Efficient Pseudorandom Correlation Generators from Ring-LPN[*]

Elette Boyle [†] Geoffroy Couteau [‡] Niv Gilboa [§] Yuval Ishai [¶] Lisa Kohl [‖]
Peter Scholl [**]

August 10, 2022

Abstract

Secure multiparty computation can often utilize a trusted source of correlated randomness to achieve better efficiency. A recent line of work, initiated by Boyle et al. (CCS 2018, Crypto 2019), showed how useful forms of correlated randomness can be generated using a cheap, one-time interaction, followed by only "silent" local computation. This is achieved via a *pseudorandom correlation generator* (PCG), a deterministic function that stretches short correlated seeds into long instances of a target correlation. Previous works constructed concretely efficient PCGs for simple but useful correlations, including random oblivious transfer and vector-OLE, together with efficient protocols to distribute the PCG seed generation. Most of these constructions were based on variants of the Learning Parity with Noise (LPN) assumption. PCGs for other useful correlations had poor asymptotic and concrete efficiency.

In this work, we design a new class of efficient PCGs based on different flavors of the *ring-LPN* assumption. Our new PCGs can generate OLE correlations, authenticated multiplication triples, matrix product correlations, and other types of useful correlations over large fields. These PCGs are more efficient by orders of magnitude than the previous constructions and can be used to improve the preprocessing phase of many existing MPC protocols.

[*]Full version of a paper published at CRYPTO 2020, `https://doi.org/10.1007/978-3-030-56880-1_14`. For a list of changes, see Section 1.2.

[†]IDC Herzliya and NTT Research, eboyle@alum.mit.edu

[‡]IRIF, couteau@irif.fr

[§]Ben-Gurion University, niv.gilboa@gmail.com

[¶]Technion, yuvali@cs.technion.ac.il

[‖]Cryptology Group, CWI Amsterdam, lisa.kohl@cwi.nl

[**]Aarhus University, peter.scholl@cs.au.dk

Contents

1 Introduction

Correlated secret randomness is a commonly used resource for secure multi-party computation (MPC) protocols. Indeed, simple kinds of correlations enable lightweight MPC protocols even when there is no honest majority. For instance, an *oblivious transfer* (OT) correlation supports MPC for Boolean circuits [GMW87,Kil88,IPS08,NNOB12], while oblivious linear-function evaluation[1] (OLE), an arithmetic variant of OT, supports MPC for arithmetic circuits [NP99,IPS09]. Other useful types of correlations include multiplication triples [Bea91] and truth-table correlations [IKM+13,DNNR17,Cou19]. Finally, *authenticated* multiplication triples serve as a powerful resource for achieving security against malicious parties [BDOZ11,DPSZ12].

A common paradigm in modern MPC protocols is to utilize the above kinds of correlations in the following way. In a preprocessing phase, before the inputs are known, the parties use an *offline protocol* to generate many instances of the correlation. These instances are then consumed by an *online protocol* to securely compute a function of the secret inputs. This approach is appealing because of the high efficiency of the online protocol. Indeed, with the above simple correlations, the online communication and computation costs are comparable to the size of the circuit being evaluated. The price one pays for the fast online protocol is a much slower and higher-bandwidth offline protocol. Even simple types of correlated randomness are expensive to generate in a secure way. This high cost becomes even higher when aiming for security against malicious parties.

Recently, a promising approach for instantiating the preprocessing phase of MPC protocols was suggested in [BCGI18,BCG+19b], relying on a new primitive called a *pseudorandom correlation generator* (PCG). Consider a target two-party correlation $\mathcal{C}$, typically consisting of many independent instances of a simple correlation as above. A PCG for $\mathcal{C}$ consists of two algorithms: $\mathsf{Gen}(1^\lambda)$, which given a security parameter λ generates a pair of short, correlated seeds $(\mathsf{k}_0, \mathsf{k}_1)$, and $\mathsf{Expand}(\mathsf{k}_\sigma)$, which deterministically stretches a seed k_σ to a long output R_σ. The intuitive security requirement is that the joint outputs (R_0, R_1) of the above process cannot be distinguished from $\mathcal{C}$ not only by an outsider, but also by an insider who learns one of the two seeds. PCGs naturally lead to protocols with an appealing *silent preprocessing* feature, by breaking the offline phase into two parts:

1. Setup. The parties run a secure protocol to distribute the seed generation of Gen. Since Gen has low computational cost and short outputs, this protocol only involves a small amount of communication, much smaller than the output of $\mathcal{C}$. Each party stores its own short seed k_σ for later use.

2. Silent expansion. Shortly before the online phase, the parties use Expand to generate long pseudorandom correlated strings (R_0, R_1) to be consumed by the online protocol. This part is referred to as *silent*, since it involves no communication.

Beyond the potential improvement in the total offline communication and computation, this blueprint has two additional advantages. First, it can substantially reduce the *storage* cost of correlated randomness by enabling efficient compression. Indeed, the parties can afford to generate and store many correlated seeds, possibly with different sets of parties, and expand them just before they are needed. Second, the cost of protecting the offline protocol against malicious parties is "amortized away," since it is only the small setup part that needs to be protected. A malicious execution of Expand is harmless.

The work of Boyle et al. [BCG+19b] constructed efficient PCGs for several kinds of useful correlations based on different assumptions that include variants of Learning Parity with Noise (LPN) [BFKL94] and Learning With Errors (LWE) [Reg05]. While the LPN-based PCG for OT

[1]An OLE correlation over a finite field $\mathbb{F}$ is a two-party correlation (r_0, r_1) where $r_0 = (a, b)$ is uniform over $\mathbb{F}^2$ and $r_1 = (x, ax + b)$ for $x \in_R \mathbb{F}$.

from [BCG+19b] has very good concrete efficiency, making it a practically appealing approach for generating many OTs [BCG+19a], this is not the case for other useful correlations such as OLE or authenticated multiplication triples. For these correlations, two different constructions were proposed in [BCG+19a]. Both are "practically feasible" but quite inefficient. In the first construction, based on homomorphic secret sharing from ring-LWE [BGI16a, BGI17, BCG+17, BKS19], the seed expansion can be at most quadratic due to the use of a pseudorandom generator with algebraic degree 2. In concrete terms, the seeds are several GBs long and can only be expanded by around 6x, giving far too much overhead for most applications. Their second construction is based directly on LPN, and has computational cost of at least $\Omega(N^2)$ for output length N, which is impractical for large N.

1.1 Our contributions

In this work, we present efficient new PCG constructions for several widely used correlations for which previous techniques had poor concrete efficiency.

Silent OLE and multiplication triple generation. Our main construction gives the first concretely efficient PCG for OLE over big finite fields $\mathbb{F}$. This PCG is based on a variant of the *ring-LPN* assumption over $\mathbb{F}$, makes a black-box use of $\mathbb{F}$, and has $\mathsf{poly}(\lambda) \cdot \log N$ seed size and $\mathsf{poly}(\lambda) \cdot \tilde{O}(N)$ computational cost for expanding the seeds into N instances of OLE. This PCG gives both an asymptotic and concrete improvement over the LPN-based construction from [BCG+19b].

We also show how to modify our PCG for OLE to produce multiplication triples and authenticated triples, used in maliciously secure MPC protocols like SPDZ [DPSZ12]. This incurs an extra overhead of only around a factor of two in seed size, seed generation, and silent expansion time. Finally, we extend the main construction to other types of useful correlations, including matrix products and circuit-dependent correlations.

Technically, one of our main innovations here is showing how to avoid the $\Omega(N^2)$ blowup from the previous LPN-based PCG for OLE from [BCG+19b]. Our method of doing this requires switching from unstructured LPN to ring-LPN over a certain kind of polynomial rings. This is analogous to early fully homomorphic encryption schemes, where switching from a construction based on LWE [BV11a] to one based on ring-LWE [BV11b] reduced the ciphertext expansion in multiplication from quadratic to linear. A key difference between LWE-based constructions and LPN-based constructions over a big field $\mathbb{F}$ is the noise distribution: Gaussian in the former and low Hamming weight noise in the latter. With LPN-style noise distribution more care is needed, and some natural PCG candidates based on Reed-Solomon codes can indeed be broken.

Concrete efficiency. Our PCGs have attractive concrete efficiency features. To give a couple of data points, in the case of OLE the parties can store a pair of seeds of size 1.25MB each, and expand them on demand to produce over a million OLEs (of size 32MB, 26x larger than the seeds) in $\mathbb{Z}_q$, where q is the product of two 62-bit primes,[2] with 128-bit security. When running on a single core of a modern laptop, we estimate this takes under 10 seconds, resulting in a throughput of over 100 thousand OLEs per second. To produce authenticated triples instead of OLE, the expansion cost roughly doubles, giving 50 thousand triples per second, while the seed size increases to 2.6MB. For comparison, estimates from [BCG+19b] for their PCG for producing authenticated triples gave a throughput of up to 7 thousand per second, but this was only possible when generating an enormous batch of 17GB worth of triples, with 3GB seeds. See below for comparison with non-silent correlation generation techniques.

[2]Our construction works in any sufficiently large finite field, or modulus that is a product of primes via the CRT. We estimated costs with a product of two primes due to better software support.

Efficient setup protocols. Recall that to avoid a trusted setup, one typically needs a setup protocol to securely distribute the PCG seed generation. We present concretely efficient setup protocols for OLE and authenticated triples, with both semi-honest and malicious security. The protocols make black-box use of lightweight cryptographic primitives, as well as of generic MPC protocols for performing binary and arithmetic computations on secret-shared values.

In practice, our PCGs and setup protocols can be used in a *bootstrapping mode*, where a portion of the PCG outputs are reserved to be used as correlated randomness for the setup procedure of the next PCG seeds. This means that the vast majority of the setup cost is amortized away over multiple instances. Concretely, we estimate that when bootstrapped in this way, the setup phase for a PCG of one million authenticated triples requires only around 4.2MB of communication per party, to produce 32MB worth of triples. The initial setup protocol for the first PCG (before bootstrapping can take place) requires around 25000 authenticated triples, plus some additional correlated randomness (OT and VOLE). This should be feasible to produce in under a minute (although with high communication cost) using standard protocols such as MASCOT [KOS16] or Overdrive [KPR18], and previous PCG protocols for OT and VOLE with malicious security [BCG+19a].

Compared with non-silent secure correlation generation protocols, we expect the overall *computational* cost of our approach to be comparable with state-of-the-art protocols based on homomorphic encryption [KPR18, JVC18, HIMV19], but with much lower *communication* costs. For instance, in the case of authenticated multiplication triples, the Overdrive protocol [KPR18] can produce around 30 thousand triples per second with malicious security. This is similar to our PCG expansion phase (modulo different hardware, environment, and so on), with the significant difference that Overdrive requires almost 2GB of communication to produce the triples. In comparison, our amortized 4.2MB communication complexity is over two orders of magnitude smaller, with the additional benefit that our short correlated seeds can be easily stored for on-demand silent expansion.

Extension to other correlations and multiple parties. Beyond multiplication triples, it can be useful to have more general "degree-two" correlations, such as inner-product triples, matrix-multiplication triples, or circuit-dependent multiplication triples [Cou19, BNO19, BGI19]. We use our PCG for OLE to obtain PCGs for these kinds of correlations, by exploiting a special "programmability" feature that enables reusing the same PCG output in multiple instances [BCG+19b]. This gives us a way to produce many independent instances of any degree-two correlation (a vast generalization of OLE and multiplication triples), with seed size that grows sublinearly with the total number of instances. Useful special cases include the types of correlations mentioned above. This construction has a bigger overhead than our PCG for OLE, and in practice seems mainly suited for small correlations such as low-dimensional matrix products. However, these can still be useful in larger computations which involve a lot of linear algebra or other repeated sub-computations.

We can also use same programmability feature of our 2-party PCGs to extend them to the *multi-party setting.* This yields practical multi-party PCGs for multiplication triples that enable an online passively-secure MPC protocol for arithmetic circuits whose cost scales linearly (rather than quadratically) with the number of parties. This transformation to the multi-party case, which originates from [BCG+19b], does not scale well to correlations with degree higher than 2. As a result, we do not get a multi-party PCG for authenticated triples (a degree-three correlation) with the same level of efficiency.

Security of ring-LPN. Our constructions rely on variants of the ring-LPN assumption over non-binary fields. Binary ring-LPN [HKL+12] is a fairly standard assumption that withstood a significant amount of cryptanalysis. However, since we also use relatively unexplored variants over different rings, we give a thorough survey of known attacks, and analyze the best strategies that apply to our setting. We find that there are only one or two additional attack possibilities from the additional structure we introduce, and these are easily countered with a small increase in the number of errors.

More precisely, settling for a PCG that generates a single OLE instance over a large ring of degree-N polynomials, our construction can be based on a conservative variant of ring-LPN where the modulus is irreducible. A big ring-OLE correlation can then be converted into N independent instances of standard OLE by communicating $O(N)$ field elements. For generating *silent* OLE over $\mathbb{F}_p$, we instead rely on a variant of ring-LPN where the modulus splits completely into N linear factors. In practice, this requires using larger parameters and increases the cost of our protocols by around a factor of two, compared with irreducible ring-LPN.

1.2 Changes Since Publication at CRYPTO 2020

This article has undergone a major revision since its original publication. As well as various minor improvements and clarifications, there are the following major changes.

- The maliciously secure DPF setup protocol in Section 5.3 has changed, since the original security proof was flawed, as pointed out by Damiano Abram. The protocol now uses a different consistency check to fix the issue.
- The DPF setup protocol has also been generalized to work over any (sufficiently large) finite field, it no longer uses the random oracle model, and the $\mathcal{F}_{\mathsf{c\text{-}SUV}}$ functionality it realizes no longer allows the adversary to guess β.
- The cost analysis has been updated to reflect the new protocol. The resulting efficiency estimates have not changed significantly.
- Some of the security analysis of the ring-LPN assumption has been revised due to minor bugs. We point out that much of this analysis is based on cost estimates for attacks that were recently shown in [LWYY22] to be overly conservative. We did not adjust our parameters to reflect [LWYY22], however, according to their cost models (combined with our optimizations for exploiting the structure of cyclotomic ring-LPN), our parameter sets for 80-bit security have between 92–112 bits of security, while our 128-bit parameter sets have 133–171 bits of security. It therefore seems that our parameters have a comfortable security margin.

1.3 Technical Overview

Construction from [BCG+19b]. Before describing our PCG for OLE, it is instructive to recall the PCG for general degree-two correlations by Boyle et al. [BCG+19b], based on LPN. The goal is to build a PCG for the correlation which gives each party a random vector $\vec{x}_i$, together with an additive secret share of the tensor product $\vec{x}_0 \otimes \vec{x}_1$. They used the dual form of LPN over a ring $\mathbb{Z}_p$, which states that the distribution

$$\left\{ H, H \cdot \vec{e} \,\middle|\, H \xleftarrow{\$} \mathbb{Z}_p^{m \times n}, \vec{e} \xleftarrow{\$} \mathbb{Z}_p^n \text{ s.t. } \mathrm{wt}(\vec{e}) = t \right\}$$

is computationally indistinguishable from uniform, where $\vec{e}$ is a sparse random vector with only t non-zero coordinates, for some $t \ll n$, and $m < n$.

The idea of the construction is that the setup algorithm gives each party a random sparse $\vec{e}_0$ or $\vec{e}_1$, and computes the tensor product $\vec{e}_0 \otimes \vec{e}_1$, which has at most t^2 non-zero coordinates. This product is then distributed to the parties via function secret sharing (FSS), by generating a pair of FSS keys for the function that outputs each entry of the product on its respective inputs from 1 to n^2. This function can be written as a sum of t^2 point functions, allowing practical FSS schemes based on distributed point functions [GI14, BGI15, BGI16b]. Note that unlike the case of PCGs for OT or Vector-OLE [BCG+19a, SGRR19], here we cannot replace FSS by the simpler punctured PRF primitive.

Given shares of $\vec{e}_0 \otimes \vec{e}_1$ and either $\vec{e}_0$ or $\vec{e}_1$, the parties expand these using LPN, computing:

$$\vec{x}_0 = H \cdot \vec{e}_0, \quad \vec{x}_1 = H \cdot \vec{e}_1, \quad \vec{z} = (H \cdot \vec{e}_0) \otimes (H \cdot \vec{e}_1) = (H \otimes H) \cdot (\vec{e}_0 \otimes \vec{e}_1)$$

where $\vec{x}_i$ is computed by party P_i, while $\vec{z}$ is computed in secret-shared form, using the shares of $\vec{e}_0 \otimes \vec{e}_1$ and the formula on the right-hand side.

Notice that both $\vec{x}_0$ and $\vec{x}_1$ are pseudorandom under LPN, which gives the desired correlation.

Optimizations and additional applications. Boyle et al. state the computational complexity of the above as $O(n^4)$ operations, due to the tensor product of H with itself. We observe that the value of $(H \cdot \vec{e}_0) \otimes (H \cdot \vec{e}_1)$ can be read directly from $H \cdot (\vec{e}_0 \cdot \vec{e}_1^\intercal) \cdot H^\intercal$, which requires much less computation and can be made even more efficient if H is a structured matrix, reducing the computational complexity to $\tilde{O}(n^2)$. We also describe two variants of the PCG which allow producing large matrix multiplication correlations with different parameter tradeoffs. While much less practical than our main constructions, we present these in Section 10 for completeness.

A first attempt. The problem with the above construction is that it produces an entire tensor product correlation, which inherently requires $\Omega(n^2)$ computation. Even if we only want to compute the diagonal entries of the tensor product output (that is, n OLEs), we do not see a way to do this any more efficiently.

The bottleneck is computing the n^2 entries of $\vec{e}_0 \otimes \vec{e}_1$ to obtain $\vec{z}$. A natural idea to reduce the computational complexity is to replace $\vec{e}_0$ and $\vec{e}_1$ by degree-n sparse polynomials e_0 and e_1, and $\vec{x}_0$ and $\vec{x}_1$ by, say, $n/2$ evaluations of e_0 and e_1 respectively. Then, $\vec{z}$ is computable in quasilinear time as evaluations of the polynomial product $e_0 \cdot e_1$ of degree $2n$. This approach does not give a secure PCG candidate, though, because $\vec{x}_0$ and $\vec{x}_1$ can be efficiently distinguished from random using algebraic decoding techniques.

An efficient PCG for OLE. Our actual approach follows the same idea, but with the underlying code chosen more carefully. Namely, let $R_p = \mathbb{Z}_p[X]/F(X)$ for some degree N polynomial $F(X)$, and let e, f be two sparse polynomials in R_p. For a random polynomial $a \in R_p$, the pair

$$(a, a \cdot e + f \mod F(X))$$

is pseudorandom under the *ring-LPN* assumption [HKL+12].

Now, given two pairs of sparse polynomials (e_0, e_1) and (f_0, f_1), each product $e_i \cdot f_j$ (without reduction modulo F) has degree $< 2N$ and only t^2 non-zero coefficients. These can again be distributed to two parties using FSS, but this time the expanded FSS outputs can be computed in *linear time* in N, instead of quadratic, since the domain size of the function being shared is only $2N$.

Given shares of $e_i \cdot f_j$, similarly to the LPN case, the parties compute expanded outputs by defining

$$x_0 = a \cdot e_0 + f_0, \quad x_1 = a \cdot e_1 + f_1, \quad z = ((1, a) \otimes (1, a)) \cdot ((e_0, f_0) \otimes (e_1, f_1))$$

The main difference here is that each tensor product is only of length 2, and can be computed in $\tilde{O}(N)$ time using fast polynomial multiplication algorithms.

This gives a PCG that compresses a single OLE over the ring R_p. To obtain a PCG for OLE over $\mathbb{Z}_p$, we again take inspiration from the fully homomorphic encryption literature, by using ciphertext-packing techniques [SV14]. We can carefully choose p and $F(X)$ such that $F(X)$ splits into N distinct, linear factors modulo p. Then R_p is isomorphic to N copies of $\mathbb{Z}_p$, and we can immediately convert a random OLE over R_p into N random OLEs over $\mathbb{Z}_p$. This works particularly well with cyclotomic rings as used in ring-LWE [LPR13], where we can e.g. use N a power of two and easily exploit FFTs for polynomial arithmetic.

Extending to authenticated multiplication triples. We show that our construction extends from OLE to authenticated multiplication triples, as used in the SPDZ protocol for maliciously secure MPC [DPSZ12, DKL$^+$13]. This follows from a simple trick, where we modify the FSS scheme to additionally multiply its outputs by a random MAC key $\alpha \in \mathbb{Z}_p$. Since this preserves sparsity of the underlying shared vector, it adds only at most a factor of two overhead on top of the basic scheme.

Distributed setup. We focus on the case of OLE correlations over R_p (the setup for authenticated triples is very similar). Recall that the seed of the PCG for OLE consists of t-sparse degree-N "error" polynomials e_0, e_1 and f_0, f_1, and FSS keys for secret-shares of the products $e_i \cdot f_j$, each represented as a coefficient vector via the sum of t^2 point functions $f_{\alpha,\beta}\colon [2N] \to \mathbb{Z}_p$. Each point function corresponds to a single monomial product from e_i and f_j. The index $\alpha \in [2N]$ of the nonzero position is the *sum* of the corresponding nonzero indices, and the payload $\beta \in \mathbb{Z}_p$ is the *product* of the corresponding payloads in e_i and f_j.

In the semi-honest setting, secure computation of this PCG generation procedure can be attained directly, using generic 2-PC for simple operations on the α and β values, as well as black-box use of a protocol for secure computation of FSS key generation, such as the efficient protocol of Doerner and shelat [Ds17].

For the malicious setting, we would wish to mimic the same protocol structure with underlying 2-PC components replaced with maliciously secure counterparts. The simple 2-PCs on α, β can be converted to malicious security with relatively minor overhead. The problem is the FSS key generation, for which efficient maliciously secure protocols currently do not exist. Generic 2-PC of the FSS key generation functionality would require expensive secure evaluations of crytographic pseudorandom generators (PRG). The semi-honest protocol of [Ds17] is black-box in a PRG; but, precisely this fact makes it difficult to ensure consistency between different steps in the face of a malicious party.

Note that this is similar to the problem that Boyle et al. faced in [BCG$^+$19a] for silent OT generation, but their setting was conceptually simpler: There, one party always knew the position α of the non-zero value of the distributed point function (indeed, for their purpose the simpler building block of a puncturable pseudorandom function sufficed). Further, they did not have to assume any correlation between path values, whereas in our setting we require that the parties behave consistently regarding the path positions *and* payloads across several instances.

In this work we show how to extend the approach of [BCG$^+$19a] to the context of distributed point functions, further addressing the mentioned issues.

Our protocol realizes a PCG-type functionality for a scaled unit vector[3] with leakage: Given authenticated values for the location of the non-zero position $\alpha \in [0..N)$ and the non-zero payload $\beta \in \mathbb{Z}_p$, the functionality allows a corrupt party to choose its output vector $\vec{y} \in \mathbb{Z}_p^N$ and delivers to the honest party the correct corresponding output $\vec{y} - (0, \ldots, \beta, \ldots, 0)$, where β is in the α-th position. The leakage on α can be captured by allowing the adversary a predicate guess on α.[4] In the setting of noise generation for (ring-)LPN, as is the case for our PCG constructions (and likely future constructions), such leakage is tolerable as, intuitively, this can be accounted for by slightly increasing the noise rate. Indeed, we prove that this functionality suffices to implement a protocol securely realizing PCG functionalities, such as the corruptible functionality for OLE

[3]Note that this corresponds to a distributed point function where we do not require the key setup on its own to be secure, but only require the protocol to securely implement the FSS functionality including expansion, as this suffices for using PCGs in the context of secure computation (see also [BCG$^+$19a]).

[4]In fact, the leakage can be characterized by predicates corresponding to bit-matching with wildcards.

and authenticated multiplication triples, based on a variant of the ring-LPN assumption that allows small amount of leakage (only 1 bit on average).

Extensions. A downside of the above construction, compared with the one from LPN, is that it is restricted to multiplication triples or OLE. It can be useful to obtain other degree-two correlations such as matrix multiplication triples, which allow multiplying two secret matrices with only $O(n^2)$ communication, instead of $O(n^3)$ from naively using individual triples. Another technique is preprocessing multiplications in a way that depends on the structure of the circuit, which allows reducing the online cost of 2-PC down to communicating just one field element per party, instead of two from multiplication triples [Bea92, DNNR17, Cou19, BNO19, BGI19]. This type of circuit-dependent preprocessing can also be expressed as a degree two correlation.

Our PCG for OLE satisfies a useful "programmability" feature, introduced by Boyle et al. [BCG+19b], allowing certain parts of the PCG output to be reused across multiple instances. This is simply due to the fact that we can reuse the polynomials e_0, e_1 or f_0, f_1 in the PCG, without harming security. This allows us to extend the PCG to build more general correlations, by using multiple programmed instances to perform every multiplication in the general correlation.

We in fact present a more general construction, which, loosely speaking, achieves the following. Given a programmable PCG for some bilinear correlation g, let f be another bilinear correlation that is computable using linear combinations of outputs of g applied to its input. Then, we can construct a PCG for f using several copies of the PCG for g, where the number of instances is given by the complexity of f written as a function of g. This gives a general way of combining PCGs to obtain correlations of increasing complexity, while allowing for different complexity tradeoffs by varying the "base" bilinear correlation f.

Multi-party PCGs. As discussed earlier, the programmability feature also immediately allows us to extend our PCGs for OLE and degree-two correlations to the multi-party setting, using the construction from [BCG+19b]. This does not apply to the PCG for authenticated multiplication triples; in Section 7.6, we sketch a possible alternative solution based on three-party distributed point functions, but these are much less efficient than the two-party setting.

Security analysis of ring-LPN. We use the ring $R_p = \mathbb{Z}_p[X]/F(X)$, for some degree N polynomial $F(X)$. There are two main ring-LPN variants we consider, depending on how the parameters are instantiated. The more conservative is when $F(X)$ is either *irreducible* in R_p (hence, R_p is isomorphic to a finite field), or at least when $F(X)$ has only very few low-degree factors, so R_p has a large subring that is a field. This type of instantiation is similar to previous recommendations for ring-LPN [HKL+12, GJL15] and post-quantum encryption schemes from quasi-cyclic codes [MBD+18]. The best known attacks are to solve the underlying syndrome decoding problem, and the additional ring structure does not seem to give much advantage. One exception is when a very large number of samples are available, when the ring structure can in some cases be exploited [BL12]. This does not apply to our setting, however, since our constructions only rely on ring-LPN with one sample[5].

The second variant, which is needed for silent OLE in $\mathbb{F}_p$, is when $F(X)$ splits modulo p into many distinct factors of low degree. Here, the main attack vector that needs to be considered is that if f_i is some degree-d factor of $F(X)$, then reducing a ring-LPN instance modulo f_i gives a new instance in smaller dimension d, albeit with a different noise distribution. The best case for the adversary is when f_i is of the form $X^d + c_i$, when this reduction does not increase the Hamming weight of the noise (although, the corresponding error rate goes up). If such sparse factors exist, then, we must also ensure that the underlying ring-LPN instance in dimension d, with new noise weight, is hard to solve.

[5]Or, two samples if security is based on ring-LPN with a uniform (not sparse) secret.

One way to counter this attack is to choose $F(X)$ to be a product of N random linear factors, ensuring that any factors of F an adversary can find are likely to be very dense. However, to improve computational efficiency, it is better to use a cyclotomic polynomial such as $F(X) = X^N + 1$ with N a power of two, as is common in the ring-LWE setting. In this case, there are many sparse factors of the form $X^{2^i} + c_i$ which can be exploited, and we must take these into account when choosing parameters. The main advantage of performing this reduction is the vector operations in attacks such as information-set decoding become cheaper, since they are all in a smaller dimension. This only has a small overall effect on attack complexity, though, since these algorithms are all exponential in the noise weight. Therefore, to counter the attack, it suffices to ensure there are enough noisy coordinates in a reduced instance, which requires only a small increase in noise weight.

Note that for $p = 2$, this strategy was also considered in Lapin [HKL+12], and it was later shown that an optimized version of this over $\mathbb{F}_2$ reduces security of some Lapin parameter sets by ≈ 10 bits [GJL15]. Our analysis over $\mathbb{F}_p$ is roughly consistent with this.

2 Preliminaries

2.1 Notation

We let λ denote a security parameter, and use the standard definitions of negligible functions, computational indistinguishability (with respect to nonuniform distinguishers), and pseudorandom generators. We use $[0..n)$ to denote the index set $\{0, \cdots, n-1\}$, as well as $[0..n] = \{0, \ldots, n\}$ and $[n] = \{1, \ldots, n\}$.

Vectors, outer sum and outer product. We use column vectors by default. For two vectors $\vec{u} = (u_1, \ldots, u_t), \vec{v} = (v_1, \ldots, v_t) \in R^t$, for some ring R, we write $\vec{u} \boxplus \vec{v}$ to mean the *outer sum* given by the length t^2 vector $(u_i + v_j)_{i\in[t], j\in[t]}$. Similarly, we define the flattened outer product (or tensor product) to be $\vec{u} \otimes \vec{v} = (u_i \cdot v_j)_{i\in[t], j\in[t]}$, that is, the vector $(v_1 \cdot \vec{u}, \ldots, v_n \cdot \vec{u})$. We denote the inner product of two vectors by $\langle \vec{u}, \vec{v} \rangle$.

2.2 Function Secret Sharing

Function secret sharing [BGI15, BGI16b] (FSS) is a succinct secret sharing of functions. More concretely, an FSS scheme randomly splits a secret function $f : I \to \mathbb{G}$, where $\mathbb{G}$ is some Abelian group, into two or more functions f_i, each represented by a key K_i, such that: (1) the sum of all function shares f_i is equal to f (namely, $\sum_i f_i(x) = f(x)$ for every input $x \in I$), and (2) each subset of the keys K_i hides f. In this work we will use 2-party FSS that we formalize below.

Definition 2.1 (Function Secret Sharing) *Let $\mathcal{C} = \{f : I \to \mathbb{G}\}$ be a class of function descriptions, where the description of each f specifies the input domain I and an Abelian group $(\mathbb{G}, +)$ as the output domain. A (2-party)* function secret sharing *(FSS) scheme for $\mathcal{C}$ is a pair of algorithms* $\mathsf{FSS} = (\mathsf{FSS.Gen}, \mathsf{FSS.Eval})$ *with the following syntax:*

- $\mathsf{FSS.Gen}(1^\lambda, f)$ *is a PPT algorithm that given security parameter λ and description of $f \in \mathcal{C}$ outputs a pair of keys (K_0, K_1). We assume that the keys specify I and $\mathbb{G}$.*
- $\mathsf{FSS.Eval}(b, K_b, x)$ *is a polynomial-time algorithm that, given a key K_b for party $b \in \{0, 1\}$, and an input $x \in I$, outputs a group element $y_b \in \mathbb{G}$.*

The scheme should satisfy the following requirements:

- **Correctness:** *For any $f \in \mathcal{C}$ and $x \in I$, we have* $\Pr[(K_0, K_1) \overset{\$}{\leftarrow} \mathsf{FSS.Gen}(1^\lambda, f) : \sum_{b\in\{0,1\}} \mathsf{FSS.Eval}(b, K_b, x) = f(x)] = 1$.

- **Security:** *For any $b \in \{0,1\}$, there exists a PPT simulator* Sim *such that for any polynomial-size function sequence $f_\lambda \in \mathcal{C}$, the distributions* $\{(K_0, K_1) \overset{\$}{\leftarrow} \mathsf{FSS.Gen}(1^\lambda, f_\lambda) : K_b\}$ *and* $\{K_b \overset{\$}{\leftarrow} \mathsf{Sim}(1^\lambda, \mathsf{Leak}(f_\lambda))\}$ *are computationally indistinguishable.*

In the constructions we use, the leakage function $\mathsf{Leak} : \{0,1\}^* \to \{0,1\}^*$ *is given by* $\mathsf{Leak}(f_\lambda) = (I, \mathbb{G})$*, namely it outputs a description of the input and output domains of f.*

We also define a full-domain evaluation algorithm, $\mathsf{FSS.FullEval}(b, K_b)$, which outputs a vector of $|I|$ group elements, corresponding to running Eval on every element x in the domain I. For the type of FSS we consider, $\mathsf{FSS.FullEval}$ is significantly faster than the generic solution of running $|I$ instances of Eval.

We will use FSS for point functions and sums of point functions, as defined below.

Definition 2.2 (Distributed Point Function (DPF) [GI14, BGI15]) *For an Abelian group $\mathbb{G}$, $\alpha \in [n]$, and $\beta \in \mathbb{G}$, the* point function *$f_{\alpha,\beta}$ is the function $f_{\alpha,\beta} : [n] \to \mathbb{G}$ defined by $f_{\alpha,\beta}(x) = 0$ whenever $x \neq \alpha$, and $f_{\alpha,\beta}(x) = \beta$ if $x = \alpha$. A* distributed point function *(DPF) is an FSS scheme for the class of point functions $\{f_{\alpha,\beta} : [n] \to \mathbb{G} \mid \alpha \in [n], \beta \in \mathbb{G}\}$.*

The best known DPF construction [BGI16b] can use any pseudorandom generator (PRG) $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$ and has the following efficiency features. For $m = \lceil \frac{\log|\mathbb{G}|}{\lambda+2} \rceil$, the key generation algorithm Gen invokes G at most $2(\lceil \log n \rceil + m)$ times, the evaluation algorithm Eval invokes G at most $\lceil \log n \rceil + m$ times, and the full-domain evaluation algorithm $\mathsf{FullEval}$ invokes G at most $n \cdot (1 + m)$ times. The size of each key is at most $\lceil \log n \rceil \cdot (\lambda + 2) + \lambda + \lceil \log_2 |\mathbb{G}| \rceil$ bits.

We will use a simple and generic extension of DPF to sums of point functions.

Definition 2.3 (FSS for sum of point functions (SPFSS)) *For $S = (s_1, \ldots, s_t) \in [n]^t$ and $\vec{y} = (y_1, \ldots, y_t) \in \mathbb{G}^t$, define the* sum of point functions *$f_{S,\vec{y}} : [n] \to \mathbb{G}$ by*

$$f_{S,\vec{y}}(x) = \sum_{i=1}^{t} f_{s_i,y_i}(x).$$

An SPFSS *scheme is an FSS scheme for the class of sums of point functions $f_{S,\vec{y}}$.*

Note that for $S = (s_1, \ldots, s_t)$, the function $f_{S,\vec{y}}$ non-zero on *at most t* points. If the elements of S are distinct, $f_{S,\vec{y}}$ coincides with a *multi-point function* for the set of points in S; however, in our usage we can have repeated elements in S. A simple realization of SPFSS is by summing t independent instances of DPF. This will typically be good enough for our purposes. Alternatively, asymptotically better constructions for full-domain evaluation can be obtained using hash functions or (probabilistic) batch codes [IKOS04, BCGI18, ACLS18, SGRR19]. To simplify notation, when generating keys for a scheme $\mathsf{SPFSS} = (\mathsf{SPFSS.Gen}, \mathsf{SPFSS.Eval})$, we write $\mathsf{SPFSS.Gen}(1^\lambda, S, \vec{y})$, instead of explicitly writing $f_{S,\vec{y}}$.

2.3 Pseudorandom Correlation Generators

A pseudorandom correlation generator (PCG) [BCGI18, BCG+19b] is a primitive with a setup algorithm that generates a pair of seeds, which can then be locally expanded to produce *correlated* pseudorandomness. To define security, we use the notions of correlation generators, and reverse-sampleable correlation generators, from [BCG+19b].

Definition 2.4 (Correlation generator) *A PPT algorithm $\mathcal{C}$ is called a* correlation generator*, if $\mathcal{C}$ on input 1^λ outputs a pair of elements in $\{0,1\}^n \times \{0,1\}^n$ for $n \in \mathsf{poly}(\lambda)$.*

Definition 2.5 (Reverse-sampleable correlation generator) *Let $\mathcal{C}$ be a correlation generator. We say $\mathcal{C}$ is* reverse sampleable *if there exists a PPT algorithm* $\mathsf{RSample}$ *such that for $\sigma \in \{0,1\}$ the correlation obtained via:*

$$\{(R'_0, R'_1)\ |(R_0, R_1) \xleftarrow{\$} \mathcal{C}(1^\lambda), R'_\sigma := R_\sigma, R'_{1-\sigma} \xleftarrow{\$} \mathsf{RSample}(\sigma, R_\sigma)\}$$

is computationally indistinguishable from $\mathcal{C}(1^\lambda)$.

The following definition of pseudorandom correlation generators can be viewed as a generalization of the definition of the pseudorandom VOLE generator in [BCGI18].

Definition 2.6 (Pseudorandom Correlation Generator (PCG)) *Let $\mathcal{C}$ be a reverse-sampleable correlation generator. A* pseudorandom correlation generator (PCG) *for $\mathcal{C}$ is a pair of algorithms* $(\mathsf{PCG.Gen}, \mathsf{PCG.Expand})$ *with the following syntax:*

- $\mathsf{PCG.Gen}(1^\lambda)$ *is a PPT algorithm that given a security parameter λ, outputs a pair of seeds $(\mathsf{k}_0, \mathsf{k}_1)$;*
- $\mathsf{PCG.Expand}(\sigma, \mathsf{k}_\sigma)$ *is a polynomial-time algorithm that given a party index $\sigma \in \{0,1\}$ and a seed k_σ, outputs a bit string $R_\sigma \in \{0,1\}^n$.*

The algorithms $(\mathsf{PCG.Gen}, \mathsf{PCG.Expand})$ *should satisfy the following:*

- **Correctness.** *The correlation obtained via:*

  $$\{(R_0, R_1) \mid (\mathsf{k}_0, \mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda), R_\sigma \leftarrow \mathsf{PCG.Expand}(\sigma, \mathsf{k}_\sigma) \text{ for } \sigma \in \{0,1\}\}$$

  is computationally indistinguishable from $\mathcal{C}(1^\lambda)$.

- **Security.** *For any $\sigma \in \{0,1\}$, the following two distributions are computationally indistinguishable:*

  $$\begin{aligned}&\{(\mathsf{k}_{1-\sigma}, R_\sigma) \mid (\mathsf{k}_0, \mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda), R_\sigma \leftarrow \mathsf{PCG.Expand}(\sigma, \mathsf{k}_\sigma)\} \text{ and}\\ &\{(\mathsf{k}_{1-\sigma}, R_\sigma) \mid (\mathsf{k}_0, \mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda), R_{1-\sigma} \leftarrow \mathsf{PCG.Expand}(\sigma, \mathsf{k}_{1-\sigma}),\\ &\qquad R_\sigma \xleftarrow{\$} \mathsf{RSample}(\sigma, R_{1-\sigma})\}\end{aligned}$$

  where $\mathsf{RSample}$ *is the reverse sampling algorithm for correlation $\mathcal{C}$.*

Note that to avoid the trivial solution where $\mathsf{PCG.Gen}$ simply outputs a sample from $\mathcal{C}$, we are only interested in constructions where the seed size is significantly shorter than the output size.

3 The Ring-LPN Assumption

In this section, we recall the ring-LPN assumption, which was first introduced (over $\mathbb{Z}_2$) in [HKL$^+$12] to build efficient authentication protocols. Since then, it has received some attention from the cryptography community [BL12, DP12, LP15, GJL15], due to its appealing combination of LPN-like structure, compact parameters, and short runtimes. Below, we provide a definition of module-LPN, which generalizes ring-LPN in the same way that the more well-known module-LWE generalizes ring-LWE. We also discuss several variants depending on the choice of ring, including cyclotomic rings over $\mathbb{Z}_p$, which have previously been used for ring-LWE.

3.1 Ring-LPN

Definition 3.1 (Ring-LPN) *Let $R = \mathbb{Z}_p[X]/F(X)$ for a prime p and degree-N polynomial $F(X) \in \mathbb{Z}_p[X]$. (We will write R_p when we want to highlight the modulus p.) For $t \in \mathbb{N}$, let $\mathcal{HW}_{R,t}$ denote the distribution of "sparse polynomials" over R obtained by sampling t noise positions $A \leftarrow [0..N)^t$ and t payloads $\vec{b} \leftarrow (\mathbb{Z}_p^*)^t$ uniformly at random, and outputting $e(X) := \sum_{j=0}^{t-1} \vec{b}[j] \cdot X^{A[j]}$. (We write $\mathcal{HW}_t$ when R is clear from the context.) For $R = R(\lambda), m = m(\lambda), t = t(\lambda)$, we say that the* ring-LPN problem *R-$\mathsf{LPN}_{R,m,t}$ is hard if for every nonuniform polynomial-time distinguishher $\mathcal{A}$, it holds that*

$$\left|\Pr[\mathcal{A}((a^{(i)}, a^{(i)} \cdot e + f^{(i)})_{i=1}^m) = 1] - \Pr[\mathcal{A}((a^{(i)}, u^{(i)})_{i=1}^m) = 1]\right| \leq \mathsf{negl}(\lambda)$$

where the probabilities are taken over $a^{(1)}, \ldots, a^{(m)}, u^{(1)}, \ldots, u^{(m)} \leftarrow R(\lambda)$ and $e, f^{(1)}, \ldots, f^{(m)} \leftarrow \mathcal{HW}_{R,t}$.

We will also use the regular *variant of R-$\mathsf{LPN}_{R,m,t}$, which is defined in the same way as above except that $\mathcal{HW}_{R,t}$ is obtained by letting A include a single random position from each block of size N/t (where here we assume that $t|N$).*

Remark 3.1 (Useful parameters) *Our constructions will only use Definition 3.1 with $m = 1$, namely one sample. Bigger values of m will be used for reducing security in this case to a variant where the secret e is uniform (see Lemma 3.4 below). The sparsity parameter t will roughly correspond to a concrete security parameter, the degree N to the length of the target correlation (or number of instances of an atomic correlation), and p to a modulus over which this correlation is defined. Useful choices of the polynomial $F(X)$ will be discussed in Section 3.2 below.*

Remark 3.2 (Noise distribution) *Note that in the default variant of our definition, the distribution over sparse polynomials is obtained by picking the t noise positions* with *replacement. This can result in collisions, and thus negatively affect the entropy introduced by the payloads. The reason for this choice is that it helps simplify some of our constructions and their analysis. The entropy loss is minor in the regime of parameters we care about, as for $t \ll N$ the probability of collisions is very small. The collisions are entirely avoided in the regular variant of the definition, which also leads to better concrete efficiency in our constructions.*

Remark 3.3 (Extension to other rings.) *Note that our restriction to prime-order fields $\mathbb{Z}_p$ is only for simplicity; R-LPN can be defined similarly over other rings (such as extension fields $\mathbb{F}_{p^d}$ or rings $\mathbb{Z}_{2^k}$ or $\mathbb{Z}_{pq}$ for primes p, q). These alternative choices are not known to introduce any significant weakness or structural difference compared to the version over prime-order fields. In fact, we obtain PCGs for OLEs and multiplication triples over extension fields $\mathbb{F}_{p^d}$ by using rings R_p defined over the base field.*

Module LPN. We will also use a natural generalization of R-LPN, where we replace $a^{(i)} \cdot e$ by the inner product $\langle \vec{a}^{(i)}, \vec{e} \rangle$ between length-$(c-1)$ vectors over R, for some constant $c \geq 2$. (The parameter c can be viewed as a *compression factor* in our construction – see more below.) We call this *module-LPN*, analogously to module-LWE. This will allow for useful efficiency tradeoffs, as according to our security analysis it will be enough to choose the total number of noise positions w such that $w = ct \approx \lambda$, and therefore increasing c allows to choose a smaller t. For the parameter regime in our constructions, increasing c will result in shorter PCG seeds at the expanse of higher running time of Expand.

Definition 3.2 (Module-LPN) *Let $c \geq 2$ be an integer and $R, \mathcal{HW}_{R,t}$ be as in Definition 3.1. Then, for $R = R(\lambda), m = m(\lambda), t = t(\lambda)$, we say that the R^c-$\mathsf{LPN}_{R,m,t}$ problem is hard if for every nonuniform polynomial-time distinguisher $\mathcal{A}$, it holds that*

$$\left|\Pr[\mathcal{A}((\vec{a}^{(i)}, \langle\vec{a}^{(i)}, \vec{e}\rangle + f^{(i)})_{i=1}^m) = 1] - \Pr[\mathcal{A}((\vec{a}^{(i)}, u^{(i)})_{i=1}^m) = 1]\right| \le \mathsf{negl}(\lambda)$$

where the probabilities are taken over $\vec{a}^{(1)}, \ldots, \vec{a}^{(m)} \leftarrow R^{c-1}$, $u^{(1)}, \cdots, u^{(m)} \leftarrow R$, $\vec{e} \leftarrow \mathcal{HW}_{R,t}^{c-1}$, $f^{(1)}, \ldots, f^{(m)} \leftarrow \mathcal{HW}_{R,t}$. We similarly define the regular variant by modifying $\mathcal{HW}_{R,t}$ as in Definition 3.1.

Equivalence to module-LPN with uniform secret. We observe that, by the same argument as for standard LWE [ACPS09], the R-LPN (resp. module-LPN) problem with a secret chosen from the error distribution is at least as hard as the corresponding R-LPN (resp. module-LPN) problem where the secret is chosen uniformly at random, if the adversary is given one additional sample (resp., $c-1$ additional samples).

Lemma 3.4 *For any $c \ge 2$, let R^c-uLPN denote the variant of R^c-LPN where the secret e is sampled uniformly at random. Then, for any $R = R(\lambda), m = m(\lambda), t = t(\lambda)$, if R^c-$\mathsf{uLPN}_{R,m+(c-1),t}$ is hard then R^c-$\mathsf{LPN}_{R,m,t}$ is hard.*

Proof. Let $m > c$. We show how a distinguisher for R^c-$\mathsf{LPN}_{R,m-(c-1),t}$ can be used to solve an instance of R^c-$\mathsf{uLPN}_{R,m,t}$, which implies the statement in the theorem. Let $(\vec{a}^{(1)}, u^{(1)}, \cdots, \vec{a}^{(m)}, u^{(m)})$ be R^c-uLPN samples, where $\vec{a}^{(i)} \in R^{c-1}$ and $u^{(i)} \in R$ for $i = 1$ to m. Assume that $(\vec{a}^{(1)}, \cdots, \vec{a}^{(c-1)})$, viewed as a matrix A of dimensions $(c-1) \times (c-1)$ over R, is invertible; this happens with high probability (at least constant) over a random choice of the $\vec{a}^{(i)}$. Denote by $A' \in R^{m-(c-1)\times(c-1)}$ the matrix whose rows are the remaining $(\vec{a}^{(c)}, \cdots, \vec{a}^{(m)})$, by $\vec{u}$ the (vertical) vector $(u^{(1)}, \cdots, u^{(c-1)})$, and by $\vec{u}'$ the (vertical) vector $(u^{(c)}, \cdots, u^{(m)})$. With these, the distinguishing game can be rewritten as follows: the adversary must distinguish between the case where

- $A\vec{e} + \vec{f} = \vec{u}$ for some $\vec{e} \in R^{c-1}$ and sparse $\vec{f} \in \mathcal{HW}_{R,t}^{c-1}$, and
- $A'\vec{e} + \vec{f'} = \vec{u}'$ for some sparse $\vec{f'} \in \mathcal{HW}_{R,t}^{m-c+1}$,

from the case where $(\vec{u}, \vec{u}')$ are random. But since A is invertible, by multiplying the first equation with $B \leftarrow A'A^{-1}$ and denoting $\vec{v} \leftarrow B\vec{u} - \vec{u}'$, the first case can be rewritten as

$$B\vec{f} - \vec{f'} = \vec{v}$$

which is distributed exactly as an instance of the R^c-$\mathsf{LPN}_{R,m-(c-1),t}$ problem; hence, a distinguisher for R^c-$\mathsf{LPN}_{R,m-(c-1),t}$ can be used to get a distinguisher for R^c-$\mathsf{uLPN}_{R,m,t}$. □

Relation to syndrome decoding. Our constructions will use module-LPN with a single sample ($m = 1$). To simplify notation and emphasize that the secret comes from the error distribution, we often combine the secret and noise value ($\vec{e}$ and f in the notation of Definition 3.2) into a single vector $\vec{e}$, replacing the previous $(\vec{a}, \langle\vec{a}, \vec{e}\rangle + f)$ by writing $(\vec{a}, \langle\vec{a}, \vec{e}\rangle)$, where

$$\vec{a} = (1, \vec{a}'), \vec{a}' \xleftarrow{\$} R^{c-1}, \vec{e} \xleftarrow{\$} \mathcal{HW}_{R,t}^{c}.$$

This formulation of module-LPN is equivalent to a variant of the syndrome decoding problem in random polynomial codes. To see this, let M_i be the $N \times N$ matrix over $\mathbb{Z}_p$ representing multiplication with the fixed element $a_i' \in R_p$, for $i = 1$ to $c-1$. Define the matrix

$$H = [\mathsf{Id}_N || M_1 || \cdots || M_{c-1}].$$

H is a parity-check matrix in systematic form for a polynomial code defined by the random elements $a_i' \in R_p$. Module-LPN can be seen as a decisional version of syndrome decoding for this code, where we assume that $(H, H \cdot \vec{e})$ is pseudorandom for an error vector $\vec{e} = (e_1, \ldots, e_c)$ with a regular structure, namely where each of the length-N blocks e_i have t non-zero entries. The code has length $N \cdot c$ and dimension $N \cdot (c-1)$; the rate of the code is therefore $(c-1)/c$. With this formulation, c can be viewed as the compression factor of the linear map $\vec{e} \to H \cdot \vec{e}$. Therefore, we generally refer to c as the *syndrome compression factor*.

3.2 Choice of the Polynomial F

The ring-LPN assumption (and more generally, the module-LPN assumption) is dependent of the choice of the underlying polynomial F. We discuss possible choices for the polynomial F, and their implications for the security of ring-LPN/module-LPN over the corresponding ring $R_p = \mathbb{Z}_p[X]/F(X)$.

Irreducible $F(X)$. The most conservative instantiation is when $F(X)$ is irreducible over $\mathbb{Z}_p$, and so R_p is a field. In this setting, no attacks are known that perform significantly better than for standard LPN.

Reducible $F(X)$. We also consider when $F(X)$ is reducible over $\mathbb{Z}_p$, and splits into several distinct factors. Here we have a few different useful instantiations.

1. *Cyclotomic $F(X)$.* Let $F(X)$ be the M-th cyclotomic polynomial, whose degree is $N = \phi(M)$ (Euler's totient function). Then, $F(X)$ splits modulo p into N/d distinct factors f_i, where each f_i is of degree d, and d is the smallest integer satisfying $p^d \equiv 1 \bmod M$. The advantage of using a cyclotomic F is that it allows for fast multiplication in R_p using FFT. We are particularly interested in the following cases.

   - *Two-power N, prime $p > 2N$.* Let N be a power of two and p a large prime such that $p \equiv 1 \bmod (2N)$ (here, $M = 2N$). Then $F(X)$ splits completely into N linear factors modulo p, so R_p is isomorphic to $\mathbb{Z}_p^N$.
   - $p = 2$. Here, each degree-d subring $\mathbb{Z}_p[X]/(f_i(X))$ is isomorphic to the finite field $\mathbb{F}_{2^d}$, hence $R_p \cong \mathbb{F}_{2^d}^{N/d}$.

   Regarding security, we observe that cyclotomic polynomials can introduce an additional weakness, due to sparse factors of $F(X)$. In Section 8.2, we analyze this further and discuss how to adjust parameters accordingly.

2. *Random factors.* A more conservative option may be to choose an $F(X)$ that splits completely into d distinct, *random* factors. For instance, for a large prime p we can pick (distinct) random elements $\alpha_1, \ldots, \alpha_N \leftarrow \mathbb{Z}_p$ and let

   $$F(X) = \prod_{i=1}^{N} (X - \alpha_i)$$

   Just as with the two-power cyclotomic case, R_p is isomorphic to $\mathbb{Z}_p^N$. Now, however, the problem may be harder since we are avoiding the structure given by roots of unity. On the other hand, the isomorphism is more expensive to compute as we can no longer make a direct use of FFT, and polynomial interpolation algorithms cost $O(N \log^2 N)$ instead of $O(N \log N)$.

4 PCGs for OLE and Authenticated Multiplication Triples

In this section, we construct PCGs for OLE and authenticated multiplication triples, based on the R^c-LPN assumption. The constructions in this section can achieve an arbitrary (a priori bounded) polynomial stretch, where the seed size scales logarithmically with the output length N and the running time of Expand scales nearly linearly with N.

4.1 PCG for OLE over R_p

We build a PCG for producing a single OLE over the ring R_p. When R_p splits appropriately, as described in Section 3.2, this can be locally transformed into a PCG for a large batch of OLEs or (authenticated) multiplication triples over a finite field $\mathbb{F}_{p^d}$ or $\mathbb{F}_p$. The OLE correlation over R_p outputs a single sample from the distribution

$$\left\{((x_0, z_0), (x_1, z_1)) \middle| x_0, x_1, z_0 \xleftarrow{\$} R_p, z_1 = x_0 \cdot x_1 - z_0\right\}$$

This can be viewed as giving the two parties additive shares of a product of two random elements x_0, x_1 of R_p, where x_σ is known to party P_σ.

Below is an informal presentation of the construction, which is described formally in Fig. 1.

The high-level idea is to first give each of the two parties a random vector $\vec{e}_0$ or $\vec{e}_1 \in R_p^c$, consisting of sparse polynomials, together with a random, additive secret sharing of the tensor product $\vec{e}_0 \otimes \vec{e}_1$ over R_p.

We view $\vec{e}_0, \vec{e}_1$ as R^c-LPN error vectors (whose first entry is implicitly the R^c-LPN secret), which will be expanded to produce outputs $x_\sigma = \langle \vec{a}, \vec{e}_\sigma \rangle$ by each party P_σ for a random, public $\vec{a} = (1, \hat{\vec{a}})$. This defines two R^c-LPN instances with independent secrets but the same $\vec{a}$ value, which are pseudorandom by a standard reduction to R^c-LPN with a single sample. To obtain shares of $x_0 \cdot x_1$, observe that when $\vec{a}$ is fixed, this is a degree 2 function in $(\vec{e}_0, \vec{e}_1)$, so can be computed locally by the parties given their shares of $\vec{e}_0 \otimes \vec{e}_1$.

The only part that remains, then, is to distribute shares of this tensor product. Recall that each entry of $\vec{e}_\sigma$ is a polynomial of degree less than N with at most t non-zero coordinates. We write these coefficients as a set of indices $A \in [0..N)^t$ and corresponding non-zero values $\vec{b} \in \mathbb{Z}_p^t$. Taking two such sparse polynomials $(A, \vec{b})$ and $(A', \vec{b}')$, notice that the product of the two polynomials is given by

$$\left(\sum_{i \in [0..t)} \vec{b}[i] \cdot X^{A[i]}\right) \cdot \left(\sum_{j \in [0..t)} \vec{b}'[j] \cdot X^{A'[j]}\right) = \sum_{i,j \in [0..t)} \vec{b}[i] \cdot \vec{b}'[j] \cdot X^{A[i]+A'[j]}$$

We can therefore express the coefficient vector of the product as a *sum of t^2 point functions*, where the (i, j)-th point function evaluates to $\vec{b}[i] \cdot \vec{b}'[j]$ at input $A[i] + A'[j]$, and zero elsewhere. This means the parties can distribute this product using a function secret sharing scheme SPFSS for sums of point functions, as defined in Definition 2.3. Recall that an SPFSS takes a sequence of points and associated vector of values, and produces two keys that represent shares of the underlying sum of point functions. If each party locally evaluates its key at every point in the domain, then it obtains a pseudorandom secret-sharing of the coefficients of the entire sparse polynomial.

There are c^2 polynomials in the tensor product, so overall we need c^2 instances of SPFSS, where each SPFSS uses t^2 point functions. Instantiating this naively using t^2 distributed point functions, we get a seed size of $\tilde{O}(\lambda(ct)^2 \log N)$ bits. Note that to achieve exponential security against the best known attacks on R^c-LPN, it is enough to choose $ct = O(\lambda)$. By increasing N, we can therefore obtain an arbitrary polynomial stretch for the PCG, where the stretch is defined as the ratio of its output length to the seed size.

More concretely, we have the following theorem.

Construction G_{OLE}

Parameters: Security parameter λ, noise weight $t = t(\lambda)$, compression factor $c \geq 2$, modulus $p = p(\lambda)$, degree $N = N(\lambda)$, and the ring $R_p = \mathbb{Z}_p[X]/F(X)$ for degree-N $F(X) \in \mathbb{Z}_p[X]$.
An FSS scheme $(\mathsf{SPFSS.Gen}, \mathsf{SPFSS.FullEval})$ for sums of t^2 point functions, with domain $[0..2N-1)$ and range $\mathbb{Z}_p$.
Public input: random polynomials $a_1, \ldots, a_{c-1} \in R_p$, used for R^c-LPN.
Correlation: After expansion, outputs $(x_0, z_0) \in R_p^2$ and $(x_1, z_1) \in R_p^2$, where $z_0 + z_1 = x_0 \cdot x_1$.

Gen: On input 1^λ:

1. For $\sigma \in \{0,1\}$ and $i \in [0..c)$, sample random vectors $A_\sigma^i \leftarrow [0..N)^t$ and $\vec{b}_\sigma^i \leftarrow (\mathbb{Z}_p^*)^t$.
2. For each $i, j \in [0..c)$, sample FSS keys $(K_0^{i,j}, K_1^{i,j}) \overset{\$}{\leftarrow} \mathsf{SPFSS.Gen}(1^\lambda, A_0^i \boxplus A_1^j, \vec{b}_0^i \otimes \vec{b}_1^j)$.
3. Let $\mathsf{k}_\sigma = \Big((K_\sigma^{i,j})_{i,j \in [0..c)}, (A_\sigma^i, \vec{b}_\sigma^i)_{i \in [0..c)} \Big)$.
4. Output $(\mathsf{k}_0, \mathsf{k}_1)$.

Expand: On input $(\sigma, \mathsf{k}_\sigma)$:

1. Parse k_σ as $\Big((K_\sigma^{i,j})_{i,j \in [0..c)}, (A_\sigma^i, \vec{b}_\sigma^i)_{i \in [0..c)} \Big)$.
2. Define (over $\mathbb{Z}_p$) the degree $< N$ polynomials, for $i \in [0..c)$
$$e_\sigma^i(X) = \sum_{j \in [0..t)} \vec{b}_\sigma^i[j] \cdot X^{A_\sigma^i[j]}$$
3. Compute $x_\sigma = \langle \vec{a}, \vec{e}_\sigma \rangle \mod F(X)$, where $\vec{a} = (1, a_1, \ldots, a_{c-1})$, $\vec{e}_\sigma = (e_\sigma^0, \ldots, e_\sigma^{c-1})$.
4. For $i, j \in [0..c)$, compute $u_{\sigma, i+cj} \leftarrow \mathsf{SPFSS.FullEval}(\sigma, K_\sigma^{i,j})$ and view this as a degree $< 2N$ polynomial, defining the length-c^2 vector $\vec{u}_\sigma \mod F(X)$.
5. Compute $z_\sigma = \langle \vec{a} \otimes \vec{a}, \vec{u}_\sigma \rangle \mod F(X)$.
6. Output (x_σ, z_σ)

Figure 1: PCG for OLE over the ring R_p, based on ring-LPN

Theorem 4.1 *Suppose that* SPFSS *is a secure FSS scheme for sums of point functions (Definition 2.3), and the* R^c-$\mathsf{LPN}_{R_p,1,t}$ *assumption (Definition 3.2) holds. Then the construction in Fig. 1 is a secure PCG for OLE over* R_p.

When instantiating SPFSS using from $\mathsf{PRG} : \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$ via the PRG-based DPF construction from [BGI16b], we have:

- Each party's seed has size at most $(ct)^2 \cdot ((\lceil \log N \rceil + 1) \cdot (\lambda + 2) + \lambda + \lceil \log p \rceil) + ct(\lceil \log N \rceil + \lceil \log p \rceil)$ bits.
- The computation of Expand can be done with at most $(4 + 2\lfloor \log p/\lambda \rfloor)N(ct)^2$ PRG operations, and $O(c^2 N \log N)$ operations in $\mathbb{Z}_p$.

Under standard ring-LPN parameters the above can be instantiated with seed size $\mathsf{poly}(\lambda) \cdot \log N$ and with Expand running in time $O(N^{1+\epsilon})$. This running time can be improved to $\tilde{O}(N)$ using batch codes, as discussed below. Increasing the syndrome compression parameter c allows choosing a smaller noise parameter t, which results in smaller seed side at the expense of increased running time.

Proof. We first argue correctness.

Let $i, j \in [0..c)$ and consider the polynomials e_0^i, e_1^j defined in Expand. We have,

$$e_0^i(X) \cdot e_1^j(X) = \sum_{k,\ell \in [0..t)} \vec{b}_0^i[k] \cdot \vec{b}_1^j[\ell] \cdot X^{A_0^i[k] + A_1^j[\ell]}$$

Therefore, the coefficients of this product can be obtained by evaluating the sum of point functions used in the (i,j)-th instance of SPFSS. This means that $u_{0,i+cj} + u_{1,i+cj}$ equals $e_0^i(X) \cdot e_1^j(X)$, and hence, $\vec{u} = \vec{e}_0 \otimes \vec{e}_1$. Looking at the outputs of expand, we then have

$$z_0 + z_1 = \langle \vec{a} \otimes \vec{a}, \vec{u}_0 + \vec{u}_1 \rangle = \langle \vec{a} \otimes \vec{a}, \vec{e}_0 \otimes \vec{e}_1 \rangle = \langle \vec{a}, \vec{e}_0 \rangle \cdot \langle \vec{a}, \vec{e}_1 \rangle = x_0 \cdot x_1$$

where the penultimate equality can be seen by inspection of the tensor products.

Each $\langle \vec{a}, \vec{e}_\sigma \rangle$ can be seen as a sample from the R^c-LPN distribution with a fixed random $\vec{a}$ and independent secret $\vec{e}_\sigma$. It follows from a standard hybrid argument that (x_0, x_1) is computationally indistinguishable from a random pair in R_p^2, under R^c-LPN. Furthermore, by the security of SPFSS, each z_σ is individually pseudorandom, so the outputs (x_0, z_0, x_1, z_1) are indistinguishable from a random OLE over R.

To show the security property, fix $\sigma = 1$ (the case $\sigma = 0$ is symmetric). For two keys $(\mathsf{k}_0, \mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda)$ with associated expanded outputs (x_0, z_0) and (x_1, z_1), we need to show that

$$\{(\mathsf{k}_1, x_0, z_0)\} \equiv \left\{(\mathsf{k}_1, \tilde{x}_0, \tilde{z}_0) | \tilde{x}_0 \xleftarrow{\$} R_p, \tilde{z}_0 = \tilde{x}_0 \cdot x_1 - z_1\right\}$$

We use a sequence of hybrids, where first we change z_0 to be computed as $x_0 \cdot x_1 - z_1$, and successively replace each FSS key $K_1^{i,j}$ in k_1 with a *simulated* key, generated with only the range and domain of the function. This is indistinguishable from the first distribution, by the correctness and security properties of the FSS scheme. Then, since K_1 and z_0 are independent of the R^c-LPN secret producing x_0, we can rely on R^c-LPN to sample x_0 at random instead of from the seed k_0. Finally, we can now switch the FSS keys back to ones generated from SPFSS.Gen, again using the FSS security property. This gives the distribution on the right. $\square$

We remark that assuming R^c-LPN holds for *regular error distributions*, the seed size can be reduced to roughly $(ct)^2 \cdot ((\log N - \log t + 1) \cdot (\lambda + 2) + \lambda + \log p) + ct(\log N + \log p)$ bits, and the number of PRG calls in Expand down to $(4 + 2\lfloor (\log p)/\lambda \rfloor)Nc^2 t$. See details below. This eliminates a factor of t from the asymptotic computational cost. Furthermore, implementing SPFSS using batch codes reduces the number of PRG calls to $O(Nc^2)$. This eliminates another factor of t, making the overall computational overhead logarithmic in N, but comes at a cost of making distributed seed generation more complex.

Obtaining OLEs over $\mathbb{F}_p$. As discussed in Section 3.2, when R and p are chosen appropriately, an OLE over R_p is locally equivalent to N OLEs over $\mathbb{F}_p$ or $\mathbb{F}_{p^d}$. Hence, this PCG immediately implies PCGs over $\mathbb{F}_{p^d}$, with the same seed size and complexity.

If we want to rely on the (apparently) more conservative version of R^c-LPN, where $F(X)$ is *irreducible* in $\mathbb{Z}_p[X]$, the parties can still use our PCG over R_p to obtain OLEs over $\mathbb{Z}_p$, but this requires $O(N)$ interaction. To do this, the parties each sample random polynomials $a, b \in \mathbb{Z}_p[X]$, each of degree $< N/2$. They then use the OLE over R_p to multiply a and b, which can be done by sending N elements of $\mathbb{Z}_p$.[6] This gives shares of $c = ab$ in R_p, which equals ab over $\mathbb{Z}_p[X]$, since no overflow occurs modulo $F(X)$ (which has degree N). Each party then locally computes evaluations of its shares of a, b and c at $N/2$ fixed, distinct, non-zero points, which gives $N/2$ secret-shared products over $\mathbb{Z}_p$ (this can be done as long as $p > N/2$).

Optimizations. We now discuss a few optimizations which apply to the basic scheme.

Optimizing the MPFSS evaluation. Naively, the computational cost of the FSS full-domain evaluation is $O((ct)^2 N)$ PRG operations. Using a regular error distribution, we can bring this down to $O(c^2 tN)$ (see below). With batch codes [IKOS04, BCGI18] or probabilistic batch codes [ACLS18, SGRR19], the full evaluation cost can be brought down to $O(c^2 N)$ operations. However, if the seed generation phase has to be created by a secure distributed protocol, setting up the seeds that support better Expand time can hurt the concrete cost of distributed seed generation.

Using regular errors. Suppose two sparse polynomials $e_0, e_1 \in Z_p^N$ are regular, that is $e_b = (e_{b,1}, \dots, e_{b,t})$, where each $e_{b,j} \in \mathbb{Z}_p^{N/t}$ has weight 1, and defines a coefficient in the range $[(j-1) \cdot (N/t), j \cdot (N/t) - 1]$. Each pair $(e_{0,i}, e_{1,j})$ gives rise to an index in $[(i+j-2) \cdot (N/t), (i+j) \cdot (N/t) - 2]$, so the product of two regular error polynomials can be represented by a t^2-point SPFSS of domain size $2N/t$. This leads to a total expansion cost of $O(c^2 tN)$ PRG operations.

Extension to multiplication triples. Recall that in an instance of an OLE correlation over $\mathbb{Z}_p$, parties P_0 and P_1 each hold a secret random $\mathbb{Z}_p$ element, and they jointly hold an additive secret-sharing of the product of the two secrets. While OLE correlations can be directly useful for some applications of secure computation [NP99, IPS09, DGN+17, GN19, CDI+19, HIMV19], it is often more convenient to use a slightly more complicated variant known as a *multiplication triple* correlation [Bea91]. In a (2-party) multiplication triple, the two parties hold *shares* of random $\mathbb{Z}_p$ elements a and b (which are known to neither party), and moreover they hold shares of the product $c = a \cdot b$. Multiplication triples are useful for 2-PC of arithmetic circuits over $\mathbb{Z}_p$ with security against semi-honest parties: each multiplication gate can be evaluated by consuming a single multiplication triple and communicating two $\mathbb{Z}_p$ elements per party. (Addition gates are "for free.") An instance of a multiplication triple correlation can be obtained in a black-box way using two instances of an OLE correlation. Concretely, writing $a = a_0 + a_1$, $b = b_0 + b_1$, and $c = a_0 b_0 + a_1 b_1 + a_0 b_1 + a_1 b_0$, one can distribute (a_σ, b_σ) to party P_σ and secret-share the cross-terms $a_0 b_1$ and $a_1 b_0$ via two independent OLE instances (the terms $a_0 b_0$ and $a_1 b_1$ can be computed locally by P_0 and P_1 respectively and added to the OLE outputs). As a result, a PCG generating N instances of multiplication triples can be obtained from a PCG generating $2N$ instances of OLE. In the next section we will see how to extend this to *authenticated* multiplication triples, which serve as a useful resource for 2-PC with security against *malicious* parties, at a slightly higher cost.

[6]This can be reduced to $N/2$, by defining a, b to be the first $N/2$ coefficients of the polynomials x_0, x_1 produced by the OLE, so that only the second half of the coefficients need to be sent in the multiplication protocol.

Extension to higher degree correlations. We can naturally extend this construction from OLE over R_p to general degree-D correlations (over R_p), for any constant D, by sharing D-way products of sparse polynomials instead of just pairwise products. However, this comes at a high cost: the seed size increases to $O((ct)^D \log N\lambda)$, and the computational cost becomes $\tilde{O}((ct)^D \cdot N)$.

4.2 Authenticated Multiplication Triples

We now show how to modify the PCG for OLE to produce *authenticated multiplication triples*, which are often used in maliciously secure MPC protocols such as the BDOZ [BDOZ11] and SPDZ [DPSZ12, DKL$^+$13] line of work. Note that although OLE can be used to build authenticated triples in a black-box way, doing this requires several OLEs and some interaction, for every triple. This is contrasted with the case of standard multiplication triples, discussed above, that can be reduced to OLE without any interaction. Our PCG avoids this interaction, with only a small overhead on top of the previous construction: the seeds are less than 2x larger, while the expansion phase has around twice the computational cost.

Secret-sharing with MACs. We use authenticated secret-sharing based on SPDZ MACs between n parties, where a secret-sharing of $x \in \mathbb{Z}_p$ is defined as:

$$[\![x]\!] = (\alpha_i, x_i, m_{x,i})_{i=1}^n \quad \text{such that } \sum_i x_i = x, \sum_i m_{x,i} = x \cdot \sum_i \alpha_i$$

Note that the MAC key shares α_i are fixed for every shared x. The MAC shares $m_{x,i}$ are used to prevent a sharing from being opened incorrectly, via a MAC check procedure from [DKL$^+$13]. An *authenticated multiplication triple* is a tuple of random sharings $([\![x]\!], [\![y]\!], [\![z]\!])$, where $x, y \xleftarrow{\$} \mathbb{Z}_p$ and $z = x \cdot y$. Our PCG outputs a single multiplication triple over the ring R_p, for $n = 2$ parties, together with additive shares of the MAC key $\alpha \in \mathbb{Z}_p$. When using the fully-reducible variant of ring-LPN, this is equivalent to N triples over $\mathbb{F}_{p^d}$ (where for suitably chosen p we can have $d = 1$).

PCG construction. The construction, given in Fig. 2, is remarkably simple. Recall that our previous construction for OLE uses FSS keys which are expanded into shares of sparse polynomials $u_{i,j} = e_i \cdot e_j \in \mathbb{Z}_p[X]$. The FSS payload was defined by some (column) vector $\vec{v} \in \mathbb{Z}_p^{t^2}$, which defines the t^2 values of the non-zero coefficients in $u_{i,j}$. We can modify this to produce *authenticated OLE* by extending the FSS range from $\mathbb{Z}_p$ to $\mathbb{Z}_p^2$, and letting the payload be $\vec{v} \cdot (1, \alpha) \in \mathbb{Z}_p^{2N \times 2}$, for a random $\alpha \in \mathbb{Z}_p$. Evaluating the FSS keys at some input k now produces shares of $(\vec{v}[k], \alpha \cdot \vec{v}[k])$. Hence, these can be used to obtain authenticated shares of $x_0 \cdot x_1$, as well as the OLE.

To extend the above to authenticated triples, the seed generation phase will now produce three sets of FSS keys. The first two sets, $(K^i_{x,0}, K^i_{x,1})$ and $(K^i_{y,0}, K^i_{y,1})$, are used to compress shares of the $2c$ sparse polynomials defined by $(A^i_0, \vec{b}^i_0)$ and $(A^i_1, \vec{b}^i_1)$. These have sparsity t, so can be compressed using t-point SPFSS, and are later expanded to produce shares and MAC shares for R_p elements x and y. The third set, $(K^i_{z,0}, K^i_{z,1})$, compresses pairwise products of the previous sparse polynomials, so each of these can be defined using t^2-point SPFSS, as in the previous construction. This gives the shares and MAC shares for the product term $z = x \cdot y$.

We omit the proof of the following theorem, which is very similar to that of Theorem 4.1. Recall that to achieve exponential security against the best known attacks on R^c-LPN, it is enough to choose $ct = O(\lambda)$, therefore choosing a larger c allows to decrease the size of t. For more details on concrete parameter choices we refer to Section 9.

Construction G_{triple}

Parameters: Security parameter λ, noise weight $t = t(\lambda)$, compression factor $c \geq 2$, modulus $p = p(\lambda)$, degree $N = N(\lambda)$, and the ring $R_p = \mathbb{Z}_p[X]/F(X)$ for degree-N $F(X) \in \mathbb{Z}_p[X]$.
An FSS scheme ($\mathsf{SPFSS.Gen}, \mathsf{SPFSS.FullEval}$) for sums of t^2 point functions, with domain $[0..2N-1)$ and range $\mathbb{Z}_p^2$.
Public input: random polynomials $a_1, \ldots, a_{c-1} \in R_p$, used for R^c-LPN.
Correlation: Authenticated triples $(\llbracket x \rrbracket, \llbracket y \rrbracket, \llbracket z \rrbracket)$, satisfying $z = x \cdot y \in R_p$, and MAC key shares $\alpha_0, \alpha_1 \in \mathbb{Z}_p$.

Gen: On input 1^λ:

1. Sample $\alpha_0, \alpha_1 \xleftarrow{\$} \mathbb{Z}_p$ and let $\alpha = \alpha_0 + \alpha_1$.
2. For $\sigma \in \{0,1\}$ and $i \in [0..c)$, sample random vectors $A^i_\sigma \leftarrow [0..N)^t$ and $\vec{b}^i_\sigma \leftarrow (\mathbb{Z}_p^*)^t$.
3. Sample the following FSS keys:
   - $(K^i_{x,0}, K^i_{x,1}) \xleftarrow{\$} \mathsf{SPFSS.Gen}(1^\lambda, A^i_0, \vec{b}^i_0 \cdot (1, \alpha))$, for $i \in [0..c)$
   - $(K^i_{y,0}, K^i_{y,1}) \xleftarrow{\$} \mathsf{SPFSS.Gen}(1^\lambda, A^i_1, \vec{b}^i_1 \cdot (1, \alpha))$, for $i \in [0..c)$
   - $(K^{i,j}_{z,0}, K^{i,j}_{z,1}) \xleftarrow{\$} \mathsf{SPFSS.Gen}(1^\lambda, A^i_0 \boxplus A^j_1, (\vec{b}^i_0 \otimes \vec{b}^j_1) \cdot (1, \alpha))$, for $i, j \in [0..c)$
4. Let $\mathsf{k}_\sigma = \Big(\alpha_\sigma, (K^i_{x,\sigma}, K^i_{y,\sigma})_{i \in [0..c)}, (K^{i,j}_{z,\sigma})_{i,j \in [0..c)}\Big)$.
5. Output $(\mathsf{k}_0, \mathsf{k}_1)$.

Expand: On input $(\sigma, \mathsf{k}_\sigma)$, where $\mathsf{k}_\sigma = \Big(\alpha_\sigma, (K^i_{x,\sigma}, K^i_{y,\sigma})_{i \in [0..c)}, (K^{i,j}_{z,\sigma})_{i,j \in [0..c)}\Big)$:

1. Compute the vectors $\vec{u}_\sigma, \vec{v}_\sigma, \vec{w}_\sigma$ and $\vec{u}'_\sigma, \vec{v}'_\sigma, \vec{w}'_\sigma$ as follows:
   - $u_{\sigma,i}, u'_{\sigma,i} \leftarrow \mathsf{SPFSS.FullEval}(\sigma, K^i_{x,\sigma})$, for $i \in [0..c)$
   - $v_{\sigma,i}, v'_{\sigma,i} \leftarrow \mathsf{SPFSS.FullEval}(\sigma, K^i_{y,\sigma})$, for $i \in [0..c)$
   - $w_{\sigma,i+cj}, w'_{\sigma,i+cj} \leftarrow \mathsf{SPFSS.FullEval}(\sigma, K^{i,j}_{z,\sigma})$, for $i, j \in [0..c)$

   viewing each $\mathsf{FullEval}$ output as a pair of degree $< 2N$ polynomials over $\mathbb{Z}_p$.
2. Compute
   $$x_\sigma = \langle \vec{a}, \vec{u}_\sigma \rangle, y_\sigma = \langle \vec{a}, \vec{v}_\sigma \rangle, z_\sigma = \langle \vec{a} \otimes \vec{a}, \vec{w}_\sigma \rangle \quad \text{and}$$
   $$m_{x,\sigma} = \langle \vec{a}, \vec{u}'_\sigma \rangle, m_{y,\sigma} = \langle \vec{a}, \vec{v}'_\sigma \rangle, m_{z,\sigma} = \langle \vec{a} \otimes \vec{a}, \vec{w}'_\sigma \rangle$$
   all modulo $F(X)$.
3. Output $(\alpha_\sigma, x_\sigma, y_\sigma, z_\sigma, m_{x,\sigma}, m_{y,\sigma}, m_{z,\sigma})$

Figure 2: PCG for authenticated triples over the ring R_p, based on Ring-LPN

Theorem 4.2 *Suppose that* SPFSS *is a secure FSS scheme for sums of point functions (Definition 2.3), and the* R^c-$\mathsf{LPN}_{R_p,1,t}$ *assumption (Definition 3.2) holds. Then the construction in Fig. 2 is a secure PCG for two-party authenticated multiplication triples over* R_p.

When instantiating SPFSS using from $\mathsf{PRG} : \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$ via the PRG-based DPF construction from [BGI16b], we have a secure PCG for two-party authenticated multiplication triples over R_p with the following complexities:

- Each party's seed has size at most $2(2ct+(ct)^2)\cdot((\lceil \log N\rceil+1)\cdot(\lambda+2)+\lambda+\lceil \log p\rceil)+\lceil \log p\rceil$ bits.
- The computation of Expand can be done with at most $(8+4\lfloor(\log p)/\lambda\rfloor)N(2ct+(ct)^2)$ PRG operations, and $O(c^2 N \log N)$ operations in $\mathbb{Z}_p$.

As with the PCG for OLE, when using ring-LPN with regular errors the seed size can be reduced, replacing $\lceil \log N\rceil + 1$ in the formula with $\log(2N/t)$, while also reducing the number of PRG operations by a factor t.

5 DPF Key Generation Protocols

Up to this point, the exposition has focused on how to obtain and use pseudorandom correlation generators (PCG), abstracted in an idealized model where the short PCG seeds are sampled by a third-party trusted dealer. In this section, we give the preliminaries that will be necessary to securely set up these seeds. In particular, we present a protocol for setting up keys for distributed point functions with malicious security (allowing some leakage on the path value). Since these protocol works over any finite field, we will present it over $\mathbb{F}_q$ for arbitrary $q \in \mathbb{N}$.

5.1 Reactive 2-PC

In the following, we assume secure computation of simple operations over $\mathbb{F}_2^\ell$ and $\mathbb{F}_q$, which are depicted in Functionality $\mathcal{F}_{\mathsf{2\text{-}PC}}$ and - for the malicious setting - in Functionality $\mathcal{F}_{\mathsf{ext\text{-}2\text{-}PC}}$ in Figure 3. Note that the arithmetic addition **BitAdd** is nontrivial, as the parties must hold *bitwise* additive secret shares, but the sum itself is over $\mathbb{Z}$. This "grade school addition" over bits can be implemented via a binary circuit for integer addition with $\log N$ AND gates, similar to previous (e.g., garbled circuit based [KSS09]) protocols. For details on implementation and efficiency considerations we refer to Section 6.3.

5.2 Semi-honest DPF Key Generation

Recall that a distributed point function (DPF) allows to generate succinct shares of the point function $f\colon [0..D) \to \mathbb{F}_q$,

$$f_{\alpha,\beta}(x) = \begin{cases} \beta & \text{if } x = \alpha \\ 0 & \text{else} \end{cases}.$$

We give the functionality for setting up this succinct shares securely in Figure 4. We use the DPF construction of [BGI16b] and implement the functionality $\mathcal{F}_{\mathsf{DPF}}$ using the protocol of [Ds17] (note that the optimized DPF construction used in [Ds17] is not secure, see Remark 5.1 for details). In the following we give an overview of the underlying DPF construction and the distributed setup protocol.

The DPF construction [BGI16b]: We start by outlining the underlying DPF construction.

Parameters: Security parameter 1^λ, a natural number D specifying the domain size, and a pseudorandom generator $\mathsf{PRG}\colon \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$.

Functionality $\mathcal{F}_{\mathsf{2\text{-}PC}}$

The functionality operates on elements of $\mathbb{F}_q$ for $q \in \mathbb{N}$ and bit-strings $\mathbb{F}_2^\ell$ for $\ell \in \mathbb{N}$. Each value stored by the functionality is associated with a unique identifier that is given to all parties. Let $[\![y]\!]_{\mathbb{F}_q}$ denote the identifier for a value $x \in \mathbb{F}_q$ and $[\![x]\!]_{\mathbb{F}_2^\ell}$ denote the identifier for a bit-string $x \in \mathbb{F}_2^\ell$ that is stored by the functionality. Note that for stored bit-strings $[\![x]\!]_{\mathbb{F}_2^\ell}$ we assume individual access to the i-th bit $[\![x_i]\!]_{\mathbb{F}_2}$ for all $i \in [0..\ell)$.

Input(P_σ, x)**:** Receive a value $x \in \mathbb{F}_q$ or $x \in \mathbb{F}_2^\ell$ or from party P_σ and store $[\![x]\!]_{\mathbb{F}_q}$ or $[\![x]\!]_{\mathbb{F}_2^\ell}$.

Add$([\![x]\!]_{\mathbb{F}_q}, [\![y]\!]_{\mathbb{F}_q})$**:** Compute $z = x + y \in \mathbb{F}_q$ and store $[\![z]\!]_{\mathbb{F}_q}$.

Add$([\![x]\!]_{\mathbb{F}_2^\ell}, [\![y]\!]_{\mathbb{F}_2^\ell})$**:** (for $x, y \in \mathbb{F}_2^\ell$) Compute $z = x \oplus y \in \mathbb{F}_2^\ell$ and store $[\![z]\!]_{\mathbb{F}_2^\ell}$.

BitAdd$([\![x]\!]_{\mathbb{F}_2^\ell}, [\![y]\!]_{\mathbb{F}_2^\ell})$**:** (for $x, y \in \mathbb{F}_2^\ell$) Compute $z = x + y \in \{0,1\}^{\ell+1}$ via arithmetic addition and store $[\![z]\!]_{\mathbb{F}_2^{\ell+1}}$.

Mult$([\![x]\!]_{\mathbb{F}_q}, [\![y]\!]_{\mathbb{F}_q})$**:** Compute $z = x \cdot y \in \mathbb{F}_q$ and store $[\![z]\!]_{\mathbb{F}_q}$.

Output$([\![x]\!]_{\mathbb{F}_q})$**:** Send the value $x \in \mathbb{F}_q$ to all parties.

Output$([\![x]\!]_{\mathbb{F}_2^\ell})$**:** Send the value $x \in \mathbb{F}_2^\ell$ to all parties.

Functionality $\mathcal{F}_{\mathsf{ext\text{-}2\text{-}PC}}$

The functionality contains all the same commands as $\mathcal{F}_{\mathsf{2\text{-}PC}}$, as well as the additional commands below.
In the following, x_σ is always the input of party P_σ.

Inv$([\![x]\!]_{\mathbb{F}_q})$**:** Compute $z = x^{-1} \in \mathbb{F}_q$ and store $[\![z]\!]_{\mathbb{F}_q}$.

MixedMult$(x_0, x_1, [\![\alpha]\!]_{\mathbb{F}_2})$**:** (for $x_0, x_1 \in \mathbb{F}_2^\ell$, $\alpha \in \mathbb{F}_2$) Compute $z = \alpha \cdot (x_0 \oplus x_1) \in \mathbb{F}_2^\ell$.

1. *If both parties are honest:* Choose $z_0 \xleftarrow{\$} \mathbb{F}_2^\ell$ at random and set $z_1 = z \oplus z_0$. Output z_σ to P_σ for $\sigma \in \{0,1\}$.
2. *If P_σ is corrupt:* Wait for input z_σ by P_σ, set $z_{1-\sigma} = z \oplus z_\sigma$ and output $z_{1-\sigma}$ to $P_{1-\sigma}$.

PassiveOutput(x_0, x_1)**:** (for $x_0, x_1 \in \mathbb{F}_2^\ell$)

1. *If both parties are honest:* Wait for input $x_0, x_1 \in \mathbb{F}_2^\ell$. Output $z = x_0 \oplus x_1 \in \mathbb{F}_2^\ell$ to both parties.
2. *If P_σ is corrupt:* Send $x_{1-\sigma}$ to P_σ and wait for input $x_\sigma \in \mathbb{F}_{2^\ell}$ by P_σ. Send $z = x_0 \oplus x_1 \in \mathbb{F}_2^\ell$ to $P_{1-\sigma}$.

Figure 3: Functionality $\mathcal{F}_{\mathsf{2\text{-}PC}}$ and extended functionality $\mathcal{F}_{\mathsf{ext\text{-}2\text{-}PC}}$ for reactive 2-PC over $\mathbb{F}_q$ and $\mathbb{F}_2$

Functionality $\mathcal{F}_{\mathsf{DPF}}$

PARAMETERS: Security parameter 1^λ, distributed point function $\mathsf{DPF} = (\mathsf{DPF.Gen}, \mathsf{DPF.Eval})$ with domain $[0..D)$ and range $\mathbb{F}_q$, where $D, q \in \mathbb{N}$.
FUNCTIONALITY: The functionality contains the same **Input**, **Add**, **Mult** and **Output** commands as $\mathcal{F}_{\mathsf{2\text{-}PC}}$, as well as the following command.
DPF: On input $[\![\alpha]\!]_{\mathbb{F}_2^\ell}$, and $[\![\beta]\!]_{\mathbb{F}_q}$:

1. Sample keys $(K_0^{\mathsf{dpf}}, K_1^{\mathsf{dpf}}) \leftarrow \mathsf{DPF.Gen}(1^\lambda, \alpha, \beta)$.
2. For $\sigma \in \{0,1\}$ output K_σ^{dpf} to P_σ.

Figure 4: Functionality for semi-honest setup of a distributed point function. Here, we parse $\alpha \in [0..D)$ as bit-string $\alpha \in \{0,1\}^{\log_2 D}$, and assume to be given a secret sharing of α over $\mathbb{F}_2^{\log_2 D}$ as explained in Figure 3.

Input: A "path" $\alpha \in [0..D)$ (in the following usually interpreted as a bit-string $\in \{0,1\}^{\log D}$) and a "payload" $\beta \in \mathbb{F}_q$.

Goal: The parties hold a compact representation of (pseudorandom) shares $\vec{y}_0, \vec{y}_1 \in \mathbb{F}_q^D$, such that $\vec{y}_0 + \vec{y}_1 = (0, \dots, \beta, \dots, 0)$, where β is in the α-th position. In other words, parties P_0, P_1 hold an additive secret sharing modulo p of the α-th unit vector scaled by payload β.

Strategy:

1. Each party P_σ holds seeds $\vec{s}_\sigma^{\,0,0}, \vec{s}_\sigma^{\,0,1} \in \{0,1\}^\lambda$. These values define a $\log D$-depth tree following the GGM paradigm:
   To get from the i-th to $i+1$-st level, for all $j \in [0..2^i)$, P_σ computes
   $$\vec{s}_\sigma^{\,i+1,2j} \| \vec{s}_\sigma^{\,i+1,2j+1} \| t_\sigma^{i+1,2j} \| t_\sigma^{i+1,2j+1} \xleftarrow{\$} \mathsf{PRG}(\vec{s}_\sigma^{\,i,j}).$$
   Ignoring the t-values for now, the seeds $\vec{s}_\sigma^{\,0,0}, \vec{s}_\sigma^{\,0,1}$ can be viewed as a compact representation of a length-D vector $(\vec{s}_\sigma^{\,\log D,0}, \dots, \vec{s}_\sigma^{\,\log D,D-1})$.
2. In order to achieve $\vec{s}_0^{\,\log D,j} = \vec{s}_1^{\,\log D,j}$ for all $j \in [0..D)\backslash\{\alpha\}$ (i.e. shares of a scaled unit vector), the idea is to *correct* the nodes leaving the position defined by α.
   If α was known to one party (say P_0), this could be achieved by giving a correction word $\mathsf{CW}^i := \vec{s}_0^{\,i,\alpha_0\alpha_1\dots\overline{\alpha_i}} \oplus \vec{s}_1^{\,i,\alpha_0\alpha_1\dots\overline{\alpha_i}}$ (i.e. the node leaving the path corresponding to α in the i-th level) to party P_0 in every level. In the last level the party additionally receives $\mathsf{CW} := (s_0^{\log D,\alpha} - s_1^{\log D,\alpha})^{-1} \cdot \beta$, where $s_\sigma^{\log D,\alpha}$ corresponds to the value $\vec{s}_\sigma^{\,\log D,\alpha}$ interpreted as a bit string. (Note that this hides β, as party P_0 as no information on any of the preceding nodes of $\vec{s}_1^{\,\log D,\alpha}$).
   Adding CW^i at position $\alpha_0\alpha_1\dots\overline{\alpha_i}$ in the i-th level and interpreting the last values in the last level of the GGM tree as $\mathbb{F}_q$ elements, where P_1 multiplies its shares by -1 and P_0 multiplies the value at the α's position by CW, the parties indeed obtain output shares of the α-th unit vector scaled by β.
3. In the general setting, neither party has knowledge of α. This is, when the t-values come into play. The idea is as follows: For each node $\vec{s}_\sigma^{\,i,j}$ at level i, the parties will hold a value $T_\sigma^{i,j}$ (derived from their t-values together with correction bits $\tau^{i,0}, \tau^{i,1}$ hold by both parties) determining whether party P_σ adds the correction word or not, such that
   (a) for children of nodes *outside the path corresponding to* α, either none or both of the parties add the correction word

(b) for children of nodes *on the path corresponding to* α, exactly one of the parties adds the correction word.

As the parties do not know about the others party's $T_\sigma^{i,j}$ value, α is not leaked by the correction words or bits.

Remark 5.1 (Difference compared with [Ds17]) *[Ds17] actually use a modification of the above description, since they choose each bit $t_\sigma^{i,j}$ as the LSB of $\vec{s}_\sigma^{i,j}$, instead of as a fresh output from the PRG. We observe that unfortunately, this tweak renders their construction insecure, since the same bits are then used in computation of the correction words CW^i. This has the effect of leaking the bit α_i whenever $\mathsf{LSB}(\vec{s}_0^{i,\alpha_0\ldots\alpha_{i-1}0} \oplus \vec{s}_1^{i,\alpha_0\ldots\alpha_{i-1}0}) \neq \mathsf{LSB}(\vec{s}_0^{i,\alpha_0\ldots\alpha_{i-1}1} \oplus \vec{s}_1^{i,\alpha_0\ldots\alpha_{i-1}1})$. This occurs with probability $1/2$, so on average leaks half of the bits of α, given just one of the two seeds.*[7] *To avoid this issue, we instead apply the setup protocol of [Ds17] to the original DPF of [BGI16b].*

The protocol of Doerner and shelat. The setup protocol of Doerner and shelat [Ds17] is based on the observation that the correction words in the i-th level can be derived from the *sums* of the values of all left (resp. right) leaves in level i, if $\alpha_i = 1$ (resp. $\alpha_i = 0$). This is due to the fact that all left (resp. right) leaves that are not a child of node $\alpha|_{i-1}$ in level $i-1$ already agree due to previous corrections. Their protocol can be described as follows:

Input: The position $\alpha \in [0..D)$ (in bit-representation) and the payload $\beta \in \mathbb{F}_q$.

Setup: Each party (locally) chooses random seeds $\vec{s}_\sigma^{0,0}, \vec{s}_\sigma^{0,1} \in \{0,1\}^\lambda$

Key generation: For each $i \in [0..D)$ the parties input the sums of their right leaves and the sums of their left leaves (as well as the sums of their right t-values and their left t-values) into a secure computation. The secure computation outputs a correction word CW^i (computed as the sum of either all left leaves or all right leaves, depending on the i-th bit of alpha) and correction bits $\tau^{i,0}, \tau^{i,1}$. The parties correct the nodes of the i-th level by adding CW^i to the j-th word whenever $T_\sigma^{i,j} = 1$. Further, both parties derive the values $T_\sigma^{i+1,2j}, T_\sigma^{i+1,2j+1}$ for the next level using the correction bits. In the last level, each party additionally inputs the sum of all nodes, from which the final correction word CW can be derived.

Working Over a Ring Instead of a Field. Although we described the above protocol only over finite fields, note that it can in fact be adapted to work over $\mathbb{Z}_q$ for any $q \in \mathbb{N}$.

5.3 Malicious DPF Key Generation

We now turn focus to malicious adversaries. When transferring the protocol of Doerner and shelat to the malicious setting, one runs into the following problem: In order to achieve a practically efficient protocol, the evaluation of the PRG has to take place locally, i.e. outside any secure evaluation. But this would allow a malicious adversary to provide inconsistent key shares during key generation.

Previous works [BCG+19a, YWL+20] solve this issue at very low cost for the simpler setting of puncturable pseudorandom functions, where one party is in knowledge of the position (say P_0) and the other party in charge of setting up the key (say P_1). The idea is to introduce a consistency check that ensures that P_1 behaved consistently *with respect to* the input value α. As the check depends on the input value α if P_1 deviates from the protocol, party P_1 can attempt guess partial information on α. If P_1 guesses correctly, it obtains (partial) leakage on α. Otherwise, the protocol aborts.

[7] In the notation of [Ds17, Fig. 1], whenever $\tau^{j,0} = \tau^{j,1} \oplus 1$, you can read off α_j by XORing $\tau^{j,1}$ with $\mathsf{LSB}(\sigma^j)$.

Functionality $\mathcal{F}_{\text{c-SUV}}$

Parameters: Length $D \in \mathbb{N}$, and a prime modulus p

Functionality: The functionality contains the same **Input**, **Add**, **Mult** and **Output** commands as $\mathcal{F}_{\text{2-PC}}$, as well as the following command.

SUV: On input $[\![\alpha]\!]_{\mathbb{F}_2^{\log D}}$ and $[\![\beta]\!]_{\mathbb{F}_q}$:

If both parties are honest:

1. If $\beta = 0$ output "$\beta = 0$" to both parties and abort.
2. Sample $\vec{y}_0 \xleftarrow{\$} \mathbb{F}_q^D$, and let $\vec{y}_1 \leftarrow (0, \ldots, 0, \beta, \ldots, 0) - \vec{y}_0$, where β is in position α (parsed as integer).
3. Output $\vec{y}_\sigma$ to party P_σ, for $\sigma \in \{0, 1\}$

If party P_σ is corrupted:

1. **Allow the adversary to determine its output shares.** Wait for input $\vec{y}_\sigma \in \mathbb{F}_q^D$ from the adversary.
2. **Allow the adversary an arbitrary guess on α.** Wait for input $P \colon \mathbb{F}_2^{\log D} \to \{0, 1\}$. If $P(\alpha) = 0$, abort.
3. If $\beta = 0$ output "$\beta = 0$" to both parties and abort.
4. Set $\vec{y}_{1-\sigma} \leftarrow (0, \ldots, 0, \beta, \ldots, 0) - \vec{y}_\sigma \in \mathbb{F}_q^D$, where β is in position α (parsed as integer).
5. Output (`success`) to the adversary and $\vec{y}_{1-\sigma}$ to the honest party.

Figure 5: Functionality for the malicious distributed setup of a scaled unit vector

We show how to achieve the same in the more general setting of distributed point functions. We do this using a new consistency check for verifying the DPF keys were computed correctly. This check, which is inspired by [YWL+20], has a similar leakage profile to previous approaches for puncturable PRFs.[8] Compared with the semi-honest protocol, we need a small extra overhead, dominated by two maliciously secure 2-PC multiplications and one inversion in $\mathbb{F}_q$, as well as around twice as many PRG evaluations as in the semi-honest case.

In Figure 5 we outline the achieved functionality. Note that this is weaker than the functionality for securely generating keys of a distributed point function for two reasons:

- For the reason elaborated on above, it allows some leakage on the noise α. In the setting of noise generation for LPN, this leakage is tolerable for the following reason: The more the adversary attempts to guess, the higher the probability the protocol aborts, resulting in a leakage of only 1 bit on average. Intuitively, this can be accounted for by slightly increasing the noise rate.
- It gives out additive secret shares of the output (instead of the DPF keys), where the adversary is given full control over its share of the input. To make the resulting restriction to polynomial-size output explicit, we refer to the functionality as a setup functionality for a *shared unit vector*. Note that combined with our PCGs, this allows implementing the corruptible OLE functionality (Fig. 9) and the corruptible functionality for generating

[8]The original version of this paper used a hash-based consistency check similar to [BCG+19a], however, as observed by Damiano Abram, that check was not sufficient in this setting.

authenticated multiplication triples (Fig. 11), where the adversary is also given control over its share of the output.

Intuition for the Protocol (Fig. 6). The protocol uses the malicious 2-PC functionality $\mathcal{F}_{\mathsf{ext\text{-}2\text{-}PC}}$ (Fig. 3), and requires that α is stored in $\mathcal{F}_{\mathsf{ext\text{-}2\text{-}PC}}$ bitwise, while β is stored as an element of $\mathbb{F}_q$. Next, it starts by running the semi-honest setup protocol, with the difference that in the secure computation used to compute the correction words CW^i, we rely on $\mathcal{F}_{\mathsf{ext\text{-}2\text{-}PC}}$ to ensure that the correct bits of α are being used. Apart from this, the secure computation stage is only semi-honest: the **PassiveOutput** command of $\mathcal{F}_{\mathsf{ext\text{-}2\text{-}PC}}$ allows corrupt parties to incorrectly open a shared value, while **MixedMult** only enforces that the bit α_i is correct, and not the string it is multiplied with.

The correction words computed in this first stage correspond to a random payload β'. Before switching this to β, we perform a consistency check. The parties sample (via coin-tossing) random values $r^0, \ldots, r^{D-1} \in \mathbb{F}_q$, and use these to compute a linear combination of the DPF outputs. The result (which is secret-shared) should equal $r^\alpha \cdot \beta'$. To check this, the parties take an additional D outputs of the DPF (by extending the depth of the tree by one), with associated random payload CW^R, and take the same linear combination, scaled by $(\mathsf{CW}^R)^{-1}$, and multiply the result with β' (which has been stored in $\mathcal{F}_{\mathsf{2\text{-}PC}}$). The parties then use $\mathcal{F}_{\mathsf{2\text{-}PC}}$ to verify that these two values are equal.

The idea of the check is that the only way a corrupt party can cheat is by guessing β', which is random in a large field $\mathbb{F}_q$, or by guessing (a few bits of) α, which is allowed by the leakage in the functionality. If the check goes through, the parties finally correct the DPF payload to β instead of β', by outputting the relevant correction value $\mathsf{CW}^L = \beta/\beta'$.

Remark 5.2 *Note that one can use the functionality $\mathcal{F}_{\mathsf{c\text{-}SUV}}$ with larger output spaces $\mathbb{F}_q^\ell$, by running the same functionality over $\mathbb{F}_{q^\ell}$ and embedding $\mathbb{F}_q^\ell$ vectors into that field. Alternatively, if q is already large enough for security, a slightly more efficient solution can be obtained by tweaking the protocol as follows. Choose a PRG with larger output length $(\{0,1\}^\lambda)^{\ell+1}$ on the last level, compute a correction word CW_i^L for each $i \in [\ell]$, and check consistency for each component individually. More precisely, for each $i \in [\ell]$, compute $S_\sigma^{L,i}$ similarly to S_σ^L in Fig. 6, using the additional PRG outputs, and check that $\beta' \cdot S^R - S^{L,i} = 0$ as required.*

In the following, we prove that our protocol $\Pi_{\mathsf{c\text{-}SUV}}$ (see Fig. 6) realizes the functionality $\mathcal{F}_{\mathsf{c\text{-}SUV}}$ (Fig. 5) for generating additive secret shares of a scaled unit vector with security against malicious adversaries.

Theorem 5.1 *If $\mathsf{PRG}\colon \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$ is a secure PRG , then the protocol $\Pi_{\mathsf{c\text{-}SUV}}$ (Fig. 6) implements the functionality $\mathcal{F}_{\mathsf{c\text{-}SUV}}$ (Fig. 5) with security against malicious adversaries in the $(\mathcal{F}_{\mathsf{ext\text{-}2\text{-}PC}}, \mathcal{F}_{\mathsf{coin}})$-hybrid model.*

Proof. We first consider the case that both parties are honest, before giving a simulator for the case that one party is corrupt.

Both parties are honest. We have to show that the outputs in a real execution of the protocol are indistinguishable from the outputs of the ideal functionality (i.e. $\vec{y}_0 \overset{\$}{\leftarrow} \mathbb{F}_q^D$, $\vec{y}_1 \leftarrow (0, \ldots, \beta, \ldots, 0) - \vec{y}_0$, where β is in position α), even to an adversary seeing all communication over the network.

Correctness. We have to prove that indeed $y_0^j + y_1^j = 0$ for all $j \neq \alpha$ and $y_0^\alpha + y_1^\alpha = \beta$. In particular, we have to show that in an honest execution the protocol does not abort. We proceed in a number of steps.

Protocol $\Pi_{\mathsf{c\text{-}SUV}}$ *(Part I)*

Parameters:

- Security parameter 1^λ; output length $D = 2^k$; finite field $\mathbb{F}_q$ (where $\log q$ is a statistical security parameter)
- $\mathsf{PRG}\colon \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$, a pseudorandom generator
- $\mathsf{Convert}_{\mathbb{F}_q} : \{0,1\}^\lambda \to \mathbb{F}_q$, maps a pseudorandom bit string into an $\mathbb{F}_q$ element
- Functionalities $\mathcal{F}_{\mathsf{ext\text{-}2\text{-}PC}}$ (Fig. 3) and $\mathcal{F}_{\mathsf{coin}}$ (coin-tossing)

Protocol: The $\boxed{\text{framed}}$ parts are run in malicious 2-PC (using commands from $\mathcal{F}_{\mathsf{ext\text{-}2\text{-}PC}}$).

Inputs:

- The parties holds a position $[\![\alpha]\!]_{\mathbb{F}_2^{\log D}}$ and a payload $[\![\beta]\!]_{\mathbb{F}_q}$ stored in $\mathcal{F}_{\mathsf{ext\text{-}2\text{-}PC}}$. We assume α is shared bitwise, so for each bit α_j, for $j \in [0..\log D)$, party P_σ holds $\alpha_{j,\sigma}$ with $\alpha_{j,0} + \alpha_{j,1} = \alpha_j$.

Key Generation Phase:

1. For $\sigma \in \{0,1\}$ party P_σ chooses a PRG seed $\vec{s}_\sigma^{\,0,0} \in \{0,1\}^\lambda$ and computes $\vec{s}_\sigma^{\,1,0}\|\vec{s}_\sigma^{\,1,1}\|t_\sigma^{1,0}\|t_\sigma^{1,1} := \mathsf{PRG}(\vec{s}_\sigma^{\,0,0})$. Party P_σ locally sets $T_\sigma^{1,0} := T_\sigma^{1,1} := \sigma$, and $\vec{z}_\sigma^{\,1,0} := \vec{s}_\sigma^{\,1,0}, \vec{z}_\sigma^{\,1,1} := \vec{s}_\sigma^{\,1,1}, u_\sigma^{1,0} := t_\sigma^{1,0}, u_\sigma^{1,1} := t_\sigma^{1,1}$.

2. For $i = 1$ to $\log D$:
   - For $b \in \{0,1\}$: $\tau^{i,b} \leftarrow \textbf{PassiveOutput}(u_0^{i,b} \oplus \alpha_{i,0}, u_1^{i,b} \oplus \alpha_{i,1} \oplus 1 \oplus b)$
   - $(\vec{y}_0^{\,i}, \vec{y}_1^{\,i}) \leftarrow \textbf{MixedMult}(\vec{z}_0^{\,i,0} \oplus \vec{z}_0^{\,i,1}, \vec{z}_1^{\,i,0} \oplus \vec{z}_1^{\,i,1}, [\![\alpha_i]\!]_{\mathbb{F}_2})$ // $= \alpha_i \cdot (\vec{z}^{\,i,0} + \vec{z}^{\,i,1})$
   - $\mathsf{CW}^i \leftarrow \textbf{PassiveOutput}(\vec{y}_0 \oplus \vec{z}_0^{\,i,1}, \vec{y}_1^{\,i} \oplus \vec{z}_1^{\,i,1})$ // $\mathsf{CW}^i = \vec{z}^{\,i,1-\alpha_i}$
   - For $\sigma \in \{0,1\}$, $j \in [0..2^i)$, party P_σ computes:
     - The next level of the GGM tree:
     $$\vec{s}_\sigma^{\,i+1,2j}\|\vec{s}_\sigma^{\,i+1,2j+1}\|t_\sigma^{i+1,2j}\|t_\sigma^{i+1,2j+1} := \mathsf{PRG}(\vec{s}_\sigma^{\,i,j} \oplus T_\sigma^{i,j} \cdot \mathsf{CW}^i)$$
     - If $i < \log D$, the sums of the left/right leaves and left/right correction bits, for $b \in \{0,1\}$:
     $$\vec{z}_\sigma^{\,i+1,b} := \bigoplus_{j=0}^{2^i-1} \vec{s}_\sigma^{\,i+1,2j+b},\ u_\sigma^{i+1,b} := \bigoplus_{j=0}^{2^i-1} t_\sigma^{i+1,2j+b},$$
     - If $i < \log D$, the next level of choice bits:
     $$T_\sigma^{i+1,2j} := t_\sigma^{i+1,2j} \oplus T_\sigma^{i,j} \cdot \tau^{i,\mathsf{Parity(j)}}, \quad T_\sigma^{i+1,2j+1} := t_\sigma^{i+1,2j+1} \oplus T_\sigma^{i,j} \cdot \tau^{i,\mathsf{Parity(j)}}$$

3. Convert the output shares to $\mathbb{F}_q$: P_σ computes:
$$z_\sigma^L := \sum_{j=0}^{D-1} s_\sigma^{j,L}, \quad z_\sigma^R := \sum_{j=0}^{D-1} s_\sigma^{j,R}$$
where $s_\sigma^{j,L} = \mathsf{Convert}_{\mathbb{F}_q}(\vec{s}_\sigma^{\,\log D+1,2j})$ and $s_\sigma^{j,R} = \mathsf{Convert}_{\mathbb{F}_q}(\vec{s}_\sigma^{\,\log D+1,2j+1})$.

(to be continued)

Figure 6: Protocol for the malicious distributed setup of scaled unit vectors (Part I)

Protocol $\Pi_{\mathsf{c\text{-}SUV}}$ *(Part II)*

Key Generation Phase: *(continued)*

4. Correct the offset of the left leaves to β and the offset of the right leaves to 0:
   - For $\sigma \in \{0,1\}$: $[\![z_\sigma^L]\!]_{\mathbb{F}_q} \leftarrow \textbf{Input}(P_\sigma, z_\sigma^L)$, $[\![z_\sigma^R]\!]_{\mathbb{F}_q} \leftarrow \textbf{Input}(P_\sigma, z_\sigma^R)$
   - $[\![\beta']\!]_p \leftarrow [\![z_0^L]\!]_{\mathbb{F}_q} - [\![z_1^L]\!]_{\mathbb{F}_q}$
   - $[\![\mathsf{CW}^L]\!]_{\mathbb{F}_q} \leftarrow [\![\beta']\!]_{\mathbb{F}_q}^{-1} \cdot [\![\beta]\!]_{\mathbb{F}_q}$, $[\![\mathsf{CW}^R]\!]_{\mathbb{F}_q} \leftarrow [\![z_0^R]\!]_{\mathbb{F}_q} - [\![z_1^R]\!]_{\mathbb{F}_q}$
   - Output $\mathsf{CW}^R \leftarrow \textbf{Output}([\![\mathsf{CW}^R]\!]_{\mathbb{F}_q})$ to both parties.

Verification Phase:

6. To verify that the sender behaved consistently and the output unit vector has the correct payload β, the parties proceed as follows:
   - The parties call $\mathcal{F}_{\mathsf{coin}}$ to obtain public random values $r^0, \dots, r^{D-1} \in \mathbb{F}_q$.
   - Each P_σ computes

$$S_\sigma^L = (-1)^\sigma \cdot \sum_{j=0}^{D-1} r^j \cdot s_\sigma^{j,L}, \quad S_\sigma^R = (-1)^\sigma \cdot (\mathsf{CW}^R)^{-1} \cdot \sum_{j=0}^{D-1} r^j \cdot s_\sigma^{j,R}$$

7. The parties check if the check values where computed correctly:
   - For $\sigma \in \{0,1\}$: $[\![S_\sigma^L]\!]_{\mathbb{F}_q} \leftarrow \textbf{Input}(P_\sigma, S_\sigma^L)$, $[\![S_\sigma^R]\!]_{\mathbb{F}_q} \leftarrow \textbf{Input}(P_\sigma, S_\sigma^R)$
   - Compute $[\![Z]\!]_{\mathbb{F}_q} = [\![\beta']\!]_{\mathbb{F}_q} \cdot ([\![S_0^R]\!]_{\mathbb{F}_q} + [\![S_1^R]\!]_{\mathbb{F}_q}) - [\![S_0^L]\!]_{\mathbb{F}_q} - [\![S_1^L]\!]_{\mathbb{F}_q}$.
   - If $Z \leftarrow \textbf{Output}([\![Z]\!]_{\mathbb{F}_q})$ does not equal 0, abort.

Output Phase:

1. Output $\mathsf{CW}^L \leftarrow \textbf{Output}([\![\mathsf{CW}^L]\!]_{\mathbb{F}_q})$ to both parties. If $\mathsf{CW}^L = 0$, abort.

8. Finally, if all checks passed, party P_σ outputs $\vec{y}_\sigma = (y_\sigma^0, \dots, y_\sigma^D - 1)$, where

$$y_\sigma^j := (-1)^\sigma \cdot s_\sigma^{j,L} \cdot \mathsf{CW}^L.$$

Figure 6: Protocol for the malicious distributed setup of scaled unit vectors (Part II)

Claim 5.3 *If the parties follow the protocol description honestly, then all $i \in [0..\log D)$ and $j \in [0..2^i)\backslash\{\alpha|_i\}$ we have $\vec{s}_0^{i,j} \oplus T_0^{i,j} \cdot \mathsf{CW}^i = \vec{s}_1^{i,j} \oplus T_1^{i,j} \cdot \mathsf{CW}^i$. Further, for the correction bits we have $T_0^{i+1,2j+b} = T_1^{i+1,2j+b}$ for all $j \in [0..2^i) \setminus \{\alpha|_i\}, b \in \{0,1\}$, as well as $T_0^{i+1,2\alpha|_i+b} = T_1^{i+1,2\alpha|_i+b} \oplus 1$ for $b \in \{0,1\}$.*

We omit the proof of this claim, which closely follows from [BGI16b].

Claim 5.4 *After an honest execution of the protocol party P_σ holds output $(y_\sigma^0, \ldots, y_\sigma^{D-1})$, such that $y_0^j + y_1^j = 0$ for all $j \neq \alpha$, and $y_0^\alpha + y_1^\alpha = \beta$.*

Proof. For all $j \neq \alpha$, by Claim 5.3 we have

$$\vec{s}_\sigma^{\log D-1,j} \oplus T_\sigma^{\log D-1,j} \cdot \mathsf{CW}^{D-1} = \vec{s}_{1-\sigma}^{\log D-1,j} \oplus T_{1-\sigma}^{\log D-1,j} \cdot \mathsf{CW}^{D-1},$$

and thus $\vec{s}_0^{\log D,2j} = \vec{s}_1^{\log D,2j}$. This implies $s_0^{j,L} = s_1^{j,L}$, and thus $y_0^j + y_1^j = s_0^j, L \cdot \mathsf{CW}^L - s_1^j, L \cdot \mathsf{CW}^L = 0$

Further, we have

$$z_0^L - z_1^L = \sum_{j=0}^{D-1} s_0^{j,L} - \sum_{j=1}^{D-1} s_1^{j,L} = s_0^{\alpha,L} - s_1^{\alpha,L}$$

and thus $y_0^\alpha + y_1^\alpha = s_0^{\alpha,L} \cdot \mathsf{CW}^L - s_1^{\alpha,L} \cdot \mathsf{CW}^L = (s_0^{\alpha,L} - s_1^{\alpha,L}) \cdot (s_0^{\alpha,L} - s_1^{\alpha,L})^{-1} \cdot \beta = \beta$. □

Claim 5.5 *If $\beta \neq 0$, then in an honest execution the protocol does not abort with overwhelming probability.*

Proof. First, note that $\beta \neq 0$ implies that $\mathsf{CW}^L \neq 0$. Further, by the same reasoning as above, 5.3 yields $\vec{s}_0^{\log D,2j+1} = \vec{s}_1^{\log D,2j+1}$, and thus $s_0^{j,R} = s_1^{j,R}$, for all $j \in [0..D)\backslash\{\alpha\}$. This implies $\mathsf{CW}^R = z_0^R - z_1^R = s_0^{\alpha,R} - s_1^{\alpha,R}$. This implies

$$\begin{aligned} S_0^R + S_1^R &= (\mathsf{CW}^R)^{-1} \cdot \sum_{j=0}^{D-1} r^j \cdot (s_0^{j,R} - s_1^{j,R}) \\ &= (s_0^{\alpha,R} - s_1^{\alpha,R})^{-1} \cdot r^\alpha \cdot (s_0^{\alpha,R} - s_1^{\alpha,R}) = r^\alpha. \end{aligned}$$

Further, we have

$$\begin{aligned} S_0^L + S_1^L &= \sum_{j=0}^{D-1} r^j \cdot (s_0^{j,L} - s_1^{j,L}) \\ &= r^\alpha \cdot (s_0^{\alpha,L} - s_1^{\alpha,L}) \\ &= r^\alpha \cdot \beta' \end{aligned}$$

which yields $Z = 0$ as required.

□

Security. It is left to show that $\vec{y}_\sigma$ is distributed uniformly for each $\sigma \in \{0,1\}$ individually, even given access to all communication between P_0 and P_1. As an honest execution of our protocol is very similar to an execution of the original protocol of Doerner and shelat (instantiated with the DPF of [BGI16b]), in the following we only give a proof sketch. Note that even conditioned on all correction words and correction bits revealed during the protocol, i.e. $\mathsf{CW}^i, \tau^{i,0}, \tau^{i,1}$ for all $i \in [0..D)$ and CW^R, it holds that $\vec{y}_\sigma$ is distributed uniformly at random, because each of the correction values is blinded by an output of the PRG, where the input is known only to $P_{1-\sigma}$ (and in particular not revealed during the protocol execution). Further, if $\beta \neq 0$, the same holds true conditioned on CW^L. Finally, note that the only value that an adversary sees in the verification step of an honest execution is a 0, therefore security follows.

Party P_σ is corrupted. We now proceed to the case that one party is corrupted. We start by giving an intuition for the proof. In the simulation, all the correction values $\tau^{i,b}, \mathsf{CW}^i$, and the right correction word CW^R, will be simulated using random values. In the consistency check, the simulator needs to extract a guess on $\alpha \in [D]$ made by the potentially cheating adversary. Since the simulator does not know α, it will define, for each $\chi \in [D]$, a simulated "honest view" that is consistent with $\alpha = \chi$ and the adversary's view so far. Using the S_σ^L, S_σ^R values provided by the adversary, the simulator then defines the predicate $P(\chi)$, which is 1 if the adversary's behaviour is consistent with the simulated honest view based on χ. It queries this guess to the functionality, if the guess is successful, the consistency check passes. The simulator can then use a consistent value of χ to extract a valid output vector that is known by the adversary. In the proof, we show that due to the random linear combination, every χ that satisfies $P(\chi) = 1$ will lead to the same extracted output, so the simulation is consistent with the real protocol.

The simulator proceeds as follows.

Key generation phase:
- For $i \in [1..\log D]$:
  - The simulator sends a random $\tau_{1-\sigma}^{i,b} \in \{0,1\}$ to P_σ, and waits to receive $\tau_\sigma^{i,b}$, then defines $\tau^{i,b} = \tau_0^{i,b} \oplus \tau_1^{i,b}$.
  - Next, the simulator awaits input $\vec{v}_\sigma^i \in \{0,1\}^\lambda$ to **MixedMult** from the adversary, as well as the adversary to input P_σ's output share $\vec{y}_\sigma^i \in \{0,1\}^\lambda$.
  - Next, to simulate **PassiveOutput** the simulator sends a random $\mathsf{CW}_{1-\sigma}^i \in \{0,1\}^\lambda$ to the adversary, then waits to receive $\mathsf{CW}_\sigma^i \in \{0,1\}^\lambda$, and sets $\mathsf{CW}^i := \mathsf{CW}_0^i \oplus \mathsf{CW}_1^i$.
- Now, the simulator awaits input $z_\sigma^L, z_\sigma^R \in \mathbb{F}_q$ from the adversary and returns a random $\mathsf{CW}^R \leftarrow \mathbb{F}_q$. It also samples a random $\mathsf{CW}^L \leftarrow \mathbb{F}_q$.
- Finally, for each $\chi \in [0..D)$ (corresponding to all possible values of α) the simulator computes the output of the honest party $P_{1-\sigma}$ that is compatible with the correction values and correction bits as sampled above.
  - The simulator sets $T_{1-\sigma,\chi}^{1,1-\chi_1} := 1 - \sigma$.
  - For $i \in [1..\log D]$:
    * For $b \in \{0,1\}$, the simulator sets $u_{1-\sigma,\chi}^{i,b} := \tau_{1-\sigma}^{i,b} \oplus \chi_i \oplus \alpha_{i,\sigma} \oplus (1 \oplus \sigma) \cdot (1 \oplus b)$.
    * The simulator sets
$$\vec{z}_{1-\sigma,\chi}^{i,1-\chi_i} := \mathsf{CW}_{1-\sigma}^i \oplus \vec{y}_\sigma^i \oplus \chi_i \cdot \vec{v}_\sigma^i.$$
  - For the first level, the simulator sets $\vec{s}_{1-\sigma,\chi}^{1,1-\chi_1} := \vec{z}_{1-\sigma,\chi}^{1,1-\chi_1}$.
  - For $i \in [1..\log D]$:
    * For $j \in [0..2^i)$, the simulator computes:
      · The next level of the GGM tree:
$$\vec{s}_{1-\sigma,\chi}^{i+1,2j} \| \vec{s}_{1-\sigma,\chi}^{i+1,2j+1} \| t_{1-\sigma,\chi}^{i+1,2j} \| t_{1-\sigma,\chi}^{i+1,2j+1} := \mathsf{PRG}(\vec{s}_{1-\sigma,\chi}^{i,j} \oplus T_{1-\sigma,\chi}^{i,j} \cdot \mathsf{CW}^i) \text{ if } j \neq \chi_i.$$
      · Next, if $i < \log D$ the simulator uses the pre-computed values $\vec{z}_{1-\sigma,\chi}^{i+1,1-\chi_{i+1}}$, $u_{1-\sigma,\chi}^{i+1,0}, u_{1-\sigma,\chi}^{i+1,1}$ to compute the missing part of the GGM tree – except for the seed value *on the χ-path*, $\vec{s}_{1-\sigma,\chi}^{i+1,\chi|_i+\chi_{i+1}}$ (where by $\chi|_i = \sum_{j=0}^{i-1} \chi_j 2^j$ we denote the i-bit prefix of χ parsed as a natural number), since we do not need this value to continue. Instead, the already chosen correction word of the next level (together with the other leave values) will implicitly define it.
$$\vec{s}_{1-\sigma,\chi}^{i+1,2\chi|_i+1-\chi_{i+1}} := \vec{z}_{1-\sigma,\chi}^{i+1,1-\chi_{i+1}} \oplus \bigoplus_{j=0,j\neq\chi|_i}^{2^i-1} \vec{s}_{1-\sigma,\chi}^{i+1,2j+1-\chi_{i+1}},$$

as well as

$$t_{1-\sigma,\chi}^{i+1,2\chi|_i+b} := u_{1-\sigma,\chi}^{i+1,b} \oplus \bigoplus_{j=0,j\neq\chi|_i}^{2^i-1} t_{1-\sigma,\chi}^{i+1,2j+b}.$$

· Further, if $i < \log D$, the simulator computes the next level of choice bits:

$$T_{1-\sigma,\chi}^{i+1,2j} := t_{1-\sigma,\chi}^{i+1,2j} \oplus T_{1-\sigma,\chi}^{i,j} \cdot \tau^{i,\mathsf{Parity(j)}}, \quad T_{1-\sigma,\chi}^{i+1,2j+1} := t_{1-\sigma,\chi}^{i+1,2j+1} \oplus T_{1-\sigma,\chi}^{i,j} \cdot \tau^{i,\mathsf{Parity(j)}}$$

- Note that the simulator cannot compute the honest party's output shares at position χ, since this would require knowledge of β. However, the simulator can compute the output share of the dishonest party P_σ, such that it is consistent with the shares of $P_{1-\sigma}$ (conditioned on $\alpha = \chi$).

$$s_{\sigma,\chi}^{\chi,L} := z_\sigma^L - \sum_{j=0,j\neq\chi}^{D-1} s_{\sigma,\chi}^{j,L}, \quad s_{\sigma,\chi}^{\chi,R} := z_\sigma^R - \sum_{j=0,j\neq\chi}^{D-1} s_{\sigma,\chi}^{j,R}$$

where $s_{\sigma,\chi}^{j,L} = \mathsf{Convert}_{\mathbb{F}_q}(\vec{s}_{1-\sigma,\chi}^{\log D+1,2j})$ and $s_{\sigma,\chi}^{j,R} = \mathsf{Convert}_{\mathbb{F}_q}(\vec{s}_{1-\sigma,\chi}^{\log D+1,2j+1})$ for $j \neq \chi$.

Verification phase:
- To simulate $\mathcal{F}_{\mathsf{coin}}$, the simulator sends random $r^0, \ldots, r^{D-1} \in \mathbb{F}_q$.
- Further, the simulator receives the "real" inputs S_σ^L, S_σ^R by the adversary.
- Next, for each $\chi \in [0..D)$ the simulator computes

$$S_{\sigma,\chi}^L = (-1)^\sigma \cdot \sum_{j=0}^{D-1} r^j \cdot s_{\sigma,\chi}^{j,L}, \quad S_{\sigma,\chi}^R = (-1)^\sigma \cdot (\mathsf{CW}^R)^{-1} \cdot \sum_{j=0}^{D-1} r^j \cdot s_{\sigma,\chi}^{j,R}$$

- For all $j \in [0..D)$ the simulator further computes the (potential) output as

$$y_{\sigma,\chi}^j := (-1)^\sigma \cdot s_{\sigma,\chi}^j, L \cdot \mathsf{CW}^L.$$

- To extract the adversary's potential guess on α, the simulator defines the predicate

$$P \colon \mathbb{F}_2^{\log D} :\to \{0,1\}, P(\chi) = 1 \Leftrightarrow S_{\sigma,\chi}^L = S_\sigma^L \wedge S_{\sigma,\chi}^R = S_\sigma^R,$$

picks a $\chi^\star$ with $P(\chi^\star) = 1$ and inputs $\vec{y}_{\sigma,\chi^\star} = (y_{\sigma,\chi^\star}^0, \ldots, y_{\sigma,\chi^\star}^{D-1})$ and P to the functionality. If such a $\chi^\star$ does not exist, the simulator sends $\vec{y}_\sigma = 0$ and $P = 0$ to the functionality.
  - If the functionality aborts, then the simulator receives α and sends to the adversary a random $Z \in \mathbb{F}_q^*$ and aborts.
  - Else, the simulator outputs $Z = 0$ to the adversary. It also learns from the functionality whether or not $\beta = 0$.

Output phase. Finally, we simulate the opening of the correction word CW^L. If $\beta = 0$, the simulator outputs 0 and aborts. Else, it sends CW^L as sampled before.

It is left to show that the simulation is indistinguishable from a real protocol execution. To this end, we first switch the real protocol distribution to a randomized distribution (Lemma 5.6), and then show that the simulation is statistically indistinguishable from the randomized distribution (Lemma 5.7).

Hybrid 1: Real behaviour of the honest party: We start by describing the honest party's behavior in a real execution, when the protocol is run with input α.

Key generation phase: • Party $P_{1-\sigma}$ chooses a PRG seed $\vec{s}^{0,0}_{1-\sigma} \in \{0,1\}^\lambda$ and computes $\vec{s}^{1,0}_{1-\sigma} \| \vec{s}^{1,1}_{1-\sigma} \| t^{1,0}_{1-\sigma} \| t^{1,1}_{1-\sigma} := \mathsf{PRG}(\vec{s}^{0,0}_{1-\sigma})$. Party $P_{1-\sigma}$ locally sets $T^{1,0}_{1-\sigma} := T^{1,1}_{1-\sigma} := 1 - \sigma$, and $\vec{z}^{1,0}_{1-\sigma} := \vec{s}^{1,0}_{1-\sigma}, \vec{z}^{1,1}_{1-\sigma} := \vec{s}^{1,1}_{1-\sigma}, u^{1,0}_{1-\sigma} := t^{1,0}_{1-\sigma}, u^{1,1}_{1-\sigma} := t^{1,1}_{1-\sigma}$.

- For $i = [1..\log D]$:
  - For $b \in \{0,1\}$, $P_{1-\sigma}$ sends $\tau^{i,b}_{1-\sigma} := u^{i,b}_{1-\sigma} \oplus \alpha_{i,1-\sigma} \oplus (1 \oplus \sigma) \cdot (1 \oplus b)$ to **PassiveOutput** and receives $\tau^{i,b}$.
  - $P_{1-\sigma}$ sends $\vec{z}^{i,0}_{1-\sigma} \oplus \vec{z}^{i,1}_{1-\sigma}$ to **MixedMult** and receives $\vec{y}^{i}_{1-\sigma}$.
  - $P_{1-\sigma}$ sends $\vec{y}^{i}_{1-\sigma} \oplus \vec{z}^{i,1}_{1-\sigma}$ to **PassiveOutput** and receives CW^i.
  - For $j \in [0..2^i)$, party $P_{1-\sigma}$ computes:
    - ∗ The next level of the GGM tree:

$$\vec{s}^{i+1,2j}_{1-\sigma} \| \vec{s}^{i+1,2j+1}_{1-\sigma} \| t^{i+1,2j}_{1-\sigma} \| t^{i+1,2j+1}_{1-\sigma} := \mathsf{PRG}(\vec{s}^{i,j}_{1-\sigma} \oplus T^{i,j}_{1-\sigma} \cdot \mathsf{CW}^i)$$

    - ∗ The sums of the left/right leaves and left/right correction bits, for $b \in \{0,1\}$:

$$\vec{z}^{i+1,b}_{1-\sigma} := \bigoplus_{j=0}^{2^i-1} \vec{s}^{i+1,2j+b}_{1-\sigma}, \; u^{i+1,b}_{1-\sigma} := \bigoplus_{j=0}^{2^i-1} t^{i+1,2j+b}_{1-\sigma},$$

    - ∗ The next level of choice bits:

$$T^{i+1,2j}_{1-\sigma} := t^{i+1,2j}_{1-\sigma} \oplus T^{i,j}_{1-\sigma} \cdot \tau^{i,\mathsf{Parity(j)}}, \quad T^{i+1,2j+1}_{1-\sigma} := t^{i+1,2j+1}_{1-\sigma} \oplus T^{i,j}_{1-\sigma} \cdot \tau^{i,\mathsf{Parity(j)}}$$

- Convert the output shares to $\mathbb{F}_q$: $P_{1-\sigma}$ computes:

$$z^L_{1-\sigma} := \sum_{j=0}^{D-1} s^{j,L}_{1-\sigma}, \quad z^R_{1-\sigma} := \sum_{j=0}^{D-1} s^{j,R}_{1-\sigma}$$

  where $s^{j,L}_{1-\sigma} = \mathsf{Convert}_{\mathbb{F}_q}(\vec{s}^{\log D+1,2j}_{1-\sigma})$ and $s^{j,R}_{1-\sigma} = \mathsf{Convert}_{\mathbb{F}_q}(\vec{s}^{\log D+1,2j+1}_{1-\sigma})$.
- Correct the offset of the left leaves to β and the offset of the right leaves to 0:
  - Party $P_{1-\sigma}$ sends $\vec{z}^L_{1-\sigma}$ and $\vec{z}^R_{1-\sigma}$ to **Input**.
  - Party $P_{1-\sigma}$ receives CW^R.

Verification Phase: • The parties call $\mathcal{F}_{\mathsf{coin}}$ to obtain $r^0, \ldots, r^{D-1} \in \mathbb{F}_q$.

- Party $P_{1-\sigma}$ computes

$$S^L_{1-\sigma} = (-1)^{1-\sigma} \cdot \sum_{j=0}^{D-1} r^j \cdot s^{j,L}_{1-\sigma}, \quad S^R_{1-\sigma} = (-1)^{1-\sigma} \cdot (\mathsf{CW}^R)^{-1} \cdot \sum_{j=0}^{D-1} r^j \cdot s^{j,R}_{1-\sigma}$$

- Party $P_{1-\sigma}$ inputs $S^L_{1-\sigma}, S^R_{1-\sigma}$ to **Input**. It receives Z. If $Z \neq 0$, party $P_{1-\sigma}$ aborts.

Output Phase: • Party $P_{1-\sigma}$ receives CW^L. If $\mathsf{CW}^L = 0$, the party aborts.

- Finally, if all checks passed, party $P_{1-\sigma}$ outputs $\vec{y}_{1-\sigma} = (y^0_{1-\sigma}, \ldots, y^D_{1-\sigma} - 1)$, where

$$y^j_{1-\sigma} := (-1)^{1-\sigma} \cdot s^{j,L}_{1-\sigma} \cdot \mathsf{CW}^L.$$

Hybrid 2: Randomized behaviour of the honest party. We now describe the honest party's behaviour, where we replace calls to PRG by random values whenever the seed is not known by the adversarial party P_σ. Changes to the first hybrid are $\boxed{\text{framed}}$.

Key generation phase: • Party $P_{1-\sigma}$ $\boxed{\text{chooses } \vec{s}^{1,0}_{1-\sigma} \| \vec{s}^{1,1}_{1-\sigma} \| t^{1,0}_{1-\sigma} \| t^{1,1}_{1-\sigma} \xleftarrow{\$} \{0,1\}^{2\lambda+2}}$ at random. Party $P_{1-\sigma}$ locally sets $T^{1,0}_{1-\sigma} := T^{1,1}_{1-\sigma} := 1-\sigma$, and $\vec{z}^{1,0}_{1-\sigma} := \vec{s}^{1,0}_{1-\sigma}$, $\vec{z}^{1,1}_{1-\sigma} := \vec{s}^{1,1}_{1-\sigma}$, $u^{1,0}_{1-\sigma} := t^{1,0}_{1-\sigma}$, $u^{1,1}_{1-\sigma} := t^{1,1}_{1-\sigma}$.

- For $i = [1..\log D]$:
  - For $b \in \{0,1\}$, $P_{1-\sigma}$ sends $\tau^{i,b}_{1-\sigma} := u^{i,b}_{1-\sigma} \oplus \alpha_{i,1-\sigma} \oplus (1 \oplus \sigma) \cdot (1 \oplus b)$ to **PassiveOutput** and receives $\tau^{i,b}$.
  - $P_{1-\sigma}$ sends $\vec{z}^{i,0}_{1-\sigma} \oplus \vec{z}^{i,1}_{1-\sigma}$ to **MixedMult** and receives $\vec{y}^{i}_{1-\sigma}$.
  - $P_{1-\sigma}$ sends $\vec{y}^{i}_{1-\sigma} \oplus \vec{z}^{i,1}_{1-\sigma}$ to **PassiveOutput** and receives CW^i.
  - For $j \in [0..2^i)$, party $P_{1-\sigma}$ computes:
    * The next level of the GGM tree:

$$\vec{s}^{i+1,2j}_{1-\sigma} \| \vec{s}^{i+1,2j+1}_{1-\sigma} \| t^{i+1,2j}_{1-\sigma} \| t^{i+1,2j+1}_{1-\sigma} \begin{cases} \boxed{\xleftarrow{\$} \{0,1\}^{2\lambda+2}} & \boxed{\text{if } j = \alpha|_i} \\ := \mathsf{PRG}(\vec{s}^{i,j}_{1-\sigma} \oplus T^{i,j}_{1-\sigma} \cdot \mathsf{CW}^i) & \text{else} \end{cases}$$

    * The sums of the left/right leaves and left/right correction bits, for $b \in \{0,1\}$:

$$\vec{z}^{i+1,b}_{1-\sigma} := \bigoplus_{j=0}^{2^i-1} \vec{s}^{i+1,2j+b}_{1-\sigma}, \quad u^{i+1,b}_{1-\sigma} := \bigoplus_{j=0}^{2^i-1} t^{i+1,2j+b}_{1-\sigma},$$

    * The next level of choice bits:

$$T^{i+1,2j}_{1-\sigma} := t^{i+1,2j}_{1-\sigma} \oplus T^{i,j}_{1-\sigma} \cdot \tau^{i,\mathsf{Parity(j)}}, \quad T^{i+1,2j+1}_{1-\sigma} := t^{i+1,2j+1}_{1-\sigma} \oplus T^{i,j}_{1-\sigma} \cdot \tau^{i,\mathsf{Parity(j)}}$$

- Convert the output shares to $\mathbb{F}_q$: $P_{1-\sigma}$ computes:

$$z^{L}_{1-\sigma} := \sum_{j=0}^{D-1} s^{j,L}_{1-\sigma}, \quad z^{R}_{1-\sigma} := \sum_{j=0}^{D-1} s^{j,R}_{1-\sigma}$$

  where $s^{j,L}_{1-\sigma} = \mathsf{Convert}_{\mathbb{F}_q}(\vec{s}^{\log D+1,2j}_{1-\sigma})$ and $s^{j,R}_{1-\sigma} = \mathsf{Convert}_{\mathbb{F}_q}(\vec{s}^{\log D+1,2j+1}_{1-\sigma})$.
- Correct the offset of the left leaves to β and the offset of the right leaves to 0:
  - Party $P_{1-\sigma}$ sends $\vec{z}^{L}_{1-\sigma}$ and $\vec{z}^{R}_{1-\sigma}$ to **Input**.
  - Party $P_{1-\sigma}$ receives CW^R.

Verification Phase: • The parties call $\mathcal{F}_{\mathsf{coin}}$ to obtain $r^0, \dots, r^{D-1} \in \mathbb{F}_q$.

- Party $P_{1-\sigma}$ computes

$$S^{L}_{1-\sigma} = (-1)^{1-\sigma} \cdot \sum_{j=0}^{D-1} r^j \cdot s^{j,L}_{1-\sigma}, \quad S^{R}_{1-\sigma} = (-1)^{1-\sigma} \cdot (\mathsf{CW}^R)^{-1} \cdot \sum_{j=0}^{D-1} r^j \cdot s^{j,R}_{1-\sigma}$$

- Party $P_{1-\sigma}$ inputs $S^{L}_{1-\sigma}, S^{R}_{1-\sigma}$ to **Input**. It receives Z. If $Z \neq 0$, party $P_{1-\sigma}$ aborts.

Output Phase: • Party $P_{1-\sigma}$ receives CW^L. If $\mathsf{CW}^L = 0$, the party aborts.

- Finally, if all checks passed, party $P_{1-\sigma}$ outputs $\vec{y}_{1-\sigma} = (y^0_{1-\sigma}, \dots, y^D_{1-\sigma} - 1)$, where

$$y^j_{1-\sigma} := (-1)^{1-\sigma} \cdot s^{j,L}_{1-\sigma} \cdot \mathsf{CW}^L.$$

We omit the proof of the following lemma here and refer to [BGI16b], since the changes only concern the underlying DPF protocol.

Lemma 5.6 *If* $\mathsf{PRG}\colon \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$ *is a secure PRG, then the real behaviour and the randomized behaviour of* $P_{1-\sigma}$ *are indistinguishable to an adversary controlling* P_σ.

Lemma 5.7 *Assume* $q = q(\lambda) = \lambda^{\omega(1)}$. *Then, the distribution generated by the simulator is statistically indistinguishable from the randomized protocol execution as described in Hybrid 2.*

Proof. Assume to be given the real $\alpha \in [0..D), \beta \in \mathbb{F}_q$. First, note that we can assume that for CW^L as sampled by the simulator it holds that $\mathsf{CW}^L \neq 0$, since this holds except with probability $1/q$. Further, we assume that for β' as generated in the randomized protocol execution, it holds $\beta' \neq 0$, which is also true except with probability $1/q$.

We begin the proof by showing that the values $\vec{s}^{\,\log D+1,2j}_{1-\sigma,\alpha}$ and $\vec{s}^{\,\log D+1,2j+1}_{1-\sigma,\alpha}$ for $j \neq \alpha$ are *perfectly* indistinguishable from those generated in a real protocol execution with randomized behaviour of $P_{1-\sigma}$.

- First, we have $T^{1,1-\alpha_1}_{1-\sigma,\alpha} = 1 = T^{1,1-\alpha}_{1-\sigma}$.

- Next, recall that for $b \in \{0,1\}$ the simulator sets $u^{1,b}_{1-\sigma,\alpha} := \tau^{1,b}_{1-\sigma} \oplus \alpha_1 \oplus \alpha_{1,\sigma} \oplus (1\oplus\sigma)\cdot(1\oplus b)$, where $\tau^{1,b}_{1-\sigma}$ is chosen uniformly at random from $\{0,1\}$. This is equivalent to choosing $u^{1,b}_{1-\sigma}$ uniformly at random and setting $\tau^{i,b}_{1-\sigma} := u^{i,b}_{1-\sigma} \oplus \alpha_{1,1-\sigma} \oplus (1\oplus\sigma)\cdot(1\oplus b)$, as done in the randomized protocol execution.

- Further, recall that the simulator sets
$$\vec{z}^{\,1,1-\alpha_i}_{1-\sigma,\alpha} := \mathsf{CW}^1_{1-\sigma} \oplus \vec{y}^{\,1}_\sigma \oplus \alpha_1 \cdot \vec{v}^{\,1}_\sigma,$$
where $\vec{v}^{\,1}_\sigma \in \{0,1\}^\lambda$ is the adversary's input to **MixedMult** and $\vec{y}^{\,1}_\sigma \in \{0,1\}^\lambda$ is the adversary's output share for P_σ. Note that in a randomized execution of the protocol, it holds $\mathsf{CW}^1_{1-\sigma} = \vec{y}^{\,1}_{1-\sigma} \oplus \vec{z}^{\,1,1}_{1-\sigma}$, where $\vec{y}^{\,1}_\sigma \oplus \vec{y}^{\,1}_{1-\sigma} = \alpha_1 \cdot (\vec{v}^{\,1}_\sigma \oplus \vec{z}^{\,1,0}_{1-\sigma} \oplus \vec{z}^{\,1,1}_{1-\sigma})$. This yields
$$\mathsf{CW}^1_{1-\sigma} = \vec{y}^{\,1}_\sigma \oplus \alpha_1 \cdot \vec{v}^{\,1}_\sigma \oplus \vec{z}^{\,1,1-\alpha_1}_{1-\sigma},$$
where $\vec{z}^{\,1,1-\alpha_1}_{1-\sigma}$ is sampled uniformly at random. Thus, as above we obtain that the simulation is perfectly indistinguishable from the randomized protocol execution.

- For $i \in [1..\log D]$, observe that again sampling $\mathsf{CW}^{i+1}, \tau^{i+1,0}, \tau^{i+1,1}$ (and thus $\vec{z}^{\,i+1,1-\alpha_i}_{1-\sigma,\alpha}$ and $u^{i+1,b}_{1-\sigma,\alpha}$) and then defining the missing $\vec{s}^{\,i+1,2\alpha|_i+1-\alpha_{i+1}}_{1-\sigma,\alpha}$, $t^{i+1,2\alpha|_i}_{1-\sigma,\alpha}$, $t^{i+1,2\alpha|_i+1}_{1-\sigma,\alpha}$ is indistinguishable to the randomized protocol execution, where $P_{1-\sigma}$ proceeds vice versa.

With the above considerations, we obtain that $\vec{s}^{\,\log D+1,2j}_{1-\sigma,\alpha}$ and $\vec{s}^{\,\log D+1,2j+1}_{1-\sigma,\alpha}$ for $j \neq \alpha$ are indistinguishable from those generated in a randomized protocol execution as required.

Next, we compute the values $s^{\alpha,L}_{1-\sigma}, s^{\alpha,R}_{1-\sigma}$, as would be computed in a randomized protocol execution (conditioned on $\beta \neq 0$), and show that the simulator outputs $Z = 0$ if and only if the randomized execution outputs $Z = 0$, except with negligible probability (corresponding to the event that the adversary successfully guessed β').

- Recall that z^L_σ, z^R_σ are the values sent to **Input** by the adversary. In a randomized protocol execution we would have
$$(-1)^\sigma \cdot z^L_\sigma + (-1)^{1-\sigma} \cdot z^L_{1-\sigma} = \beta',$$
and thus
$$z^L_{1-\sigma} = (-1)^{1-\sigma} \cdot \beta' + z^L_\sigma,$$
as well as
$$z^R_{1-\sigma} = (-1)^{1-\sigma} \cdot \mathsf{CW}^R + z^R_\sigma.$$

This yields

$$s_{1-\sigma}^{\alpha,L} = z_{1-\sigma}^{L} - \sum_{j=0,j\neq\alpha}^{D-1} s_{1-\sigma}^{j,L}, \quad s_{1-\sigma}^{\alpha,R} = z_{1-\sigma}^{R} - \sum_{j=0,j\neq\alpha}^{D-1} s_{1-\sigma}^{j,R}$$

where $s_{1-\sigma}^{j,L} = \mathsf{Convert}_{\mathbb{F}_q}(\vec{s}_{1-\sigma}^{\log D+1,2j})$ and $s_{1-\sigma}^{j,R} = \mathsf{Convert}_{\mathbb{F}_q}(\vec{s}_{1-\sigma}^{\log D+1,2j+1})$ for $j \neq \alpha$.

- Let now $S_{1-\sigma}^{L}$, $S_{1-\sigma}^{R}$ be the values as computed by the honest party in a randomized protocol execution, and $S_{\sigma,\alpha}^{L}$, $S_{\sigma,\alpha}^{R}$ the values computed by the simulator. Then, we have

$$\begin{aligned} S_{1-\sigma}^{L} + S_{\sigma,\alpha}^{L} &= (-1)^{1-\sigma} \cdot r^{\alpha} \cdot s_{1-\sigma}^{\alpha,L} + (-1)^{\sigma} \cdot r^{\alpha} \cdot s_{\sigma,\alpha}^{\alpha,L} \\ &= (-1)^{1-\sigma} \cdot r^{\alpha} \cdot z_{1-\sigma}^{L} + (-1)^{\sigma} \cdot r^{\alpha} \cdot z_{\sigma}^{L} \\ &= (-1)^{1-\sigma} \cdot r^{\alpha} \cdot ((-1)^{1-\sigma} \cdot \beta' + z_{\sigma}^{L}) + (-1)^{\sigma} \cdot r^{\alpha} \cdot z_{\sigma}^{L} \\ &= r^{\alpha} \cdot \beta' \end{aligned}$$

and

$$\begin{aligned} S_{1-\sigma}^{R} + S_{\sigma,\alpha}^{R} &= (-1)^{1-\sigma} \cdot (\mathsf{CW}^{R})^{-1} \cdot r^{\alpha} \cdot s_{1-\sigma}^{\alpha,R} + (-1)^{\sigma} \cdot (\mathsf{CW}^{R})^{-1} \cdot r^{\alpha} \cdot s_{\sigma,\alpha}^{\alpha,R} \\ &= r^{\alpha}. \end{aligned}$$

We thus have that the randomized protocol execution outputs $Z = 0$ if and only if $S_{\sigma}^{L} = S_{\sigma,\alpha}^{L}$ and $S_{\sigma}^{R} = S_{\sigma,\alpha}$, unless the adversary successfully guessed β'. Since β' is distributed perfectly uniformly at random from the adversary's point of view in the randomized protocol execution at this point (since it didn't yet learn CW^{L}), this only happens with probability $1/q$. Further, if the simulator aborts and the adversary did not successfully guess β', it perfectly simulates the distribution of Z, since again β' is distributed uniformly at random.

It is left to show that the output of the honest party induced by the simulation is indistinguishable from the output of the honest party in the randomized protocol execution. To that end, we have to show that

1. $\vec{y}_{\sigma,\alpha} + \vec{y}_{1-\sigma} = (0,\ldots,0,\beta,\ldots,0)$, where $\vec{y}_{\sigma,\alpha}$ is as computed by the simulator and $\vec{y}_{1-\sigma}$ is the value computed in the randomized protocol execution, and
2. $\vec{y}_{\sigma,\chi^*} = \vec{y}_{\sigma,\alpha}$ for all χ^* with $P(\chi^\star) = 1$.

This can be proven as follows.

1. By the above considerations we have

$$y_{\sigma,\alpha}^{j} + y_{1-\sigma}^{j} = (-1)^{\sigma} \cdot s_{\sigma,\alpha}^{j,L} \cdot \mathsf{CW}^{L} + (-1)^{1-\sigma} \cdot s_{1-\sigma}^{j,L} \cdot \mathsf{CW}^{L} = 0$$

for all $j \neq \alpha$. Further, we have

$$\begin{aligned} y_{\sigma,\alpha}^{\alpha} + y_{1-\sigma}^{\alpha} &= (-1)^{\sigma} \cdot s_{\sigma,\alpha}^{\alpha,L} \cdot \mathsf{CW}^{L} + (-1)^{1-\sigma} \cdot s_{1-\sigma}^{\alpha,L} \cdot \mathsf{CW}^{L} \\ &= (-1)^{\sigma} \cdot z_{\sigma}^{L} \cdot \mathsf{CW}^{L} + (-1)^{1-\sigma} \cdot z_{1-\sigma}^{L} \cdot \mathsf{CW}^{L} \\ &= \beta' \cdot \mathsf{CW}^{L} \\ &= \beta, \end{aligned}$$

which yields the required.

2. Since $r^0, \ldots, r^{D-1}$ are sampled at random over $\mathbb{F}_q$, independently of the rest of the protocol execution, we have that if $S_{1-\sigma,\chi^\star}^{L} = S_{1-\sigma,\alpha}^{L}$ and $S_{1-\sigma,\chi^\star}^{L} = S_{1-\sigma,\alpha}^{L}$, then $s_{\sigma,\alpha}^{j,L} = s_{\sigma,\chi}^{j,L}$ for all $j \in [0..D)$, except with negligible probability. This concludes the proof.

□

□

Functionality $\mathcal{F}_{\mathsf{OLE\text{-}Setup}}$

PARAMETERS: Security parameter 1^λ, $\mathsf{PCG}_{\mathsf{OLE}} = (\mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Gen}, \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand})$ as per Figure 1.
FUNCTIONALITY:

1. Sample $(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Gen}(1^\lambda)$.
2. Output k_σ to party P_σ, for $\sigma \in \{0, 1\}$

Figure 7: Generic functionality for the distributed setup of OLE PCG seeds

6 PCG Setup Protocols

In this section, we present the complete secure two-party setup protocols for our PCGs based on module-LPN, using the protocols from the previous section as building blocks. In Section 6.1, we show how to securely realize (against a semi-honest adversary) the randomized functionality $\mathcal{F}_{\mathsf{OLE\text{-}Setup}}$ that executes the seed generation for our PCG construction $\mathsf{PCG}_{\mathsf{OLE}}$ from Section 4, and outputs the corresponding PCG seeds to each party. This can in turn be used to realize a functionality for the secure generation of OLE correlations, by having the parties simply expand their received PCG seeds locally.

For the malicious case, we do not directly realize the PCG seed generation functionality, which would be much more challenging. Instead, we realize the random OLE generation functionality $\mathcal{F}_{\mathsf{mal\text{-}OLE}}$, in which a corrupt adversary can choose his output $(x_\sigma, z_\sigma) \in R_p^2$ and the honest party receives a random consistent value $(x_{1-\sigma}, z_{1-\sigma})$, i.e. for which $z_0 + z_1 = x_0 \cdot x_1$ (or the parties receive a random sample from the correlation given honest behavior; see Figure 9 for full description). As discussed in [BCG+19b], such a protocol can *directly* serve as a substitute for ideal OLE correlations in a wide range of higher-level applications, already proven to remain secure given this functionality. More precisely, in Section 5.3, we address both the secure generation of OLEs and authenticated multiplication triples in the presence of active adversaries, securely realizing the functionality $\mathcal{F}_{\mathsf{mal\text{-}triple}}$ at little extra cost.

Finally, in Section 6.3 we analyze the efficiency of our protocols. For concrete numbers, we refer to Table 3 in Section 9.2.

6.1 Semi-Honest Distributed Setup for OLE

We present a protocol for securely executing the seed-generation functionality $\mathcal{F}_{\mathsf{OLE\text{-}Setup}}$ (Figure 7) with respect to our PCG construction $\mathsf{PCG}_{\mathsf{OLE}}$ from Section 4. Recall that in the $\mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Gen}$ procedure (see Figure 1), each party receives a succinct description $(A^i_\sigma, \vec{b}^i_\sigma)_{i\in[0..c)}$ of t-sparse "noise vectors" $e^0_\sigma, \ldots, e^{c-1}_\sigma$ each of length N, as well as a collection of $(ct)^2$ distributed point function (DPF) keys as a compact representation of all possible products $e^i_0 \cdot e^j_1$.

Note that the protocols works over the prime field $\mathbb{F}_p$, rather than an extension field $\mathbb{F}_q$ (as in $\mathcal{F}_{\mathsf{DPF}}$). This is because the PCGs we use need to run over R_p for a prime p, to be useful for the case where R_p fully splits into linear factors (cf. Section 3.2).

Theorem 6.1 *The protocol $\Pi_{\mathsf{OLE\text{-}Setup}}$ (Fig. 8) securely realizes the OLE PCG seed functionality $\mathcal{F}_{\mathsf{OLE\text{-}Setup}}$ (Fig. 7) with security against semi-honest adversaries in the $(\mathcal{F}_{\mathsf{2\text{-}PC}}, \mathcal{F}_{\mathsf{DPF}})$-hybrid model.*

Proof. Observe that the protocol $\Pi_{\mathsf{OLE\text{-}Setup}}$ is directly a secure evaluation of the computation steps of $\mathcal{F}_{\mathsf{OLE\text{-}Setup}}$ (see description of $\mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Gen}$ as given in Figure 1 within Section 4), with the exception that the FSS key generation for the sum of point functions $\mathsf{SPFSS.Gen}(1^\lambda, A^i_0 \boxplus$

Protocol $\Pi_{\mathsf{OLE\text{-}Setup}}$

Parameters: Security parameter 1^λ, natural number $N = 2^k$, prime p, distributed point function $\mathsf{DPF} = (\mathsf{DPF.Gen}, \mathsf{DPF.Eval})$ with domain $[0..2N)$ and range $\mathbb{F}_p$. Further, we assume access to the functionalities $\mathcal{F}_{\mathsf{2\text{-}PC}}$ (Fig. 3) and $\mathcal{F}_{\mathsf{DPF}}$ (Fig. 4).
Protocol:

1. For $\sigma \in \{0,1\}, i \in [0..c)$, party P_σ samples random vectors $A^i_\sigma \leftarrow [0..N)^t$ (where each entry of A^i_σ is viewed as a length-$\log N$ bit-string) and $\vec{b}^i_\sigma \leftarrow (\mathbb{F}_p^*)^t$. Note that as outlined in Figure 1, each pair $A^i_\sigma, \vec{b}^i_\sigma$ defines a t-sparse polynomial $e^i_\sigma \in R_p$.
2. For $\sigma \in \{0,1\}, i \in [0..c)$ and $k \in [0..t)$:
   //P_σ inputs the k-th non-zero position and corresponding payload of e^i_σ.
   $$[\![A^i_\sigma[k]]\!]_{\mathbb{F}_2^\ell} \leftarrow \mathbf{Input}(P_\sigma, A^i_\sigma[k]) \text{ and } [\![\vec{b}^i_\sigma[k]]\!]_p \leftarrow \mathbf{Input}(P_\sigma, \vec{b}^i_\sigma[k])$$
3. For every $i, j \in [0..c)$ and $k, l \in [0..t)$ (in parallel) the parties do the following:
   (a) $[\![\alpha^{i,j}_{k,l}]\!]_{\mathbb{F}_2^{\ell+1}} \leftarrow \mathbf{BitAdd}([\![A^i_0[k]]\!]_{\mathbb{F}_2^\ell}, [\![A^j_1[l]]\!]_{\mathbb{F}_2^\ell})$ //Compute the $k+tl$-th position of $e^i_0 \cdot e^j_1$.
   (b) $[\![\beta^{i,j}_{k,l}]\!]_p \leftarrow \mathbf{Mult}([\![\vec{b}^i_0[k]]\!]_p, [\![\vec{b}^j_1[l]]\!]_p)$ //Compute the $k+tl$-th payload of $e^i_0 \cdot e^j_1$.
   (c) For each $i, j \in [0..c)$ and $k, l \in [0..t)$, call $\mathcal{F}_{\mathsf{DPF}}$ with domain size $[0..2N)$ on input $[\![\alpha^{i,j}_{k,l}]\!]_{\mathbb{F}_2^{\ell+1}}$ and $[\![\beta^{i,j}_{k,l}]\!]_p$, and let $K^{i,j}_{\sigma,k,l}$ denote the output to party P_σ.
   //Compute compressed additive secret shares of $e^i_0 \cdot e^j_1$.
4. Party P_σ outputs $\mathsf{k}_\sigma = \Big((K^{i,j}_{\sigma,k,l})_{i,j\in[0..c),k,l\in[0..t)}, (A^i_\sigma, \vec{b}^i_\sigma)_{i\in[0..c)}\Big)$.

Figure 8: Distributed setup of PCG seeds for OLE in the $(\mathcal{F}_{\mathsf{2\text{-}PC}}, \mathcal{F}_{\mathsf{DPF}})$-hybrid model, against semi-honest adversaries. $[\![x]\!]_{\mathbb{F}_2^\ell}, [\![x]\!]_p$ denote additive shares of bit-strings or $\mathbb{F}_p$ elements.

$A^j_1, \vec{b}^i_0 \otimes \vec{b}^j_1$) is instantiated directly by the DPF key generation for every individual nonzero component. As discussed in Section 2.2, this is a valid instantiation of SPFSS, and hence the claim holds. $\square$

6.2 PCG Setup Protocols with Malicious Security

Module-LPN with static leakage. The protocols in this section rely on a stronger variant of module-LPN, where the adversary is allowed to query (on average) one bit of information on the secret error vectors. We need to allow this, because of the leakage in the functionality $\mathcal{F}_{\mathsf{c\text{-}SUV}}$ used to generate DPF keys with malicious security. Note that this style of assumption with leakage is very similar to the assumption of standard LPN with static leakage as used in [HOSS18a, BCG+19a].

Definition 6.2 (Module-LPN with static leakage) *Let $R_p = \mathbb{Z}_p[X]/F(X)$ for some prime p and degree-N polynomial $F(X) \in \mathbb{Z}[X]$, and let $c, t \in \mathbb{N}$ with $c \geq 2$. Let $\mathcal{HW}_t$ be the distribution over R_p that is obtained via sampling t noise positions $A \leftarrow [0..N)^t$ as well as t payloads $\vec{b} \leftarrow \mathbb{Z}_p^t$ uniformly at random, and outputting $e(X) := \sum_{j=0}^{t-1} \vec{b}[j] \cdot X^{A[j]}$. For $b \in \{0,1\}$ let $G_b(\lambda)$ be the following game:*

- *Let $e_0, \dots, e_{c-1} \leftarrow \mathcal{HW}_t$ and let $A^0, \dots, A^{c-1} \in [0..N)^t$ be the corresponding positions of non-zero coefficients.*

- ***Allow the adversary arbitrary guesses on the noise positions* before *seeing the module-LPN sample.*** *The adversary $\mathcal{A}$ can make an arbitrary number of (adaptive) queries, each consisting of indices $i \in [0..c)$, $k \in [0..c)$ and a predicate $P\colon [0..N) \to \{0,1\}$. For each query, if $P(A^i[k]) = 0$ then abort, otherwise, send* (`success`) *to the adversary and continue.*

- *Set $u_0 = \langle \vec{a}, \vec{e} \rangle \mod F(X)$, where $\vec{e} = (e_0, \ldots, e_{c-1})$ and draw $u_1 \leftarrow R_p$ at random.*

- *Return u_b to the adversary.*

The R^c-$\mathsf{LPN}_{p,t}$ problem with static leakage *for $\vec{a} \in R_p^c$ is hard if for any PPT adversary $\mathcal{A}$, it holds that*

$$\left|\Pr[\mathcal{A}^{G_0(\lambda)} = 1] - \Pr[\mathcal{A}^{G_1(\lambda)} = 1]\right| \leq \mathsf{negl}(\lambda)$$

where the probabilities are taken over $e_0, \ldots, e_{c-1} \leftarrow \mathcal{HW}_t$ and the randomness of $\mathcal{A}$.

In the above, if the adversary was restricted to querying predicates that either take 1 on all values or 0 on at least half of the values (in other words: if the adversary attempts to guess some noise coordinate, the protocol will abort with probability at least 1/2), then we can reduce the ring LPN with static leakage and noise parameter $t + \kappa$ to our basic ring LPN assumption with noise parameter t. The idea is that the adversary can attempt a guess on at most κ noise coordinates, where κ is a statistical security parameter, as otherwise the protocol will abort with probability $> 1 - 2^{-\kappa}$. This observation would allow to reduce security of a slight tweak of the protocol $\Pi_{\mathsf{mal\text{-}OLE}}$ (Figure 10), where our protocol $\Pi_{\mathsf{c\text{-}SUV}}$ is used to generate *random* instances of scaled unit vectors (which are derandomized subsequently), to the standard module-LPN assumption. This is due to the fact that if α is random from the point of the view of the adversary, one can show that in the protocol $\Pi_{\mathsf{c\text{-}SUV}}$ an adversary actually can only give a wildcard guess on α, i.e. guess a subset of the bits, which corresponds to giving a predicate which takes 0 on at least half of the inputs.

This said, we want to stress that the described reduction is very loose in the sense that even if the adversary only attempts to guess a single bit on the noise position, the reduction "gives up" the correspoding noise position completely. Indeed, for our concrete protocol, we conjecture that it's not necessary to increase the noise rate in the presence of this limited leakage. For a more detailed discussion regarding attacks on ring LPN with static leakage, see Section 8.6.

Distributed generation of OLEs in the malicious setting. Building on the previous protocols, we now present our PCG protocol for generating OLE correlations with distributed setup (Fig. 10) and show that it securely implements the corruptible OLE functionality (Fig. 9) with security against malicious adversaries.

Theorem 6.3 *Let $R_p = \mathbb{Z}_p[X]/F(X)$ for some prime p and degree-N polynomial $F(X) \in \mathbb{Z}[X]$, and let $c, t \in \mathbb{N}$. If the R^c-$\mathsf{LPN}_{p,t}$ problem with static leakage is hard, then the protocol $\Pi_{\mathsf{mal\text{-}OLE}}$ (Fig. 10) implements the functionality $\mathcal{F}_{\mathsf{mal\text{-}OLE}}$ (Fig. 9) with security against malicious adversaries in the $\mathcal{F}_{\mathsf{2\text{-}PC}}, \mathcal{F}_{\mathsf{c\text{-}SUV}}$-hybrid model.*

Proof. We start with considering the case that both parties are honest.

Both parties are honest. Note that all communication of the parties is via ideal functionalities, so we only need to prove that the final output of a real protocol execution is indistinguishable from the output of the ideal functionality.

For $\sigma \in \{0,1\}$, let $e_\sigma^i(X) = \sum_{k=0}^{t-1} \vec{b}_\sigma^i[k] \cdot X^{A_\sigma^i[k]} \in R_p$ be the noise polynomial defined by A_σ^i and $\vec{b}_\sigma^i$. Then, $e_0^i \cdot e_1^j$ is the polynomial with coefficient $\vec{b}_0^i[k] \cdot \vec{b}_1^j[l]$ at position $A_0^i[k] + A_1^j[l]$ for all

Functionality $\mathcal{F}_{\text{mal-OLE}}$

Parameters: Security parameter 1^λ, modulus p, and the ring $R_p = \mathbb{Z}_p[X]/F(X)$, where $F(X)$ has degree N.
Functionality:
If both parties are honest:

1. Sample $x_0, x_1 \leftarrow R_p$.
2. Sample $z_0 \xleftarrow{\$} R_p$, and let $z_1 \leftarrow x_0 \cdot x_1 - z_0$.
3. Output (x_σ, z_σ) to party P_σ, for $\sigma \in \{0,1\}$

If party P_σ is corrupted:

1. Wait for input $(x_\sigma, z_\sigma) \in R_p^2$ from the adversary.
2. Sample $x_{1-\sigma} \leftarrow R_p$ and set $z_{1-\sigma} \leftarrow x_0 \cdot x_1 - z_\sigma$.
3. Output $(x_{1-\sigma}, z_{1-\sigma})$ to the honest party.

Figure 9: Corruptible OLE Functionality

$i,j \in [0..c)$ and $k,l \in [0..t)$. Thus, by the properties of $\mathcal{F}_{\text{c-SUV}}$, it holds $g_{0,i+cj} + g_{1,i+cj} = e_0^i \cdot e_1^j$, and therefore $x_0 \cdot x_1 = z_0 + z_1$ as required.

Finally, $R^c\text{-}\mathsf{LPN}_{p,t}$ implies the indistinguishability of $(x_0, x_1), (z_0, z_1)$ from an OLE triple generated at random similarly to the proof of Theorem 4.1.

Party P_σ is corrupted. For the notation in the following we assume $\sigma = 1$, the case $\sigma = 0$ is obtained by switching (i,k) with (j,l). The simulator proceeds as follows. For $j \in [0..c)$ and $l \in [0..t)$ it waits for input $A_\sigma^j, \dots, A_\sigma^j \in [0..N)^t$ and $\vec{b}_\sigma^j \in \mathbb{Z}_p^t$ of the adversary. Otherwise, the simulator sets $e_\sigma^j \in R_p$ to be degree $< N$ polynomial defined by $A_\sigma^j, \vec{b}_\sigma^j$, and saves $\vec{e}_\sigma := (e_\sigma^0, \dots e_\sigma^{c-1}) \in R_p^c$.

The simulator proceeds to simulate the calls to $\mathcal{F}_{\text{c-SUV}}$ as follows. For each $j \in [0..c)$, $l \in [0..t)$ for which $\vec{b}_\sigma^i[k] = 0$ it outputs "$\beta = 0$" to the adversary and aborts on this instance. Next, for all other instances it awaits a guess B of size at most $2N$ of the adversary. The simulator replies with "$\perp$" (regardless of the set B) and continues. Subsequently, for $i,j \in [0..c)$ and $k,l \in [0..t)$ corresponding to a non-aborting instance the simulator waits for input $\vec{g}_{\sigma,k,l}^{i,j}$ and predicate $P_{k,l}^{i,j} \colon [0..2N) \to \{0,1\}$ by the adversary. Now, the simulator chooses random $A^0, \dots, A^{c-1} \xleftarrow{\$} [0..N)^t, \vec{b}^0, \dots, \vec{b}^{c-1} \xleftarrow{\$} \mathbb{Z}_p^t \backslash \{0\}$ and proceeds as follows: For each $i,j \in [0..c)$, $k,l \in [0..t)$, in the (i,j,k,l)-th invocation of $\mathcal{F}_{\text{c-SUV}}$, if $P_{k,l}^{i,j}(A^i[k] + A_\sigma^j[l]) = 0$, the simulator aborts on all instances and outputs $(A^i[k] + A_\sigma^j[l], \vec{b}^i[k] \cdot \vec{b}_\sigma^j[l])$. Otherwise, it outputs $(\texttt{success})$ and continues. Next, if the simulator did not abort on any instance, for each $i,j \in [0..c)$, $k,l \in [0..t)$ the simulator defines the vector

$$\vec{g}_\sigma^{i,j} = \sum_{j,k=1}^{t} \vec{g}_{\sigma,k,l}^{i,j}.$$

The simulator interprets each $\vec{g}_\sigma^{i,j}$ as a degree $< 2N$ polynomial $g_{\sigma,i+cj}$ over Z_p, and sets $\vec{g}_\sigma := (g_\sigma^0, \dots, g_\sigma^{c^2-1})$.

Protocol $\Pi_{\mathsf{mal\text{-}OLE}}$

Parameters: Security parameter 1^λ, natural number $N = 2^k$, prime p, access to functionality $\mathcal{F}_{\mathsf{c\text{-}SUV}}$ with length $2N$, prime modulus p, polynomial $F(X) \in \mathbb{Z}[X]$ of degree N, ring $R = \mathbb{Z}_p[X]/F(X)$, and parameters $c, t \in \mathbb{N}$ (corresponding to the syndrome compression factor and noise rate of the LPN assumption, respectively).
Public input: random polynomials $a_1, \ldots, a_{c-1} \in R_p$, used for R^c-LPN
Correlation: After expansion, outputs $(x_0, z_0) \in R_p^2$ and $(x_1, z_1) \in R_p^2$, where $z_0 + z_1 = x_0 \cdot x_1$.
Protocol:

1. For $\sigma \in \{0,1\}, i \in [0..c)$, party P_σ samples random vectors $A_\sigma^i \leftarrow [0..N)^t$ (where each entry of A_σ^i is viewed as a length-$\log N$ bit-string) and $\vec{b}_\sigma^i \leftarrow (\mathbb{Z}_p^*)^t$. Note that each pair $A_\sigma^i, \vec{b}_\sigma^i$ defines a t-sparse polynomials $e_\sigma^i(X) = \sum_{k=0}^{t-1} \vec{b}_\sigma^i[k] \cdot X^{A_\sigma^i[k]} \in R_p$.
2. For $\sigma \in \{0,1\}, i \in [0..c)$ and $k \in [0..t)$:
$$[\![A_\sigma^i[k]]\!]_2 \leftarrow \mathbf{Input}(P_\sigma, A_\sigma^i[k]) \text{ and } [\![\vec{b}_\sigma^i[k]]\!]_p \leftarrow \mathbf{Input}(P_\sigma, \vec{b}_\sigma^i[k]).$$
3. For every $i, j \in [0..c)$ and $k, l \in [0..t)$ (in parallel) the parties do the following:
   (a) $[\![\alpha_{k,l}^{i,j}]\!]_2 \leftarrow \mathbf{BitAdd}([\![A_0^i[k]]\!]_2, [\![A_1^j[l]]\!]_2)$.
   (b) Compute $[\![\beta_{k,l}^{i,j}]\!]_p \leftarrow \mathbf{Mult}([\![\vec{b}_0^i[k]]\!]_p, [\![\vec{b}_1^j[l]]\!]_p)$.
   (c) For each $i, j \in [0..c)$ and $k, l \in [0..t)$, call $\mathcal{F}_{\mathsf{c\text{-}SUV}}$ with length $2N$ on input $[\![\alpha_{k,l}^{i,j}]\!]_2$ and $[\![\beta_{k,l}^{i,j}]\!]_p$, and let $\vec{g}_{\sigma,k,l}^{i,j}$ denote the output to party P_σ.
4. For each $i, j \in [0..c)$, P_σ defines the vector
$$\vec{g}_\sigma^{i,j} = \sum_{j,k=0}^{t-1} \vec{g}_{\sigma,k,l}^{i,j}.$$
5. Party P_σ computes $x_\sigma = \langle \vec{a}, \vec{e}_\sigma \rangle \mod F(X)$, where $\vec{a} = (1, a_1, \ldots, a_{c-1})$, $\vec{e}_\sigma = (e_\sigma^0, \ldots, e_\sigma^{c-1})$.
   And, interpreting each $\vec{g}_\sigma^{i,j}$ as a degree $< 2N$ polynomial $g_{\sigma,i+cj}$ over Z_p, and setting $\vec{g}_\sigma := (g_0, \ldots, g_{c^2-1})$, party P_σ computes
$$z_\sigma = \langle \vec{a} \otimes \vec{a}, \vec{g}_\sigma \rangle \mod F(X).$$
6. Party P_σ outputs (x_σ, z_σ)

Figure 10: PCG protocol securely implementing the corruptible OLE functionality

Finally, the simulator forwards

$$x_\sigma := \langle \vec{a}, \vec{e}_\sigma \rangle \mod F(X)$$

and

$$z_\sigma := \langle \vec{a} \otimes \vec{a}, \vec{g}_\sigma \rangle \mod F(X)$$

to the functionality.

We conclude the proof by showing that an adversary $\mathcal{A}$ distinguishing the simulated protocol execution from a real execution can be turned into an adversary $\mathcal{B}$ on the module-LPN assumption with static leakage. Adversary $\mathcal{B}$ proceeds exactly as the simulation until the point, where it receives the predicate guesses $P^{i,j}_{k,l}$ from the adversary within the call of $\mathcal{F}_{\mathsf{c\text{-}SUV}}$. Instead of choosing $A^0, \dots, A^{c-1} \xleftarrow{\$} [0..N)^t, \vec{b}^0, \dots, \vec{b}^{c-1} \xleftarrow{\$} \mathbb{Z}_p^t \backslash \{0\}$, for each $i, j \in [0..c)$, $k, l \in [0..t)$ the simulator defines the predicate $Q^{i,j}_{k,l} \colon [0..N) \to \{0,1\}$ via $Q^{i,j}_{k,l}(A) = 0$, iff $P^{i,j}_{k,l}(A + A^j_\sigma[l]) = 0$ and forwards indices i, k and predicate $Q^{i,j}_{k,l}$ to the module-LPN game. If the game aborts on some index $i^\star, j^\star \in [0..c)$, $k^\star, l^\star \in [0..t)$, $\mathcal{B}$ samples $A^0, \dots, A^{c-1} \xleftarrow{\$} [0..N)^t$ such that $Q^{i^\star,j^\star}_{k^\star,l^\star}(A^{i^\star}[k^\star]) = 0$ and $Q^{i,j}_{k,l}(A^i[k]) = 1$ for all previously queried tuples (i, j, k, l). Further, $\mathcal{B}$ samples $\vec{b}^0, \dots, \vec{b}^{c-1} \xleftarrow{\$} \mathbb{Z}_p^t \backslash \{0\}$, returns $(A^{i^\star}[k^\star] + A^{j^\star}_\sigma[l^\star], \vec{b}^i[k] \cdot \vec{b}^j_\sigma[l])$ to $\mathcal{A}$ and aborts. Note that $\mathcal{B}$ can sample $A^0, \dots, A^{c-1}$ as required by sampling fresh tuples $A^0, \dots, A^{c-1} \xleftarrow{\$} [0..N)^t$ until the predicates take the values as required. For all events that happen with noticeable probability, this yields an adversary in expected polynomial time.

If the module LPN game does not abort, $\mathcal{B}$ receives $u \in R_p$ from the experiment and sets $x_{1-\sigma} := u$. The adversary $\mathcal{B}$ computes $z_{1-\sigma} = x_0 \cdot x_1 - z_\sigma$ (where x_σ and z_σ are computed as in the simulation). Finally, $\mathcal{B}$ outputs 0 if and only if the adversary $\mathcal{A}$ returns "real".

Note that in case $b = 0$ (i.e. the module-LPN game returns a real module-LPN sample), adversary $\mathcal{B}$ simulates the real protocol execution except with negligible probability: First, note that the probability that the adversary gets any of the guesses B to the functionality $\mathcal{F}_{\mathsf{c\text{-}SUV}}$ right is negligible, since p is superpolynomial and $|B| \leq 2N$. Next, observe that the output distribution simulated by $\mathcal{B}$ corresponds to the real output distribution, and the module-LPN game aborts if and only if the functionality $\mathcal{F}_{\mathsf{c\text{-}SUV}}$ would have aborted on at least one of its calls. Therefore, $\mathcal{B}$ simulates the real protocol execution except with negligible probability. In case $b = 1$, $\mathcal{B}$ perfectly simulates the simulation, since the probability that the module-LPN game aborts equals the probability that the simulation aborts in an ideal execution. This concludes the proof.

□

Distributed generation of authenticated multiplication triples in the malicious setting. We finally present the distributed setup for our PCG protocol to generate authenticated multiplication triples $\Pi_{\mathsf{mal\text{-}triple}}$ (Fig. 12). To additionally generate shares of the MACs, the protocol uses $\mathcal{F}_{\mathsf{DPF}}$ with outputs in $\mathbb{F}_p^2$; this can be done using the modification in Remark 5.2.

Note that the corruptible functionality $\mathcal{F}_{\mathsf{mal\text{-}triple}}$ (similar to $\mathcal{F}_{\mathsf{c\text{-}SUV}}$) gives the adversary full control over its share of the output. Recall that this is sufficient to use our PCG construction as a "plug-in" replacement in a wide range of natural MPC protocols that use correlated randomness, such as [DPSZ12]. As the proof of the following is very similar to the proof of Theorem 6.3, we omit it here.

Theorem 6.4 *Let $R = \mathbb{Z}_p[X]/F(X)$ for some prime p and degree-N polynomial $F(X) \in \mathbb{Z}[X]$, and let $c, t \in \mathbb{N}$. If the R^c-$\mathsf{LPN}_{p,t}$ problem with static leakage is hard, then the protocol $\Pi_{\mathsf{mal\text{-}triple}}$ (Fig. 12) implements the functionality $\mathcal{F}_{\mathsf{mal\text{-}triple}}$ (Fig. 11) with security against malicious adversaries in the $\mathcal{F}_{\mathsf{2\text{-}PC}}, \mathcal{F}_{\mathsf{c\text{-}SUV}}$-hybrid model.*

Functionality $\mathcal{F}_{\mathsf{mal\text{-}triple}}$

PARAMETERS: Security parameter 1^λ, modulus p, and the ring $R = \mathbb{Z}[X]/F(X)$, where $F(X)$ has degree N.
FUNCTIONALITY:
If both parties are honest:

1. Sample $\gamma_0, \gamma_1 \leftarrow \mathbb{Z}_p$ and let $\gamma = \gamma_0 + \gamma_1$.
2. Sample $x_0, x_1, y_0, y_1 \leftarrow R_p$ and let $x = x_0 + x_1$, $y = y_0 + y_1$.
3. Compute $z = x \cdot y$.
4. Choose $m_{x,0}, m_{y,0}, m_{z,0} \leftarrow R_p$ and define $m_{x,1} := \gamma \cdot x - m_{x,0}$, $m_{y,1} := \gamma \cdot y - m_{y,0}$ and $m_{z,1} := \gamma \cdot z - m_{z,0}$.
5. Output $(\gamma_\sigma, x_\sigma, y_\sigma, z_\sigma, m_{x,\sigma}, m_{y,\sigma}, m_{z,\sigma})$ to party P_σ, for $\sigma \in \{0,1\}$

If party P_σ is corrupted:

1. Wait for input $(\gamma_\sigma, x_\sigma, y_\sigma, z_\sigma, m_{x,\sigma}, m_{y,\sigma}, m_{z,\sigma}) \in \mathbb{Z}_p \times R_p^6$ from the adversary.
2. Sample $\gamma_{1-\sigma} \leftarrow \mathbb{Z}_p$ and $x_{1-\sigma}, y_{1-\sigma} \leftarrow R_p$. Set $\gamma := \gamma_0 + \gamma_1$, $x := x_0 + x_1$, $y := y_0 + y_1$ and $z := x \cdot y$. Compute $z_{1-\sigma} \leftarrow z - z_\sigma$, as well as $m_{x,1-\sigma} \leftarrow \gamma \cdot x - m_{x,\sigma}$, $m_{y,1-\sigma} \leftarrow \gamma \cdot y - m_{y,\sigma}$ and $m_{z,1-\sigma} \leftarrow \gamma \cdot z - m_{z,\sigma}$.
3. Output $(\gamma_{1-\sigma}, x_{1-\sigma}, y_{1-\sigma}, z_{1-\sigma}, m_{x,1-\sigma}, m_{y,1-\sigma}, m_{z,1-\sigma})$ to the honest party.

Figure 11: Corruptible functionality for authenticated triples

6.3 Efficiency Analysis

In this section we analyze the efficiency of the presented protocols for distributed setup. For an overview of concrete efficiency for the generation of OLEs in the semi-honest and malicious setting, we refer to Table 3 in Section 9.

We start this section by providing the costs for implementing the secure functionalities $\mathcal{F}_{\mathsf{2\text{-}PC}}$ and $\mathcal{F}_{\mathsf{ext\text{-}2\text{-}PC}}$ from Section 5. We assume that secret values in these functionalities are additively secret-shared between the two parties. In the case of malicious security, the shares are also authenticated with information-theoretic MACs as in [BDOZ11, NNOB12], which ensures correct opening of shares.

Input(P_σ, x)**:** For semi-honest security, this has no communication cost as the parties define their respective shares to be x and zero. For malicious security, P_σ needs to have an authenticated random value, which can be preprocessed, and then sends ℓ bits for $x \in \{0,1\}^\ell$ or $\log p$ bits for $x \in \mathbb{Z}_p$, to use the random value to authenticate x.

Add: This is a local operation.

Mult$(\llbracket x \rrbracket_M, \llbracket y \rrbracket_M)$**:** Uses one multiplication triple over $\mathbb{Z}_M$, with an online communication cost of $2 \log M$ bits per party (from two calls to **Output**).

BitAdd: Evaluates a binary circuit on inputs of length ℓ bits, using ℓ AND gates. This consumes ℓ multiplication triples over $\mathbb{Z}_2$, giving an online communication cost of 2ℓ bits per party.

Output$(\llbracket x \rrbracket_M)$**:** This requires sending $\log M$ bits per party. With malicious security, the parties

Protocol $\Pi_{\mathsf{mal\text{-}triple}}$ *(Part I)*

Parameters: Security parameter 1^λ, natural number $N = 2^k$, prime p, access to functionality $\mathcal{F}_{\mathsf{c\text{-}SUV}}$ with length $2N$, prime modulus p, polynomial $F(X) \in \mathbb{Z}[X]$ of degree N, ring $R = \mathbb{Z}_p[X]/F(X)$, and parameters $c, t \in \mathbb{N}$ (corresponding to the syndrome compression factor and noise rate of the LPN assumption, respectively).
Public input: random polynomials $a_1, \ldots, a_{c-1} \in R_p$, used for R^c-LPN
Protocol:

1. The parties jointly setup $\gamma \in \mathbb{Z}_p$ and t-sparse polynomials $e^0, ..., e^{c-1}$ and $f^0, \ldots, f^{c-1}$, which will be used to generate x and y, respectively.
   (a) For $\sigma \in \{0,1\}$, party P_σ chooses $\gamma_\sigma \in \mathbb{Z}_p$, and inputs $[\![\gamma_\sigma]\!]_p \leftarrow \mathsf{Input}(P_\sigma, \gamma_\sigma)$. The parties compute $[\![\gamma]\!]_p \leftarrow [\![\gamma_0]\!]_p + [\![\gamma_1]\!]_p$.
   (b) For $\sigma \in \{0,1\}, i \in [0..c)$, party P_σ chooses $A^i_{e,\sigma} \leftarrow [0..N)^t$, $\vec{b}^i_{e,\sigma} \leftarrow (\mathbb{Z}_p^*)^t$. For $k \in [0..t)$:
      i. $[\![A^i_{e,\sigma}[k]]\!]_2 \leftarrow \mathbf{Input}(P_\sigma, A^i_{e,\sigma}[k])$ //P_σ inputs k-th non-zero position of e^i_σ
      ii. $[\![\vec{b}^i_{e,\sigma}[k]]\!]_p \leftarrow \mathbf{Input}(P_\sigma, \vec{b}^i_{e,\sigma}[k])$ //P_σ inputs the k-th payload of e^i_σ
      iii. $[\![A^i_e[k]]\!]_2 \leftarrow [\![A^i_{e,0}[k]]\!]_2 \oplus [\![A^i_{e,1}[k]]\!]_2$ //compute k-th position of $e^i = e^i_0 + e^i_1$
      iv. $[\![\vec{b}^i_e[k]]\!]_p \leftarrow [\![\vec{b}^i_{e,0}[k]]\!]_p + [\![\vec{b}^i_{e,1}[k]]\!]_p$ //compute k-th payload of $e^i = e^i_0 + e^i_1$
      v. $[\![\vec{m}^i_e[k]]\!]_p \leftarrow \mathbf{Mult}([\![\vec{b}^i_e[k]]\!]_p, [\![\gamma]\!]_p)$ //multiply k-th payload of e^i by MAC key γ
      vi. Call $\mathcal{F}_{\mathsf{c\text{-}SUV}}$ with length N on input $[\![A^i_e[k]]\!]_2$ and $[\![\vec{b}^i_e[k], \vec{m}^i_e[k]]\!]_p$ such that party P_σ receives output $(\vec{e}^i_{\sigma,k}, \vec{u}^i_{\sigma,k}) \in \mathbb{Z}_p^N \times \mathbb{Z}_p^N$.
      //convert to additive shares of $e^i, \gamma \cdot e^i$
   (c) Repeat step (b) for $f^0, \ldots, f^{c-1}$ with result $(\vec{f}^i_{\sigma,k}, \vec{v}^i_{\sigma,k}) \in \mathbb{Z}_p^N \times \mathbb{Z}_p^N$.
2. For every $i, j \in [0..c)$ and $k, l \in [0..t)$ (in parallel) the parties do the following:
   (a) $[\![\alpha^{i,j}_{k,l}]\!]_2 \leftarrow \mathbf{BitAdd}([\![A^i_e[k]]\!]_2, [\![A^j_f[l]]\!]_2)$ //compute $(k+tl)$-th non-zero position of $e^i \cdot f^j$
   (b) $[\![\beta^{i,j}_{k,l}]\!]_p \leftarrow \mathbf{Mult}([\![\vec{b}^i_e[k]]\!]_p, [\![\vec{b}^j_f[k]]\!]_p)$ //compute $(k+tl)$-th payload of $e^i \cdot f^j$
   (c) $[\![m^{i,j}_{k,l}]\!]_p \leftarrow \mathbf{Mult}([\![\beta^{i,j}_{k,l}]\!]_p, [\![\gamma]\!]_p)$ //multiply $(k+tl)$-th payload of $e^i \cdot f^j$ by MAC key γ
   (d) For each $i, j \in [0..c)$ and $k, l \in [0..t)$, call $\mathcal{F}_{\mathsf{c\text{-}SUV}}$ with length $2N$ on input $[\![\alpha^{i,j}_{k,l}]\!]_2$ and $[\![\beta^{i,j}_{k,l}, m^{i,j}_{k,l}]\!]_p$, and let $(\vec{g}^{i,j}_{\sigma,k,l}, \vec{w}^{i,j}_{\sigma,k,l})$ denote the output to party P_σ.
   //convert to additive shares of $e^i \cdot f^j$, $\gamma \cdot e^i \cdot f^j$
3. For each $i \in [0..c)$, P_σ defines the vectors
$$\vec{e}^i_\sigma = \sum_{k=0}^{t-1} \vec{e}^i_{\sigma,k}, \quad \vec{u}^i_\sigma = \sum_{k=0}^{t-1} \vec{u}^i_{\sigma,k}, \quad \vec{f}^i_\sigma = \sum_{k=0}^{t-1} \vec{f}^i_{\sigma,k} \quad \text{and} \quad \vec{v}^i_\sigma = \sum_{k=0}^{t-1} \vec{v}^i_{\sigma,k}.$$
4. For each $i, j \in [0..c)$, P_σ defines the vectors
$$\vec{g}^{i,j}_\sigma = \sum_{j,k=0}^{t-1} \vec{g}^{i,j}_{\sigma,k,l} \quad \text{and} \quad \vec{w}^{i,j}_\sigma = \sum_{j,k=0}^{t-1} \vec{w}^{i,j}_{\sigma,k,l}.$$

Figure 12: PCG protocol implementing the corruptible functionality for authenticated triples (Part I)

Protocol $\Pi_{\textsf{mal-triple}}$ *(Part II)*

PROTOCOL: *(continued)*

5. Interpret each $\vec{e}^{\,i}_\sigma$ as a degree $< N$ polynomial $e_{\sigma,i}$ over $\mathbb{Z}_p$, and set $\vec{e}_\sigma := (e_0, \ldots, e_{c-1})$ (accordingly for $\vec{u}^{\,i}_\sigma, \vec{f}^{\,i}_\sigma$ and $\vec{v}^{\,i}_\sigma$).
6. Interpret each $\vec{g}^{\,i,j}_\sigma$ as a degree $< 2N$ polynomial $g_{\sigma,i+cj}$ over $\mathbb{Z}_p$, and set $\vec{g}_\sigma := (g_0, \ldots, g_{c^2-1})$ (accordingly for $\vec{w}^{\,i,j}_\sigma$).
7. Compute
$$x_\sigma = \langle \vec{a}, \vec{e}_\sigma \rangle, y_\sigma = \left\langle \vec{a}, \vec{f}_\sigma \right\rangle, z_\sigma = \langle \vec{a} \otimes \vec{a}, \vec{g}_\sigma \rangle \quad \text{and}$$
$$m_{x,\sigma} = \langle \vec{a}, \vec{u}_\sigma \rangle, m_{y,\sigma} = \langle \vec{a}, \vec{v}_\sigma \rangle, m_{z,\sigma} = \langle \vec{a} \otimes \vec{a}, \vec{w}_\sigma \rangle$$
all modulo $F(X)$.
8. Output $(\gamma_\sigma, x_\sigma, y_\sigma, z_\sigma, m_{x,\sigma}, m_{y,\sigma}, m_{z,\sigma})$

Figure 12: PCG protocol implementing the corruptible functionality for authenticated triples (Part II)

also need to send and check the MACs on shares, however, this can be amortized for a large batch of openings by checking a random linear combination, so we ignore this.

Inv($[\![x]\!]_p$)**:** This can be done using one multiplication triple, and two openings.[9]

MixedMult($x_0, x_1, [\![\alpha]\!]_2$)**:** With semi-honest security, this uses two string-OTs of length ℓ, plus $\ell + 1$ bits of communication from each party. With malicious security, when α is already authenticated with MACs, this can be done with ℓ bits of communication from each party without any additional preprocessing. To see this, recall that a share α_σ is authenticated by giving P_σ a MAC $M = \alpha_\sigma K + K'$, where $K, K' \in \{0,1\}^\lambda$ are random MAC keys known to $P_{1-\sigma}$. By hashing M and K' respectively, the parties obtain secret shares of $\alpha_\sigma \cdot z$, where $z = H(K') \oplus H(K' \oplus K)$ is known to $P_{1-\sigma}$. These can be converted into shares of $\alpha_\sigma \cdot x_{1-\sigma}$ with ℓ bits of communication; the complete multiplication is then obtained symmetrically.

PassiveOutput(x_0, x_1)**:** ($x_0, x_1 \in \{0,1\}^{\log M}$) Here, the parties exchange shares by sending $\log M$ bits each.

Cost of Preprocessing. We assume that multiplication triples over $\mathbb{Z}_p$ can be obtained by bootstrapping from a previous PCG instance, so essentially obtained at no cost. Similarly, for the **Input** phase we can obtain authenticated random values with vector-OLE (over $\mathbb{Z}_p$) or correlated OT (over $\mathbb{Z}_2$), using the efficient PCGs from previous works [BCGI18, BCG+19b, BCG+19a]. With semi-honest security, we can also obtain multiplication triples over $\mathbb{Z}_2$ using a PCG for OT. However, we cannot do this for malicious security, since we do not have an efficient PCG for authenticated triples over $\mathbb{Z}_2$. Therefore, we build these from correlated OT using the TinyOT protocol from [FKOS15, HSS17], for a cost of 32 bits of communication per party and 54 correlated OTs, for each triple.

Main Subprotocols. Before stating the complexities of distributed key generation, we give an overview of the complexity of the subprotocols $\Pi_{\textsf{DPF}}$ (based on [Ds17])/ $\Pi_{\textsf{c-SUV}}$ (see Figure 6) for

[9] Given a triple $[\![a]\!], [\![b]\!], [\![c]\!]$ with $c = ab$, first open $x + a$, then compute $[\![xb]\!] = (x + a)[\![b]\!] - [\![c]\!]$, open xb and finally compute $[\![z]\!] = [\![b]\!](xb)^{-1}$.

implementing $\mathcal{F}_{\mathsf{DPF}}$ (semi-honest)/ $\mathcal{F}_{\mathsf{c\text{-}SUV}}$ (malicious) with domain size D and output shares in $\mathbb{Z}_p^\ell$ (see Remark 5.2 for obtaining outputs in $\mathbb{Z}_p^\ell$). Note that for the following we assume $\lambda > \log p$, otherwise the parties have to perform an additional $D \cdot \lfloor (\log p)\lambda \rfloor$ PRG operations.

Basic protocol: the basic protocol has the following complexity:

- $2D$ evaluations of PRG //Step 2
- $\log D$ **Mult**/**MixedMult** over $\{0,1\}^\lambda \times \{0,1\}$ //Step 2
- $\log D$ **Output**/**PassiveOutput** over $\{0,1\}^{\lambda+2}$ //Step 2
- $\ell + 1$ **Input** (per party) + $\ell + 1$ **Output** over $\mathbb{Z}_p$ //Step 4
- ℓ **Inv** + ℓ **Mult** over $\mathbb{Z}_p$ //Step 4

Consistency check: (only required for malicious security)

- ℓ **Mult** over $\mathbb{Z}_p$ //Step 7
- $\ell + 1$ **Input** (per party) + ℓ **Output** over $\mathbb{Z}_p$ //Step 7

With implementing the functionalities for secure 2-party computation as described, this adds up to the following complexity:

Correlated randomness: Consuming correlated randomness in the form of:

- semi-honest: 2ℓ multiplication triples + $2 \log D$ λ-string-OTs
- malicious: 3ℓ multiplication triples + $2\times$ length-$(2\ell + 2)$ vector-OLE (both over $\mathbb{Z}_p$)

Computation: Computation is dominated by $2D$ PRG evaluations.

Communication: Communication (per party) is dominated by:

- $(2\lambda + 3) \log D + 5\ell \log p$ bits
- malicious: extra $(4\ell + 2) \log p$ bits

Cost of distributed setup. The complexities of our protocol $\Pi_{\mathsf{OLE\text{-}Setup}}$ (Fig. 8/ $\Pi_{\mathsf{mal\text{-}OLE}}$ (Fig. 10 implementing $\mathcal{F}_{\mathsf{OLE\text{-}Setup}}$ (semi-honest)/ $\mathcal{F}_{\mathsf{mal\text{-}OLE}}$ (malicious) to produce N OLEs are as follows:

1. ct **Input** over $\{0,1\}^{\log N}$ and $\mathbb{Z}_p$ (per party) //positions + payloads of e_0^i, e_1^i
2. $(ct)^2$ **BitAdd** over $\{0,1\}^{\log N}$ + $(ct)^2$ **Mult** over $\mathbb{Z}_p$ //positions + payloads of $e_0^i \cdot e_1^j$
3. $(ct)^2$ invocations of $\mathcal{F}_{\mathsf{DPF}}$/ $\mathcal{F}_{\mathsf{c\text{-}SUV}}$ with length $2N$ over $\mathbb{Z}_p$ ($\ell = 1$) //convert to shares of $e_0^i \cdot e_1^j$

For protocol $\Pi_{\mathsf{mal\text{-}triple}}$ (Fig. 12) implementing $\mathcal{F}_{\mathsf{mal\text{-}triple}}$ (malicious)the costs are as follows:

1. $2ct$ **Input** over $\{0,1\}^{\log N}$ and $\mathbb{Z}_p$ (per party) //positions + payloads of $e_0^i, e_1^i, f_0^i, f_1^i$
2. $2ct + (ct)^2$ **Mult** over $\mathbb{Z}_p$ //multiply payloads of $e_0^i, e_1^i, f_0^i, f_1^i$ by MAC γ
3. $2ct$ $\mathcal{F}_{\mathsf{c\text{-}SUV}}$ with length N and output $\mathbb{Z}_p^2$ //convert to shares of $e_0^i + e_1^i$, $f_0^i + f_1^i$
4. $(ct)^2$ **BitAdd** over $\{0,1\}^{\log N}$ + $(ct)^2$ **Mult** over $\mathbb{Z}_p$ //positions + payloads of $e^i \cdot f^j$
5. $(ct)^2$ invocations of $\mathcal{F}_{\mathsf{DPF}}$/ $\mathcal{F}_{\mathsf{c\text{-}SUV}}$ with length $2N$ over $\mathbb{Z}_p^2$ ($\ell = 2$) //convert to shares of $e^i \cdot f^j$

Altogether, we obtain the following theorems:

Theorem 6.5 (OLE Distributed Generation) *Assuming hardness of R^c-$\mathsf{LPN}_{p,1,t}$ (Def. 3.2), there exists a protocol securely realizing functionality $\mathcal{F}_{\mathsf{OLE\text{-}Setup}}$/ $\mathcal{F}_{\mathsf{mal\text{-}OLE}}$ to produce N OLEs against semi-honest adversaries/ malicious adversaries with the following complexity (note that costs not specified in the following apply to both):*

Correlated randomness: *Consuming correlated randomness in the form of:*

1. *malicious: $2ct\log N$ correlated OTs + $2\times$ length-ct vector-OLE (in $\mathbb{Z}_p$)*
2. - *semi-honest: $2(ct)^2\log N$ bit-OTs + $(ct)^2$ OLEs over $\mathbb{Z}_p$*
   - *malicious: $54(ct)^2\log N$ correlated OTs + $(ct)^2$ authenticated triples over $\mathbb{Z}_p$*
3. - *$4(ct)^2$ OLEs + $2(ct)^2\log(2N)$ λ-string-OTs*
   - *malicious: $3(ct)^2$ authenticated triples + $2\times$ length-$4(ct)^2$ vector-OLE over $\mathbb{Z}_p$*

Computation: *Computation is dominated by $2(ct)^2N$ PRG evaluations.*

Communication: *Communication (per party) is dominated by:*

1. *malicious: $ct(\log N + \log p)$ bits*
2. - *semi-honest: $2(ct)^2\log N$ + $2(ct)^2\log p$ bits*
   - *malicious: $34(ct)^2\log N$ + $2(ct)^2\log p$ bits*
3. - *$(ct)^2((2\lambda+3)\log(2N) + 5\log p)$ bits*
   - *malicious: additional $(ct)^2\cdot 6\log p)$ bits*

Theorem 6.6 (Authenticated Triples Distributed Generation) *Assuming hardness of the R^c-$\mathsf{LPN}_{p,1,t}$ assumption (Def. 3.2), there exists a protocol securely realizing functionality $\mathcal{F}_{\mathsf{mal\text{-}triple}}$ to produce N authenticated triples against malicious adversaries with the following complexity:*

Correlated randomness: *Consuming correlated randomness in the form of:*

1. *$4ct\log N$ correlated OTs + $2\times$ length-$2ct$ vector-OLE (in $\mathbb{Z}_p$)*

2. & 4. *$54(ct)^2\log N$ bit OTs + $2(ct+(ct)^2)$ authenticated triples over $\mathbb{Z}_p$*

3. & 5. *$6(2ct+(ct)^2)$ authenticated triples over $\mathbb{Z}_p$ + $2\times$ length-$6(2ct+(ct)^2)$ vector-OLE over $\mathbb{Z}_p$*

Computation: *Computation is dominated by $2(2ct+(ct)^2)N$ PRG evaluations.*

Communication: *Communication (per party) is dominated by:*

1. *$2ct(\log N + \log p)$ bits*

2. & 4. *$34(ct)^2\log N$ + $4(ct+(ct)^2)\log p$ bits*

3. & 5. *$(2ct+(ct)^2)\cdot((2\lambda+3)\log(2N) + 10\log p)$ bits*

Remark 6.1 (Optimizing efficiency) *If N is a power of 2, the above protocols can be optimized by taking the noise positions mod N and therefore reducing the input length to $\mathcal{F}_{\mathsf{DPF}}/\mathcal{F}_{\mathsf{c\text{-}SUV}}$ from $2N$ to N.*

Alternatively, by relying on the hardness of R^c-LPN with a regular *error distribution (where each noisy coordinate lies in a fixed interval of size N/t), one can replace all occurrences of N in the efficiency estimates above by N/t.*

Recall that except for a one-time setup cost the generation of correlated randomness is *almost for free* in terms of communication: For the generation of OT and vector-OLE with semi-honest and even malicious security, we can build on the silent OT and vector-OLE extensions from [BCG+19b, BCG+19a]. Further, for the generation of OLE and authenticated multiplication triples we can bootstrap and therefore only need to generate these correlations from scratch once. For generating OLEs with malicious security one can bootstrap using the PCG for authenticated multiplication triples (at comparatively little extra cost).

7 Extensions and Applications

In this section we extend our PCG for OLE in several directions. First, we build PCGs for inner product correlations from OLE over R_p, with the advantage that we do not need to rely on the full reducibility of $F(X)$, and can also obtain correlations over $\mathbb{F}_2$. Secondly, we present a method for building PCGs for general bilinear correlations, such as matrix multiplication, in a black-box way from our previous PCG. Finally, we show that all of these PCGs for degree two correlations can be extended in a natural way to the multi-party setting.

7.1 Bilinear Correlations

The class of bilinear correlations we consider is as follows.

Definition 7.1 (Simple Bilinear Correlation) *Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ be Abelian groups and $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be a bilinear map. We define the* simple bilinear correlation *for e by the distribution $\mathcal{C}_e$ over $(\mathbb{G}_1 \times \mathbb{G}_T) \times (\mathbb{G}_2 \times \mathbb{G}_T)$ of the form*

$$\mathcal{C}_e = \left\{((r_0, s_0), (r_1, s_1)) \mid r_0 \xleftarrow{\$} \mathbb{G}_1, r_1 \xleftarrow{\$} \mathbb{G}_2, s_0 \xleftarrow{\$} \mathbb{G}_T, s_1 = e(r_0, r_1) - s_0\right\}.$$

We denote by $\mathcal{C}_e^n$ the correlation that outputs n independent samples from $\mathcal{C}_e$, i.e.

$$\mathcal{C}_e^n = \left\{((R_0, S_0), (R_1, S_1)) \mid R_0 \xleftarrow{\$} \mathbb{G}_1^n, R_1 \xleftarrow{\$} \mathbb{G}_2^n, S_0 \xleftarrow{\$} \mathbb{G}_T^n, S_1 = e(R_0, R_1) - S_0\right\},$$

where we define $e\colon \mathbb{G}_1^n \times \mathbb{G}_2^n \to \mathbb{G}_T^n$ as the bilinear map obtained by applying e componentwise.

This covers several common correlations like OT and OLE, for example, OLE over a ring R can be obtained with $\mathbb{G}_1 = \mathbb{G}_2 = \mathbb{G}_T = (R, +)$ and $e(x, y) = x \cdot y$. Also, note that two independent bilinear correlations can be locally converted to produce an additively secret-shared instance of the correlation — for example, two OLEs are locally equivalent to one multiplication triple.

7.2 Inner Product Correlations

An inner product correlation is a simple bilinear correlation with the inner product map over $\mathbb{Z}_p$. These can be used to compute inner products in an MPC online phase, in a similar way to using multiplication triples. Inner products are common in tasks involving linear algebra, like privately evaluating or training machine learning models such as SVMs and neural networks, and a single inner product can also be used to measure the similarity between two input vectors.

We remark that given n random OLEs in $\mathbb{F}_p$, it is easy to locally convert these into a length-n inner product correlation, so we can build a PCG for inner products of any length using a PCG for OLE. However, the constructions in this section *do not* rely on the fully-reducible ring-LPN assumption that is needed for OLE in $\mathbb{F}_p$; instead, we use the ring-OLE construction from Fig. 1 over more conservative rings, which do not split completely into linear factors. Further, the constructions in this section generalize to rings $\mathbb{Z}_p$ for arbitrary $p \in \mathbb{N}$ (including $p = 2$).

Lemma 7.1 *Let $R_p = \mathbb{Z}_p[X]/F(X)$, where $F(X)$ is a degree-N polynomial with non-zero constant coefficient. Then, a single OLE over R_p can be locally converted into an inner product correlation over $\mathbb{Z}_p^N$.*

Proof. For $a \in R_p$, define the matrix over $\mathbb{Z}_p$ that corresponds to multiplication by a, as

$$M_a = \begin{pmatrix} | & | & & | \\ a & aX & \cdots & aX^{N-1} \\ | & | & & | \end{pmatrix}$$

where each aX^i is reduced modulo $F(X)$ and represented as a vector of coefficients in $\mathbb{Z}_p^N$.

We have that for any $b \in R_p$ represented as a vector of coefficients, the coefficient vector of ab is $M_a \cdot b$. Also, it is easy to see that if a is uniformly distributed in R_p then the first row of M_a, denoted $M_a[0]$, is uniform over $\mathbb{Z}_p^N$, since aX^i has a_{N-i} appearing in its constant term. So given an OLE $(a, c), (b, d)$ over R_p, where $c - d = ab$, the parties can define an inner product correlation $(M_a[0], c_0), (b, -d_0)$, where $c_0 - d_0 = \langle M_a[0], b\rangle$. $\square$

Note that, for the special case of $F(X) = X^n + 1$, the vector $M_a[0]$ can be computed as $(a_0, -a_{n-1}, \ldots, -a_1)$, without any modular reductions.

Corollary 7.2 (Large inner product from irreducible ring-LPN) *Suppose the $R\text{-}\mathsf{LPN}_{p,1,t}$ assumption holds for $R = \mathbb{Z}_p[X]/F(X)$, where $F(X)$ is degree N and irreducible over $\mathbb{Z}_p$. Then there is a PCG for the length-N inner product correlation over $\mathbb{Z}_p$, where the seeds have size $O(\lambda t^2 \log N)$ bits, and the computational complexity of the* Expand *operation is $\tilde{O}(N)$ operations in $\mathbb{Z}_p$, plus $O(t^2 N)$ PRG operations.*

Corollary 7.3 (Small inner products from reducible ring-LPN) *Suppose the $R\text{-}\mathsf{LPN}_{p,1,t}$ assumption holds for $R = \mathbb{Z}_p[X]/F(X)$, where $F(X)$ is degree N and splits into N/d distinct factors of degree d. Then there is a PCG for producing N/d instances of length-d inner product correlation over $\mathbb{Z}_p$, with the same seed size and complexity as above.*

The latter construction has two benefits over naively using OLE over $\mathbb{F}_p$ to generate an inner product. Firstly, OLE in $\mathbb{F}_p$ requires that R splits fully into *linear factors*, whereas for inner products the factors can be degree-d (and irreducible), which is a much more conservative assumption; in particular, the dimension-reduction attack we consider in Section 8 is less effective. Secondly, we can also use this to generate inner products over small fields such as $\mathbb{F}_2$, whereas we cannot efficiently obtain OLEs over $\mathbb{F}_2$ with our present constructions.

7.3 Bilinear Correlations from Programmable PCG for OLE

We can build a PCG to create a large batch of samples from *any* simple bilinear correlation, using the PCG for OLE from Section 4. To do this, we exploit the fact that this PCG is *programmable*, which, roughly speaking, means that one party can "reuse" its input a or b in several instances of the PCG, while maintaining security. Boyle et al. [BCG+19b] previously used this property to construct multi-party PCGs from several instances of programmable two-party PCGs; unlike their work, we exploit the property for a different purpose in the two-party setting.

In the following, we recall the definition of programmability, and show that our PCG for OLE satisfies this definition.

Definition 7.2 (Programmable PCG) *A tuple of algorithms* $\mathsf{PCG} = (\mathsf{PCG.Gen}, \mathsf{PCG.Expand})$ *following the syntax of a standard PCG, but where* $\mathsf{PCG.Gen}(1^\lambda)$ *takes additional random inputs $\rho_0, \rho_1 \in \{0,1\}^\star$, is a* programmable PCG *for a simple bilinear 2-party correlation $\mathcal{C}_e^n$ (specified by $e \colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$) if the following holds:*

- **Correctness.** *The correlation obtained via:*

$$\left\{ ((R_0,S_0),(R_1,S_1)) \;\middle|\; \begin{array}{l} \rho_0,\rho_1 \xleftarrow{\$} \$, (\mathsf{k}_0,\mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda,\rho_0,\rho_1), \\ (R_\sigma,S_\sigma) \leftarrow \mathsf{PCG.Expand}(\sigma,\mathsf{k}_\sigma) \text{ for } \sigma \in \{0,1\} \end{array} \right\}$$

is computationally indistinguishable from $\mathcal{C}_e^n(1^\lambda)$*.*

- ***Programmability.*** *There exist public efficiently computable functions* $\phi_0\colon \{0,1\}^\star \to \mathbb{G}_1^n$*,* $\phi_1\colon \{0,1\}^\star \to \mathbb{G}_2^n$ *such that*

$$\Pr\left[\begin{array}{l} \rho_0,\rho_1 \leftarrow \$, (\mathsf{k}_0,\mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda,\rho_0,\rho_1) \\ (R_0,S_0) \leftarrow \mathsf{PCG.Expand}(0,\mathsf{k}_0), \\ (R_1,S_1) \leftarrow \mathsf{PCG.Expand}(1,\mathsf{k}_1) \end{array} : \begin{array}{l} R_0 = \phi_0(\rho_0) \\ R_1 = \phi_1(\rho_1) \end{array} \right] \geq 1 - \mathsf{negl}(\lambda),$$

where $e\colon \mathbb{G}_1^n \times \mathbb{G}_2^n \to \mathbb{G}_T^n$ *is the bilinear map obtained by applying e componentwise.*

- ***Programmable security.*** *The distributions*

$$\left\{ (\mathsf{k}_1,(\rho_0,\rho_1)) \;\middle|\; \rho_0,\rho_1 \leftarrow \$, (\mathsf{k}_0,\mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda,\rho_0,\rho_1) \right\} \quad \textit{and}$$
$$\left\{ (\mathsf{k}_1,(\rho_0,\rho_1)) \;\middle|\; \rho_0,\rho_1,\tilde{\rho}_0 \leftarrow \$, (\mathsf{k}_0,\mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda,\tilde{\rho}_0,\rho_1) \right\}$$

as well as

$$\left\{ (\mathsf{k}_0,(\rho_0,\rho_1)) \;\middle|\; \rho_0,\rho_1 \leftarrow \$, (\mathsf{k}_0,\mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda,\rho_0,\rho_1) \right\} \quad \textit{and}$$
$$\left\{ (\mathsf{k}_0,(\rho_0,\rho_1)) \;\middle|\; \rho_0,\rho_1,\tilde{\rho}_1 \leftarrow \$, (\mathsf{k}_0,\mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda,\rho_0,\tilde{\rho}_1) \right\}$$

are computationally indistinguishable.

We start by showing that a programmable PCG is a PCG in the standard sense.

Lemma 7.4 *Let* $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ *be a bilinear map. Then, a programmable PCG* $\mathsf{PCG} = (\mathsf{PCG.Gen},\mathsf{PCG.Expand})$ *for the bilinear 2-party correlation* $\mathcal{C}_e^n$ *is also a PCG for* $\mathcal{C}_e^n$ *in the sense of Definition 2.6 (where the sampling of* ρ_0,ρ_1 *happens inside* $\mathsf{PCG.Gen}$*).*

Proof. We have to show that the PCG satisfies standard security. More precisely, we have to prove that if $\mathsf{PCG} = (\mathsf{Gen},\mathsf{Expand})$ satisfies correctness, programmability and programmable security, then the distributions

$$\mathcal{D}^{\mathsf{real}} := \left\{ (\mathsf{k}_1,(R_0,S_0)) \;\middle|\; \begin{array}{l} \rho_0,\rho_1 \leftarrow \$, (\mathsf{k}_0,\mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda,\rho_0,\rho_1) \\ (R_0,S_0) \leftarrow \mathsf{PCG.Expand}(0,\mathsf{k}_0) \end{array} \right\} \quad \text{and}$$

$$\mathcal{D}^{\mathsf{sim}} := \left\{ (\mathsf{k}_1,(R_0,S_0)) \;\middle|\; \begin{array}{l} \rho_0,\rho_1 \leftarrow \$, (\mathsf{k}_0,\mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda,\rho_0,\rho_1) \\ (R_1,S_1) \leftarrow \mathsf{PCG.Expand}(1,\mathsf{k}_1) \\ R_0 \xleftarrow{\$} \mathbb{G}_1, S_0 = e(R_0,R_1) - S_1 \end{array} \right\}$$

are computationally indistinguishable (the case k_0 is symmetric). We proceed the proof via a sequence of hybrid distributions:

$$\mathcal{D}_1 := \left\{ (\mathsf{k}_1,(R_0,S_0)) \;\middle|\; \begin{array}{l} \rho_0,\rho_1 \leftarrow \$, (\mathsf{k}_0,\mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda,\rho_0,\rho_1) \\ (R_1,S_1) \leftarrow \mathsf{PCG.Expand}(1,\mathsf{k}_1) \\ R_0 = \phi_0(\rho_0), S_0 = e(R_0,R_1) - S_1 \end{array} \right\}$$

$$\mathcal{D}_2 := \left\{ (\mathsf{k}_1,(R_0,S_0)) \;\middle|\; \begin{array}{l} \rho_0,\rho_1 \leftarrow \$, (\mathsf{k}_0,\mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda,\rho_0,\rho_1) \\ (R_1,S_1) \leftarrow \mathsf{PCG.Expand}(1,\mathsf{k}_1) \\ \tilde{\rho}_0 \leftarrow \$, R_0 = \phi_0(\tilde{\rho}_0), S_0 = e(R_0,R_1) - S_1 \end{array} \right\}$$

The distribution $\mathcal{D}^{\mathsf{real}}$ and $\mathcal{D}_1$ are computationally close by the programmability and correctness of the PCG.

Next, let $\mathcal{A}$ be a distinguisher between the distribution $\mathcal{D}_1$ and $\mathcal{D}_2$. Then, we can construct an adversary $\mathcal{B}$ on the programmable security of PCG as follows. The adversary $\mathcal{B}$ obtains $(\mathsf{k}_1, (\rho_0, \rho_1))$ from its experiment. It computes $(R_1, S_1) \leftarrow \mathsf{PCG.Expand}(1, \mathsf{k}_1)$, $R_0 = \phi_0(\rho_0)$ and $S_0 = e(R_0, R_1) - S_1$ and forwards $(\mathsf{k}_1, (R_0, S_0))$ to $\mathcal{A}$. Finally, $\mathcal{B}$ forwards the reply of $\mathcal{A}$ to its own experiment. If $\mathcal{B}$ obtained the real ρ_0 from its own experiment, then the resulting distribution is identically to $\mathcal{D}_1$, whereas if $\mathcal{B}$ obtained a simulated ρ_0, $\mathcal{B}$ simulates $\mathcal{D}_2$. Therefore, $\mathcal{B}$ successfully breaks the programmable security, whenever $\mathcal{A}$ successfully distinguishes $\mathcal{D}_1$ and $\mathcal{D}_2$.

Finally, $\mathcal{D}_2$ and $\mathcal{D}^{\mathsf{sim}}$ are computationally indistinguishable by the correctness of the PCG. □

Lemma 7.5 *The PCG construction for ring-OLE from Fig. 1 is programmable.*

Proof. To allow programmability, we tweak the Gen algorithm as follows. For $\sigma \in \{0,1\}$ the additional input ρ_σ can be sampled as $\{A^i_\sigma, \vec{b}^i_\sigma\}_{i \in [0..c)}$ in Fig. 1, representing a vector of sparse polynomials $\vec{e} = (e^0_\sigma, \ldots, e^{c-1}_\sigma)$. Notice that the first part (i.e. x_σ) of both expanded outputs can now be obtained from just ρ_σ, by first expanding $\{A^i_\sigma, \vec{b}^i_\sigma\}_{i \in [0..c)}$ to $\vec{e}_\sigma$ and then computing $x_\sigma = \langle \vec{a}, \vec{e}_\sigma \rangle \mod F(X)$. This defines the functions ϕ_0 and ϕ_1, and the correctness property follows in the same way as the proof of correctness in Theorem 4.1. The programmable security property can also be proven similarly to the proof of Theorem 4.1, with a reduction to the FSS scheme and ring-LPN assumption. □

Below we describe the main result, and some applications.

Decomposition of bilinear maps. Let $f : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ and $g : \mathbb{G}_1^u \times \mathbb{G}_2^v \to \mathbb{G}_T^w$ be bilinear maps over the additive groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$. We will consider ways of computing g that are restricted to a fixed number of calls to f on the components of the inputs to g, followed by linear combinations in $\mathbb{G}_T$ of the results of the f evaluations.

Definition 7.3 (Simple f-decomposition) *Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ be additive abelian groups, viewed as $\mathbb{Z}$-modules. Let $f : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ and $g : \mathbb{G}_1^u \times \mathbb{G}_2^v \to \mathbb{G}_T^w$ be non-degenerate bilinear maps. We say that g has a* simple f-decomposition *if there exist $\gamma \in \mathbb{N}$, $W \in \mathbb{Z}^{w \times \gamma}$ and $\alpha_i \in [u], \beta_i \in [v]$, for $i \in [\gamma]$, such that for all $a = (a_1, \ldots, a_u) \in \mathbb{G}_1^u$ and $b = (b_1, \ldots, b_v) \in \mathbb{G}_2^v$, it holds that*

$$g(x_0, x_1) = W \cdot \begin{pmatrix} f(a_{\alpha_1}, b_{\beta_1}) \\ \vdots \\ f(a_{\alpha_\gamma}, b_{\beta_\gamma}) \end{pmatrix}$$

We say that the f-complexity *of this decomposition of g is given by $n_f(g) := \gamma$.*

Note that if $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are all a (commutative) ring R and f is multiplication in R, then any g has a simple f-decomposition of complexity $u \cdot v$. However, it can still be useful to find a different f that achieves lower complexity.

We now show that any map g with a simple f-decomposition can be used to construct a PCG for the simple bilinear correlation $\mathcal{C}_g$, given a programmable PCG for $\mathcal{C}_f$. The construction is given in Fig. 13. The idea is that for each invocation of f used in an evaluation of g, we will use a separate instance of the PCG for f, programmed to use the correct portions of the input to g. Then, we can obtain the expanded g output by applying the linear map W to all the expanded outputs of f. We also use a PRG to randomize the final output shares, to ensure that there are no correlations introduced when multiplying by W.

For the security proof, we use an extension of the programmable security property, which considers several PCG instances, given in the following lemma.

Construction $G_{\mathsf{bil}}^{\mathsf{prog}}$

Parameters: Security parameter 1^λ, a programmable PCG $(\mathsf{PCG}_f.\mathsf{Gen}, \mathsf{PCG}_f.\mathsf{Expand})$ for the simple bilinear correlation $\mathcal{C}_f^n$ with programmibiliy maps $\phi_0\colon \{0,1\}^\star \to \mathbb{G}_1^n$, $\phi_1\colon \{0,1\}^\star \to \mathbb{G}_2^n$, and a pseudorandom generator $\mathsf{PRG} : \{0,1\}^\lambda \to \mathbb{G}_T^{n\times w}$.
The bilinear map $g : \mathbb{G}_1^u \times \mathbb{G}_2^v \to \mathbb{G}_T^w$ has an f-decomposition with parameters $(\gamma, \alpha_1, \dots, \alpha_\gamma, \beta_1, \dots, \beta_\gamma, W)$.
Correlation: $\mathcal{C}_g^n$ produces n instances of the simple bilinear correlation for g. That is, $\mathcal{C}_g^n$ outputs tuples $((X_0, Y_0), (X_1, Y_1))$ such that it holds $Y_0 + Y_1 = g(X_0, X_1)$, where by $g\colon \mathbb{G}_1^{n\times u} \times \mathbb{G}_2^{n\times v} \to \mathbb{G}_T^{n\times w}$ we denote the transpose of the map g applied to X_0 and X_1 row-wise.

Gen: On input 1^λ:

1. Sample random $\rho_0^1, \dots, \rho_0^u$ and $\rho_1^1, \dots, \rho_1^v$ according to the programmability property of PCG_f. // This choice will define the j-th column of X_0 as $\phi_0(\rho_0^j)$ for all $j \in [u]$, and the k-th column of X_1 as $\phi_1(\rho_1^k)$ for all $k \in [v]$.
2. Sample a PRG seed $\mathsf{k_{prg}} \leftarrow \{0,1\}^\lambda$.
3. For $i = 1, \dots, \gamma$, sample seeds $(\mathsf{k}_0^i, \mathsf{k}_1^i) \leftarrow \mathsf{PCG}_f.\mathsf{Gen}(1^\lambda, \rho_0^{\alpha_i}, \rho_1^{\beta_i})$.
4. Output $\mathsf{k}_0 = (\mathsf{k_{prg}}, \{\mathsf{k}_0^i\}_{i\in[\gamma]})$ and $\mathsf{k}_1 = (\mathsf{k_{prg}}, \{\mathsf{k}_1^i\}_{i\in[\gamma]})$

Expand: On input $(\sigma, \mathsf{k}_\sigma)$:

1. Compute $(R_\sigma^i, S_\sigma^i) \leftarrow \mathsf{PCG}_f.\mathsf{Expand}(\sigma, \mathsf{k}_\sigma^i)$, for $i \in [\gamma]$
2. If $\sigma = 0$, define the matrix $X_0 = (A_1 \| \dots \| A_u) \in \mathbb{G}_1^{n\times u}$, where $A_j = R_0^i$ for some i where $\alpha_i = j$ (if more than one α_i matches, pick arbitrarily)[a]
3. If $\sigma = 1$, define the matrix $X_1 = (B_1 \| \dots \| B_v) \in \mathbb{G}_2^{n\times v}$, where $B_j = R_1^i$ for some i where $\beta_i = j$
4. Output X_σ and $Y_\sigma = (S_\sigma^1 \| \dots \| S_\sigma^\gamma) \cdot W^\top + (-1)^\sigma \cdot \mathsf{PRG}(\mathsf{k_{prg}}) \in \mathbb{G}_T^{n\times w}$

[a]Note that such an α_i always exists, otherwise g would be degenerate. Further, if $\alpha_i = \alpha_{i'} = j$, then $\vec{k}_0^i$ and $\vec{k}_0^{i'}$ were sampled using the same seed ρ_0^j, and thus $R_0^i = R_0^{i'} = \phi_0(\rho_0^j)$ by programmability of PCG_f.

Figure 13: PCG for general bilinear correlations defined by the map g

Lemma 7.6 (Multi-instance programmability) *Let* $\mathsf{PCG} = (\mathsf{PCG.Gen}, \mathsf{PCG.Expand})$ *be a programmable PCG as in Definition 7.2. Then, for any* $d = \mathsf{poly}(\lambda)$*, the distributions*

$$\left\{ \left(\mathsf{k}_1^i, (\rho_0, \rho_1^i)\right)_{i=1}^d \;\middle|\; \rho_0, \rho_1^i \leftarrow \$, (\mathsf{k}_0^i, \mathsf{k}_1^i) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda, \rho_0, \rho_1^i) \right\} \quad \textit{and}$$
$$\left\{ \left(\mathsf{k}_1^i, (\rho_0, \rho_1^i)\right)_{i=1}^d \;\middle|\; \rho_0, \rho_1^i, \tilde{\rho}_0^i \leftarrow \$, (\mathsf{k}_0^i, \mathsf{k}_1^i) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda, \tilde{\rho}_0^i, \rho_1^i), \right\}$$

are computationally indistinguishable. A symmetric property holds for k_0^i.

Proof. This follows a standard hybrid argument, where the reduction loses a factor of d in advantage. In the j-th hybrid, pick randomness $\rho_0, \rho_1^i \leftarrow \$$ and for $i \leq j$ also $\tilde{\rho}_0^i \leftarrow \$$, for $i \leq j$ compute the key as $(k_0^i, k_1^i) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda, \tilde{\rho}_0^i, \rho_1^i)$, and for $i > j$ compute the key as $(k_0^i, k_1^i) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda, \rho_0, \rho_1^i)$. Then give out $\left(\mathsf{k}_1^i, (\rho_0, \rho_1^i)\right)_{i=1}^d$. Given a distinguisher for any two hybrids j and $j+1$, it is straightforward to construct a distinguisher for the programmable security property of PCG, with the same advantage. □

Theorem 7.4 *Let f and g be bilinear maps as above, and suppose that g has a simple f-decomposition with f-complexity $n_f(g)$. Furthermore, let* $\mathsf{PCG}_f = (\mathsf{PCG}_f.\mathsf{Gen}, \mathsf{PCG}_f.\mathsf{Expand})$ *be a programmable PCG for* $\mathcal{C}_f^n$*. Then there exists a programmable PCG* $\mathsf{PCG}_g = (\mathsf{PCG}_g.\mathsf{Gen}, \mathsf{PCG}_g.\mathsf{Expand})$ *for* $\mathcal{C}_g^n$*, with the following properties:*

- $\mathsf{PCG}_g.\mathsf{Gen}$ *runs* $n_f(g)$ *executions of* $\mathsf{PCG}_f.\mathsf{Gen}$*, and its key sizes are* $n_f(g)$ *times that of* PCG_f.
- $\mathsf{PCG}_g.\mathsf{Expand}$ *runs* $n_f(g)$ *executions of* $\mathsf{PCG}_f.\mathsf{Expand}$*, and n evaluations of the linear map W from the f-decomposition of g.*

Proof. We give an explicit consturction of PCG_g in Figure 13. We consider separately the correctness and security properties of the PCG definition.

Correctness. Let (X_0, Y_0) and (X_1, Y_1) be a pair of outputs from $\mathsf{PCG}_g.\mathsf{Expand}$. Let ϕ_0, ϕ_1 be the programmibility maps, and $\rho_0^1, \dots, \rho_0^u$ and $\rho_1^1, \dots, \rho_1^v$ as chosen by Gen on input 1^λ. Let $A_j = \phi_0(\rho_0^j) \in \mathbb{G}_1^n$ and $B_k = \phi_1(\rho_1^k) \in \mathbb{G}_2^n$ for all $j \in [u], k \in [v]$. Then, by the programmibility property of PCG_f it holds that $(R_0^1, \dots, R_0^\gamma) = (A_{\alpha_1}, \dots, A_{\alpha_\gamma})$ and $(R_1^1, \dots, R_1^\gamma) = (B_{\beta_1}, \dots, B_{\beta_\gamma})$ with overwhelming probability. Also, by construction in PCG_g, we have $X_0 = (A_1 \| \dots \| A_u)$ and $X_1 = (B_1 \| \dots \| B_v)$. From the correctness of PCG_f, with overwhelming probability

$$Y_0 + Y_1 = (S_0^1 + S_1^1 \| \dots \| S_0^\gamma + S_1^\gamma) \cdot W^\top = (f(A_{\alpha_1}, B_{\beta_1}) \| \dots \| f(A_{\alpha_\gamma}, B_{\beta_\gamma})) \cdot W^\top = g(X_0, X_1),$$

where f is considered to be applied entry-wise to the inputs. Furthermore, X_0 and X_1 are both pseudorandom by the correctness property of PCG_f, and any individual Y_σ is pseudorandom due to the security of PRG; hence, the outputs of PCG_g are computationally indistinguishable from random outputs of the correlation C_g^n.

Programmability. The programable randomness can be defined as $\rho_0 = (\rho_0^1, \dots, \rho_0^u)$ and $\rho_1 = (\rho_1^1, \dots, \rho_1^v)$. The programmability maps are defined by applying ϕ_0 and ϕ_1 for PCG_f componentwise. The programmability property then follows directly from the programmability of PCG_f.

Programmable Security. We first consider the case $\sigma = 1$. Recall that by Lemma 7.4 programmable security together with correctness and programmability automatically implies standard PCG security. We need to show indistinguishability of the two distributions

$$\mathcal{D}^{\mathsf{real}} := \left\{ (\mathsf{k}_1, (\rho_0, \rho_1)) \;\middle|\; \rho_0, \rho_1 \xleftarrow{\$} \$, (\mathsf{k}_0, \mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG}_g.\mathsf{Gen}(1^\lambda, \rho_0, \rho_1) \right\} \quad \text{and}$$

$$\mathcal{D}^{\mathsf{sim}} := \left\{ (\mathsf{k}_1, (\rho_0, \rho_1)) \;\middle|\; \rho_0, \rho_1, \tilde{\rho}_0 \xleftarrow{\$} \$, (\mathsf{k}_0, \mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG}_g.\mathsf{Gen}(1^\lambda, \tilde{\rho}_0, \rho_1) \right\}$$

Recall that $\mathsf{k}_1 = (\mathsf{k}_{\mathsf{prg}}, \{\mathsf{k}_1^i\}_{i \in [\gamma]})$, where $(\mathsf{k}_0^i, \mathsf{k}_1^i) \leftarrow \mathsf{PCG}_f.\mathsf{Gen}(\rho_0^{\alpha_i}, \rho_1^{\beta_i})$, and $\rho_0 = \{\rho_0^i\}_{i \in [u]}, \rho_1 = \{\rho_1^i\}_{i \in [v]}$. We use a hybrid argument, where the j-th experiment is as follows.

Hybrid distribution $\mathcal{D}_0$. Give out $(\mathsf{k}_1, (\rho_0, \rho_1))$ as computed in the actual construction.

Hybrid distribution $\mathcal{D}_j$, for $j = 1, \dots, u$. Sample randomness $\rho_0^1, \dots, \rho_0^u, \rho_1^1, \dots, \rho_1^v$ as in the construction, as well as random $\tilde{\rho}_0^1, \dots, \tilde{\rho}_0^\gamma$. For each $i \in [\gamma]$, if $\alpha_i \leq j$ then sample $(\mathsf{k}_0^i, \mathsf{k}_1^i) \xleftarrow{\$} \mathsf{PCG}_f.\mathsf{Gen}(1^\lambda, \tilde{\rho}_0^i, \rho_1^{\beta_i})$. Otherwise, if $\alpha_i > j$, sample $(\mathsf{k}_0^i, \mathsf{k}_1^i)$ using $(\rho_0^{\alpha_i}, \rho_1^{\beta_i})$ as in the construction. Output $\mathsf{k}_1 = \{\mathsf{k}_1^i\}_{i \in [\gamma]}$ and the randomness $\rho_0 = \{\rho_0^i\}_{i \in [u]}, \rho_1 = \{\rho_1^i\}_{i \in [v]}$.

Note that $\mathcal{D}_0$ is identical to $\mathcal{D}^{\mathsf{real}}$ and $\mathcal{D}_u$ is identical to $\mathcal{D}^{\mathsf{sim}}$. It is left to consider the difference between hybrids $\mathcal{D}_j$ and $\mathcal{D}_{j+1}$. For any $i \in [\gamma]$ such that $\alpha_i = j + 1$, in hybrid distribution $\mathcal{D}_{j+1}$ we use fresh randomness $\tilde{\rho}_0^i$ to sample k_1^i, whereas in $\mathcal{D}_j$ we use the real randomness $\rho_0^{\alpha_i}$. However, any adversary who distinguishes these can be used to break the multi-instance security property (Lemma 7.6) of the programmable PCG, where the d keys are defined to be those of the indices where $\alpha_i = j + 1$.
Finally, notice that the case of $\sigma = 0$ proceeds symmetrically, with a sequence of v hybrids argued in the same way.

□

7.4 Application: Matrix Multiplication Triples

We can use the general bilinear construction to build a PCG for generating a large batch of matrix multiplication triples. For matrices of dimensions $n_1 \times n_2$ and $n_2 \times n_3$, the PCG seed size is around $n_1 \cdot n_3$ times larger than the PCG for OLE. This means it will likely be practical for small-to-medium matrices.

Let $g : \mathbb{Z}_p^{n_1 \times n_2} \times \mathbb{Z}_p^{n_2 \times n_3} \to \mathbb{Z}_p^{n_1 \times n_3}$ be the matrix multiplication map, and let $f : \mathbb{Z}_p^{n_2} \times \mathbb{Z}_p^{n_2} \to \mathbb{Z}_p$ be the inner product map over $\mathbb{Z}_p$. These maps fit the requirements of Theorem 7.4, by using $\mathbb{G}_1 = \mathbb{G}_2 = \mathbb{Z}_p^{n_2}$, $\mathbb{G}_T = \mathbb{Z}_p$, $u = n_1, v = n_3, w = n_1 \cdot n_3$ and appropriately flattening matrices into one-dimensional vectors.

Multiplication of $n_1 \times n_2$ and $n_2 \times n_3$ matrices is easily decomposed as a sequence of $n_1 \cdot n_3$ inner products of length n_2, where each inner product is taken from a consecutive portion of the two inputs. This shows that the matrix multiplication map g has a linear f-decomposition with f-complexity $n_1 \cdot n_3$.

To obtain a PCG for g, we use the PCG for the length-n_2 inner product correlation f, which can be based on R-LPN when $F(X)$ splits into degree-n_2 factors (Corollary 7.3).

Corollary 7.7 *Suppose R-$\mathsf{LPN}_{p,1,t}$ holds for $R = \mathbb{Z}_p[X]/F(X)$, where $F(X)$ is degree N and splits into N/n_2 distinct factors of degree n_2. Then there is a PCG for producing $n = N/n_2$ correlations for $(n_1 \times n_2) \times (n_2 \times n_3)$ matrix multiplication over $\mathbb{Z}_p$. The PCG requires $n_1 \cdot n_3$ copies of the PCG for n inner products of length n_2, and has seed size and computational cost around $n_1 \cdot n_3$ times that of the PCG for inner product.*

Example parameters. We now give some example parameters for generating matrix triples, based on the analysis in Section 9. For instance, when producing 8-dimensional square matrix triples in a batch of size ≈ 130000, over a 128-bit field, the PCG seeds have size around 83MB. These can be expanded to produce matrix correlations of around 400MB, giving a 5-fold expansion factor. Increasing the dimension to 16, the seed size grows to 330MB, while the PCG output has size 810MB. Going to dimension 32 and beyond, the seeds start to become much larger, although better expansion rates could be obtained when producing a much larger batch of triples. In general, this shows that in practice, the construction is only likely to be useful for small matrix dimensions; however, these could still be used as a building block in performing larger matrix multiplications in applications. For larger matrices, more interactive approaches such as recent work based on homomorphic encryption [CKR+20] appear to be more practical.

Remark 7.8 *Instead of using a PCG for inner product, we could instead directly use a programmable PCG for OLE to build matrix multiplications, by applying Theorem 7.4 with f as $\mathbb{Z}_p$ multiplication. However, this would require $n_1 \cdot n_2 \cdot n_3$ instances of the base PCG, giving a much worse expansion factor. This shows the advantage of finding a suitable f such that the desired correlation g has low f-complexity, compared with the naive approach.*

7.5 Application: Circuit-Dependent MPC Preprocessing

Circuit-dependent preprocessing is a variation on the standard multiplication triples technique, which is based on Beaver's circuit randomization technique [Bea92] and extended in more recent works [DNNR17, KKW18, BNO19, BGI19]. The idea is to preprocess multiplications in a way that depends on the structure of the circuit, and leads to an online phase that requires just *one opening per multiplication gate*, instead of two when using multiplication triples.

With the PCG for general bilinear correlations, we can generate circuit-dependent preprocessing for a large batch of identical circuits. This can be useful, for instance, when executing the same function many times on different inputs, or when a larger computation contains many small, repeated instances of a particular sub-circuit.

Let C be an arithmetic circuit over $\mathbb{F}$ consisting of fan-in two addition and multiplication gates.

In the offline phase each wire w in the circuit is assigned a value r_w such that:

- if w is an input wire, $r_w \overset{\$}{\leftarrow} \mathbb{F}$ is chosen at random
- if w is the output wire of a multiplication gate, $r_w \overset{\$}{\leftarrow} \mathbb{F}$ is chosen at random
- if w is the output wire of an addition gate with input wires u and v, then $r_w = r_u + r_v$.

Further, each multiplication gate is assigned a value $s_{u,v}$ as follows:

- if the multiplication gate has input wires u and v, then $s_{u,v} = r_u \cdot r_v$.

The goal of the offline phase is for the parties to obtain random additive shares of r_w for all input wires and output wires of multiplication gates, as well as $s_{u,v}$ for all multiplication gates.

Let G denote the set of multiplication gates. Then, for a multiplication gate $g \in G$ with input wires u and v, we can write $s_{u,v}$ as $\sum_{i \in L_g} r_i \cdot \sum_{j \in R_g} r_j$, where L_g and R_g are the sets of output wires of multiplication gates that pass only through addition gates before reaching g. All $s_{u,v}$ values can therefore be computed as a bilinear function f_C of $\vec{r}$, where $\vec{r}$ consist of the random values assigned to the input wires and the output wires of multiplication gates.

Plugging in the previous construction for general bilinear correlations, we obtain a PCG for f_C out of several instances of a PCG for OLE, where the total number of instances is $\sum_{g \in G} |L_g| \cdot |R_g|$.

7.6 Application: Multi-Party PCGs for Bilinear Correlations

In [BCG$^+$19b][Theorem 41], Boyle et al. showed that any two-party, programmable PCG for a simple bilinear correlation can be used to build a multi-party PCG for an additively secret-shared version of the same correlation.

Plugging in Lemma 7.5 and the results of the previous section, we obtain N-party PCGs for (unauthenticated) multiplication triples, matrix triples and circuit-dependent preprocessing over $\mathbb{Z}_p$ based on ring-LPN, for any polynomial number of parties N. Each party's seed contains $2(N-1)$ seeds of the underlying two-party PCG, plus $N-1$ seeds for a PRG. The expansion procedure of the PCG consists of expanding the $2(N-1)$ PCG seeds, as well as the PRG seeds.

Authenticated triples in the multi-party setting. Unfortunately, the transformation of Boyle et al. only applies to degree-2 correlations, and we do not see a way to directly apply it to our PCG for *authenticated* multiplication triples, which is a degree-3 correlation. To see the challenge in extending their construction, consider the case of n parties who wish to obtain additive shares of $\alpha \cdot (x \cdot y)$, where α, x, y are additively shared. To do this using pairwise correlations, it seems we need a way to obtain shares of $x_i \cdot y_j \cdot \alpha_k$, for every $i, j, k \in [n]$, where P_i holds x_i, P_j holds y_j and P_k holds α_k. We cannot do this with just a pairwise correlation between P_i and P_j, since α_k must be known only to P_k. On the other hand, given a function secret sharing scheme for *3 parties*, one could modify our authenticated triples construction from Fig. 2 to make this work. However, known constructions of 3-party distributed point functions are much more expensive, with a seed size of $O(\sqrt{\lambda N})$ rather than $O(\lambda \log N)$ [BGI15].

8 Security Analysis

We analyze the security of module-LPN against various attacks. In the following, we consider a R^c-uLPN instance over a ring $R = \mathbb{F}[X]/F(X)$ (where $F(X)$ is a degree-N polynomial) with c samples, and a regular noise of total weight w (that is, $w = t \cdot c$, where t is the number of nonzero coordinates of each noise $e_i \in R$); therefore, the adversary gets $(\vec{a}_i, \langle \vec{a}_i, \vec{s} \rangle + e_i)_{i=1}^c$, where $\vec{s}$ and each $\vec{a}_i$ are random over R^{c-1}, and each $\vec{e}_i$ is sampled from $\mathcal{HW}_{w/c}$, and must distinguish $(\langle \vec{a}_i, \vec{s} \rangle + e_i)_{i \le c}$ from a random element of R^c. The corresponding code is a linear code with dimension $(c-1) \cdot N$ and length $c \cdot N$, whose parity-check matrix $H \in \mathbb{F}^{N \times c \cdot N}$ is c-compressing. Note that by our reduction from R-LPN to R-uLPN (Lemma 3.4), this effectively reduces to distinguishing $(\vec{a}', \langle \vec{a}', \vec{s}' \rangle + e')$ from random, where $\vec{a}' \xleftarrow{\$} R^{c-1}$ and $\vec{s}' \xleftarrow{\$} (\mathcal{HW}_{w/c}^{c-1})$ is $(c-1)w/c$-sparse.

We will consider two alternatives for the underlying field: either $\mathbb{F}$ is a very small field (e.g. $\mathbb{F} = \mathbb{F}_2$), or $\mathbb{F}$ is a large field (e.g. $\mathbb{F} = \mathbb{Z}_p$ where p is a large prime, for example $p \approx 2^{128}$). Eventually, we will consider both the cases where $F(X)$ is an irreducible polynomial over $\mathbb{F}[X]$, and the case where $F(X)$ is fully reducible over $\mathbb{F}[X]$ (typically, this will be the case when F is a two-power cyclotomic polynomial and $\mathbb{F} = \mathbb{Z}_p$ where p is a large prime).

We can also consider two error distributions: the e_i can be either random weight-t errors, or regular weight-t errors (where N coordinates of e_i are divided into t blocks of length N/t, and a single random noise is added to a random coordinate of each block). Below, we analyze various attacks with respect to the uniform noise distribution. However, none of the attacks we describe in the following sections performs better when using a regular noise distribution.

Bottom Line. To provide a short, high-level summary of the conclusion of this section: when F is irreducible over a large field $\mathbb{F}$, no known attacks perform significantly better than those on standard LPN, with a very limited number of samples $O(n)$, where n is the dimension. In this setting, attacks such as BKW do not apply, but information set decoding (ISD) and statistical decoding (SD) do. Both have complexity exponential in the number of noisy coordinates. When

F is reducible, however, the adversary can reduce the instance modulo some factor and get a new LPN instance, which might be easier to solve if the factor is sparse (as it reduces the dimension without increasing the noise). Hence, the cost of ISD and SD must be evaluated for each reduction modulo a sparse factor, and the cost of the attack is the smallest cost accross all factors. Our use of structured matrices means that the DOOM attack might apply, but it only reduces security by about $(\log N)/2$ bits. Eventually, over types of structural attacks, like algebraic decoding attacks, do not seem to apply to our setting.

8.1 Generic Attacks on LPN

We denote by $G = [\mathsf{Id}_N || - A_1^\intercal || \cdots - A_{c-1}^\intercal]^\intercal$ the generating matrix of the linear code associated to our R-LPN instance, which generates a code with a c-compressing parity-check matrix $H = [A_1 || \cdots A_{c-1} || \mathsf{Id}_N]$ where the A_i are the multiplication matrices of a_i mod $F(X)$ for independent random $a_i \in R$.

Existing Generic Attacks. In spite of its extensive use in cryptography, few cryptanalytic results are known for the general LPN assumption. We outline below the main known attacks.

Attacks on Syndrome Decoding. The problem we consider is best seen as an instance of the syndrome decoding problem, where the goal is to recover $\vec{s}$ given $H \cdot \vec{s}$ (where, in our case, $H = [A_1 || \cdots A_{c-1} || \mathsf{Id}_N]$ is the parity-check matrix of G). Existing attacks on syndrome decoding rely either on improvements over a natural Gaussian elimination attack, called information set decoding (ISD), or on a parity-check attack exploiting low-weight codewords in the dual code, known as statistical decoding attacks [AJ01, Ove06, FKI06, DAT17], or low-weight parity-check attacks [Zic17].

- **Gaussian Elimination.** For the standard LPN assumption with w noisy coordinates, the Gaussian elimination attack requires on average $(1/(1 - w/(cN)))^{(c-1)N}$ iterations, where the adversary must invert a $(c-1)N \times (c-1)N$ submatrix of G, which takes time $O(((c-1)N)^{2.8})$ using Strassen's matrix multiplication algorithm. However, since the special structure of G allows for fast (quasilinear) matrix-vector multiplication, standard techniques allow for solving a linear system of equations defined by a random submatrix of G in time $O(((c-1)^2 N^2 \log((c-1)N))$. Hence, the cost (counted as a number of arithmetic operations over $\mathbb{F}$) of the generic Gaussian elimination attack on R-LPN is (assuming for simplicity that c is a constant, as it will be in all our instantiations):

$$O\left(\left(\frac{1}{1-\frac{w}{cN}}\right)^{(c-1)N} \cdot ((c-1)N)^2 \log N\right) = O(e^{\frac{(c-1)w}{c}} \cdot ((c-1)N)^2 \log N),$$

  where the equality holds when $N \gg w$. Note that the above attack assumes a standard noise distribution. However, using a regular noise distribution does not change the efficiency of the attack: if the noise vector is divided into w blocks of length cN/w with a single noise in each block, the natural adaptation of the above Gaussian elimination attack to this setting works by trying to find $(c-1)N/w$ non-noisy coordinate in each of the w blocks (finding more non-noisy coordinates in a given block can only decrease the success probability of the attack). This means that the success probability of the attack is given by

$$\left(\left(1 - \frac{1}{cN/w}\right)^{(c-1)N/w}\right)^w = \left(1 - \frac{w}{cN}\right)^{(c-1)N}$$

  which leads exactly to the same cost for the attacker. We note that the same observation applies to all variants of ISD we are aware of.

- **Information Set Decoding.** Among the best algorithms for syndrome decoding are improvements of Prange's ISD algorithm (which is itself an improvement over the Gaussian elimination attack given above), which attempts to find a size-w subset of the rows of H that spans $H \cdot \vec{e}$. When the LPN instance has high dimension $(c-1)N$, cN samples, and very low error rate (which is the case in our scenario, since we consider a fixed amount w of noisy coordinates and $N \gg w$), according to the analysis of [TS16], all known variants of ISD (e.g. [Pra62, Ste88, FS09, BLP11, MMT11, BJMM12, MO15]) have essentially the same asymptotic complexity $c^{w(1+o(1))}$ (ignoring the $O((c-1)^2N^2 \log N)$ polynomial cost of solving a linear system). Therefore, their gain compared to the initial algorithm of Prange vanishes in our setting and the cost of these attacks is well approximated by

$$O\left(c^{w\cdot(1+o(1))} \cdot (c-1)^2N^2 \cdot \log N\right).$$

- **Statistical Decoding.** Eventually, all the previous attacks recover the secret $\vec{s}$. If one simply wants to distinguish $\vec{b} = A \cdot \vec{s} + \vec{e}$ from random, there exists an alternative, incomparable line of attacks, known as *statistical decoding* attacks [AJ01]. These attacks are based on the following observation: by the singleton bound, the minimal distance of the code generated by H is at most $(c-1)\cdot N+1$, hence there must be a parity-check equation for G of weight $(c-1)\cdot N+1$. Then, if $\vec{b}$ is random, it passes the check with probability at most $1/|\mathbb{F}|$, whereas if $\vec{b}$ is a noisy encoding, it passes the check with probability at least $1/|\mathbb{F}| + ((N-1)/cN)^w$. Note that this attack works especially well when $\mathbb{F}$ is very large, since then a random $\vec{b}$ has negligible probability to pass the check. Improved variants of the algorithm describe optimized methods to quickly find a relatively large number of parity-check equations with a sufficiently small weight. For the sake of providing conservative estimates, however, we will assume in our analysis that the adversary has already pre-computed an arbitrary number of parity-check equations (since these equations depend solely on H), and that all such equations have minimal weight $N+1$. Under these conservative assumptions, the cost (counted as a number of arithmetic operation) of statistical decoding is lower-bounded by

$$O\left(\left(\frac{cN}{N-1}\right)^w \cdot N\right) \approx O\left(c^w \cdot N\right).$$

  Here again, the estimations are made using the standard noise distribution. However, it does not seem feasible for an attacker to exploit a regular noise distribution: intuitively, the best the attacker can do to exploit this structure requires finding low-weight codewords in the dual code whose nonzero coordinates are 'equally well-spread' accross all coordinates. But at quick calculation similar to the one we did for Gaussian elimination shows that, even if finding many such optimally low-weight well-spread codewords was feasible (which is not clear), the running time of the attack would still remain identital to our estimate above.

Attacks on LPN (Using Many Samples). When the number of samples can be very large, as is generally the case in the LPN literature, there exist improved attacks based on time-space tradeoffs. Below, we briefly recall existing attacks. However, as our overview below illustrates, all these attacks require a *superlinear* number of samples $\omega(D)$ in the dimension D (even the sample-optimized variant of BKW of [Lyu05]), while the variant we consider has $cN = \frac{c}{c-1} \cdot D$ samples. Hence, none of the attacks below does apply in our scenario. We refer the reader to [EKM17] for a more comprehensive overview. Given an LPN instance with dimension $(c-1)N$, w noisy coordinates, and $q = cN$ samples, we let $r \leftarrow w/q$ denote the *noise rate* of the instance.

- **The BKW algorithm [BKW00].** This algorithm is a variant of Gaussian elimination which achieves subexponential complexity even for high-noise LPN (e.g. constant noise rate), but requires a subexponential number of samples: the attack solves LPN over $\mathbb{F}_2$ in time $2^{O((c-1)N/\log((c-1)N/r))}$ using $2^{O((c-1)N/\log(c-1)(N/r))}$ samples.

- **Hybrid attacks [EKM17].** The authors of [EKM17] conducted an extended study of the security of LPN, and described combinations and refinements of the previous three attacks (called the *well-pooled Gauss attack*, the *hybrid attack*, and the *well-pooled MMT attack*). All these attacks achieve subexponential time complexity, but require as many sample as their time complexity.
- **Scaled-down BKW [Lyu05].** This algorithm is a variant of the BKW algorithm, tailored to LPN with polynomially-many samples. It solves LPN in time
$$2^{O((c-1)N/\log\log((c-1)N/r))},$$
using $((c-1)N)^{1+\varepsilon}$ samples (for any constant $\varepsilon > 0$) and has worse performance in time and number of samples for larger fields.

8.2 Taking Advantage of Reducible F

When $F(X)$ is reducible, the above attacks can be improved if the adversary finds sufficiently sparse polynomial factors f_i of F. Indeed, when this is the case, the adversary obtains new LPN instances by computing $\vec{a} \cdot s + \vec{e} \bmod f_i$, where the new noise $\vec{e} \bmod f_i$ remains sparse since f_i is sparse. This reduces the problem to an LPN instance in smaller dimension, which can also be solved by any of the above attacks. Below, we consider the best case (for the adversary), when there is a factor f_i of degree $n = N/k$, for some k, which has sparsity 1. This happens, for example, when $F(X)$ is the m-th cyclotomic polynomial, of degree $N = \phi(m)$, $p = 1 \bmod 2N$, and N is a power of two. In this case, $F(X) = X^N + 1$ splits completely into N linear factors modulo p, but also has sparse factors of the form $X^{2^i} + c_i$ due to properties of roots of unity.[10]

Reducing a R-LPN sample mod f_i brings the dimension down to N/k, while the total number of noisy coordinates now lies between w/k and w (depending on how many errors are added together). To study the effectiveness of this attack, we need to first analyze the new noise rate, and then the performance of the best known R-LPN attacks for the new set of parameters.

Estimating the reduced error rate. Suppose each of the $t = w/c$ errors in one entry of $\vec{e} = (e_1, \cdots, e_c)$ is chosen independently and uniformly from $[N]$ (this is only a small change from the original distribution. Then, reducing an error polynomial modulo some 1-sparse f_i of degree $n = N/k$ just means each error ends up in a random position in the reduced length-n vector. For each $j \in [n]$, define the random variable E_j to be 1 if position j in the reduced polynomial has zero errors, and 0 otherwise. It follows that the expected the number of error-free positions is $\sum_j \mathbb{E}[E_j] = n \cdot \Pr[E_j = 1] = n \cdot (1 - 1/n)^t$. Therefore, summing up across the c error polynomials, we get a total of
$$cn(1 - (1 - \frac{1}{n})^t)$$
expected errors in the reduced LPN instance.

Analysis for Two-Power Cyclotomics. In our estimations below, we will focus on the important case where $F(X)$ is a two-power cyclotomic, which is one of our main candidates for building a pseudorandom correlation generator for OLE correlations over $\mathbb{F} = \mathbb{Z}_p$, and which is also the best-case scenario for the attacker: for $i = 1$ to $\log N$, there exists 1-sparse factors of $F(X)$ of degree 2^i, and each of them gives rise to an LPN instance of dimension $n_i = (c-1) \cdot 2^i$, with $q_i = c \cdot 2^i$ samples, and with an expected number of errors $w_i = cn_i(1 - (1 - \frac{1}{n_i})^{w/c})$.

[10]For example, if c is a square root of -1 modulo p (which exists, since with two-power cyclotomics, $p \equiv 1 \bmod 4$) then $X^N + 1 = (X^{N/2} + c)(X^{N/2} - c)$.

Gaussian Elimination. By picking an optimal choice of i, this R-LPN instance can be solved in time

$$O\left(\min_{1\leq i\leq \log N}\left(\frac{1}{1-\frac{w_i}{q_i}}\right)^{n_i}\cdot n_i^2\log n_i\right).$$

Information Set Decoding. When $F(X)$ is reducible, we cannot generally assume that the dimension is much larger than the number of noisy coordinates, since the adversary can reduce the dimension. By picking an appropriate i, the adversary can therefore find an LPN instance where some of the improved ISD algorithm perform better than Prange's original algorithm. Due to the extended literature on ISD, it is difficult to evaluate precisely all existing attacks on a given instance. However, a simplified and general estimation of the efficiency of ISD algorithms was given in [HOSS18b], based on similar analysis given in [FS09, Sen11a, HS13, TS16]. This general analysis builds upon the fact that most state-of-the-art ISD algorithms share a common structure, from which a general lower bound on the cost of the attack can be derived. Here, we simply reproduce the conclusions of [HOSS18b], restricted to our specific setting, and refer the reader to [HOSS18b] for details on the analysis. We note that [HOSS18b] does not aim at precisely estimating the cost of the attacks, but at providing a reasonably sharp and general lower bound on their costs. In general, the cost of modern ISD algorithms for a parity-check matrix H_i, with dimension $n_i=(c-1)\cdot 2^i$, $q_i=c\cdot 2^i$ samples, and w_i noisy coordinates, is lower bounded by

$$\min_{p,q}\left\{\frac{\min\left\{2^{n_i},\binom{q_i}{w_i}\right\}}{\binom{n_i-q}{w_i-p}}\cdot\left(\frac{K_1+K_2}{\binom{n_i+q}{p}}+\frac{w_i\cdot(n_i-q)}{2^q}\right)\right\},$$

where the values (p,q) satisfy $0\leq q\leq 2^i$ and $0\leq p\leq n_i+q$, K_1 denotes the cost of performing a Gaussian elimination on a submatrix of H_i with n_i-q columns, and K_2 denotes the running time of a specific sub-algorithm, which varies accross different attacks. Since H is well structured, we assume to be conservative that performing Gaussian elimination on the submatrix of H_i can be done in time $(n_i-q)^2\log(n_i-q)$. Regarding K_2, according to the analysis of [HOSS18b], it can be lower bounded by

$$\binom{(n_i+q)/2}{p/8}$$

when using the algorithm of [BJMM12], which seems to provide the best efficiency on the instances we consider (more recent algorithms improve over [BJMM12], but at the cost of large hidden constants that render them less practical, or only for very high noise rates). Putting everything together, a lower bound on the cost of ISD algorithm for a two-power cyclotomic F is given by

$$\min_{\substack{1\leq i\leq\log N\\0\leq q\leq n_i\\0\leq p\leq n_i+q}}\left\{\frac{\min\left\{2^{n_i},\binom{q_i}{w_i}\right\}}{\binom{n_i-q}{w_i-p}}\cdot\left(\frac{(n_i-q)^2\log(n_i-q)+\binom{(n_i+q)/2}{p/8}}{\binom{n_i+q}{p}}+\frac{w_i\cdot(n_i-q)}{2^q}\right)\right\}.$$

Statistical Decoding. By picking an optimal choice of i, a reducible R-LPN instance can be solved with a statistical decoding algorithm in time

$$O\left(\min_{1\leq i\leq\log N}\left(\frac{q_i}{n_i-1}\right)^{w_i}\cdot 2^i\right).$$

Changing the Structure of the Noise. In our analysis above, we rely on an estimation of the expected amount of noise after reduction modulo a 1-sparse factor f_i of F. However, this ignores the possibility that, in some rare cases, the reduction modulo f_i might lead to an instance with a much smaller amount of noise than the expected number, in which case the attacks above will work much more efficiently. We note that there is a natural approach to avoid these "weak parameters", by sampling the noise such that its reduction modulo *any* 1-sparse factor f_i has weight at least the expected quantity t_i. Typically, when $F = X^N + 1$, the 1-sparse factors are all of the form $X^{2^i} + c_i$, and sampling the noise in this way is very easy; one can for example use a simple rejection sampling approach. Note that it is sufficient to consider a single factor $X^{2^i} + c_i$ for the smallest i which the adversary can consider (typically, $i = 6$ or 7 in our instances), because reducing the noise modulo $X^{2^i} + c_i$ amounts to computing a linear combination of the consecutive length-2^i subvectors of the noise vector. Hence, reducing modulo any factor with a larger i' amounts to computing a linear combination of concatenations of $2^{i'-i}$ of these subvectors, which implies that the total noise cannot decrease more than when reducing modulo $X^{2^i} + c_i$. Since a random noise vector will have more than t_i noisy coordinates after reduction modulo f_i with probability $\approx 1/2$, this rejection sampling approach reduces by a single bit the total entropy of the noise vector.

We note that there might possibly be other, less sparse factors, which the adversary could use. For such factors, the sampled noise vector is not guaranteed to maintain a target expected number of nonzero coordinates after modular reduction. However, the noise reduction achieved by modular reduction comes from collisions between the noisy coordinates in the reduced instance, but this number of collision (as shown in our computation of the expected number t_i of collisions) is typically quite small. On the other hand, reducing modulo a d-sparse factor increases the amount of noise by a factor d; when $d > 1$, we expect that this will systematically lead to an increased total amount of noise, and is not a viable adversarial strategy. Therefore, reductions modulo 1-sparse factors seems to be the main concern, and sampling the noise vector as we suggest eliminates the unlikely event of a weak noise with respect to one of these factors.

8.3 Algebraic Attacks on Fully-Reducible F

Another type of attacks are the algebraic attacks that exploit the structure of the underlying code. Many such algebraic decoding attacks have been devised in the literature, and fall in a unified framework developed in [Pel92,Kot92] of distinguishing attacks based on componentwise product of codes. Examples of such attacks include [PMCMM11,MCP12,FGUO+13,CGGU+13, MCMMP14] (and many more), and were often used to break some variants of the McEliece cryptosystem. Assume again that $F(X) = X^N + 1$ is a 2-power cyclotomic polynomial, with N linear factors $f_i(X) = X + c_i$. In this setting, the generating matrix associated to $(\vec{s} \mapsto \langle \vec{a}, \vec{s} \rangle \bmod f_i)_{i \leq N}$ is of the form $G = V \cdot [\mathsf{Id}_N || - A_1^\intercal || \cdots - A_{c-1}^\intercal]^\intercal$, where V is a Vandermonde matrix (since Vandermonde matrices capture polynomial evaluation, and reducing a polynomial modulo a linear factor of the form $X + c_i$ is equivalent to evaluating the polynomial at c_i). Algebraic decoding attacks would allow to break the reducible R-LPN assumption if G is strongly multiplicative (roughly, G generates a strongly multiplicative code if the entry-wise product of each pairs of columns of G spans a vector space of dimension $d < (cN)^2$). However, with G as above, it is easily seen that the pointwise products of pairs of columns of G span the whole $\mathbb{F}^{(cN)^2}$ with overwhelming probability over the choice of $A_1, \cdots, A_{c-1}$, because the pointwise products of two columns in each A_i are distinct to each other with overwhelming probability. Therefore, algebraic decoding attacks do not seem to apply to our LPN variant.

8.4 Attacks Using the Quasi-Cyclic Structure of the Code

When the underlying code has a quasi-cyclic structure, there is an additional attack which must be accounted for, which enhances the ISD family of attacks with a $\sqrt{(c-1)N}$ computational speedup: the DOOM (Decoding One Out of Many) attack [Sen11b]. We note that the structure of our codes is not exactly a quasi-cyclic structure; however, they have a structure that closely resemble the structure of quasi-cyclic codes (e.g. when $F = X^N + 1$, our code matrix is quasi-cyclic up to the fact that the odd-numbered blocks are multiplied by a factor -1). Therefore, even though we are not aware of an extension of the DOOM attack to the kind of structured codes we consider, we assume to be conservative that the DOOM attack can be extended to work in our setting, and take into account the corresponding speedup for the ISD attacks in our estimations. Similarly, when F is reducible, we assume that the DOOM attack speeds up the ISD attacks by a $\sqrt{n_i}$ factor over the reducted instance of dimension n_i.

8.5 Attacks over Small Fields

In this section, we assume the $\mathbb{F} = \mathbb{F}_2$, which is the setting we consider in our applications that rely on R-LPN small fields. The attack we describe works also over larger fields, but performs more poorly, since it requires to brute-force over all possible secrets.

A natural approach to attack R-LPN over a small field is to brute-force over all possible choices of $\vec{s} \in \mathcal{HW}_{w/c}^{c-1}$. For each candidate vector $\vec{s}$, the attacker checks whether $\langle \vec{a}, \vec{s} \rangle + \vec{b}$ is a regular w/c-sparse vector. The number of sparse secrets from $\mathcal{HW}_{w/c}^{c-1}$ over $\mathbb{F}_2$ is

$$\binom{(c-1)\cdot N}{(c-1)\cdot \frac{w}{c}},$$

and grows to $\binom{((c-1)\cdot N)^{|\mathbb{F}|-1}}{((c-1)\cdot \frac{w}{c})^{|\mathbb{F}|-1}}$ over arbitrary fields. Each $\langle \vec{a}, \vec{s} \rangle$ for some candidate secret $\vec{s}$ requires $O((c-1)\cdot N \log N)$ multiplications over $\mathbb{F}$, hence the running time of the attack is lower bounded by

$$O\left(\binom{(c-1)\cdot N}{(c-1)\cdot \frac{w}{c}} \cdot (c-1)\cdot N \log N\right).$$

Note that when $\mathbb{F} = \mathbb{F}_2$, the smallest integer d such that $2^d = 1 \bmod m$ (where m is such that $\phi(m) = N$, *i.e.*, F is the m-th cyclotomic polynomial) is at least $\log N$, hence F splits into at most N/d factors and when this is the case, $\mathbb{F}_2[X]/F(x) = R \cong \mathbb{F}_{2^d}^{N/d}$. When applying the brute-force attack to a reduced instance, the cost grows as

$$\binom{((c-1)\cdot d)^{|\mathbb{F}_{2^d}|-1}}{((c-1)\cdot \frac{w}{c})^{|\mathbb{F}_{2^d}|-1}} \cdot (c-1)\cdot N \log N \approx \binom{((c-1)\cdot d)^{N-1}}{((c-1)\cdot \frac{w}{c})^{N-1}} \cdot (c-1)\cdot N \log N.$$

Since N will be very large (e.g. about 2^{20}) in our instantiations, this attack is never feasible.

Improved Small Field Attacks. We note, however, that when $\mathbb{F} = \mathbb{F}_2$ and F is reducible, one can significantly refine the naive brute-force attack which we described above. Since this setting is exactly the setting of the Lapin authentication protocol [HKL$^+$12], which is perhaps the flagship application of R-LPN, it has been the subject of extensive cryptanalysis in [BL12,GJL15], which managed to break some candidate parameters of the original proposal. We will not cover their attacks in detail, but note that parameters which were conjectured to provide 80 bits of security in the original proposal (where the estimation was based on standard attacks on LPN, ignoring the ring structure), were shown to provide only 70 bits of security in [GJL15]. Therefore, when using $\mathbb{F} = \mathbb{F}_2$ and a reducible F, the state-of-the-art attack of [GJL15] should be taken into account and the security margin must be increased by at least a comparable factor.

8.6 Attacks on R-LPN with Static Leakage

Eventually, our malicious distributed setup protocol in Section 5.3 relies on the R-LPN assumption *with static leakage* (Definition 6.2). In this variant, security is based on the following game: after the c noise vectors $\vec{e}_0, \cdots \vec{e}_{c-1}$ sampled, the adversary is allowed to submit $c \cdot t$ arbitrary predicates $P_k^i : [0, N) \mapsto \{0, 1\}$, where P_k^i takes as input the position of the k-th nonzero entry of $\vec{e}_i$. If any P_k^i returns 0, we abort and the adversary looses the game. Otherwise, we sens "success" to the adversary, and he can now attempt to distinguish whether he got a random vector or a noisy codeword.

As outlined in 5.3, R-LPN with static leakage can be reduced to R-LPN without leakage. However, this comes at a strong loss in the reduction, and updating parameters to reflect this reduction would decrease the efficiency of our schemes. While the resulting efficiency would still be acceptable, we observe that all the attacks described in this section do not perform significantly better against the R-LPN with static leakage assumption. The reason is that in all the attacks mentioned in this section, the choices made by the adversary when trying to distinguish $\vec{b} = A\vec{s} + \vec{e}$ from random (e.g. picking candidate non-noisy coordinates in the Gaussian elimination attack, or choosing low-weight parity-check vectors in the statistical decoding attack) are made *independently of* $\vec{b}$. Therefore, in all these attacks, the success probability of the adversary can be computed by sampling the noise vector $\vec{e}$ *after the adversary made his choices*; indeed, this is exactly how we estimate the complexity of the attack in our asymptotic estimations.

This implies that, for all the above attacks, the success probability of the adversary can be obtained by fixing his choice of predicates P_k^i, his choice of attack parameters (e.g. candidate non-noisy coordinates in the Gaussian elimination attack, low-weight parity-check vectors in the statistical decoding attack) and computing $p_0 p_1$ with

- the probability p_0 that all P_k^i return 1, over a random choice of $\vec{e}$;
- the probability p_1 that the attack succeeds when $\vec{e}$ is sampled *conditioned on all* P_k^i *returning 1.*

However, the above probability is upper bounded by the probability that the attack succeeds for a random noise vector not conditioned on the output of the predicates. The takeway message is that any attack that selects its attack parameters without using $\vec{b}$ (but possibly using A and all other parameters of the system) cannot succeed better at breaking R-LPN with static leakage than at breaking the standard non-leaky R-LPN. In light of the fact that all known attacks that apply to our setting have this feature, it seems that, for the same parameters, R-LPN with static leakage offers the same concrete level of security as R-LPN without leakage.

9 Efficiency Analysis

In this section, based on our security analysis from Section 8, we discuss concrete choices of parameters for which the corresponding ring-LPN problems are secure against the attacks we considered. We then analyse the concrete efficiency of our PCGs in terms of seed size, communication complexity of the setup protocol, and estimated computational costs of seed expansion.

We focus on the case of a large finite field $\mathbb{F} = \mathbb{Z}_p$, for a prime p with $\log p \approx 128$, as is commonly used in MPC implementations [KOS16, KPR18]. Further, for improved efficiency, we always choose the ring-LPN noise vectors to have a regular structure (as was done e.g. in [BCGI18, BCG$^+$19b, BCG$^+$19a]), which does not introduce any known weaknesses.

Estimating Attack Costs. For large fields, we focus on the statistical decoding and information set decoding (ISD) families of attacks (the latter being always at least as efficient as the Gaussian elimination attack), combined with the speedup obtained with the DOOM attack against quasi-cyclic codes. For statistical decoding, we compute our estimations with a conservative lower bound of $n \cdot (cN/(N-1))^w$ arithmetic operations. For ISD, we used the parameter estimation tool developed for the LEDA candidate [BBC$^+$19] in the NIST post-quantum competition[11]. This software takes as input the parameters (dimension, number of sample, number of noisy coordinates, block-size of the quasi-cyclic matrices) of the instance, and outputs the

[11] `https://github.com/LEDAcrypt/LEDAtools`

complexity of attacking the instance with several ISD variants, namely those of Prange [Pra62], Lee and Brickell [LB88], Leon [Leo88], Stern [Ste88], Finiasz and Sendrier [FS09], May, Meurer, and Thomae [MMT11], and Becker, Joux, May, and Meyer [BJMM12], while taking into account the speedup of the DOOM attack when the code is quasi-cyclic. Furthermore, observing that the C++ code of the software implements matrix inversion using standard (cubic time) Gaussian elimination, we modified the matrix inversion used in the code to account for polynomial speedups obtained by using fast (quasi-quadratic time) algorithms for inversion of structured matrices.

9.1 Comparing Reducible and Irreducible Ring-LPN

We start by comparing parameters for the PCGs based on the reducible and irreducible variants of ring-LPN. Recall that in the reducible case, $F(X)$ splits completely into linear factors modulo p, so we can obtain OLEs or triples over $\mathbb{Z}_p$ (or an extension field $\mathbb{F}_{p^d}$). To improve computational efficiency, we use the cyclotomic polynomial $F(X) = X^N + 1$, for N a power of two. In the case where $F(X)$ is irreducible, we only produce a single, large OLE/triple over $\mathbb{Z}_p[X]/F(X)$.

Reducible Ring-LPN Parameters. We consider a field $\mathbb{F} = \mathbb{Z}_p$ of size $|\mathbb{F}| \approx 2^{128}$. As in our applications, we focus on the case where there is a factor f_i of degree $n = N/k$, for some k, which has sparsity 1, such as when N is a power of two and $F(X) = X^N + 1$ splits completely into N linear factors modulo p. From the analysis in Section 8.2, we can reduce an instance modulo a 1-sparse factor f_i of degree $n = 2^i$, reducing the expected number of noisy coordinates to

$$w_i = w - cn + (c(n-1) + w) \cdot \left(1 - \frac{1}{n}\right)^{w/c-1},$$

the dimension to $n_i = (c-1)\cdot 2^i$, and the number of samples to $q_i = c\cdot 2^i$. In our experiments, we found that the optimal behavior for the adversary was always to pick the smallest i such that the new weight w_i of the noise is not higher than the dimension n_i (such that the reduced instance is still uniquely decodable and is not statistically close to random). For security parameter $\lambda = 80$ (resp. $\lambda = 128$), the smallest such i is $i = 6$ (resp. $i = 7$). In Table 1, we provide various choices of parameters (λ, N, c, w) such that the best attack on any reduced instance requires at least 2^λ multiplications over a field $\mathbb{F}$ of size $|\mathbb{F}| \approx 2^{128}$. Observe that increasing N does not allow increasing the noise weight, since the best attack always exploits the structure of $F(X)$ by reducing to a much smaller dimension. The table also presents the concrete seed sizes and computational requirements for our PCG for OLE, based on Theorem 4.1 (and with optimizations due to the regular error distribution).

Our conservative estimates of the running time of the statistical decoding attack have better asymptotic complexity than the ISD attacks, according to our analysis in Section 8; and indeed, we found statistical decoding to always give the best available attack. Given that statistical decoding should not generally perform better than ISD [DAT17], this suggests that our estimation of the cost of statistical decoding might be overly conservative, meaning that our parameters might be slightly pessimistic. When using a smaller field $\mathbb{F}'$, the parameters in Table 1 change as follows: the size of the seed, as it is counted as a number of group elements, grows roughly by a factor $\log_2 |\mathbb{F}|/\log_2 |\mathbb{F}'|$, the stretch and the number of PRG calls decrease by the same factor (e.g. about a factor 2 when using $\mathbb{F}$ with $\log_2 |\mathbb{F}'| \approx 64$, meaning that the running time for generating N OLEs over $\mathbb{F}'$ decrease by a factor 2 compared to OLEs over $\mathbb{F}$). The reduction in the number of PRG calls requires encoding multiple field elements into a PRG output; for example, using AES, a single PRG call produces 128 pseudorandom bits, which suffices to "pack" two elements over a 64-bit field using an appropriate encoding.

Table 1: Concrete parameters and seed size (per party, counted as equivalent number of field elements) for our PCG for OLE over $\mathbb{Z}_p$ from reducible ring-LPN, where $p = 1 \bmod 2N$, $\log p \approx 128$, for various λ, N, syndrome compression factor c, and number of noisy coordinates w. 'Stretch', computed as $2N/(\text{seed size})$, is the ratio between storing a full random OLE (i.e., $2N$ field elements) and the smaller PCG seed. #PRG calls is computed as $4 \cdot Ncw$. Parameters are chosen to achieve λ-bits of security against known attacks (see Section 9.1 for the details on how the bit-security is estimated). This setting is useful for generating batches of N OLE correlations or authenticated triples over $\mathbb{Z}_p$, or small inner-product correlations (Section 7.2). See Section 9.1 for estimations of how to update the table for smaller field sizes.

λ	N	c	w	(i, w_i)	Seed size	Stretch	# R-mults	#PRG calls
80	2^{20}	2	97	$(6, 74)$	$2^{17.4}$	12	4	$2^{29.6}$
80	2^{20}	4	40	$(6, 37)$	$2^{15.0}$	65	16	$2^{29.3}$
80	2^{20}	8	26	$(6, 25)$	$2^{13.9}$	139	64	$2^{29.7}$
128	2^{20}	2	152	$(7, 121)$	$2^{18.6}$	5	4	$2^{30.2}$
128	2^{20}	4	64	$(7, 60)$	$2^{16.3}$	27	16	$2^{30.0}$
128	2^{20}	8	41	$(7, 40)$	$2^{15.1}$	59	64	$2^{30.4}$
80	2^{25}	2	97	$(6, 74)$	$2^{17.7}$	306	4	$2^{34.6}$
80	2^{25}	4	40	$(6, 37)$	$2^{15.3}$	1654	16	$2^{34.3}$
80	2^{25}	8	26	$(6, 25)$	$2^{14.2}$	3623	64	$2^{34.7}$
128	2^{25}	2	152	$(7, 121)$	$2^{19.0}$	130	4	$2^{35.2}$
128	2^{25}	4	64	$(7, 60)$	$2^{16.6}$	673	16	$2^{35.0}$
128	2^{25}	8	41	$(7, 40)$	$2^{15.4}$	1513	64	$2^{35.4}$

Irreducible Case. We consider a field $\mathbb{F} = \mathbb{Z}_p$ of size $|\mathbb{F}| \approx 2^{128}$. We consider various choices of parameters (N, c, w), such that the time complexity of the best attack, with statistical decoding or any attack from the ISD family, takes time at least 2^λ (using all the optimizations discussed previously). As with the reducible case, our analysis in Section 8 (which may be too pessimistic) shows the statistical decoding attack to have the best complexity. Given parameters (λ, N, c, w), our PCG for generating one pseudorandom OLE correlation over R has seeds of size equivalent to

$$w \cdot (1 + \log N / \log |\mathbb{F}|) + w^2 \cdot ((\log N + 1)(\lambda + 2) + \lambda + \log |\mathbb{F}|) / \log |\mathbb{F}|$$

elements of $\mathbb{F}$. When the error distribution is regular, the seed size can be reduced to

$$w \cdot (1 + \log N / \log |\mathbb{F}|) + w^2 \cdot (\log(2Nc/w) \cdot (\lambda + 2) + \lambda + \log |\mathbb{F}|) / \log |\mathbb{F}|.$$

The computational complexity of the expansion algorithm is dominated by c^2 multiplications over R, and $4Ncw$ calls to a PRG for evaluating FullEval in the underlying SPFSS. The results are represented on Table 2. When using a smaller field $\mathbb{F}'$, the size of the seed, as it is counted as a number of group elements, grows roughly by a factor $\log_2 |\mathbb{F}| / \log_2 |\mathbb{F}'|$, the stretch and the number of PRG calls decrease by the same factor (e.g. about a factor 2 when using $\mathbb{F}$ with $\log_2 |\mathbb{F}'| \approx 64$, meaning that the running time for generating N OLEs over $\mathbb{F}'$ decrease by a factor 2 compared to OLEs over $\mathbb{F}$). The reduction in the number of PRG calls requires encoding multiple field elements into a PRG output; for example, using AES, a single PRG call produces 128 pseudorandom bits, which suffices to "pack" two elements over a 64-bit field using an appropriate encoding.

Compared with the reducible case in Table 1, notice that when $N = 2^{20}$, using an irreducible $F(X)$ allows the noise weight w to be around 10–40% smaller, for the same level of security. This saving is slightly more pronounced for larger N, since in the reducible version, an increase in N does not allow any reduction in the noise weight.

Table 2: Size of the seed (per party, counted as an equivalent number of field elements) for generating an OLE correlation over $R = \mathbb{F}[X]/F(X)$, where F is irreducible, $\lceil \log_2 |\mathbb{F}| \rceil = 128$ and $\deg F = N$ for various λ, N, syndrome compression factor c, and number of noisy coordinates w. 'Stretch' is computed as $2N/$(seed size), and refers to the ratio between storing a full random OLE (i.e., $2N$ field elements) and storing the smaller seed. #PRG calls is computed as $4 \cdot Ncw$. Parameters are chosen to achieve λ-bits of security against known attacks (see Section 9.1 for the details on how the bit-security is estimated). Note that increasing c always decreases the seed size, but increases the running time of the expansion algorithm. This setting is useful for generating large OLE correlations or authenticated triples over R, inner-product correlations (Section 7.2), or batch matrix product correlations (Section 7.4). See Section 9.1 for estimations of how to update the table for smaller field sizes.

λ	N	c	w	seed size	stretch	# R-mults	#PRG calls
80	2^{20}	2	60	$2^{16.0}$	32	4	$2^{28.9}$
80	2^{20}	4	30	$2^{14.2}$	114	16	$2^{28.9}$
80	2^{20}	8	20	$2^{13.1}$	238	64	$2^{29.3}$
128	2^{20}	2	108	$2^{17.6}$	10	4	$2^{29.8}$
128	2^{20}	4	54	$2^{15.8}$	37	16	$2^{29.8}$
128	2^{20}	8	36	$2^{14.7}$	76	64	$2^{30.2}$
80	2^{25}	2	55	$2^{16.1}$	941	4	$2^{33.8}$
80	2^{25}	4	28	$2^{14.3}$	3344	16	$2^{33.8}$
80	2^{25}	8	19	$2^{13.3}$	6834	64	$2^{34.2}$
128	2^{25}	2	103	$2^{17.9}$	279	4	$2^{34.7}$
128	2^{25}	4	52	$2^{16.0}$	1006	16	$2^{34.7}$
128	2^{25}	8	35	$2^{15.0}$	2085	64	$2^{35.1}$

9.2 Estimated Costs and Runtimes for OLE and Triple Generation

We now look more closely at the concrete costs of setting up and expanding the PCG seeds. Here, we consider the task of producing $N = 2^{20}$ OLEs or triples over $\mathbb{Z}_p$, using reducible ring-LPN. (For smaller numbers of outputs, see the discussion in Section 9.3.)

Methodology. We estimate both the computational cost of expanding a PCG seed, as well as the communication cost required to distribute the PCG seeds with either passive or active security. When measuring communication, we do not include the one-time setup phase for bootstrapping the protocol with an initial batch of OLEs or multiplication triples (in practice, these can be created with a non-PCG based protocol such as ring-LWE).

For computation, the main costs in the expansion step are the DPF full-domain evaluations, and polynomial operations over R_p. We separately benchmarked these using the DPF code from [BCG+19a], and NFLLib [ABG+16] for polynomial arithmetic with a 124-bit modulus p, which is a product of two 62-bit primes (such that R_p splits completely into linear factors, by the CRT). The benchmarks were run on a single core of an Intel i7-7600U 2.8GHz processor.

To estimate the communication complexity of setting up the seeds, we first assume that as a one-time setup, the parties already have access to a single pair of PCG seeds (for multiplication triples). We then measure the cost of bootstrapping this to produce another seed, based on the analysis from Section 6.3.

Cost Estimates for $N = 2^{20}$ Correlations. The results are in Table 3 for OLE, and Table 4 for authenticated triples. Note that compared with Table 1, the noise weight w has been rounded so it is divisible by c, so that $t = w/c$ is an integer. We see that as the module-LPN compression factor c increases, the polynomial arithmetic gets more expensive, while the DPF cost first decreases at $c = 4$, and then goes back up at $c = 8$. This is because the DPF complexity scales

Table 3: Estimated costs for our PCG for producing $N = 2^{20}$ OLEs in $\mathbb{Z}_p$, with $\log p \approx 124$. Seed size is the size of one party's seed, setup comm. measures the per-party communication required to setup the PCG seeds (ignoring costs for correlated randomness that can come from a previous PCG).

λ	c	w	Seed size (MB)	Setup comm. (MB)		Runtimes for Expand (s)		
				passive	active	R-mult (s)	DPF eval. (s)	Total (s)
80	2	96	2.69	6.14	6.69	0.4	9.8	10.2
80	4	40	0.52	1.17	1.28	1.4	7.5	8.9
80	8	32	0.35	0.78	0.86	5.3	9.3	14.6
128	2	152	6.37	14.63	15.92	0.4	12.6	13.0
128	4	64	1.26	2.86	3.12	1.4	8.6	10.0
128	8	40	0.55	1.22	1.34	5.3	14.4	19.7

Table 4: Estimated costs for our PCG for producing $N = 2^{20}$ authenticated triples in $\mathbb{Z}_p$, with $\log p \approx 124$. Seed size is the size of one party's seed, setup comm. measures the per-party communication required to setup the PCG seeds (ignoring costs for correlated randomness that can come from a previous PCG).

λ	c	w	Seed size (MB)	Setup comm. (MB)	Runtimes for Expand (s)		
					R-mult (s)	DPF eval. (s)	Total (s)
80	2	96	5.49	8.77	0.8	19.6	20.4
80	4	40	1.09	1.68	2.8	15.0	17.8
80	8	32	0.74	1.13	10.6	18.6	29.2
128	2	152	12.91	20.97	0.8	25.2	26.0
128	4	64	2.60	4.09	2.8	17.2	20.0
128	8	40	1.14	1.75	10.6	28.8	39.4

with c^2t, so doubling c only reduces its cost if t can be reduced by more than a factor of 4. The best choice for speed seems to be $c = 4$, where we are able to silently expand over 100 thousand OLEs per second at the 128-bit security level, with a seed size of around 1MB. When generating authenticated triples instead of OLEs, the seed size and runtimes increase by roughly a factor of two, while the setup communication cost is only slightly larger.

9.3 Comparison with OLE From Ring-LWE

In Table 5, we compare the cost of our OLE protocol from ring-LPN with the passively secure OLE protocol from ring-LWE by Baum et al [BEPU$^+$20]. For a 120-bit plaintext modulus, the protocol implemented in [BEPU$^+$20] has an average communication cost of 420 bits per party, for each OLE. Excluding the one-time setup, our ring-LPN based protocol with $c = 4$ improves upon the communication complexity of ring-LWE when producing $N = 65536$ or more OLEs. With $c = 2$, when N is 32768 or 65536, the ring-LPN protocol requires more OLEs as preprocessing than it produces as output, so is not beneficial; instead, the asymptotic improvement in communication starts to take effect when N is half a million or more.

10 PCG for OLE from Standard LPN

In this section, we provide new constructions of PCG for OLE and matrix product correlations. Unlike the other PCG constructed in this work, the PCGs of this section do not rely on the R-LPN assumption, but on the standard LPN assumption (or alternatively, some variant of it with a structured parity-check matrix H allowing for fast multiplication with H). The PCG for OLE we obtain is typically less efficient than those based on R-LPN, but rely on more

Table 5: Comparing costs of our passively secure OLE protocol from ring-LPN with ring-LWE. # OTs and #OLEs measure the amount of correlated randomness required to run one instance of the protocol. Communication is averaged per-party

N	c	w	Ring-LPN costs			Ring-LWE costs [BEPU+20]
			# OTs	# OLEs	Comm. (MB)	Comm. (MB)
32768	2	152	877952	115520	10.9	1.72
65536	2	152	970368	115520	11.6	3.44
131072	2	152	1062784	115520	12.4	6.88
262144	2	152	1155200	115520	13.1	13.78
524288	2	152	1247616	115520	13.9	27.5
32768	4	64	188416	20480	2.19	1.72
65536	4	64	204800	20480	2.33	3.44
131072	4	64	221184	20480	2.46	6.88
262144	4	64	237568	20480	2.59	13.8
524288	4	64	253952	20480	2.73	27.5

well-studied assumption, hence can be seen as a conservative alternative. Furthermore, our LPN-based PCGs for matrix multiplication correlations is incomparable to our R-LPN-based PCG for batch matrix multiplication: while the later generates large batch of small-to-moderate size matrix multiplication triples, the former allows to generate one (potentially very large) matrix multiplication triple.

Our starting point is the LPN-based construction of PCGs for bilinear correlations from Boyle et al. [BCG+19b], over an arbitrary field $\mathbb{F}$, which we recall below. In [BCG+19b], this construction was estimated to be mainly of theoretical interest: generating $m = O(n)$ OLEs with this PCG requires $O(n^4)$ arithmetic operations, and a seed of size $O(t^2 \log n)$. When n is large enough for the seed to provide a nontrivial compression factor, the computational overhead is already impractical. In this section, we will give an optimized variant of this construction which requires $O(n^\omega)$ arithmetic operations, where ω is the matrix multiplication exponent. This cost can be further reduced to $O(n^2 \log n)$ by relying on any variant of LPN with structured parity-check matrices allowing for fast ($o(n^2)$) matrix-vector products, such as a Toeplitz or a quasi-cyclic parity-check matrix.

10.1 The Construction of [BCG+19b]

Theorem 10.1 (From [BCG+19b], Section 6) *Suppose the* $\mathsf{dual\text{-}LPN}_{m,n,t}$ *assumption holds relative to* H*, and that* SPFSS *is a secure sum of point function secret sharing scheme. Then the construction* G_{bil} *(Fig. 14) is a secure PCG for general bilinear correlations.*

Efficiency. Instantiating the SPFSS as in [BCGI18], the setup algorithm of G_{bil} outputs seeds of size $t^2 \cdot (\lceil \log n \rceil (\lambda + 2) + \lambda + \log_2 |\mathbb{F}|)$ bits, which amounts to $\tilde{O}(t^2 \log n)$ field elements over a large field ($\log_2 |\mathbb{F}| = O(\lambda)$). Expanding the seed involves $(tn)^2$ PRG evaluations and $O(m \cdot n)^2 = O(n^4)$ arithmetic operations.

10.2 Optimized Construction

The main source of inefficiency in the above construction stems from the fact that to compute a bilinear function of two (pseudo)random strings, the expansion algorithm must obtain the tensor product between the pseudorandom strings $(\vec{x}_0, \vec{x}_1)$. In the construction, it holds that

$$\vec{x}_\sigma = H \cdot \vec{e}_\sigma \text{ for } \sigma = 0, 1$$

Construction G_{bil}

PARAMETERS: $1^\lambda, m, n, t, \in \mathbb{N}$, where $n > m$; a field $\mathbb{F}$. A parity-check matrix $H \in \mathbb{F}^{m\times n}$ for a code of dimension $n-m$ and number of samples m over $\mathbb{F}$. A bilinear function $B_{\vec{c}} : \mathbb{F}^n \times \mathbb{F}^n \to \mathbb{F}, (\vec{\alpha}, \vec{\beta}) \mapsto \langle \vec{c}, (\vec{\alpha} \otimes \vec{\beta})\rangle$.

Gen: On input 1^λ:

1. Pick two random sparse vectors $\vec{e}_0, \vec{e}_1 \xleftarrow{\$} \mathbb{F}^n$ with $\mathrm{wt}(\vec{e}_0) = \mathrm{wt}(\vec{e}_1) = t$. Define f to be the sum of t^2 point functions whose evaluation on its entire domain is $\vec{e}_0 \otimes \vec{e}_1$.
2. Compute $(K_0, K_1) \xleftarrow{\$} \mathsf{SPFSS.Gen}(1^\lambda, f)$.
3. Let $\mathsf{k}_0 \leftarrow (K_0, \vec{e}_0)$ and $\mathsf{k}_1 \leftarrow (K_1, \vec{e}_1)$.
4. Output $(\mathsf{k}_0, \mathsf{k}_1)$.

Expand: On input $(\sigma, \mathsf{k}_\sigma)$, parse k_σ as $(K_\sigma, \vec{e}_\sigma)$. Set $\vec{x}_\sigma \leftarrow H \cdot \vec{e}_\sigma$. Compute $\vec{u}_\sigma \leftarrow \mathsf{SPFSS.FullEval}(\sigma, K_\sigma)$ in $\mathbb{F}_p^{n^2}$, and set $\vec{z}_\sigma \leftarrow \langle \vec{c}, (H \otimes H) \cdot \vec{u}_\sigma \rangle$. Output $(\vec{x}_\sigma, \vec{z}_\sigma)$.

Figure 14: PCG for Bilinear Correlations

hence

$$\begin{aligned}\vec{x}_0 \otimes \vec{x}_1 &= (H \cdot \vec{e}_0) \otimes (H \cdot \vec{e}_1)\\ &= (H \otimes H) \cdot (\vec{e}_0 \otimes \vec{e}_1).\end{aligned}$$

The tensor product between noise vectors $\vec{e}_0 \otimes \vec{e}_1$ is compressed using SPFSS. Then, the $O(n^4)$ overhead comes from multiplying this length-n^2 vector with an $m^2 \times n^2$ matrix. However, computing the above can be done much more efficiently by observing that the $n \times n$ square matrix $\vec{x}_0 \cdot \vec{x}_1^\intercal$ contains exactly the same entries as the tensor product between $\vec{x}_0$ and $\vec{x}_1$. Hence, any bilinear function of $(\vec{x}_0, \vec{x}_1)$ can be trivially computed as a linear combination between the components of $\vec{x}_0 \cdot \vec{x}_1^\intercal$. Furthermore, we can compute $\vec{x}_0 \cdot \vec{x}_1^\intercal$ much more efficiently using the following identity:

$$\begin{aligned}\vec{x}_0 \cdot \vec{x}_1^\intercal &= (H \cdot \vec{e}_0) \cdot (H \cdot \vec{e}_1)^\intercal\\ &= H \cdot (\vec{e}_0 \cdot \vec{e}_1^\intercal) \cdot H^\intercal.\end{aligned}$$

As before, $\vec{e}_0 \cdot \vec{e}_1^\intercal$ can be generated from an $O(t^2 \log n)$-long seed using SPFSS. But now, computing $H \cdot (\vec{e}_0 \cdot \vec{e}_1^\intercal) \cdot H^\intercal$ requires only $O(n^\omega)$ operations, where ω is the matrix multiplication exponent. Furthermore, if H is a structured matrix, this can be computed even more efficiently; for example, taking H to be a Toeplitz matrix or a quasi-cyclic matrix leads to a computation cost of $O(n^2 \log n)$ for generating $m = O(n)$ OLEs, under well-established variants of the LPN assumption. This provides a less efficient, but still feasible more conservative alternative to our construction based on the splittable variant of R-LPN. Hence, we directly get:

Theorem 10.2 *Suppose the* $\mathsf{dual\text{-}LPN}_{m,n,t}$ *assumption holds relative to random Toeplitz matrices, and that* SPFSS *is a secure SPFSS scheme. Then the construction described in this section is a secure PCG for general bilinear correlations with seeds of size* $\tilde{O}(t^2 \log n)$ *field elements over a large field* ($\log_2 |\mathbb{F}| = O(\lambda)$)*. Expanding the seed involves* $O((tn)^2)$ *PRG evaluations and* $O(n^2 \log n)$ *arithmetic operations.*

10.3 LPN-Based PCG for Matrix Multiplication

In this section, we describe two new variants of the PCG described in the previous section, which allow to generate large matrix multiplication triples over $\mathbb{F}$ (with different parameters tradeoffs across the two constructions).

First Construction. The first construction directly generalizes the construction above to generating pseudorandom correlations $(X_0, Z_0) \in \mathbb{F}^{m\times m} \times \mathbb{F}^{m\times m}$ and $(X_1, Z_1) \in \mathbb{F}^{m\times m} \times \mathbb{F}^{m\times m}$ such that $X_0 \cdot X_1^\intercal = Z_0 + Z_1$.

PARAMETERS: $1^\lambda, m, n, t, p \in \mathbb{N}$, where $n > m$. A parity-check matrix $H \in \mathbb{F}^{m\times n}$ for a code of dimension $n - m$ and number of samples m over $\mathbb{F}$.

Gen: On input 1^λ:

1. Pick two random matrices $E_0, E_1 \in \mathbb{F}^{n\times m}$ such that each column of E_0, E_1 contains exactly t nonzero entries.
2. Compute SPFSS keys (K_0, K_1) which generate shares of $E_0 \cdot E_1^\intercal$.
3. Let $\mathsf{k}_0 \leftarrow (K_0, E_0)$ and $\mathsf{k}_1 \leftarrow (K_1, E_1)$.
4. Output $(\mathsf{k}_0, \mathsf{k}_1)$.

Expand: On input $(\sigma, \mathsf{k}_\sigma)$, parse k_σ as (K_σ, E_σ). Set $X_\sigma \leftarrow H \cdot E_\sigma$. Compute $U_\sigma \leftarrow \mathsf{SPFSS.FullEval}(\sigma, K_\sigma)$ in $\mathbb{F}^{n\times n}$ and set $Z_\sigma \leftarrow H \cdot U_\sigma \cdot H^\intercal$. Output (X_σ, Z_σ).

The correctness and security of the construction follows by the same argument as for G_{bil} (Section 10.2). Regarding efficiency, the expansion cost is optimal, $O(n^\omega)$ using random parity-check matrices, or $O(n^2 \cdot \log n)$ using Toeplitz or quasi-cyclic matrices. Note that this means that generating a pseudorandom $m \times m$ matrix multiplication triples via this method is even faster than computing the product of random $m \times m$ matrices in the clear (for $m = O(n)$). On the downside, the size of the seed grows now as $O(t^2 n^2 \log n)$; i.e., the expansion is limited to subquadratic in the seed size. The next construction allows to achieve much better seed size (logarithmic in n), but has a higher computational complexity.

Second Construction. The second construction is similar in spirit to the previous one; it achieves a much smaller seed size, albeit with a larger computational cost $O(n^4 \log n)$. The high level idea is to generate each $m \times m$ matrix X_σ by constructing a length-m^2 vector $\vec{x}_\sigma = H \cdot \vec{e}_\sigma$, where H is an $m^2 \times n^2$ parity-check matrix (e.g. a Toeplitz or quasi-cyclic matrix), and $\vec{e}_\sigma$ is a length-n^2 t-sparse vector. Let $H_i \in \mathbb{F}^{m\times n^2}$ the matrix comprising the rows $(i-1)\cdot m+1$ to $i \cdot m$ of the matrix H, i.e. H is horizontally parsed as (very flat) matrices $H_1, \dots H_m \in F^{m\times n^2}$. Then, we define X_σ to be the $m \times m$ matrix whose i-th column is equal two $H_i \cdot \vec{e}_\sigma$. In other words, the i-th column of X_σ contains the entries $(i-1)\cdot m + 1$ to $i \cdot m$ of $\vec{x}_\sigma$. We let $\mathsf{mat} : \mathbb{F}^{m^2} \mapsto \mathbb{F}^{m\times m}$ denote the operator mapping an m^2-vector to an $m \times m$ matrix.

PARAMETERS: $1^\lambda, m, n, t, p \in \mathbb{N}$, where $n > m$. A parity-check matrix $H \in \mathbb{F}^{m^2\times n^2}$ for a code of dimension $n^2 - m^2$ and number of samples m^2 over $\mathbb{F}$.

Gen: On input 1^λ:

1. Pick two random vectors $\vec{e}_0, \vec{e}_1 \in \mathbb{F}^{n^2}$ with exactly t nonzero entries.
2. Compute SPFSS keys (K_0, K_1) which generate shares of $\vec{e}_0 \cdot \vec{e}_1^\intercal$.

3. Let $\mathsf{k}_0 \leftarrow (K_0, \vec{e}_0)$ and $\mathsf{k}_1 \leftarrow (K_1, \vec{e}_1)$.

4. Output $(\mathsf{k}_0, \mathsf{k}_1)$.

Expand: On input $(\sigma, \mathsf{k}_\sigma)$, parse k_σ as $(K_\sigma, \vec{e}_\sigma)$. Set $X_\sigma \leftarrow \mathsf{mat}(H \cdot \vec{e}_\sigma)$. Compute $U_\sigma \leftarrow \mathsf{SPFSS.FullEval}(\sigma, K_\sigma)$ in $\mathbb{F}^{n^2 \times n^2}$ and set

$$Z_\sigma \leftarrow \sum_{i=1}^{m} H_i \cdot U_\sigma \cdot H_i^\intercal.$$

Output (X_σ, Z_σ).

Security follows from the same argument as previously. For correctness, note that by definition we can write $X_\sigma = [H_1 \cdot \vec{e}_\sigma \mid \cdots \mid H_m \cdot \vec{e}_\sigma]$, which implies $X_0 \cdot X_1^\top = \sum_{i=1}^m H_i \cdot \vec{e}_0 \cdot (H_i \cdot \vec{e}_1)^\top = \sum_{i=1}^m H_i \cdot (\vec{e}_0 \cdot \vec{e}_1^\top) \cdot H_i^\top$ and thus correctness follows.

The seed size is now much smaller, $O(t^2 \log n)$; however, the computational complexity is dominated by the matrix product $H \cdot U_\sigma$, which takes time $O(n^4 \log n)$ if H is a structured matrix. This provides a much better tradeoff than when generating OLEs: the cost of generating $m = O(n)$ OLEs with (our improved version of) G_{bil} is $\tilde{O}(n^2)$, quadratically larger than the cost of generating them in the clear. However, the cost of generating a large random matrix multiplication correlation is $M(n) = O(n^\omega)$, meaning that the cost of our algorithm is below $O(M\sqrt{M})$ when the matrix multiplication algorithm is implemented with e.g. Strassen's algorithm.

11 Acknowledgements

We would like to thank Vadim Lyubashevsky, Chris Peikert, Ronny Roth and Jean-Pierre Tillich for helpful discussions and pointers. We also thank Damiano Abram for various corrections, including pointing out a security flaw in a previous version of the malicious DPF protocol.

E. Boyle, N. Gilboa, and Y. Ishai supported by ERC Project NTSC (742754). E. Boyle additionally supported by ISF grant 1861/16, AFOSR Award FA9550-17-1-0069, ERC Project HSS (852952), and a Google Research Award. G. Couteau supported by ERC Project PREP-CRYPTO (724307) and ANR SCENE. N. Gilboa additionally supported by ISF grant 1638/15, ISF 2951/20, ERC grant 876110, and a grant by the BGU Cyber Center. Y. Ishai additionally supported by NSF-BSF grant 2015782, BSF grant 2018393, ISF grant 2774/20, and a grant from the Ministry of Science and Technology, Israel and Department of Science and Technology, Government of India. L. Kohl is funded by NWO Gravitation project QSC. Most research of L. Kohl was done while at Technion, supported by ERC Project NTSC (742754). Research of L. Kohl was done in part while at Karlsruhe Institute of Technology, supported by ERC Project PREP-CRYPTO (724307) and DFG grant HO 4534/2-2. P. Scholl supported by the Danish Independent Research Council under Grant-IDs 6108-00169 (FoCC) and 0165-00107B (C3PO), and an Aarhus University Research Foundation starting grant.

References

[ABG+16] Carlos Aguilar Melchor, Joris Barrier, Serge Guelton, Adrien Guinet, Marc-Olivier Killijian, and Tancrède Lepoint. NFLlib: NTT-based fast lattice library. In Kazue Sako, editor, *CT-RSA 2016*, volume 9610 of *LNCS*, pages 341–356. Springer, Heidelberg, February / March 2016.

[ACLS18] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy*, pages 962–979. IEEE Computer Society Press, May 2018.

[ACPS09] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 595–618. Springer, Heidelberg, August 2009.

[AJ01] Abdulrahman Al Jabri. A statistical decoding algorithm for general linear block codes. In *IMA International Conference on Cryptography and Coding*, pages 1–8. Springer, 2001.

[BBC$^+$19] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. Design of LEDAkem and LEDApkc instances with tight parameters and bounded decryption failure rate. 2019. `https://www.ledacrypt.org/archives/official_comment.pdf`.

[BCG$^+$17] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, and Michele Orrù. Homomorphic secret sharing: Optimizations and applications. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2105–2122. ACM Press, October / November 2017.

[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*, pages 291–308. 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 Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 489–518. Springer, Heidelberg, August 2019.

[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*, pages 896–912. ACM Press, October 2018.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 169–188. Springer, Heidelberg, May 2011.

[Bea91] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO '91*, pages 420–432, 1991.

[Bea92] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, Heidelberg, August 1992.

[BEPU$^+$20] Carsten Baum, Daniel Escudero, Alberto Pedrouzo-Ulloa, Peter Scholl, and Juan Ramón Troncoso-Pastoriza. Efficient protocols for oblivious linear function evaluation from ring-LWE. In Clemente Galdi and Vladimir Kolesnikov, editors, *SCN 20*, volume 12238 of *LNCS*, pages 130–149. Springer, Heidelberg, September 2020.

[BFKL94] Avrim Blum, Merrick L. Furst, Michael J. Kearns, and Richard J. Lipton. Cryptographic primitives based on hard learning problems. In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 278–291. Springer, Heidelberg, August 1994.

[BGI15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 337–367. Springer, Heidelberg, April 2015.

[BGI16a] Elette Boyle, Niv Gilboa, and Yuval Ishai. Breaking the circuit size barrier for secure computation under DDH. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 509–539. Springer, Heidelberg, August 2016.

[BGI16b] 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*, pages 1292–1303. ACM Press, October 2016.

[BGI17] Elette Boyle, Niv Gilboa, and Yuval Ishai. Group-based secure computation: Optimizing rounds, communication, and computation. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 163–193. Springer, Heidelberg, April / May 2017.

[BGI19] Elette Boyle, Niv Gilboa, and Yuval Ishai. Secure computation with preprocessing via function secret sharing. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part I*, volume 11891 of *LNCS*, pages 341–371. Springer, Heidelberg, December 2019.

[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, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 520–536. Springer, Heidelberg, April 2012.

[BKS19] Elette Boyle, Lisa Kohl, and Peter Scholl. Homomorphic secret sharing from lattices without FHE. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 3–33. Springer, Heidelberg, May 2019.

[BKW00] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. In *32nd ACM STOC*, pages 435–440. ACM Press, May 2000.

[BL12] Daniel J Bernstein and Tanja Lange. Never trust a bunny. In *International Workshop on Radio Frequency Identification: Security and Privacy Issues*, pages 137–148. Springer, 2012.

[BLP11] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Smaller decoding exponents: Ball-collision decoding. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 743–760. Springer, Heidelberg, August 2011.

[BNO19] Aner Ben-Efraim, Michael Nielsen, and Eran Omri. Turbospeedz: Double your online SPDZ! Improving SPDZ using function dependent preprocessing. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19*, volume 11464 of *LNCS*, pages 530–549. Springer, Heidelberg, June 2019.

[BV11a] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In Rafail Ostrovsky, editor, *52nd FOCS*, pages 97–106. IEEE Computer Society Press, October 2011.

[BV11b] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 505–524. Springer, Heidelberg, August 2011.

[CDI$^+$19] Melissa Chase, Yevgeniy Dodis, Yuval Ishai, Daniel Kraschewski, Tianren Liu, Rafail Ostrovsky, and Vinod Vaikuntanathan. Reusable non-interactive secure computation. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 462–488. Springer, Heidelberg, August 2019.

[CGGU$^+$13] Alain Couvreur, Philippe Gaborit, Valérie Gauthier-Umana, Ayoub Otmani, and Jean-Pierre Tillich. Distinguisher-based attacks on public-key cryptosystems using reed-solomon codes. *arXiv preprint arXiv:1307.6458*, 2013.

[CKR$^+$20] Hao Chen, Miran Kim, Ilya P. Razenshteyn, Dragos Rotaru, Yongsoo Song, and Sameer Wagh. Maliciously secure matrix multiplication with applications to private deep learning. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 31–59. Springer, Heidelberg, December 2020.

[Cou19] Geoffroy Couteau. A note on the communication complexity of multiparty computation in the correlated randomness model. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 473–503. Springer, Heidelberg, May 2019.

[DAT17] Thomas Debris-Alazard and Jean-Pierre Tillich. Statistical decoding. In *2017 IEEE International Symposium on Information Theory (ISIT)*, pages 1798–1802. IEEE, 2017.

[DGN$^+$17] Nico Döttling, Satrajit Ghosh, Jesper Buus Nielsen, Tobias Nilges, and Roberto Trifiletti. TinyOLE: Efficient actively secure two-party computation from oblivious linear function evaluation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2263–2276. ACM Press, October / November 2017.

[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 Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, Heidelberg, September 2013.

[DNNR17] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. The TinyTable protocol for 2-party secure computation, or: Gate-scrambling revisited. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 167–187. Springer, Heidelberg, August 2017.

[DP12] Ivan Damgård and Sunoo Park. How practical is public-key encryption based on LPN and ring-LPN? Cryptology ePrint Archive, Report 2012/699, 2012. `https://eprint.iacr.org/2012/699`.

[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, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

[Ds17] Jack Doerner and abhi shelat. Scaling ORAM for secure computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 523–535. ACM Press, October / November 2017.

[EKM17] Andre Esser, Robert Kübler, and Alexander May. LPN decoded. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 486–514. Springer, Heidelberg, August 2017.

[FGUO+13] Jean-Charles Faugere, Valérie Gauthier-Umana, Ayoub Otmani, Ludovic Perret, and Jean-Pierre Tillich. A distinguisher for high-rate mceliece cryptosystems. *IEEE Transactions on Information Theory*, 59(10):6830–6844, 2013.

[FKI06] Marc PC Fossorier, Kazukuni Kobara, and Hideki Imai. Modeling bit flipping decoding based on nonorthogonal check sums with application to iterative decoding attack of mceliece cryptosystem. *IEEE Transactions on Information Theory*, 53(1):402–411, 2006.

[FKOS15] Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to MPC with preprocessing using OT. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part I*, volume 9452 of *LNCS*, pages 711–735. Springer, Heidelberg, November / December 2015.

[FS09] Matthieu Finiasz and Nicolas Sendrier. Security bounds for the design of code-based cryptosystems. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 88–105. Springer, Heidelberg, December 2009.

[GI14] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 640–658. Springer, Heidelberg, May 2014.

[GJL15] Qian Guo, Thomas Johansson, and Carl Löndahl. A new algorithm for solving ring-lpn with a reducible polynomial. *IEEE Transactions on Information Theory*, 61(11):6204–6212, 2015.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[GN19] Satrajit Ghosh and Tobias Nilges. An algebraic approach to maliciously secure private set intersection. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 154–185. Springer, Heidelberg, May 2019.

[HIMV19] Carmit Hazay, Yuval Ishai, Antonio Marcedone, and Muthuramakrishnan Venkitasubramaniam. LevioSA: Lightweight secure arithmetic computation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 327–344. ACM Press, November 2019.

[HKL+12] Stefan Heyse, Eike Kiltz, Vadim Lyubashevsky, Christof Paar, and Krzysztof Pietrzak. Lapin: An efficient authentication protocol based on ring-LPN. In Anne Canteaut, editor, *FSE 2012*, volume 7549 of *LNCS*, pages 346–365. Springer, Heidelberg, March 2012.

[HOSS18a] Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Concretely efficient large-scale MPC with active security (or, TinyKeys for TinyOT). In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 86–117. Springer, Heidelberg, December 2018.

[HOSS18b] Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. TinyKeys: A new approach to efficient multi-party computation. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 3–33. Springer, Heidelberg, August 2018.

[HS13] Yann Hamdaoui and Nicolas Sendrier. A non asymptotic analysis of information set decoding. *IACR Cryptology ePrint Archive*, 2013:162, 2013.

[HSS17] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 598–628. Springer, Heidelberg, December 2017.

[IKM$^+$13] Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, Claudio Orlandi, and Anat Paskin-Cherniavsky. On the power of correlated randomness in secure computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 600–620. Springer, Heidelberg, March 2013.

[IKOS04] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In László Babai, editor, *36th ACM STOC*, pages 262–271. ACM Press, June 2004.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 572–591. Springer, Heidelberg, August 2008.

[IPS09] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Secure arithmetic computation with no honest majority. In *TCC'09*, pages 294–314, 2009.

[JVC18] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *USENIX 2018*, pages 1651–1669, 2018.

[Kil88] Joe Kilian. Founding cryptography on oblivious transfer. In *20th ACM STOC*, pages 20–31. ACM Press, May 1988.

[KKW18] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 525–537. ACM Press, October 2018.

[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*, pages 830–842. ACM Press, October 2016.

[Kot92] Ralf Kotter. An unified description of an error locating procedure for linear codes. *Proc. IAACCT, Voneshta Voda, Bulgaria*, 1992.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 158–189. Springer, Heidelberg, April / May 2018.

[KSS09] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In Juan A. Garay, Atsuko Miyaji, and Akira Otsuka, editors, *CANS 09*, volume 5888 of *LNCS*, pages 1–20. Springer, Heidelberg, December 2009.

[LB88] Pil Joong Lee and Ernest F Brickell. An observation on the security of mceliece's public-key cryptosystem. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 275–280. Springer, 1988.

[Leo88] Jeffrey S Leon. A probabilistic algorithm for computing minimum weights of large error-correcting codes. *IEEE Transactions on Information Theory*, 34(5):1354–1359, 1988.

[LP15] Helger Lipmaa and Kateryna Pavlyk. Analysis and implementation of an efficient ring-LPN based commitment scheme. In Michael Reiter and David Naccache, editors, *CANS 15*, LNCS, pages 160–175. Springer, Heidelberg, December 2015.

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for ring-LWE cryptography. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 35–54. Springer, Heidelberg, May 2013.

[LWYY22] Hanlin Liu, Xiao Wang, Kang Yang, and Yu Yu. The hardness of LPN over any integer ring and field for PCG applications. Cryptology ePrint Archive, Paper 2022/712, 2022.

[Lyu05] Vadim Lyubashevsky. The parity problem in the presence of noise, decoding random linear codes, and the subset sum problem. In *Approximation, randomization and combinatorial optimization. Algorithms and techniques*, pages 378–389. Springer, 2005.

[MBD+18] Carlos Aguilar Melchor, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, and Gilles Zémor. Efficient encryption from random quasi-cyclic codes. *IEEE Trans. Information Theory*, 64(5):3927–3943, 2018.

[MCMMP14] Irene Márquez-Corbella, Edgar Martínez-Moro, and Ruud Pellikaan. On the unique representation of very strong algebraic geometry codes. *Designs, Codes and Cryptography*, 70(1-2):215–230, 2014.

[MCP12] Irene Márquez-Corbella and Ruud Pellikaan. Error-correcting pairs for a public-key cryptosystem. *arXiv preprint arXiv:1205.3647*, 2012.

[MMT11] Alexander May, Alexander Meurer, and Enrico Thomae. Decoding random linear codes in $\tilde{\mathcal{O}}(2^{0.054n})$. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 107–124. Springer, Heidelberg, December 2011.

[MO15] Alexander May and Ilya Ozerov. On computing nearest neighbors with applications to decoding of binary linear codes. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 203–228. Springer, Heidelberg, April 2015.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, Heidelberg, August 2012.

[NP99] Moni Naor and Benny Pinkas. Oblivious transfer and polynomial evaluation. In *31st ACM STOC*, pages 245–254. ACM Press, May 1999.

[Ove06] Raphael Overbeck. Statistical decoding revisited. In Lynn Margaret Batten and Reihaneh Safavi-Naini, editors, *ACISP 06*, volume 4058 of *LNCS*, pages 283–294. Springer, Heidelberg, July 2006.

[Pel92] Ruud Pellikaan. On decoding by error location and dependent sets of error positions. *Discrete Mathematics*, 106:369–381, 1992.

[PMCMM11] Ruud Pellikaan, Irene Márquez-Corbella, and Edgar Martínez-Moro. Evaluation of public-key cryptosystems based on algebraic geometry codes. In *Third International Castle Meeting on Coding Theory and Applications (3ICMTA, Cardona Castle, Barcelona, Spain, pages 199–204*, 2011.

[Pra62] Eugene Prange. The use of information sets in decoding cyclic codes. *IRE Transactions on Information Theory*, 8(5):5–9, 1962.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th ACM STOC*, pages 84–93. ACM Press, May 2005.

[Sen11a] Nicolas Sendrier. Decoding one out of many. In *International Workshop on Post-Quantum Cryptography*, pages 51–67. Springer, 2011.

[Sen11b] Nicolas Sendrier. Decoding one out of many. In Bo-Yin Yang, editor, *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011*, pages 51–67. Springer, Heidelberg, November / December 2011.

[SGRR19] Phillipp 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*, pages 1055–1072. ACM Press, November 2019.

[Ste88] Jacques Stern. A method for finding codewords of small weight. In *International Colloquium on Coding Theory and Applications*, pages 106–113. Springer, 1988.

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

[TS16] Rodolfo Canto Torres and Nicolas Sendrier. Analysis of information set decoding for a sub-linear error weight. In *International Workshop on Post-Quantum Cryptography*, pages 144–161. Springer, 2016.

[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*, pages 1607–1626. ACM Press, November 2020.

[Zic17] Lior Zichron. Locally computable arithmetic pseudorandom generators. Master's thesis, School of Electrical Engineering, Tel Aviv University, 2017.