shares. These are implicitly defined in steps 2 of Fig. 3, which sample the non-zero coefficients and values of the polynomials.

Then, we use 2-party distributed (sums of) point functions to compress additive shares of the cross-products $\alpha_j \cdot u_i^r, \alpha_j \cdot v_i^r$ and $u_i^r \cdot v_j^s$, in steps 3–4. This allows the parties to obtain shares of the MACs $\alpha x, \alpha y$, as well as the product xy.

Finally, to obtain shares of αxy, we decompose this into a sum of products $\alpha_i \cdot x_j \cdot y_k$, for every $i, j, k \in [n]$. By distributing shares of each term $\alpha_i \cdot u_j^r \cdot v_k^s$ using the 3-party DPF from Section 3, the parties can locally recover shares of αxy in the evaluation stage.

Note that due to the leakage in our 3-party DPF, if P_j and P_k are both corrupted, they learn something about the indices of the non-zero entries in $\alpha_i \cdot u_j^r \cdot v_k^s$. However, since these indices are independent of α_i, this leakage does not give away anything that wasn't already known to P_j and P_k.

In [AS21, Appendix B],we prove the following.

Theorem 2. *Suppose that* $\mathsf{DSPF}^2_{N,t}$, DSPF^2_{2N,t^2} *and* DSPF^3_{2N,t^2} *are secure distributed sums of point functions, and the* R^c*-LPN*$_t$ *assumption (Definition 1) holds. Then the construction in Fig. 3–4 is a secure PCG for n-party authenticated triples over* $R = \mathbb{F}[X]/F(X)$.

Efficiency. Note that we can optimize the construction slightly, with the observation that in any 3-party DPF instance where two of i, j, k are equal, we can instead use a 2-party DPF. This reduces the total number of 3-party DSPFs from $c^2n^2(n-1)$ down to $c^2n(n-1)(n-2)$. Each DSPF has t^2 points and a domain of size $2N$. There are also $O(c^2n^2)$ 2-party DSPFs, however, since these have logarithmic key size, their cost is dominated by the 3-party instances.

As a further optimization, we can rely on module-LPN with a *regular error distribution* [BCG+20], where each of the t non-zero entries in an error vector is sampled to be within a fixed range of length N/t. This reduces the domain size of the DPFs from $2N$ and N down to $2N/t$ and N/t, respectively.

In Section 6, we analyze the concrete parameters of our PCG, and the efficiency of our protocol for securely setting up the seeds and producing triples.

5 Distributed Setup for the 3-Party DPF

We now present an actively secure protocol that permits to distribute the keys of the 3-party DPF described in Section 3. We start by giving an overview of the

PCG$_{\text{triple}}$

Let $\mathbb{F}$ be a prime field and let N be the number of generated triples. Let t and c be the parameters of the Module-LPN assumption.

Gen: On input $\mathbb{1}^\lambda$, do the following:

1. Sample MAC key shares $\alpha_i \xleftarrow{\$} \mathbb{F}$, for every $i \in [n]$.
2. For every $i \in [n]$, $r \in [c]$, sample $\boldsymbol{\omega}_i^r, \boldsymbol{\eta}_i^r \xleftarrow{\$} [N]^t$ and $\boldsymbol{\beta}_i^r, \boldsymbol{\gamma}_i^r \xleftarrow{\$} \mathbb{F}^t$.
3. For every $i, j \in [n]$ with $i \neq j$, $r \in [c]$, compute

$$\left(U_{i,j}^{r,0}, U_{i,j}^{r,1}\right) \leftarrow \mathsf{DSPF}_{N,t}^2.\mathsf{Gen}\left(\mathbb{1}^\lambda,\ \boldsymbol{\omega}_i^r,\ \alpha_j \cdot \boldsymbol{\beta}_i^r\right),$$
$$\left(V_{i,j}^{r,0}, V_{i,j}^{r,1}\right) \leftarrow \mathsf{DSPF}_{N,t}^2.\mathsf{Gen}\left(\mathbb{1}^\lambda,\ \boldsymbol{\eta}_i^r,\ \alpha_j \cdot \boldsymbol{\gamma}_i^r\right).$$

4. For every $i, j \in [n]$ with $i \neq j$, $r, s \in [c]$, compute

$$\left(C_{i,j}^{r,s,h}\right)_{h\in[2]} \leftarrow \mathsf{DSPF}_{2N,t^2}^2.\mathsf{Gen}\left(\mathbb{1}^\lambda,\ \boldsymbol{\omega}_i^r \boxplus \boldsymbol{\eta}_j^s,\ \boldsymbol{\beta}_i^r \otimes \boldsymbol{\gamma}_j^s\right).$$

5. For every $i, j, k \in [n]$ with i, j, k not all equal, for $r, s \in [c]$, compute

$$\left(W_{i,j,k}^{r,s,h}\right)_{h\in[3]} \leftarrow \mathsf{DSPF}_{2N,t^2}^3.\mathsf{Gen}\left(\mathbb{1}^\lambda,\ \boldsymbol{\omega}_j^r \boxplus \boldsymbol{\eta}_k^s,\ \alpha_i \cdot (\boldsymbol{\beta}_j^r \otimes \boldsymbol{\gamma}_k^s)\right).$$

6. For every $i \in [n]$, output the seed

$$\kappa_i \leftarrow \left(\begin{array}{c} \alpha_i, \left(\boldsymbol{\omega}_i^r, \boldsymbol{\beta}_i^r\right)_{r\in[c]}, \left(\boldsymbol{\eta}_i^r, \boldsymbol{\gamma}_i^r\right)_{r\in[c]}, \left(U_{i,j}^{r,0}, U_{j,i}^{r,1}\right)_{\substack{j\neq i \\ r\in[c]}}, \left(V_{i,j}^{r,0}, V_{j,i}^{r,1}\right)_{\substack{j\neq i \\ r\in[c]}} \\ \left(C_{i,j}^{r,s,0}, C_{j,i}^{r,s,1}\right)_{\substack{j\neq i \\ r,s\in[c]}}, \left(W_{i,j,k}^{r,s,0}, W_{k,i,j}^{r,s,1}, W_{j,k,i}^{r,s,2}\right)_{\substack{(j,k)\neq(i,i), \\ r,s\in[c]}} \end{array}\right)$$

Eval: On input the seed κ_i, do the following:

1. For every $r \in [c]$, define the two polynomials

$$u_i^r(X) = \sum_{l\in[t]} \boldsymbol{\beta}_i^r[l] \cdot X^{\boldsymbol{\omega}_i^r[l]}, \quad v_i^r(X) = \sum_{l\in[t]} \boldsymbol{\gamma}_i^r[l] \cdot X^{\boldsymbol{\eta}_i^r[l]}$$

2. For every $r \in [c]$, compute

$$\widetilde{u}_i^r = \alpha_i \cdot u_i^r + \sum_{j\neq i} \left(\mathsf{DSPF}_{N,t}^2.\mathsf{FullEval}(U_{i,j}^{r,0}) + \mathsf{DSPF}_{N,t}^2.\mathsf{FullEval}(U_{j,i}^{r,1})\right)$$
$$\widetilde{v}_i^r = \alpha_i \cdot v_i^r + \sum_{j\neq i} \left(\mathsf{DSPF}_{N,t}^2.\mathsf{FullEval}(V_{i,j}^{r,0}) + \mathsf{DSPF}_{N,t}^2.\mathsf{FullEval}(V_{j,i}^{r,1})\right)$$

(viewing outputs of $\mathsf{FullEval}$ as degree $N-1$ polynomials over $\mathbb{F}$)

Fig. 3. PCG$_{\text{triple}}$ - Part 1

3. For every $r, s \in [c]$, compute
$$w_i^{r,s} = u_i^r \cdot v_i^s + \sum_{j \neq i} \left(\mathsf{DSPF}^2_{2N,t^2}.\mathsf{FullEval}(C_{i,j}^{r,s,0}) + \mathsf{DSPF}^2_{2N,t^2}.\mathsf{FullEval}(C_{j,i}^{r,s,1}) \right)$$
4. For every $r, s \in [c]$, compute
$$\widetilde{w}_i^{r,s} = \sum_{\substack{j,k \\ (i,i)\neq(j,k)}} \Bigg(\mathsf{DSPF}^3_{2N,t^2}.\mathsf{FullEval}(W_{i,j,k}^{r,s,0}) + \mathsf{DSPF}^3_{2N,t^2}.\mathsf{FullEval}(W_{k,i,j}^{r,s,1}) + \mathsf{DSPF}^3_{2N,t^2}.\mathsf{FullEval}(W_{j,k,i}^{r,s,2}) \Bigg) + \alpha_i \cdot u_i^r(X) \cdot v_i^s(X)$$
5. Define the vectors of polynomials $\boldsymbol{u}_i = (u_i^1, \ldots, u_i^c)$, $\boldsymbol{v}^i = (v_i^1, \ldots, v_i^c)$, similarly for $\widetilde{\boldsymbol{u}}_i, \widetilde{\boldsymbol{v}}_i$.
Let $\boldsymbol{w}_i = (w_i^{0,0}, \ldots, w_i^{c-1,0}, w_i^{0,1}, \ldots, w_i^{c-1,1}, \ldots, w_i^{c-1,c-1})$, and similarly define $\widetilde{\boldsymbol{w}}_i$.
6. Compute the final shares
$$x_i = \langle \boldsymbol{a}, \boldsymbol{u}_i \rangle, y_i = \langle \boldsymbol{a}, \boldsymbol{v}_i \rangle, z_i = \langle \boldsymbol{a} \otimes \boldsymbol{a}, \boldsymbol{w}_i \rangle \quad \text{and}$$
$$m_{x,i} = \langle \boldsymbol{a}, \widetilde{\boldsymbol{u}}_i \rangle, m_{y,i} = \langle \boldsymbol{a}, \widetilde{\boldsymbol{v}}_i \rangle, m_{z,i} = \langle \boldsymbol{a} \otimes \boldsymbol{a}, \widetilde{\boldsymbol{w}}_i \rangle$$
in $\mathbb{F}_p[X]/\big(F(X)\big)$.
7. Output $(\alpha_i, x_i, y_i, z_i, m_{x,i}, m_{y,i}, m_{z,i})$.

Fig. 4. PCG$_{\text{triple}}$ - Part 2

passively secure approach; later we will delve into the details, including active security.

High-level overview. The protocol permits to derive a 3-party secret-sharing of the unit-vector
$$\underbrace{(\overbrace{0, 0, \ldots, 0}^{\omega}, \beta, 0, 0, \ldots, 0)}_{N}$$
given secret-shared special position and non-zero value $[[\omega]]_2$ and $[[\beta]]$. Writing $\omega = \omega'\sqrt{N} + \omega''$, and following the blueprint of our 3-party DPF, the protocol samples a random $\Delta \in \mathbb{F}_{2^\lambda}$ and shares the unit vectors
$$\boldsymbol{y} = \underbrace{(\overbrace{0, 0, \ldots, 0}^{\omega'}, 1, 0, 0, \ldots, 0)}_{\sqrt{N}}, \qquad \boldsymbol{Y} = \underbrace{(\overbrace{0, 0, \ldots, 0}^{\omega'}, \Delta, 0, 0, \ldots, 0)}_{\sqrt{N}}$$
between the last two parties using a 2-party DPF. The shares of $\boldsymbol{Y}$ are regarded as vectors of seeds.

As we mentioned in Section 3, to derive the remaining seeds, we rely on oblivious transfer (OT). Observe that for every position $j \in [\sqrt{N}]$, the first party has to generate two random seeds. Moreover, for every j, the last two parties have to learn one of these seeds each. The discovered seeds coincide if and only if $j = \omega'$. We setup these seeds by running two sets of OTs, where the first party is sender in both, and the other two parties play receiver in one set each. The receivers' choice bits are determined based on the shares of $\boldsymbol{y}$, which are random bits that coincide if and only if $j = \omega'$.

Assuming the availability of a 3-party secret sharing of

$$\boldsymbol{v} = (\underbrace{\overbrace{0, 0, \ldots, 0}^{\omega''}, \beta, 0, 0, \ldots, 0}_{\sqrt{N}}),$$

the generation of the correction word is very simple: each party can just retrieve its share of $\boldsymbol{v}$ and add or subtract the expansions of its seeds. The correction word is obtained by broadcasting and adding the results. Once we have the correction word, the DPF setup phase is complete. The only remaining question, then, is how to derive the secret-sharing of $\boldsymbol{v}$: since it is a unit-vector, we will use recursion.

We now discuss the protocol more in detail, including the details of recursion and active security. To simplify the presentation, we introduce some notation and building blocks.

Double exponential representation. We assume that N is a double exponential, that is, $N = 2^{2^h}$ for some $h \in \mathbb{N}$. In practice, this choice is rather restrictive as the value of N grows very quickly. However, we only make this assumption to simplify the description of a recursive step in our protocol, and this step can easily be adapted to the case $N = 2^m$ without significantly affecting the overall complexity[3].

We define the double exponential function $\mathrm{dE}(\cdot)$ as

$$\mathrm{dE}(k) := \begin{cases} 2 & \text{if } k = -1, \\ 2^{2^k} & \text{otherwise.} \end{cases}$$

We also use the following decomposition of integers, using a double exponential basis. Its proof can be found in [AS21, Appendix C].

Lemma 1. *Any $\omega \in [N]$ can be written in a unique way as*

$$\omega = x(-1) + \sum_{i \in [h]} x(i) \cdot \mathrm{dE}(i)$$

for some $x(i) \in [\mathrm{dE}(i)]$ (depending on ω), for $i \in [h] \cup \{-1\}$, i.e. $0 \le x(i) < 2^{2^i}$ if $i \in [h]$ and $x(-1) \in \{0, 1\}$.

[3] The protocol is more efficient when m is divisible by a power of 2.

Notation. Given a number $\omega \in [N]$, we denote its j-th bit by ω_j, whereas the j-th element of its double exponential notation is indicated by $\omega(j)$. Let $\mathcal{K} := [h] \cup \{-1\}$ and define

$$\mathcal{T} := \Big\{ (k, j) \mid k \in \mathcal{K}, j \in [\mathrm{dE}(k)] \Big\}.$$

In the protocol we use h PRGs. The k-th one will be $G_k : \{0,1\}^\lambda \longrightarrow \mathbb{F}^{\mathrm{dE}(k)}$. We will also rely on a tweakable correlation-robust hash function

$$H : \{0,1\}^\lambda \times \{0,1\}^* \longrightarrow \{0,1\}^\lambda.$$

An important fact is that the protocol requires the cardinality of the field $\mathbb{F}$ to be sufficiently close to 2^λ. More specifically, consider the map $\mathrm{Enc} : \{0,1\}^\lambda \longrightarrow \mathbb{F}$ sending every string $(x_0, x_1, \dots, x_{\lambda-1})$ to $\sum_{i\in[\lambda]} x_i \cdot 2^i$. Let U be the uniform distribution over $\{0,1\}^\lambda$ and let V be the uniform distribution over $\mathbb{F}$. In order to be secure, the protocol requires the statistical distance between V and $\mathrm{Enc}(U)$ to be negligible in the security parameter. It is possible to prove that this condition is satisfied if and only if $|p - 2^\lambda|/2^\lambda$, where $p = |\mathbb{F}|$, is negligible in λ.

We also define a set of sign bits u_i^l with $i \in [3]$ and $l \in \{0,1\}$, by

$$u_i^l := \begin{cases} 1 & \text{if } i = 1, \text{ or } i = 2 \text{ and } l = 0 \\ -1 & \text{otherwise.} \end{cases}$$

These parameters will indicate whether we need to add or subtract the expansion of the seeds in the 3-party DPF keys (see Section 3).

Finally, we define some matrices used to translate between different representations. In the protocol, we use the set of matrices $(B_k)_{k\in\mathcal{K}}$, which allow us to map an N-dimensional unit vector having special position $\omega \in [N]$ into a $\mathrm{dE}(k)$-dimensional unit vector with special position $\omega(k)$ and the same non-zero value. We also use a matrix $C \in \mathbb{F}_2^{\log(N)\times|\mathcal{T}|}$. This allows us to retrieve a binary representation of $\eta \in [N]$, given the unit-vector

$$\underbrace{(\overbrace{0, 0, \dots, 0}^{\eta(k)}, 1, 0, 0, \dots, 0)}_{\mathrm{dE}(k)}$$

for every $k \in \mathcal{K}$. A formal description of the matrices $(B_k)_k$ and C can be found in [AS21, Appendix D].

5.1 Resources

The protocol we are going to present relies on an authenticated Random-OT functionality, which we instantiate using similar techniques to the TinyOT protocol [NNOB12]. We assume that every pair of parties (P_i, P_j) has access to an instance $\mathcal{F}^{i,j}_{\text{auth-ROT}}$ of this resource. The functionality $\mathcal{F}^{i,j}_{\text{auth-ROT}}$ provides

Random-OT tuples, i.e. upon every call, P_i, the sender, obtains two random values $X_0, X_1 \in \{0,1\}^\lambda$, whereas P_j, the receiver, obtains a random choice bit b and the value X_b. Additionally, $\mathcal{F}^{i,j}_{\text{auth-ROT}}$ permits to perform linear operations on the choice bits it stored. The results of these computations are output to P_i and their correctness is guaranteed even when P_j is corrupted. Finally, the resource can output random bits to P_j. The latter can be used in combination with the choice bits in the computations. A formal description of $\mathcal{F}^{i,j}_{\text{auth-ROT}}$ can be found in [AS21, Appendix F], where we also show how to implement it with low communication complexity using a Correlated-OT functionality.

In the protocol, we also use a black-box multiparty computation functionality $\mathcal{F}_{\text{MPC}}$ which allows n parties to perform computations over the prime field $\mathbb{F}$ and over $\mathbb{F}_2$. A complete description can be found in [AS21, Figure 13].The functionality stores the inputs and results of the computations internally, providing the parties with handles. Each of the stored values is associated with one of the domains $\mathbb{F}$ and $\mathbb{F}_2$ to which the element must belong. In the first case, the handle of x is denoted by $[[x]]$, whereas in the second case, the handle is denoted by $[[x]]_2$. Sometimes, we will abuse the notation and we will write $[[x]]_2$ even if $x \notin \{0,1\}$, in that case, it is understood that the functionality stored x bit by bit and the number of such bits depends only on the actual domain of x. The functionality $\mathcal{F}_{\text{MPC}}$ also features a 2-party DPF functionality, which, on input the indexes of two parties i, j, a value $[[\beta]]$ in $\mathbb{F}$ or $\mathbb{F}_{2^\lambda}$, a power of 2 M and $[[\omega]]_2 \in [M]$, outputs to P_i and P_j a 2-party secret-sharing of the M-dimensional unit vector having β in the ω-th position. The group on which the secret-sharing is defined coincides with the field to which β belongs.

Finally, we will use a functionality $\mathcal{F}_{\text{Rand}}$ which provides all the parties with random values sampled from the queried domains.

5.2 The Protocol

The functionality that our construction is going to implement is described in Figure 5. Observe that when the second and the third party are both corrupted the special position of the unit vector is leaked to the adversary. Since the protocol is based on the 3-party DPF described in Section 3, a leakage of this type was unavoidable. The functionality also allows the adversary to test the inputs in several occasions, every incorrect guess leading to an abort. In the triple generation protocol, the non-zero value β will be uniformly distributed in $\mathbb{F}^\times$, so any attempt of the adversary to guess it will fail with overwhelming probability. The leakage about the special position will not instead constitute a problem as it will be absorbed by the hardness of Module-LPN.

We can finally present our protocol. Its formal description can be found in Figures 6, 7 and 8.

Recursion. The protocol uses the 3-party DPF described in Figure 2 recursively in h levels indexed by $k = 0, 1, \ldots, h-1$. Once the k-th level is completed, the

$\mathcal{F}_{\text{3-DPF}}$

MPC Functionality. The functionality features the procedures Input, LinComb, Mult, Output, 2-DPF and Abort as in $\mathcal{F}_{\text{MPC}}$.

3-DPF. On input $[[\beta]] \in \mathbb{F}$, $[[\omega]]_2 \in [N]$ and three indexes $\sigma_0, \sigma_1, \sigma_2 \in [n]$:

1. If P_{σ_0}, P_{σ_1} and P_{σ_2} are all corrupted, send ω and β to the adversary. If only P_{σ_1} and P_{σ_2} are corrupted, send ω to the adversary.
2. If there exists $i \in [3]$ such that P_{σ_i} is corrupted, for three times, wait for $I \subseteq [N]$ from the adversary. If $\omega \notin I$, send ω to the adversary and abort.
3. If $\beta = 0$, output ZERO to the parties and to the adversary and stop.
4. If there exists $i \in [3]$ such that P_{σ_i} is corrupted, wait for the adversary.
   - If the latter sends $(\beta_\eta, \boldsymbol{v'_\eta})_{\eta \in [N]}$, check whether $\beta_\omega = \beta$. If this is the case, set $\widehat{\boldsymbol{v}} \leftarrow \boldsymbol{v'_\omega}$. Otherwise, send (ω, β) to the adversary and abort.
   - If the latter sends $(I''', \widehat{\boldsymbol{v}})$, check whether $\omega \in I'''$. If this is not the case, send ω to the adversary and abort.
5. If P_{σ_0}, P_{σ_1} and P_{σ_2} are all honest, set $\widehat{\boldsymbol{v}} \leftarrow \boldsymbol{0}$.
6. For every $\sigma_i \in \mathcal{H}$, sample a random $\boldsymbol{v_i} \in \mathbb{F}^N$ subject to
$$\sum_{\sigma_i \in \mathcal{H}} \boldsymbol{v_i} + \widehat{\boldsymbol{v}} = (\underbrace{\overbrace{0, 0, \dots, 0}^{\omega \text{ elements}}, \beta, 0, 0, \dots, 0}_{N \text{ elements}}).$$
7. Output $\boldsymbol{v_i}$ to party P_{σ_i} for every $\sigma_i \in \mathcal{H}$.

Fig. 5. The 3-party DPF functionality

parties obtain a secret-sharing over $\mathbb{F}$ of the unit vector

$$\boldsymbol{v_{k+1}} := (\underbrace{\overbrace{0, 0, \dots, 0}^{\widehat{\omega}(k+1)}, \beta, 0, 0, \dots, 0}_{\text{dE}(k+1)}),$$

$$\text{where} \qquad \widehat{\omega}(k+1) := \omega(-1) + \sum_{i=0}^{k} \omega(i) \cdot \text{dE}(i).$$

Observe that $\widehat{\omega}(h) = \omega$.

More in detail, suppose that the parties possess a secret-sharing over $\mathbb{F}$ of $\boldsymbol{v_k}$. We aim to use it to securely generate 3-party DPF keys for the unit vector $\boldsymbol{v_{k+1}}$ (see Figure 2). Using the evaluation algorithm, the parties can then expand the keys to obtain a secret-sharing of $\boldsymbol{v_{k+1}}$.

Rearranging $\boldsymbol{v_{k+1}}$ into a matrix. First of all, observe that $\boldsymbol{v_{k+1}}$ is an $N_k :=$ dE$(k+1)$-dimensional unit vector, whose special position is $\widehat{\omega}(k+1)$. Notice that

$$\widehat{\omega}(k+1) = \omega(k) \cdot \sqrt{N_k} + \widehat{\omega}(k) \quad \text{and} \quad 0 \leq \omega(k), \widehat{\omega}(k) < \text{dE}(k) = \sqrt{N_k}.$$

In other words, when we rearrange $\boldsymbol{v_{k+1}}$ into a square matrix, following the procedure described in Section 3, the special position ends up at the intersection between the $\omega(k)$-th row and the $\widehat{\omega}(k)$-th column. Observe that it is easy to obtain a secret-sharing of $\omega(k)$ over $\mathbb{F}_2$ given a secret-sharing of ω over $\mathbb{F}_2$. Indeed, $\omega(k)$ is described by a 2^k-bit substring of the bit representation of ω.

The vectors $\boldsymbol{y_0}$*,* $\boldsymbol{y_1}$ *and* $\boldsymbol{y_2}$*.* Following the blueprint of the 3-party DPF described in Section 3, the first ingredient needed to generate $\boldsymbol{v_{k+1}}$ is the vectors $\boldsymbol{y_{k,1}}$ and $\boldsymbol{y_{k,2}}$, i.e. a secret-sharing over $\mathbb{F}$ of

$$\boldsymbol{y^k} := (\underbrace{\overbrace{0, 0, \ldots, 0}^{\omega(k)}, 1, 0, 0, \ldots, 0}_{\mathrm{dE}(k)}) = \boldsymbol{y_{k,1}} + \boldsymbol{y_{k,2}}.$$

At the beginning of our protocol, using the 2-party DPF procedure in $\mathcal{F}_{\mathrm{MPC}}$, the second and third party obtain a secret-sharing of the unit vector

$$\boldsymbol{y} := (\underbrace{\overbrace{0, 0, \ldots, 0}^{\omega}, 1, 0, 0, \ldots, 0}_{N}).$$

By locally applying the matrix B_k on the shares, this also gives the shares $\boldsymbol{y_{k,1}}$ and $\boldsymbol{y_{k,2}}$. We recall that B_k maps an N-dimensional unit vector having special position $\omega \in [N]$ into a $\mathrm{dE}(k)$-dimensional unit vector with special position $\omega(k)$ and the same non-zero value.

From $\mathbb{F}$*-secret-sharing to binary secret-sharing.* In the previous paragraph, we described how it is possible to obtain a 2-party secret-sharing over $\mathbb{F}$ of the unit vector $\boldsymbol{y^k}$. In order to securely generate the seeds in the DPF key, we will need to convert this to a 2-party secret-sharing over the binary field $\mathbb{F}_2$. Using a standard trick , we can do this conversion without any interaction.

Recall that $|\mathbb{F}| = p$ for a large prime p. Suppose that the two parties have shares $b_1, b_2 \in [p]$, where $b_1 + b_2 \equiv b \bmod p$, for some $b \in \{0, 1\}$, as we do for each entry of $\boldsymbol{y^k}$. If the shares are random, then with overwhelming probability both of them are non-zero, so over the integers, $2 \leq b_1 + b_2 < 2p$ and therefore $b_1 + b_2 = b + p$. Reducing both sides modulo 2, we get that $(b_1 \bmod 2) \oplus (b_2 \bmod 2) = b \oplus 1$. In other words, for every $j \in [\mathrm{dE}(k)]$, the second and the third parties can obtain bits $b^j_{k,1} := y^j_{k,1} \bmod 2$ and $b^j_{k,2} := y^j_{k,2} \bmod 2$ that coincide if and only if $j = \omega(k)$.

This procedure works only if both b_1 and b_2 are non-zero, and for that reason, the parties abort if this is not satisfied. When the second and the third party are both honest, the property condition holds with overwhelming probability. If one of the parties is corrupted, however, $\mathcal{F}_{\mathrm{MPC}}$ allows the adversary to choose its shares. An attacker can exploit this fact to retrieve information about ω, indeed, it can select its shares so that the protocol aborts only if ω assumes particular values (selective failure attack). The corresponding leakage is modelled in step 2 of $\mathcal{F}_{3\text{-DPF}}$ (see Figure 5).

$\Pi_{\text{3-DPF}}$

The environment has access to the the procedures Input, LinComb, Mult, Output and 2-DPF of $\mathcal{F}_{\text{MPC}}$.

3-DPF. On input a number $[[\omega]]_2 \in [N]$, $[[\beta]] \in \mathbb{F}$ and indexes $\sigma_0, \sigma_1, \sigma_2 \in [n]$:
Generation of $\boldsymbol{y_{k,0}}$, $\boldsymbol{y_{k,1}}$ and $\boldsymbol{y_{k,2}}$.

1. The parties call 2-DPF over $\mathbb{F}$ with special position $[[\omega]]_2$, non-zero element 1 and indexes σ_1 and σ_2. If the latter aborts, the parties abort. Let $\boldsymbol{y_i}$ be the output received by P_{σ_i} for $i \in \{1,2\}$.
2. For every $k \in \mathcal{K}$ and $i \in \{1,2\}$, P_{σ_i} computes $\boldsymbol{y_{k,i}} \leftarrow B_k \cdot \boldsymbol{y_i}$. P_{σ_0} sets $\boldsymbol{y_{k,0}} \leftarrow \boldsymbol{0}$.

From $\mathbb{F}$ secret-sharing to binary secret-sharing.

3. If there exists $(k,j) \in \mathcal{T}$ and $i \in \{1,2\}$ such that $y^j_{k,i} = 0$, P_{σ_i} aborts.
4. For every $k \in \mathcal{K}$ and $i \in \{1,2\}$, P_{σ_i} sets $\boldsymbol{b_{k,i}} \leftarrow \boldsymbol{y_{k,i}} \bmod 2$.

Seed generation - Part 1.

5. The parties generate $[[\Delta]]_2 \leftarrow \mathsf{Rand}(\mathbb{F}_{2^\lambda})$.
6. The parties call 2-DPF over $\mathbb{F}_{2^\lambda}$ with special position $[[\omega]]_2$, non-zero element $[[\Delta]]_2$ and indexes σ_1 and σ_2. If the latter aborts, the parties abort. Let $\boldsymbol{T_i}$ be the output received by P_{σ_i} for $i \in \{1,2\}$.
7. For every $k \in \mathcal{K}$ and $i \in \{1,2\}$, P_{σ_i} computes $\boldsymbol{T_{k,i}} \leftarrow B_k \cdot \boldsymbol{T_i}$.
8. For every $(k,j) \in \mathcal{T}$ and $i \in \{1,2\}$, P_{σ_i} computes $Y^j_{k,i} \leftarrow H\big(T^j_{k,i}, (k,j)\big)$.

Seed generation - Part 2.

9. For ever $(k,j) \in \mathcal{T}$:
   (a) P_{σ_0} and P_{σ_1} call $\mathcal{F}^{\sigma_0,\sigma_1}_{\text{auth-ROT}}$ with P_{σ_0} as sender. P_{σ_0} obtains $(X^j_k[0], X^j_k[1]) \in \mathbb{F}_{2^\lambda} \times \mathbb{F}_{2^\lambda}$, P_{σ_1} receives $(t^j_{k,1}, X^j_k[2]) \in \mathbb{F}_2 \times \mathbb{F}_{2^\lambda}$, where $X^j_k[2] = X^j_k[t^j_{k,1}]$.
   (b) P_{σ_1} sends $w^j_{k,1} \leftarrow b^j_{k,1} \oplus t^j_{k,1}$ to P_{σ_0}.
   (c) P_{σ_0} and P_{σ_2} call $\mathcal{F}^{\sigma_0,\sigma_2}_{\text{auth-ROT}}$ with P_{σ_0} as sender. P_{σ_0} obtains $(W^j_k[0], W^j_k[1]) \in \mathbb{F}_{2^\lambda} \times \mathbb{F}_{2^\lambda}$, P_{σ_2} receives $(t^j_{k,2}, W^j_k) \in \mathbb{F}_2 \times \mathbb{F}_{2^\lambda}$, where $W^j_k = W^j_k[t^j_{k,2}]$.
   (d) P_{σ_2} sends $w^j_{k,2} \leftarrow b^j_{k,2} \oplus t^j_{k,2}$ to P_{σ_0}.
   (e) P_{σ_0} sends to P_{σ_2}
   $$Z^j_k[0] \leftarrow W^j_k[w^j_{k,2}] \oplus X^j_k[w^j_{k,1}], \qquad Z^j_k[1] \leftarrow W^j_k[w^j_{k,2} \oplus 1] \oplus X^j_k[w^j_{k,1} \oplus 1].$$
   (f) P_{σ_2} computes $X^j_k[3] \leftarrow W^j_k \oplus Z^j_k[b^j_{k,2}]$
10. For every $(k,j) \in \mathcal{T}$
    – P_{σ_0} sets $S^j_{k,0,0} \leftarrow X^j_k[0]$ and $S^j_{k,0,1} \leftarrow X^j_k[1]$.
    – P_{σ_1} sets $S^j_{k,1,0} \leftarrow X^j_k[2]$ and $S^j_{k,1,1} \leftarrow Y^j_{k,1}$.
    – P_{σ_2} sets $S^j_{k,2,0} \leftarrow X^j_k[3]$ and $S^j_{k,2,1} \leftarrow Y^j_{k,2}$.

Fig. 6. $\Pi_{\text{3-DPF}}$ - Part 1

First check.

11. For every $i \in \{1,2\}$ and $k \in \mathcal{K}$, P_{σ_0} and P_{σ_i} call LinearCombination in $\mathcal{F}^{\sigma_0,\sigma_i}_{\text{auth-ROT}}$ to compute
$$t_{k,i} \leftarrow \bigoplus_{j \in [\mathrm{dE}(k)]} [[t^j_{k,i}]]_2.$$
12. For every $k \in \mathcal{K}$, P_{σ_0} computes
$$\psi_k \leftarrow t_{k,1} \oplus t_{k,2} \oplus \bigoplus_{j \in [\mathrm{dE}(k)]} (w^j_{k,1} \oplus w^j_{k,2}).$$
If any of them is different from 1, it makes the protocol abort.

Second check.

13. For every $i \in \{1,2\}$, P_{σ_0} and P_{σ_i} call Random in $\mathcal{F}^{\sigma_0,\sigma_i}_{\text{auth-ROT}}$ for λ times. Let $R_i \in \{0,1\}^\lambda$ be the binary string obtained by P_{σ_i}.
14. For every $i \in \{1,2\}$, P_i inputs R_i into $\mathcal{F}_{\text{MPC}}$ with domain $\mathbb{F}_2$.
15. The parties call $\mathcal{F}_{\text{Rand}}$ to obtain a random matrix $V \in \mathbb{F}_2^{\lambda \times \log(N)}$
16. For every $i \in \{1,2\}$, P_{σ_0} and P_{σ_i} call LinearCombination in $\mathcal{F}^{\sigma_0,\sigma_i}_{\text{auth-ROT}}$ to compute $\Phi_i \leftarrow [[R_i]]_2 \oplus V \cdot C \cdot [[\boldsymbol{t_i}]]_2$ where $\boldsymbol{t_i}$ is the $|\mathcal{T}|$-dimensional vector having $t^j_{k,i}$ in the $(\mathrm{dE}(k)+j)$-th position.
17. P_{σ_0} computes $\Phi \leftarrow \Phi_1 \oplus \Phi_2 \oplus V \cdot C \cdot (\boldsymbol{w} \oplus \boldsymbol{1})$ where $\boldsymbol{w} \oplus \boldsymbol{1}$ is the $|\mathcal{T}|$-dimensional vector having $w^j_{k,1} \oplus w^j_{k,2} \oplus 1$ in the $(\mathrm{dE}(k)+j)$-th position.
18. Using $\mathcal{F}_{\text{MPC}}$ the parties open $\Phi' \leftarrow [[R_1]]_2 \oplus [[R_2]]_2 \oplus V \cdot [[\omega]]_2$. If $\Phi \neq \Phi'$, P_{σ_0} makes the protocol abort.

Base case.

19. For every $j \in [2]$ the parties set
$$x^j_0 \leftarrow \mathrm{Enc}(X^j_{-1}[0]), \quad x^j_1 \leftarrow \mathrm{Enc}(X^j_{-1}[1]), \quad x^j_2 \leftarrow \mathrm{Enc}(X^j_{-1}[2]),$$
$$x^j_3 \leftarrow \mathrm{Enc}(X^j_{-1}[3]), \quad x^j_4 \leftarrow \mathrm{Enc}(Y^j_{-1,1}), \quad x^j_5 \leftarrow \mathrm{Enc}(Y^j_{-1,2}).$$
P_{σ_0} sets $s^j_0 \leftarrow -x^j_0 - x^j_1$. P_{σ_1} sets $s^j_1 \leftarrow x^j_2 + x^j_4$. P_{σ_2} sets $s^j_2 \leftarrow x^j_3 - x^j_5$. Then, for each $i \in [3]$, P_{σ_i} sets $z_i \leftarrow s^0_i \oplus s^1_i$.
20. The parties perform the following operations
$$[[z_i]] \leftarrow \mathsf{Input}(P_{\sigma_i}, z_i) \quad \forall i \in [3]$$
$$[[CW]] \leftarrow ([[z_0]] + [[z_1]] + [[z_2]])^{-1} \cdot [[\beta]]$$
$$CW \leftarrow \mathsf{Output}([[CW]])$$
If $CW = 0$, the parties stop and output ZERO. If the operation cannot be performed due to a zero denominator, all the parties stop and output $\perp$.
21. Each party P_{σ_i} sets $v^j_{0,i} \leftarrow s^j_i \cdot CW$ for $j \in \{0,1\}$. Let $\boldsymbol{v_{0,i}} := (v^0_{0,i}, v^1_{0,i})$ for every $i \in [3]$.

Fig. 7. $\Pi_{\text{3-DPF}}$ - Part 2

Generation of the correction words.

22. For each $k \in [h]$ the parties compute the following operations
    (a) For every $i \in [3]$, P_{σ_i} broadcasts
$$\boldsymbol{CW_{k,i}} \leftarrow \boldsymbol{v_{k,i}} - \sum_{j \in [\mathrm{dE}(k)]} u_i^0 \cdot G_k(S^j_{k,i,0}) - \sum_{j \in [\mathrm{dE}(k)]} u_i^1 \cdot G_k(S^j_{k,i,1}).$$
    (b) The parties set $\boldsymbol{CW_k} \leftarrow \boldsymbol{CW_{k,0}} + \boldsymbol{CW_{k,1}} + \boldsymbol{CW_{k,2}}$.
    (c) Each party P_{σ_i} sets for every $j \in [\mathrm{dE}(k)]$
$$\boldsymbol{v^j_{k+1,i}} \leftarrow u_i^0 \cdot G_k(S^j_{k,i,0}) + u_i^1 \cdot G_k(S^j_{k,i,1}) + y^j_{k,i} \cdot \boldsymbol{CW_k}.$$
    Let $\boldsymbol{v_{k+1,i}} := (\boldsymbol{v^0_{k+1,i}} \parallel \boldsymbol{v^1_{k+1,i}} \parallel \ldots \parallel \boldsymbol{v^{\mathrm{dE}(k)-1}_{k+1,i}})$.

Final check.

23. The parties call $\mathcal{F}_{\mathrm{Rand}}$ to sample $\boldsymbol{\chi} = (\chi_0, \chi_1, \ldots, \chi_{N-1}) \in \mathbb{F}^N$.
24. Perform the following operations
    (a) $[[d_i]] \leftarrow \mathsf{Input}(P_{\sigma_i}, \langle \boldsymbol{\chi}, \boldsymbol{y_i} \rangle)$ for each $i \in \{1, 2\}$.
    (b) $[[\zeta_i]] \leftarrow \mathsf{Input}(P_{\sigma_i}, \langle \boldsymbol{\chi}, \boldsymbol{v_{h,i}} \rangle)$ for each $i \in [3]$.
    (c) $[[\rho]] \leftarrow [[\zeta_0]] + [[\zeta_1]] + [[\zeta_2]] - ([[d_1]] + [[d_2]]) \cdot [[\beta]]$
    (d) $\rho \leftarrow \mathsf{Output}([[\rho]])$
    (e) If $\rho \neq 0$, P_{σ_i} outputs ABORT and stops. Otherwise, P_{σ_i} outputs $\boldsymbol{v_{h,i}}$.

Fig. 8. $\Pi_{3\text{-DPF}}$ - Part 3

The seed generation - Part 1. We now turn to the task of generating the PRG seeds used in the DPF. We start with the method for obtaining the last seeds of the second and the third party, which, following the idea from Section 3, we compress using a 2-party DPF. Recall that these seeds coincide for every position $j \neq \omega(k)$, whereas, when $j = \omega(k)$, they are independent. Using the 2-party DPF command of $\mathcal{F}_{\mathrm{MPC}}$, we can obtain a 2-party secret-sharing over $\mathbb{F}_{2^\lambda}$ of

$$(\underbrace{\overbrace{0, 0, \ldots, 0}^{\omega}, \Delta, 0, 0, \ldots, 0}_{N}).$$

where Δ is sampled randomly by $\mathcal{F}_{\mathrm{MPC}}$. Then, by applying the matrix B_k, this can be translated into shares of

$$(\underbrace{\overbrace{0, 0, \ldots, 0}^{\omega(k)}, \Delta, 0, 0, \ldots, 0}_{\mathrm{dE}(k)}).$$

for any $k \in \mathcal{K}$. The only problem is that, in this way, the entries of the shares in the special position are not independent, due to the fixed correlation Δ. Therefore, to turn these shares into independent, random seeds, we apply the correlation-robust hash function H to each entry.

The seed generation - Part 2. Generating and distributing the remaining seeds is more complex. We have previously explained how the second and third parties derive, for each $j \in [\mathrm{dE}(k)]$, bits $b^j_{k,1}$ and $b^j_{k,2}$ that coincide if and only if $j = \omega(k)$. Now, for every $j \in [\mathrm{dE}(k)]$, the second and the third party must learn one of the seeds of the first party. The discovered seeds will coincide if and only if $j = \omega(k)$. We can therefore generate and distribute the remaining seeds using oblivious transfer (OT). Specifically, for every $j \in [\mathrm{dE}(k)]$, the first and the second party can obtain their missing seeds by means of a "sender-random" OT, i.e. an OT where the sender's messages, corresponding to the seeds of the first party, are random, while the receiver can choose its input. The first party will be the sender, while the second party will be the receiver with choice bit $b^j_{k,1}$. The third party can then receive its missing seed by means of another, now standard, OT. The sender, corresponding to the first party, will choose its messages to be the same as in the "sender-random" OT, while the choice bit of the receiver, the third party, will be $b^j_{k,2}$. The two OTs are implemented using the random-OT functionality $\mathcal{F}_{\text{auth-ROT}}$. Note that this functionality ensures that the choice bits are authenticated, which we rely on later, to check consistency of this stage and achieve active security.

The correction word (Fig. 8). After obtaining the seeds, the only missing piece of the DPF key is the correction word. Computing it is rather straightforward as each party can just retrieve its share of $\boldsymbol{v_k}$ and add or subtract the expansions of its seeds using G_k[4]. The correction word is obtained by broadcasting and adding the results. Observe that if recursion had not been used, at this point of the protocol, the parties would have needed to generate a secret-sharing of the $\sqrt{N}$-dimensional vector $\boldsymbol{v_{h-1}}$. Direct approaches would have required $O(\sqrt{N})$ communication, recursion instead allows us to compute that with $O(\sqrt[4]{N})$ complexity.

The base case $k = 0$. We have explained how to derive a secret-sharing of $\boldsymbol{v_{k+1}}$ given a secret-sharing of $\boldsymbol{v_k}$. It remains to describe how to deal with the base case, i.e. how to derive a secret-sharing of $\boldsymbol{v_0}$. Observe that $\boldsymbol{v_0}$ is a 2-dimensional unit vector, where β occupies the $\omega(-1)$-th position.

By using the same procedure described in the seed generation, for each position of $\boldsymbol{v_0}$, the parties can obtain pairs of elements in $\{0,1\}^\lambda$ of the form

$$\begin{array}{llll}\textbf{if } \boldsymbol{j \neq \omega(-1)} \textbf{ :} & \{a^j_{-1}, b^j_{-1}\}, & (b^j_{-1}, c^j_{-1}), & (a^j_{-1}, c^j_{-1}), \\ \textbf{if } \boldsymbol{j = \omega(-1)} \textbf{ :} & \{a^j_{-1}, d^j_{-1}\}, & (d^j_{-1}, b^j_{-1}), & (d^j_{-1}, c^j_{-1}).\end{array}$$

This time, we do not regard them as seeds anymore, but using the encoding map Enc, we convert them into random elements in the field $\mathbb{F}$. Observe that by combining the elements with the coefficients u^b_i, we can derive a secret-sharing of zero when $j \neq \omega(-1)$ and a secret-sharing of a random value $z \in \mathbb{F}$ when $j = \omega(-1)$. Obtaining a secret-sharing of $\boldsymbol{v_0}$ is now easy, we simply need to

[4] Whether we need to add or subtract is specified by the sign multipliers u^b_i.

multiply each secret-sharing we have just computed by $\beta \cdot z^{-1}$. The operation can be performed using $\mathcal{F}_{\text{MPC}}$.

Achieving active security. The protocol we just described allows the adversary to deviate in several points. In order to regain control on the execution, we relied on three different checks. Only the combination of all of them guarantees security.

The first issue we encounter is in the seed generation. An adversary corrupting both the second and third party can indeed discover all the seeds of the first party by always inputting different choice bits in the OTs. With this attack, the adversary would be able to retrieve β once the correction word is computed. So, we designed the first check to fail in these situations. Specifically, using $\mathcal{F}_{\text{auth-ROT}}$, the check recomputes the sum of the OT bits input by the receivers for every recursion level k. If the result is different from 1, the protocol aborts.

When the second or third party are corrupted, by cleverly choosing the choice bits of the OTs, the adversary can move the non-zero value β to a different position $\eta \neq \omega$. The second check makes sure that this attack fails with overwhelming probability. This is achieved by recomputing η from the OT inputs using the matrix C and $\mathcal{F}_{\text{auth-ROT}}$. The result is obliviously compared to ω using $\mathcal{F}_{\text{MPC}}$, the protocol aborts when they do not match.

The third check, inspired by [YWL+20], is probably the most important. Essentially, it draws a random N-dimensional vector $\boldsymbol{\chi} \in \mathbb{F}^N$ and checks that the result of the linear combination $\langle \boldsymbol{\chi}, \boldsymbol{v_h} \rangle$ coincides with $\chi_\omega \cdot \beta$. The procedure counteracts any malicious behaviour in the generation of the correction words. Moreover, in combination with the first check, it makes sure that, for every level k, there exists only one position for which the choice bits of the OTs coincide. On the other hand, the third check causes some leakage which is modelled in step 4 of $\mathcal{F}_{\text{3-DPF}}$ (see Figure 5). We prove the following in [AS21, Appendix D].

Theorem 3. *Let $N = \mathrm{dE}(h)$ be a double power of 2 and assume that $\mathbb{F}$ is a security-parameter-dependent prime field of cardinality p such that $|p - 2^\lambda|/2^\lambda$ is negligible in λ. Let $G_k : \{0,1\}^\lambda \longrightarrow \mathbb{F}^{\mathrm{dE}(k)}$ be a PRG for every $k \in [h]$ and let $H : \{0,1\}^\lambda \times \{0,1\}^* \longrightarrow \{0,1\}^\lambda$ be a tweakable correlation-robust hash function. Then the protocol $\Pi_{\text{3-DPF}}$ UC-realises $\mathcal{F}_{\text{3-DPF}}$ in the $(\mathcal{F}_{\text{MPC}}, \mathcal{F}_{\text{auth-ROT}}, \mathcal{F}_{\text{Rand}})$-hybrid model. Moreover, if all the parties are honest, $\Pi_{\text{3-DPF}}$ aborts with negligible probability.*

Complexity. The protocol $\Pi_{\text{3-DPF}}$ achieves low communication complexity. As a matter of fact, in [BCG+20], the authors described how to implement the 2-party DPF procedure of $\mathcal{F}_{\text{MPC}}$ with $O\big(\log(N) \cdot \mathsf{poly}(\lambda)\big)$ communication. We also observe that the seed generation needs $O(\sqrt{N})$ OTs. Hence, $\Pi_{\text{3-DPF}}$ has $O\big(\sqrt{N} \cdot \mathsf{poly}(\lambda)\big)$ communication complexity. A more detailed analysis of efficiency can be found in [AS21, Section 7.1].

6 Offline Phase

We can finally describe our Offline phase protocol Π_{Offline}, which achieves sublinear communication complexity. It can be broken down into 3 procedures: an initialisation procedure in which the MAC key α is generated and secret-shared, a triple generation procedure and an input mask generation procedure. The latter is used to produce, for every $j \in [n]$, random authenticated secret-shared elements $[\![a_j]\!]$ whose value is known only to party P_j. As for multiplication triples, input masks constitute an essential part of SPDZ as they are needed to provide the inputs of the computation.

The protocol Π_{Offline} closely resembles $\mathsf{PCG}_{\text{triple}}$. For this reason, we now give only an informal description of its operation and we refer to [AS21, Section 6.1] for a more thorough analysis. The protocol permits to generate N multiplication triples with $O\big(\sqrt{N} \cdot \mathsf{poly}(\lambda)\big)$ communication complexity and N input masks with $O\big(\log(N) \cdot \mathsf{poly}(\lambda)\big)$ communication complexity. The bottleneck of the triple generation is the 3-party DPF. If future research proves the existence of 3-party DPFs with logarithmic key size, we will probably be able to design multiparty triple generation protocols with logarithmic communication complexity.

Multiplication triples. The protocol uses the functionality $\mathcal{F}_{\text{3-DPF}}$ as a black box. During the initialisation procedure, each party P_i samples a random share α_i for the MAC key and inputs it in $\mathcal{F}_{\text{3-DPF}}$, fundamentally committing to its choice.

The multiplication triples are derived by executing the seed generation and the evaluation of $\mathsf{PCG}_{\text{triple}}$ inside $\mathcal{F}_{\text{3-DPF}}$: at the very beginning, each party P_i samples random special positions $\boldsymbol{\omega_i^r}, \boldsymbol{\eta_i^r} \in [N]^t$ and random non-zero elements $\boldsymbol{\beta_i^r}, \boldsymbol{\gamma_i^r} \in \mathbb{F}^t$ for every $r \in [c]$. These values are input in $\mathcal{F}_{\text{3-DPF}}$. Using $\mathsf{2\text{-}DPF}$ and $\mathsf{3\text{-}DPF}$, it is then possible for P_i to obtain $\boldsymbol{u_i}, \boldsymbol{\widetilde{u}_i}, \boldsymbol{v_i}, \boldsymbol{\widetilde{v}_i}, \boldsymbol{w_i}$ and $\boldsymbol{\widetilde{w}_i}$. Finally, by sampling a random $\boldsymbol{a} \in R^c$ using $\mathcal{F}_{\text{Rand}}$, the parties can compute the final output, i.e. random authenticated secret-shared elements $[\![x]\!], [\![y]\!], [\![z]\!] \in R$ such that $z = x \cdot y$. We recall that $R = \mathbb{F}[X]/\big(F(X)\big)$ where $F(X)$ is a degree-N polynomial. If $F(X)$ has N distinct roots in $\mathbb{F}$, the tuple $\big([\![x]\!], [\![y]\!], [\![z]\!]\big)$ can be converted into N random multiplication triples by evaluating the shares[5] over the roots of $F(X)$.

Input Masks. The generation of inputs masks is very similar but simpler. At the beginning, the party P_j to which the masks are addressed samples random special positions $\boldsymbol{\omega^r} \in [N]^t$ and random non-zero elements $\boldsymbol{\beta^r} \in \mathbb{F}^t$ for every $r \in [c]$, inputting them in $\mathcal{F}_{\text{3-DPF}}$. These values will be used to define the sparse polynomial

$$u^r(X) \leftarrow \sum_{l \in [t]} \beta^r[l] \cdot X^{\omega^r[l]}$$

Later on, for every $i \neq j$, P_i and P_j can obtain a secret-sharing of $\alpha_i \cdot u^r(X)$ using $\mathsf{2\text{-}DPF}$. Finally, by sampling a random $\boldsymbol{a} \in R^c$ using $\mathcal{F}_{\text{Rand}}$ and relying on the hardness of Module LPN, the shares can be converted into a random authenticated secret-shared element $[\![x]\!] \in R$. Since P_j knows $u^r(X)$ for every $r \in [c]$, it can also learn x. As a last operation, the shares are rerandomised using a PRG. Observe that from $[\![x]\!]$, we can derive N masks using the same trick described for the multiplication triples.

[5] The shares are elements of R and therefore polynomials

Leakage. The main difference between Π_{Offline} and $\mathsf{PCG}_{\text{triple}}$ is that every execution of 2-DPF and 3-DPF has additional leakage. At first, it might seem that the main issue arises when the last two players P_j and P_k of the 3-party DPF procedure are corrupted. In such cases, the special positions $(\boldsymbol{\omega}_{\boldsymbol{j}}^{\boldsymbol{r}} \boxplus \boldsymbol{\eta}_{\boldsymbol{k}}^{\boldsymbol{s}})_{r,s\in[c]}$ are indeed revealed to the adversary. Notice, however, that this is no problem at all, as the leaked values were chosen by the adversary itself at the beginning of the protocol.

Regarding the remaining leakage, observe that when the adversary tries the guess any non-zero element during 3-DPF, the procedure aborts with overwhelming probability. Indeed, the non-zero values are uniformly distributed in $\mathbb{F}^{\times}$, assuming that at least one party involved in 3-DPF is honest. Moreover, any leakage concerning the special positions is absorbed by the hardness of module-LPN with static leakage. For a formal discussion, see [AS21, Appendix E].

6.1 Concrete Efficiency

Table 1. Estimated seed size for producing N triples with the 3-party PCG over a 128-bit field, with 80-bit computational security.

N	2^{20}			2^{24}			2^{28}		
c	2	4	8	2	4	8	2	4	8
$w = ct$	96	40	32	96	40	32	96	40	32
Comm. (MB)	308	114	109	1120	417	418	4329	1641	1650
Stretch	0.16	0.44	0.46	0.72	1.93	1.92	2.98	7.85	7.81

In Table 1 , we estimate the concrete communication cost of our protocol, for several sets of parameters with $n = 3$ parties and 80-bit computational security. For further details and parameters for 128-bit security, we refer to the full version [AS21, Section 7.2]. The "stretch" of the protocol is defined as the ratio of the size of the uncompressed triples ($3N$ field elements) and the total communication cost. We see that, when producing around a million triples ($N = 2^{20}$), the stretch is still less than 1, meaning that the PCG does not achieve a good compression factor. Nevertheless, even at this level, we do achieve a protocol for generating triples with much lower communication than methods based on alternative techniques. For instance, using the Overdrive protocol based on homomorphic encryption [KPR18] requires almost 2GB of communication to generate the same number of triples, which is more than 10x our protocol.

When moving to larger batch sizes, the stretch improves, going up to almost 8x with $N = 2^{28}$ and $c \in \{4, 8\}$. This gives a strong saving in communication, but comes with larger computational costs due to the higher degree polynomial operations needed for arithmetic in the ring R. In practice, since these operations cost $O(N \log N)$, it seems likely that the smaller sizes of $N \leq 2^{24}$ will be preferable.

Acknowledgements

We thank the anonymous reviewers for their feedback, which helped to improve the paper.

References

AS21. Damiano Abram and Peter Scholl. Low-Communication Multiparty Triple Generation for SPDZ from Ring-LPN. Cryptology ePrint Archive, 2021, 2021.

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 *ACM CCS 2019*. ACM Press, November 2019.

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 *CRYPTO 2019, Part III*, LNCS. Springer, Heidelberg, August 2019.

BCG$^+$20. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from ring-LPN. In *CRYPTO 2020, Part II*, LNCS. Springer, Heidelberg, August 2020.

Bea92. Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO'91*, LNCS. Springer, Heidelberg, August 1992.

BGI15. Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *EUROCRYPT 2015, Part II*, LNCS. Springer, Heidelberg, April 2015.

BGI16. Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *ACM CCS 2016*. ACM Press, October 2016.

BKKO20. Paul Bunn, Jonathan Katz, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient 3-party distributed ORAM. In *SCN 20*, LNCS. Springer, Heidelberg, September 2020.

DKL$^+$13. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS 2013*, LNCS. Springer, Heidelberg, September 2013.

DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO 2012*, LNCS. Springer, Heidelberg, August 2012.

GI14. Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *EUROCRYPT 2014*, LNCS. Springer, Heidelberg, May 2014.

KOS16. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In *ACM CCS 2016*. ACM Press, October 2016.

KPR18. Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT 2018, Part III*, LNCS. Springer, Heidelberg, April / May 2018.

NNOB12. Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *CRYPTO 2012*, LNCS. Springer, Heidelberg, August 2012.

SV14. N. P. Smart and F. Vercauteren. Fully homomorphic simd operations. *Des. Codes Cryptography*, 71(1):57–81, April 2014.

YWL$^+$20. Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In *ACM CCS 20*. ACM Press, November 2020.