Efficient Pseudorandom Correlation Generators: Silent OT Extension and More *

Geoffroy Couteau, Elette Boyle, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Scholl, Idc Herzliya

Efficient Pseudorandom Correlation Generators: Silent OT Extension and More*

Elette Boyle[1], Geoffroy Couteau[2], Niv Gilboa[3], Yuval Ishai[4], Lisa Kohl[2], and Peter Scholl[5]

[1] IDC Herzliya
[2] Karlsruhe Institute of Technology
[3] Ben-Gurion University of the Negev
[4] Technion
[5] Aarhus University

Abstract. Secure multiparty computation (MPC) often relies on sources of correlated randomness for better efficiency and simplicity. This is particularly useful for MPC with no honest majority, where input-independent correlated randomness enables a lightweight "non-cryptographic" online phase once the inputs are known. However, since the amount of correlated randomness typically scales with the circuit size of the function being computed, securely generating correlated randomness forms an efficiency bottleneck, involving a large amount of communication and storage.
A natural tool for addressing the above limitations is a *pseudorandom correlation generator* (PCG). A PCG allows two or more parties to securely generate long sources of useful correlated randomness via a local expansion of correlated short seeds and no interaction. PCGs enable MPC with *silent preprocessing*, where a small amount of interaction used for securely sampling the seeds is followed by silent local generation of correlated pseudorandomness.
A concretely efficient PCG for Vector-OLE correlations was recently obtained by Boyle et al. (CCS 2018) based on variants of the learning parity with noise (LPN) assumption over large fields. In this work, we initiate a systematic study of PCGs and present concretely efficient constructions for several types of useful MPC correlations. We obtain the following main contributions:

- **PCG foundations.** We give a general security definition for PCGs. Our definition suffices for any MPC protocol satisfying a stronger security requirement that is met by existing protocols. We prove that a stronger security requirement is indeed necessary, and justify our PCG definition by ruling out a stronger and more natural definition.
- **Silent OT extension.** We present the first concretely efficient PCG for oblivious transfer correlations. Its security is based on a variant of the binary LPN assumption and any correlation-robust hash function. We expect it to provide a faster alternative to the IKNP OT extension protocol (Crypto '03) when communication is the bottleneck. We present several applications, including protocols for non-interactive zero-knowledge with bounded-reusable preprocessing from binary LPN, and concretely efficient related-key oblivious pseudorandom functions.
- **PCGs for simple 2-party correlations.** We obtain PCGs for several other types of useful 2-party correlations, including (authenticated) one-time truth-tables and Beaver triples. While the latter PCGs are slower than our PCG for OT, they are still practically feasible. These PCGs are based on a host of assumptions and techniques, including specialized homomorphic secret sharing schemes and pseudorandom generators tailored to their structure.
- **Multiparty correlations.** We obtain PCGs for multiparty correlations that can be used to make the circuit-dependent communication of MPC protocols scale *linearly* (instead of quadratically) with the number of parties.

1 Introduction

Correlated secret randomness is a valuable resource for secure multi-party computation (MPC). A simple example is a common random key that is given to two parties, who can later use it as a one-time pad for secure message transmission. In the context of MPC, a more useful example is a random *oblivious transfer* (OT) correlation, in which one party is given a pair of random bits (more generally, strings) (s_0, s_1) and the other party is given the pair (r, s_r) for a random bit r. The OT correlation can serve as a basis for general MPC protocols with no honest majority [GMW87, Kil88, IPS08]. Other kinds of two-party correlations that are useful

* This is a full version of [BCG+19].

for MPC include *oblivious linear-function evaluation* (OLE) correlations [NP06, IPS09, ADI+17], *multiplication triples* (also known as "Beaver triples") [Bea91, BDOZ11, DPSZ12], and *one-time truth tables* [IKM+13, DNNR17, DKS+17].

The above types of correlated randomness are commonly used to implement efficient MPC protocols in the *preprocessing model*. Such protocols consist of an offline, input-independent *preprocessing phase*, where many independent instances of the correlated randomness are generated, followed by a fast *online phase* that consumes this correlated randomness for the purpose of securely evaluate a given function of the inputs. In many cases, the online phase is "information-theoretic"[6] and its computational complexity is only a small-constant times higher than that of an insecure function evaluation. Most importantly for the present work, the online phase of such protocols typically outperforms all competing approaches in terms of concrete efficiency.

A major challenge in implementing such offline-online protocols is that the preprocessing phase needs to *securely* generate and store a large amount of correlated randomness. This is typically done by using a special-purpose interactive MPC protocol, which involves a significant amount of communication and computation for each gate of a circuit that should be evaluated in the online phase. A dream goal would be to replace this source of correlated randomness with *short* correlated seeds, which can be "silently" expanded *without any interaction* to produce a large amount of *pseudorandom* correlated randomness. This process should emulate an ideal process for generating the target correlation not only from the point of view of outsiders, but also from the point of view of *insiders* who can observe the correlated seeds. We refer to such an object as a *pseudorandom correlation generator*, or PCG for short.

A bit more precisely, a two-party PCG is defined as follows. Let (R_0, R_1) be a target correlation, defined by some efficient sampling algorithm $\mathcal{C}$ that on input 1^λ outputs a pair of correlated strings (r_0, r_1). For instance, $\mathcal{C}(1^\lambda)$ may output $n = \lambda^3$ independent instances of an OT correlation. A PCG is a pair of efficient algorithms $(\mathsf{Gen}, \mathsf{Expand})$ such that:

- Gen samples a pair of short correlated seeds $(\mathsf{k}_0, \mathsf{k}_1) \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$,
- Expand is a *local* deterministic seed expansion algorithm mapping k_i to $r_i \leftarrow \mathsf{Expand}(i, \mathsf{k}_i)$, where $|r_i| > |\mathsf{k}_i|$.

We would like the outputs (r_0, r_1) resulting from this process to be "indistinguishable" from an ideal sample (R_0, R_1) generated by $\mathcal{C}(1^\lambda)$, even to a party who receives one of the seeds k_b.

A useful special case of PCG was recently considered by Boyle et al. [BCGI18], who constructed (under variants of the Learning Parity with Noise assumption [BFKL93]) a concretely efficient PCG for the *vector OLE* (VOLE) correlation. The VOLE correlation over a field $\mathbb{F}$ samples a random scalar $x \in \mathbb{F}$ and vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{F}^n$, and outputs $r_0 = (\boldsymbol{u}, \boldsymbol{v})$ to one party (the "sender") and $r_1 = (x, \boldsymbol{w} = \boldsymbol{u}x + \boldsymbol{v})$ to the other party (the "receiver"). The VOLE correlation is useful for secure computation of functions that employ scalar-vector products over large fields, such as ones arising in the context of linear algebra and keyword search [ADI+17].

Designing efficient PCGs for a wider class of correlations is strongly motivated by the goal of improving the efficiency of *general* MPC in the preprocessing model, where the preprocessing phase is used to securely generate the PCG seeds. We refer to this as *MPC with silent preprocessing*. More concretely, such a protocol consists of three phases: (1) an interactive *setup phase* for securely distributing the seed generation algorithm Gen; in the end of this phase, which involves a small amount of communication, only the short seeds are stored for later use; (2) a silent *seed expansion* phase, where the seeds are expanded into long correlated randomness via a local computation of Expand and without any interaction; (3) a final *online phase* where the correlated randomness is consumed to evaluate a given function of the inputs. One could employ Phase 1 when deciding that an MPC interaction *might* take place in the future, Phase 2 when

[6] This can be formalized by requiring that the joint states of the parties in the end of the offline phase can be swapped by *computationally indistinguishable* states, given which the online protocol is secure against computationally unbounded parties.

interaction seems likely to take place in the near future, and Phase 3 to carry out the MPC interaction once the inputs are available. The low communication footprint of silent preprocessing can eliminate traffic analysis attacks that anticipate future MPC plans. Finally, another benefit of the PCG-based approach is that it can help reduce the cost of protecting MPC protocols against malicious parties. Indeed, since Phase 2 does not involve any interaction, it suffices to protect Phase 1 and Phase 3 against malicious parties, which is typically much cheaper.

Several different kinds of PCG constructions are implicit in the MPC literature. These include PCGs for simple multi-party linear correlations from any pseudorandom generator [GI99, CDI05], for general correlations from indistinguishability obfuscation [HW15, HIJ+16], for so-called "bilinear" correlations from homomorphic secret sharing [BCG+17], for restricted variants of OT correlations from key-homomorphic pseudorandom functions [Sch18] and, most recently, for VOLE correlation from LPN [BCGI18]. With the exception of linear multi-party correlations [GI99, CDI05] and VOLE correlations [BCGI18], none of these prior constructions seem appealing from a practical point of view. In particular, there was no prior approach (even a heuristic one) for constructing a concretely efficient PCG for OT correlations.

1.1 Our Contributions

In this work, we initiate a more systematic study of pseudorandom correlation generators. Our contributions are on both the foundational side, where we present new definitions, impossibility results and connections with other primitives, and the applied side, with concretely efficient constructions for commonly used MPC correlations, including OT correlations and others. Our most practical PCG constructions handle restricted (yet still useful) classes of correlations, while our more general constructions can handle much larger classes of correlations, at the expense of a bigger seed size and higher computational costs (and, for some of them, public-key-style assumptions such as lattice-based or pairing-based cryptography).

We now give a more detailed account of our contributions. Unless noted otherwise, we refer to MPC with computational security against semi-honest (i.e., passive) and static (i.e., non-adaptive) adversaries who may corrupt an arbitrary subset of parties.

1.1.1 Foundations of Pseudorandom Correlation Generators

Our first goal is to present a general security definition for the intuitive notion of PCG described above. As pointed out in [GI99], this is not quite as straightforward as one might imagine, and previous works side-stepped the problem by taking an ad-hoc approach. To motivate our general definition, we start by discussing the most natural alternative.

Ruling Out a Simulation-Based Definition. Recall that the ultimate desire would be that in any protocol, one can securely replace an ideal correlated randomness functionality $\mathcal{C}$ with pseudo-randomness obtained from expanding the correlated seeds of a PCG for $\mathcal{C}$. This would indeed follow from a natural simulation-based security definition for PCG as a computationally secure, dealer-assisted protocol for computing the randomized functionality defined by $\mathcal{C}$. Concretely, in the two-party case, the simulation-based definition requires the existence of a simulator S such that the *real* distribution $(\mathsf{k}_b, \mathsf{Expand}(\mathsf{k}_{1-b}))$, where $(\mathsf{k}_0, \mathsf{k}_1)$ are generated by Gen (capturing the view of a corrupted party b jointly with the output of the uncorrupted party $1-b$) is computationally indistinguishable from the *ideal* distribution $(S(r_b), r_{1-b})$, where (r_0, r_1) are sampled by $\mathcal{C}$. Unfortunately, we show (building on [HW15], and extending an informal argument from [GI99]) that such a definition is impossible to realize even for simple correlations. Intuitively, the impossibility follows from the fact that in the real distribution k_b "explains" the output of the honest party in an efficiently verifiable way, whereas such an explanation of r_{1-b} cannot be generated from r_b in the ideal distribution.

A General PCG Definition. To get around the above impossibility, we present a relaxed indistinguishability-based definition of PCG security, generalizing the specialized security definition for the VOLE correlation from [BCGI18]. Our definition requires that given its PCG key k_b, corrupted party b cannot distinguish the *true* expanded output of the honest party $r_{1-b} = \mathsf{Expand}(1-b, \mathsf{k}_{1-b})$ from a *random* output r_{1-b} consistent with the correlation $\mathcal{C}$ and its own expanded output $r_b = \mathsf{Expand}(b, \mathsf{k}_b)$. In other words, we replace the ideal distribution in the above simulation-based definition by $(\mathsf{k}_b, [r_{1-b} \,|\, R_b = \mathsf{Expand}(\mathsf{k}_b)])$. Note that the latter distribution involves reverse-sampling from R_{1-b} conditioned on a fixed value for R_b, which may not be well-defined. However, in this work we only consider *additive* correlations, where (R_0, R_1) are additive secret shares (over a finite Abelian group) of a sample from some core distribution. For such additive correlations, the reverse-sampling is well-defined and is computationally efficient. More broadly, our general PCG definition is meaningful when this reverse-sampling is efficient.

Limitations. Our PCG definition is not good enough for generating correlated randomness in *all* applications. Indeed, the impossibility of the simulation-based definition discussed above implies such simple counterexamples for randomized functionalities. Concretely, for any $\mathcal{C}$ to which the impossibility result applies, there is a trivial MPC protocol for $\mathcal{C}$ given correlated randomness from $\mathcal{C}$ in which each party outputs its correlated randomness. However, the impossibility result shows that using *any* PCG for $\mathcal{C}$ would render this simple protocol insecure. We show, under standard cryptographic assumptions, that a similar impossibility holds even if one restricts attention to MPC for *deterministic* functionalities. Concretely, we show a protocol which uses correlated randomness $\mathcal{C}$ to realize a deterministic functionality with statistical security against malicious parties, but which becomes completely *insecure* (even against semi-honest parties) when $\mathcal{C}$ is replaced by a specific PCG for $\mathcal{C}$ that meets our indistinguishability-based definition.

A Plug-and-Play Use of PCG. We complement the above negative results by a positive result, showing that our PCG definition does suffice to imply our "ultimate desire" in the context of most applications. Concretely, we put forward a slightly stronger security requirement for MPC with preprocessing, such that in any protocol satisfying this requirement, a PCG can be used as a drop-in replacement for correlated randomness. The stronger security requirement asserts that security still hold even if the ideal correlation functionality (R_0, R_1) is replaced by a *corruptible* functionality that allows corrupted party b to pick its own randomness r_b^*, and then delivers to the uncorrupted party a sample r_{1-b} from the conditional distribution $[r_{1-b} \,|\, R_b = r_b^*]$. It fortunately turns out that natural MPC protocols in the preprocessing model already satisfy this stronger security requirement. This allows for a plug-and-play use of PCGs in many application scenarios.

Relation with Homomorphic Secret Sharing. A (two-party) homomorphic secret sharing (HSS) scheme [BGI16a, BGI+18] for a function class $\mathcal{F}$ splits a secret x into two shares (x_0, x_1), such that given any $f \in \mathcal{F}$ one can efficiently evaluate *additive* shares of $f(x)$ via local computation on the shares. We show a two-way relation between PCG and HSS. First, we show that a PCG for any *additive* correlation (as defined above) can be reduced to HSS for a related function class $\mathcal{F}$, generalizing and formalizing a previous observation from [BCG+17]. In particular, HSS for general circuits implies PCG for all additive correlations, which include most of the useful MPC correlations as special cases. (This is only a feasibility result, which does not directly imply concretely efficient constructions.) Second, we show that some converse is also true: a PCG for the degree-d "tensoring" correlation, obtained by picking a random vector $X \in \mathcal{R}^n$ and outputting additive shares of all products of at most d entries of X, implies HSS for the class $\mathcal{F}$ of degree-d (n-variate) polynomials over $\mathcal{R}^n$, where the share size grows linearly with n and the homomorphic evaluation time grows linearly with n^d.

1.1.2 Silent OT Extension

A central contribution of this work is the first *concretely efficient* construction of PCG for the oblivious transfer (OT) correlation. From an asymptotic point of view, our PCG can achieve an arbitrary polynomial stretch, assuming: (1) The *binary* Learning Parity with Noise (LPN) assumption [BFKL93] with a conservative choice of parameters, and (2) A correlation-robust hash function [IKNP03]. The hash function primitive, which is only used in a black-box way, can be instantiated in practice by a general-purpose hash function or block cipher. Assuming LPN with a linear number of samples and inverse-polynomial noise rate holds for the dual of a near-linear time encodable code (such as the codes proposed in [HKL+12, DI14, ABD+16, ADI+17]), which is still a conservative assumption, the *computational* complexity of Expand is nearly linear in the output length.[7]

In a nutshell, our efficient PCG for OT applies the PCG for VOLE from [BCGI18] over a large extension field $\mathbb{F}_{2^\lambda}$, except for restricting the sender's output $\boldsymbol{u}$ to be over the base field. This yields n correlated instances of random OT that can be converted into standard OT by using a correlation-robust hash function, as in [IKNP03]. See Section 2 for more details.

By applying a secure two-party protocol for distributing Gen, we obtain a *silent OT extension* protocol that generates n pseudo-random OT instances using a small number of OTs, with a total of $O(n^\epsilon)$ bits of communication for any $\epsilon > 0$. This should be compared with existing OT extension protocols [Bea96, IKNP03] that do not require the LPN assumption but where the communication complexity is bigger than n.

Concrete Efficiency. Our LPN-based PCG for OT is very attractive in terms of concrete efficiency, and we expect it to outperform state-of-the-art OT extension protocols [IKNP03, ALSZ13, KK13] in settings where communication is the bottleneck. To give a few data points, our PCG can expand a pair of seeds of length 10KB into a million instances of random 128-bit string-OT, of total size 16MB (receiver) and 32MB (sender), in an estimated[8] time of around a second on a single core of a modern CPU. Alternatively, seeds of length 7KB can be expanded into 65 thousand OTs at roughly half the amortized computational cost. Factoring in the cost of securely distributing Gen (with semi-honest security, building on [Ds17]), the amortized communication complexity of our silent OT extension protocol is 0–3 bits *for each random 128-bit string-OT*. To put that into context, state-of-the-art OT extension protocols [IKNP03, ALSZ13] require 128 bits of communication per random 128-bit string-OT and can generate around 10 million OTs per second [GKWY19] over a fast network, so the price we pay for the (much) lower communication complexity seems quite modest. Even for the easier case of random *bit*-OT, the best previous OT extension protocol [KK13] required roughly 80 bits of communication per OT.

1.1.3 Other PCG Constructions

We present an assortment of practically feasible PCGs for other useful two-party correlations, based on a variety of underlying tools and assumptions.

– **PCG for Constant-Degree Polynomials from LPN.** We show that a generalization of the LPN-based VOLE generator from [BCGI18] can be used to obtain a PCG for any constant-degree additive correlation, namely a correlation that additively secret-shares a vector of degree-d polynomials of a random $X \in \mathbb{F}^n$ for some constant $d \geq 2$. This PCG relies on LPN over $\mathbb{F}$ in a similar noise regime as the PCG for OT from Section 1.1.2. In fact, by increasing the computation time (but still keeping it polynomial), one can use the LPN

[7] In Section 1.1.3 below we describe an alternative LPN-based approach to constructing PCG for OT that dispenses with assumption (2), but requires at least quadratic computation in the output length n.

[8] We caution that we have not implemented our constructions. Our estimates are based on counting basic operations and estimating their cost; the actual running times may vary due to other costs we neglected such as cache misses. We leave the task of optimizing and implementing our constructions to future work.

assumption in a parameter regime that is not known to imply public-key encryption [Ale03], let alone OT. The main caveat is that even for generating simple degree-d correlations, such as $\Omega(n)$ Beaver triples ($d = 2$), the *computational* complexity of Expand is bigger than n^d. While much slower than our PCG for OT, this construction may still be practically feasible for $d = 2$ even with reasonably large n. We leave the question of obtaining more efficient variants of this construction to future work.

As discussed in Section 1.1.1, this PCG construction implies (2-party) HSS schemes for constant-degree polynomials from LPN. By additionally assuming a standard OT protocol, it implies secure two-party computation protocols for constant-degree polynomials in which the communication complexity is nearly linear in the input size. Using the techniques from [BGI16a, Cou19], it also implies an "almost-sublinear" general secure computation protocol: for any constant $c > 1$ and layered boolean circuit of size s (and assuming binary LPN and OT), there is a secure two-party computation protocol with polynomial computation and total communication bounded by s/c. We stress again that these are mainly feasibility results because of the high computational cost of this PCG construction.

– **PCG for One-Time Truth Tables from any PRG.** One-time truth tables (OTTT) are a type of correlation that allow secure evaluation of a public lookup table in MPC, on a secret-shared input [IKM+13, DNNR17, DKS+17], and are well-suited to computations such as the S-box of AES. For MPC with active security, the correlation outputs need to be authenticated with information-theoretic MACs, as in the recent TinyTable protocol [DNNR17]. We present a very simple PCG for authenticated OTTT using only a distributed point function (DPF) [GI14, BGI16b], which in turn can be efficiently constructed from any pseudorandom generator (PRG). This PCG follows naturally from a building block of the silent OT extension construction (as we explain in Section 2). It compresses the storage cost of an authenticated OTTT from $O(\lambda n)$ bits down to $O(\lambda \log n)$ bits, for a table of size n, giving a reduction in size of over 20x for a length-256 table such as the AES S-box. There is a concretely efficient protocol to distribute the seed generator Gen with semi-honest security by using the distributed DPF key generation protocol from [Ds17]. While a similar protocol with malicious security is considerably more expensive, even a naive approach based on general-purpose secure computation (e.g., using recent protocols such as [KRRW18]) is feasible in practice, enabling the compressed storage benefit of the PCG-based approach.

– **PCGs from Homomorphic Secret Sharing.** We give practically feasible PCG constructions for OLE and (authenticated) Beaver triple correlations, which are useful for arithmetic MPC protocols such as SPDZ [DPSZ12]. For these constructions we use HSS based on ring-LWE [BGV12, DHRW16, BKS19] and the BGN (pairing-based) cryptosystem [BGN05, BGI16a, BCG+17]. To expand the seeds, we rely on a multivariate quadratic (MQ) assumption based PRG, which limits the stretch to sub-quadratic, but allows for reasonable computational efficiency. For example, with our ring-LWE-based PCG we estimate that one should be able to expand a pair of 3GB seeds into 17GB of authenticated Beaver triples in a 128-bit field, at a rate of around 6 thousand triples per second; various tradeoffs are possible between seed size and computation time, and we also explore an iterative variant which produces triples in small batches. Securely computing Gen to distribute the seeds is relatively cheap compared to the expansion phase, and the overall performance should be comparable to recent work on actively secure triple generation with much more interaction [KPR18]. With BGN, we estimate around 200ms for computing an OLE correlation over $\mathbb{Z}_N$ for small N (say, $N < 10$). Although much more expensive than our silent OT extension, an advantage of the ring-LWE-based constructions, beyond the richer class of correlations, is that they can be extended to the *multi-party* setting, as we discuss next.

1.1.4 PCGs for Multi-Party Correlations

Finally, we present a general transformation for extending certain classes of PCGs from the 2-party to the multi-party setting. This can be applied to PCGs for simple bilinear correlations, including VOLE and Beaver triples, giving the first non-trivial, efficient PCG constructions in the multi-party setting. The transformation applies to most of our 2-party PCGs, including the LPN-based PCG for constant-degree correlations.

On top of the silent preprocessing feature, an appealing application of our multi-party PCGs is in obtaining secure M-party computation protocols with *total* communication complexity $O(Ms + M^2 \cdot s^\epsilon)$ (for circuit size s and constant $0 < \epsilon < 1$). The $O(Ms)$ term is the cost of the (information-theoretic) online phase, and the $O(M^2 \cdot s^\epsilon)$ term is the cost of distributing the PCG seed generation, which is the only part of the protocol requiring pairwise communication. This should be contrasted with OT-based MPC protocols, which have total communication complexity $\Omega(M^2 s)$ [GMW87, HOSS18] . Protocols with such communication complexity (without the silent preprocessing feature) could previously be based on different flavors of somewhat homomorphic encryption [FH96, CDN01, DPSZ12]. We get the first such protocol that only relies on LPN and OT, and the first practically feasible protocol that has sublinear-communication offline phase and information-theoretic online phase.

Table 1. Summary of the New PCG Constructions

PCG	Section 5	Section B	Section 6	Section 4.4	Sections 7.3,C,D,E	Sections 7.4,F
Assumption	LPN	PRG*	LPN	deg-d HSS + MQ/LPN	SXDH + LPN	LWE + MQ
Correlations	OT*	OTTT*	deg-d	degree-$d/2$	small-ring deg-2	deg-d
Efficiency	1M OT/s†	-	-	-	5 OLE/s‡	7000 ABT/s**
Multiparty (bilinear corr.)	✗	✗	✔	✔	✗	✔

* PRG stands for an arbitrary pseudorandom generator, OT for random oblivious transfer, and OTTT for authenticated one-time truth-table correlation.

† Estimated (approximate) cost over one core of a standard laptop, with average communication of 2.6 bits/OT. See Tables 3 and 4.

‡ Estimated (approximate) cost over one core of a standard laptop, for OLE correlation over a small (constant size) ring. See Section E.3, and Tables 9 and 10.

** ABT stands for authenticated Beaver triple. Estimated (approximative) cost over one core of a standard laptop. See Table 2.

1.1.5 Additional Applications

From our silent OT extension protocol, we obtain the following additional results:

- *Oblivious Pseudorandom Functions (OPRFs).* An OPRF [FIPR05] is a two-party protocol for securely evaluating a pseudorandom function, whose key is known by one party, on a secret input known by the second party. OPRFs serve as the main building blocks in recent protocols for private set intersection [KKRT16]. Our silent OT construction can be used to obtain a form of batch OPRF with cost as little as 1 bit of communication per OPRF evaluation on a random input, leading to around a factor two reduction in communication for these protocols.
- *Reusable-Preprocessing NIZK.* Consider the following setting for non-interactive zero knowledge (NIZK) with reusable interactive setup: In an offline setup phase, before the statements to be proved are known, the prover and the verifier interact to securely generate correlated random seeds. The seeds can then be used to prove any polynomial number of statements by

having the prover send a single message to the verifier for each statement. Such a notion was recently constructed in [BCGI18], building on [CDI+18], using their PCG for VOLE. Our silent OT extension can be used to obtain an improved reusable-preprocessing NIZK system for NP, under the standard LPN assumption over $\mathbb{F}_2$. As compared to the reusable NIZK of [BCGI18], our NIZK relies on a more standard assumption (LPN over $\mathbb{F}_2$ versus large $\mathbb{F}$), and the setup cost is independent of both the number of statements and their size (whereas in [BCGI18], the setup cost was independent of the number of statements, but grows linearly with a bound on their size). On the down side, our OT-based NIZK protocols do not have the computational complexity advantages of the VOLE-based constructions from [BCGI18].

– *Efficient Secure Matrix Multiplication.* As a stepping stone towards silent OT extension, we construct a PCG for a generalization of VOLE called *subfield VOLE*. This can be seen as a form of batch VOLE where the $\boldsymbol{u}$ value is reused across several instances, and can be applied to compute secret-shared tensor products and matrix multiplication more efficiently. Compared with naively using a PCG for standard VOLE, we reduce the seed size by at least a $O(\log n)$ factor.

Finally, our PCG for OTTT yields the following application.

– *Improved 2-PC with Sublinear Online Communication.* Standard approaches to secure computation with preprocessing (e.g., SPDZ) still require online communication that is linear in the circuit size. Recently, Couteau [Cou19] demonstrated asymptotic feasibility of information-theoretic secure 2-party computation (2-PC) in the preprocessing model for a natural class of circuits (namely, "layered" circuits), with sublinear online communication, $O(s/\log\log s)$ for circuit size s. However, this comes at the cost of generating and storing $O(s^2)$ bits of correlated randomness.
Our compressed one-time truth-table (OTTT) construction allows one to match the asymptotic complexity of [Cou19], while reducing the amount of correlated randomness from quadratic to quasilinear in the circuit size, in exchange for settling for computational security and assuming the existence of one-way functions.

1.2 Paper Organization

We begin in Section 2 with an overview of our techniques, followed by preliminaries in Section 3. In Section 4, we present our PCG definition and foundational results. Section 5 contains our silent OT extension construction, and applications to OPRFs and NIZK from LPN. Section 6 provides an LPN-based construction of PCG for general constant-degree correlations. We then present generic constructions of PCGs from specific classes of PRGs (Section 4.4); we instantiate this framework for more complex correlations based on group-based and lattice-based HSS, in Section 7.3 and Section 7.4, respectively. Finally, in Section 8 we construct general, multi-party PCGs for simple bilinear correlations based on any "programmable" 2-party PCG.

2 Technical Overview of Constructions

In this section we give a high-level overview of the techniques that underly our different PCG constructions.

2.1 Background

Our PCG constructions rely on different types of *homomorphic secret sharing* (HSS) and *function secret sharing* (FSS) schemes. Informally, HSS is a form of secret sharing that allows a secret x to be split up into shares $\mathsf{k}_0, \mathsf{k}_1$, such that a party holding k_i can *locally* obtain an additive secret share of $f(x)$, for some function f. FSS is the dual notion: starting with a function f, and

splitting into shares f_0, f_1 such that each share f_i hides f, but can be used to obtain an additive sharing of $f(x)$ for some public input x.[9] FSS for a class of point functions (i.e., functions f which evaluate to 0 on all but a single input) is called a *distributed point function* [GI14], and can be constructed very efficiently based on a pseudorandom generator (PRG) [BGI16b]. There are HSS constructions for branching programs based on DDH [BGI16a] or lattices [BKS19], or general circuits from strong forms of fully homomorphic encryption [DHRW16].

2.2 Overall methodology

At a high level, our constructions can all be seen as examples of the following blueprint: construct an HSS scheme that can homomorphically evaluate the composition of a pseudorandom generator (PRG) with a function f that uses the expanded randomness to compute the desired correlation. This can be used to obtain PCGs for any *additive* correlation; i.e., that outputs random additive shares of some distribution. Of course, the main challenge lies in instantiating this *efficiently*, since plugging in even a low-degree PRG to an off-the-shelf HSS scheme is typically not practical. We instead use specialized HSS constructions that pair well with our carefully chosen PRGs.

As a stepping stone, our constructions implicitly construct a *compressible* form of HSS, which allows the sharing of inputs from some distribution $\mathcal{D}$, such that the share size is *smaller than* an uncompressed output of $\mathcal{D}$, and we can still compute some useful function f on the expanded inputs. We typically choose $\mathcal{D}$ to be a sparse distribution on vectors, or another similarly compressible distribution. We then convert these long $\mathcal{D}$-vectors to slightly shorter (but still long) *random*-looking vectors, by homomorphically multiplying by a compressive linear map. Under a suitable LPN-type assumption, this combination of expanding the compressed $\mathcal{D}$-vector followed by linear compression acts as a PRG in the above blueprint, and we can proceed to homomorphically compute the desired correlation.

For example, when $\mathcal{D}$ samples a sparse, low-weight vector $\boldsymbol{e}$ over $\mathbb{F}_2$, and the linear map is a random matrix H, then distinguishing $(H, \boldsymbol{e} \cdot H)$ from random is as hard as the problem of decoding a random binary linear code, which corresponds to the standard LPN assumption [BFKL93, Ale03]. Another example is when $\mathcal{D}$ outputs a *tensor product* of two short, uniform vectors. Recovering the short vectors given only $(\boldsymbol{x} \otimes \boldsymbol{y}) \cdot H$ is the problem of solving a random system of multivariate quadratic equations (MQ problem), which is believed to be hard for a suitable choice of parameters [MI88, Wol05, BGP06, AHI+17]. In particular, the decision version of MQ is polynomially reducible to its search version [BGP06].

We remark that the resulting PRGs do not necessarily conform to standard metrics of simplicity, such as low degree or low locality, and in isolation may appear somewhat unnatural. This exemplifies an interesting observation that "HSS-friendliness" may indeed be a new type of metric that does not directly align with those previously studied.

2.3 Silent OT Extension

As a building block for silent OT extension, we start by constructing a PCG for a two-party correlation we call *subfield vector oblivious linear evaluation* (subfield VOLE). This correlation works over a field $\mathbb{F}_q$, and a subfield $\mathbb{F}_p$, where $q = p^r$. It first samples a random $x \in \mathbb{F}_q$, $\boldsymbol{u} \in \mathbb{F}_p^n, \boldsymbol{v} \in \mathbb{F}_q^n$, then outputs $(\boldsymbol{u}, \boldsymbol{v})$ to the sender and $(x, \boldsymbol{w} = \boldsymbol{u}x + \boldsymbol{v})$ to the receiver.[10] Our construction is a generalization of the vector-OLE construction from [BCGI18]: when $p = q$ the correlation is exactly vector-OLE, but using $q > p$ opens up additional applications. For example, viewing $x \in \mathbb{F}_q$ as a vector $\boldsymbol{x} \in \mathbb{F}_p^r$, subfield VOLE can be seen as computing additive shares of the $r \times n$ tensor product $\boldsymbol{x} \otimes \boldsymbol{u}$, which can be useful for secure two-party matrix multiplication,

[9] FSS is actually equivalent to HSS for a related class of functions, but we differentiate between the two for convenience, depending on the applications.

[10] We view elements of $\mathbb{F}_p$ embedded into $\mathbb{F}_q$ throughout, so that the multiplication $\boldsymbol{u} \cdot x$ happens over $\mathbb{F}_q$.

and other linear algebra tasks. Compared with using r copies of VOLE [BCGI18] to achieve the same task, we reduce the seed size by a $O(\log n)$ factor and obtain more efficient computation.

To build a PCG for subfield VOLE, we consider a compressible distribution $\mathcal{D}$ that outputs random sparse vectors of weight t and length n'. First, notice that we can compress a secret-sharing of the j-th unit vector $\boldsymbol{e}_j \in \{0,1\}^{n'}$, using a distributed point function (DPF) for the point $(j, 1)$: evaluating a DPF key on input i produces a random share of 0 on all inputs except $i = j$, where it outputs a share of 1. Hence, performing all n' evaluations results in shares of the entire vector $\boldsymbol{e}_j$. This easily extends to weight t vectors, by naively using t DPFs and summing up the shares of the t unit vectors (this step can be optimized with a multi-point DPF as described in [BCGI18]).

Although it may appear that this only allows us to compress sparse vectors, and not perform any useful HSS computations afterwards, we observe that with a small tweak we can use this to build HSS for the family of randomized functions

$$\mathcal{F} = \{f_H : \mathbb{F}_q \to \mathbb{F}_q^n, x \mapsto x \cdot \boldsymbol{e} \cdot H \mid \boldsymbol{e} \xleftarrow{\$} \mathcal{HW}_t, H \in \mathbb{F}_p^{n' \times n}\} \tag{1}$$

where $\mathcal{HW}_t$ is the distribution that outputs a random weight-t vector over $\mathbb{F}_p^{n'}$ (with each entry either 0 or uniform). We remark that a naive description of the class $\mathcal{F}$ gives functions with very high degree, which could *not* be evaluated using simple HSS schemes, which highlights the importance of tailoring a specific solution.

To upgrade the above sketch to get HSS for $\mathcal{F}$, we make one small modification: using t DPFs that output shares over $\mathbb{F}_q$, we specify the i-th DPF by the point $(j_i, y_i \cdot x)$ for some random index j_i and $y_i \in \mathbb{F}_p^*$, instead of $(j_i, 1)$ as before. When evaluating the DPFs, the parties now obtain additive shares of $\boldsymbol{e} \cdot x$, where $\boldsymbol{e}$ contains all t y_i's in random positions. Since additive secret sharing is linear, any linear map H can then be locally applied on the shares.

If $H \in \mathbb{F}_p^{n' \times n}$ is a compressive linear map with $n < n'$, the vector $\boldsymbol{u} = \boldsymbol{e} \cdot H$ is pseudorandom under a suitable form of the LPN (or syndrome decoding) assumption. Concretely, we require that a t-noisy random codeword in the code whose *parity check* matrix is H is pseudorandom. This immediately yields a subfield VOLE generator, where each party's seed contains a set of DPF seeds, and the sender additionally gets the points (j_i, y_i), and the receiver gets x, since additive shares of $x \cdot \boldsymbol{u}$ can be locally converted to the $(\boldsymbol{v}, \boldsymbol{w})$ components of a VOLE correlation.

Our next observation, inspired by the OT extension protocol of Ishai et al. [IKNP03], is that subfield VOLE already gives as a restricted form of oblivious transfer, known as correlated OT or Δ-OT. If we run subfield VOLE over $\mathbb{F}_2$, embedded in $\mathbb{F}_{2^r}$, then the VOLE sender obtains a set of pairs $u_i \xleftarrow{\$} \mathbb{F}_2, v_i \xleftarrow{\$} \mathbb{F}_{2^r}$, while the VOLE receiver gets $x \xleftarrow{\$} \mathbb{F}_{2^r}$ and $w_i = x \cdot u_i + v_i$, for $i = 1, \ldots, n$. Now switch the roles of sender and receiver, so the VOLE sender becomes an OT receiver with choice bit u_i and string v_i. If $u_i = 0$ then $v_i = w_i$, whilst if $u_i = 1$ then $v_i = w_i - x$, hence, this is exactly a 1-out-of-2 OT where the OT sender's (formerly VOLE receiver's) messages are all of the form $(w_i, w_i - x)$.

On its own, this type of Δ-OT is already useful for many applications such as garbled circuits and secure computation with information-theoretic MACs [WRK17a, NNOB12]. However, most importantly, following [IKNP03], the parties can locally convert such a correlated OT into an OT on random strings, using a hash function that is pseudorandom under correlated inputs. This gives us a PCG for random oblivious transfer, where the seed size is essentially that of t distributed point functions, or $O(t\lambda \log n)$ bits. Combining this with an efficient secure protocol for setting up a pair of DPF keys [Ds17], we obtain our silent OT extension protocol, which produces n pseudorandom string-OTs with $o(n)$ bits of communication.

2.4 One-Time Truth Tables

We next show how to adapt the above approach to produce authenticated, one-time truth table correlations, which can be used to efficiently perform table lookups in MPC [IKM+13, DNNR17,

DKS+17]. This construction is straightforward given the above description of our subfield-VOLE generator, so we informally explain it here and defer the complete description to Section B of the Appendix.

The correlation we want to produce, for a lookup table $T : [n] \to \{0,1\}^m$, is an additive secret-sharing of

$$\left(\alpha, \{y_i, \gamma_i\}_{i\in[n]}\right) \quad \text{where} \quad y_i = T(s + i \bmod n), \gamma_i = y_i \cdot \alpha \in \mathbb{F}_{2^\lambda} \tag{2}$$

for $\alpha \xleftarrow{\$} \mathbb{F}_{2^\lambda}, s \xleftarrow{\$} [n]$. Here, the y_i's are equal to T shifted by a random offset s, while the γ_i's are information-theoretic MACs on y_i under the key α, used to obtain active security in the MPC protocol.

Our starting point is the observation from [KOR+17] that the y_i's can be generated locally, given secret-shares of a random unit vector. This is because, if $\boldsymbol{e}_s \in \{0,1\}^n$ is the s-th unit vector, then we have

$$T(s + i \bmod n) = \sum_{j=1}^{n} \boldsymbol{e}_s[j] \cdot T(i + j \bmod n)$$

which is linear in $\boldsymbol{e}_s$. We can further obtain the γ_i's (namely, the authenticated $\gamma_i = y_i \cdot \alpha$) if we additionally have secret-shares of the corresponding scaled vector $\alpha \cdot \boldsymbol{e}_s$.

The core observation is that a DPF gives precisely a *compressed* secret sharing of such a secret vector $(1||\alpha) \cdot \boldsymbol{e}_s \in (\{0,1\}^{1+\lambda})^n$: requiring only $O(\lambda \log n)$ bits in the place of $O(\lambda n)$.

More concretely, this leads to the following, simple approach for a PCG to generate shares of (2): use the previous DPF-based construction of HSS for the family in (1) over $\mathbb{F}_2$, with $t = 1$, $x = (1\|\alpha)$ for $\alpha \xleftarrow{\$} \mathbb{F}_{2^\lambda}$, and H the linear map induced by T in the equation above. The resulting PCG has seed size essentially the same as one DPF, which is $O(\lambda \log n)$ bits. This gives a large compression over the previous, practical approach from [DNNR17], which required $O(\lambda n)$ bits per table. Expanding the PCG is relatively cheap in practice, since in 2-PC applications only a single entry of each table is ever used, and this can be computed on-the-fly with $O(n)$ PRG evaluations.

A downside of this construction is that it seems difficult to produce the necessary PCG seeds with good concrete efficiency in the malicious setting, since the only known approach in this setting requires evaluating a PRG inside 2-PC [Ds17]. However, our result is still interesting for a preprocessing phase with semi-honest security, or when a trusted dealer is present. Alternatively, if one can afford the cost of distributing Gen with malicious security via general-purpose 2-PC, the resulting correlated seeds only require a small amount of storage, and their local expansion is (automatically) secure against malicious parties.

2.5 PCGs for Constant-Degree Polynomials from LPN

We construct PCGs for constant-degree polynomials, using again function secret sharing for multi-point functions together with LPN. At a high level, the construction builds upon the fact that given two sparse vectors $\boldsymbol{a}, \boldsymbol{b}$, their tensor product $\boldsymbol{a} \otimes \boldsymbol{b}$ is sparse as well, hence shares of $\boldsymbol{a} \otimes \boldsymbol{b}$ can be compressed using an FSS, as for vector-OLE generators and silent OT extension. Then, a compressive mapping can be applied to obtain $\boldsymbol{x} \otimes \boldsymbol{y}$ from $\boldsymbol{a} \otimes \boldsymbol{b}$, where $\boldsymbol{x} = (\boldsymbol{a} \cdot H)$ and $\boldsymbol{y} = (\boldsymbol{b} \cdot H)$ are *pseudorandom* under the LPN assumption, thanks to the bilinearity of the tensor product (and linearity of H). This immediately leads to a PCG for bilinear functions, which can be easily generalized to a PCG for constant-degree polynomials. However, the share size grows as $O(t^d)$, where t is the number of noisy coordinates in the LPN instance, and d is the degree of the polynomial. The computation cost grows as $O(n^{2d})$, where n is the input size.

2.6 PCGs from Ring-LWE and BGN-based HSS

We construct PCGs for more general two-party correlations, building upon the specific structure of homomorphic encryption-based HSS schemes [DHRW16, BKS19] and group-based HSS schemes [BGI16a, BGI17, BCG+17]. Our key observation is that in both HSS schemes encodings of large pseudorandom strings can be compressed efficiently using an "HSS-friendly PRG" as described in Section 2.2. For the ring-LWE based construction, we obtain compression with a PRG based on the multivariate quadratic equations problem, and present several ways of optimizing this with batching techniques for homomorphic encryption, which lead to different tradeoffs for seed size and computational cost.

The group-based approach requires more involved techniques: The underlying HSS scheme uses two types of encodings, where so-called level-1 encodings are ElGamal ciphertexts, and level-2 encodings are shares of $\mathsf{sk} \cdot \boldsymbol{x}$ for a vector $\boldsymbol{x}$, where sk is the secret key of the homomorphic encryption scheme. Then, a special HSS operation allows to compute level-2 encodings of bilinear functions applied to a level-1 encoding and a level-2 encoding. Using two parallel instances of the PCG for vector-OLE of [BCGI18] allows us to efficiently compress shares of $\boldsymbol{y}$ and $\mathsf{sk} \cdot \boldsymbol{y}$, where $\boldsymbol{y}$ is a pseudorandom vector and sk is a shared value, to only $O(\lambda t \log n)$ bits, under the LPN assumption with t noisy coordinates. Furthermore, encrypting short random sparse vectors suffices for homomorphically evaluating a specific LPN-based PRG directly on the level-1 encodings, as long as they support evaluation of degree-2 functions. This can be ensured by using BGN-style pairing-based encryption for the group-based HSS. Since the HSS comes with an inverse-polynomial error probability, we further develop a new method to efficiently remove the faulty outputs, building upon our silent OT extension protocol.

For both schemes, we discuss various optimizations and provide detailed efficiency estimations.

2.7 Multi-Party PCGs

As our final contribution, we construct *multi-party* PCGs for a useful class of bilinear correlations. Concretely, for a given bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, we consider M-party correlations of the form $\{(a_i, b_i, c_i)\}_{i\in[M]}$, consisting of additive secret shares of random elements $a \in \mathbb{G}_1$, $b \in \mathbb{G}_2$, and their image $c = e(a, b) \in \mathbb{G}_T$. For appropriate choice of groups and bilinear operation, this captures M-party OT, M-party vector OLE, M-party Beaver triples, and more.

Our construction approach provides a semi-generic transformation from any PCG for a corresponding *2-party* correlation $\{(a, c_1), (b, c_2)\}$ for random a, b, and $c_1 + c_2 = e(a, b)$, if the PCG satisfies an additional programmability property. Roughly, this property requires a way of "reusing" the inputs a and b across instances without compromising security.

The M-party construction leverages this structure by executing $M(M-1)$ pairwise instances of the underlying 2-party PCG, for all the "cross-terms." Namely, we think of each a_i and b_i from the final M-party correlation as playing the role of a or b in the 2-party correlation, with all possible partners. The desired M-party additive shares c_i can then be derived by combining $c_{ii} = a_i b_i$ (computable locally) together with $\{c_{ij}, c_{ji}\}_{j\in[M]\setminus\{i\}}$ resulting from the 2-party correlations for pairs (a_i, b_j) and (a_j, b_i). The resulting M-party PCG keys consist of $M(M-1)$ keys from the 2-party PCG, together with short expandable shares of 0 for rerandomization.

We observe that the necessary programmability property is satisfied by our subfield VOLE construction and the 2-party VOLE PCG from [BCGI18], as well as the 2-party bilinear PCGs constructed in this work (including OT and Beaver triples) from group-based and lattice-based HSS (Section 7) and from LPN (Section 6). As a corollary, we obtain M-party variants of these correlations with quadratic blowup in computation and share size. Interestingly, our silent OT extension construction does *not* seem to support the necessary programmability, since the resulting sender message pairs are implicitly defined as a function of the receiver's bit selections.

3 Preliminaries

We say that a function $\mathsf{negl}\colon \mathbb{N} \to \mathbb{R}^+$ is *negligible* if it vanishes faster than every inverse polynomial. For two families of distributions $X = \{X_\lambda\}$ and $Y = \{Y_\lambda\}$ indexed by a security parameter $\lambda \in \mathbb{N}$, we write $X \overset{\mathrm{c}}{\approx} Y$ if X and Y are *computationally indistinguishable* (namely, any family of circuits of size $\mathsf{poly}(\lambda)$ has a negligible distinguishing advantage), $X \overset{\mathrm{s}}{\approx} Y$ if they are *statistically indistinguishable* (namely, the above holds for arbitrary distinguishers), and $X \equiv Y$ if the two families are identically distributed.

Notation. We usually denote matrices with capital letters (A, B, C) and vectors with bold lowercase $(\boldsymbol{x}, \boldsymbol{y})$. By default, vectors are assumed to be row vectors. We write $A|_{i,j}$ to denote the entry (i, j) of a matrix A. Given a vector $\boldsymbol{x}$ of length $|\boldsymbol{x}| = n$, the notation $\mathsf{HW}(x)$ denotes the Hamming weight $\boldsymbol{x}$, *i.e.*, the number of its nonzero entries. Given a distribution $\mathcal{D}$, we denote by $\mathsf{Im}(\mathcal{D})$ the image of $\mathcal{D}$ (i.e., its support set).

3.1 Function Secret Sharing

Informally, an FSS scheme for a class of functions $\mathcal{C}$ is a pair of algorithms $\mathsf{FSS} = (\mathsf{FSS.Gen}, \mathsf{FSS.Eval})$ such that:

- $\mathsf{FSS.Gen}$ given a function $f \in \mathcal{C}$ outputs a pair of keys (K_0, K_1);,
- $\mathsf{FSS.Eval}$, given K_b and input x, outputs y_b such that y_0 and y_1 form additive shares of $f(x)$.

The security requirement is that each key K_b computationally hide f, except for revealing the input and output domains of f. We formalize this below.

Definition 1 (Function Secret Sharing; adapted from [BGI16b]). *A 2-party* function secret sharing *(FSS) scheme for a class of functions $\mathcal{C} = \{f : I \to \mathbb{G}\}$ with input domain I and output domain an abelian group $(\mathbb{G}, +)$, is a pair of PPT algorithms $\mathsf{FSS} = (\mathsf{FSS.Gen}, \mathsf{FSS.Eval})$ with the following syntax:*

- $\mathsf{FSS.Gen}(1^\lambda, f)$*, given security parameter λ and description of a function $f \in \mathcal{C}$, outputs a pair of keys (K_0, K_1);*
- $\mathsf{FSS.Eval}(b, K_b, x)$*, given party index $b \in \{0, 1\}$, key K_b, and input $x \in I$, outputs a group element $y_b \in \mathbb{G}$.*

Given an allowable leakage function $\mathsf{Leak} : \{0,1\}^* \to \{0,1\}^*$, the scheme FSS should satisfy the following requirements:

- **Correctness:** *For any $f : I \to \mathbb{G}$ 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.*

Unless otherwise specified, we assume that for $f : I \to \mathbb{G}$, the allowable leakage $\mathsf{Leak}(f)$ outputs $(I, \mathbb{G})$, namely a description of the input and output domains of f.

Remark 2. In any FSS scheme for a sufficiently rich class of functions (including point functions), each of the two evaluation functions $F^b_K(x) = \mathsf{FSS.Eval}(b, K, x)$ is a pseudorandom function [BGI15]. Some of our constructions will use this property.

Some applications of FSS require applying the evaluation algorithm on *all inputs*. Following [BGI16b, BCGI18], given an FSS scheme $(\mathsf{FSS.Gen}, \mathsf{FSS.Eval})$, we denote by $\mathsf{FSS.FullEval}$ an algorithm which, on input a bit b, and an evaluation key K_b (which defines the input domain I), outputs a list of $|I|$ elements of $\mathbb{G}$ corresponding to the evaluation of $\mathsf{FSS.Eval}(b, K_b, \cdot)$ on every input $x \in I$ (in some predetermined order). While $\mathsf{FSS.FullEval}$ can always be realized with $|I|$ invocations of $\mathsf{FSS.Eval}$, it is typically possible to obtain a more efficient construction. Below, we recall some results from [BGI16b] on FSS schemes for useful classes of functions.

3.1.1 Distributed Point Functions

A distributed point function (DPF) [GI14] is an FSS scheme for the class of point functions $f_{\alpha,\beta} : \{0,1\}^\ell \to \mathbb{G}$ which satisfy $f_{\alpha,\beta}(\alpha) = \beta$, and $f_{\alpha,\beta}(x) = 0$ for any $x \neq \alpha$. A sequence of works [GI14, BGI15, BGI16b] has led to highly efficient constructions of DPF schemes from any pseudorandom generator (PRG), which can be implemented in practice using block ciphers such as AES.

Theorem 3 (PRG-based DPF [BGI16b], Theorems 3.3 and 3.4). *Given a PRG $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$, there exists a DPF for point functions $f_{\alpha,\beta} : \{0,1\}^\ell \to \mathbb{G}$ with key size $\ell \cdot (\lambda+2) + \lambda + \lceil \log_2 |\mathbb{G}| \rceil$ bits. For $m = \lceil \frac{\log |\mathbb{G}|}{\lambda+2} \rceil$, the key generation algorithm* Gen *invokes G at most $2(\ell+m)$ times, the evaluation algorithm* Eval *invokes G at most $\ell+m$ times, and the full evaluation algorithm* FullEval *invokes G at most $2^\ell(1+m)$ times.*

Note that a naive construction of FullEval from Eval would require $2^\ell(\ell+m)$ invocations of G.

3.1.2 FSS for Multi-Point Functions

Similarly to [BCGI18], we use FSS for *multi-point functions*. A k-point function evaluates to 0 everywhere, except on k specified points. When specifying multi-point functions we often view the domain of the function as $[n]$ for $n = 2^\ell$ instead of $\{0,1\}^\ell$.

Definition 4 (Multi-Point Function [BCGI18]). *An (n,t)-multi-point function over an abelian group $(\mathbb{G},+)$ is a function $f_{S,\boldsymbol{y}} : [n] \to \mathbb{G}$, where $S = (s_1, \cdots, s_t)$ is an ordered subset of $[n]$ of size t and $\boldsymbol{y} = (y_1, \cdots, y_t) \in \mathbb{G}^t$, defined by $f_{S,\boldsymbol{y}}(s_i) = y_i$ for any $i \in [t]$, and $f_{S,y}(x) = 0$ for any $x \in [n] \setminus S$.*

We assume that the description of S includes the input domain $[n]$ so that $f_{S,\boldsymbol{y}}$ is fully specified.

A *Multi-Point Function Secret Sharing* (MPFSS) is an FSS scheme for the class of multi-point functions, where a point function $f_{S,\boldsymbol{y}}$ is represented in a natural way. We assume that an MPFSS scheme leaks not only the input and output domains but also the number of points t that the multi-point function specifies. An MPFSS can be easily obtained by adding t instances of DPF; optimized constructions of MPFSS, using batch codes [IKOS04] to speed up the full domain evaluation algorithm, were presented in [BCGI18].

3.2 Homomorphic Secret Sharing

We consider homomorphic secret sharing (HSS), a dual form of FSS introduced in [BGI16a]. HSS can be viewed as the natural secret-sharing analogue of fully homomorphic encryption. In this work, we consider a *secret-key* variant of HSS in which a common secret key is used to share multiple inputs, and the output is shared additively over an Abelian group. Furthermore, we will be mainly interested in HSS schemes that support the evaluation of *low-degree* multivariate polynomials on shared input vectors. Informally, a degree-d HSS is a triple of algorithms $(\mathsf{Gen}, \mathsf{Share}, \mathsf{Eval})$ such that:

- Gen generates a secret key sk and an evaluation key ek,
- Share uses the secret key to share an input vector into (s_0, s_1), and
- Eval, given share s_b, evaluation key ek, and a description of a function f of algebraic degree d, outputs y_b such that $y_0 + y_1 = f(x)$.

Security states that a single share s_b together with ek computationally hide the input x.

More formally, we consider degree-d HSS over a finite ring $\mathcal{R}$. In this work we will consider rings $\mathcal{R}$ that are either finite fields or rings $\mathbb{Z}_m$ of integers modulo m. We view the ring as being

implicitly defined by the security parameter λ, and assume that the bit-length of ring elements is at most polynomial in λ.

Definition 5 (Degree-d Homomorphic Secret Sharing). *A (2-party, secret-key)* Degree-d Homomorphic Secret Sharing (HSS) *scheme over a ring $(\mathcal{R} = \mathcal{R}(\lambda), +, \cdot)$ is a triple of PPT algorithms $\mathsf{HSS} = (\mathsf{HSS.Gen}, \mathsf{HSS.Share}, \mathsf{HSS.Eval})$ with the following syntax:*

- $\mathsf{HSS.Gen}(1^\lambda)$*: On input a security parameter 1^λ, the key generation algorithm outputs a secret key sk and an evaluation key ek.*
- $\mathsf{HSS.Share}(\mathsf{sk}, x)$*: Given secret key sk and secret input value $x \in \mathcal{R}^n$, the sharing algorithm outputs a pair of shares (s_0, s_1). We assume that a description of the ring $\mathcal{R}$ and the input length n are included in each of (s_0, s_1).*
- $\mathsf{HSS.Eval}(b, \mathsf{ek}, s_b, P)$*: On input party index $b \in \{0, 1\}$, evaluation key ek, share s_b of an input vector $x \in \mathcal{R}^n$, and degree-d arithmetic circuit P over $\mathcal{R}$ with n inputs and m outputs, the (deterministic) homomorphic evaluation algorithm outputs $y_b \in \mathcal{R}^m$, constituting party b's share over $\mathcal{R}$ of an output $y \in \mathcal{R}^m$.*

The algorithms $(\mathsf{HSS.Gen}, \mathsf{HSS.Share}, \mathsf{HSS.Eval})$ *should satisfy the following correctness and security requirements:*

- **Correctness:** *For every polynomial $\mathsf{poly}(\lambda)$ there exists a negligible $\mathsf{negl}(\lambda)$ such that for every λ, input $x \in \mathcal{R}^n$ (where $\mathcal{R} = \mathcal{R}(\lambda)$), and degree-$d$ arithmetic circuit P of size $\mathsf{poly}(\lambda)$ we have:*

$$\Pr[y_0 + y_1 \neq P(x)] \leq \mathsf{negl}(\lambda),$$

where probability is taken over

$$(\mathsf{sk}, \mathsf{ek}) \leftarrow \mathsf{HSS.Gen}(1^\lambda); (s_0, s_1) \leftarrow \mathsf{HSS.Share}(\mathsf{sk}, x);$$
$$y_b \leftarrow \mathsf{HSS.Eval}(b, \mathsf{ek}, s_b, P),\ b \in \{0, 1\}.$$

- **Security:** *For any $b \in \{0, 1\}$, pair of input sequences $x_\lambda, x'_\lambda \in \mathcal{R}^n$ of polynomial length $n(\lambda)$, the distribution ensembles $C_b(\lambda, x_\lambda)$ and $C_b(\lambda, x'_\lambda)$ are computationally indistinguishable, where $C_b(\lambda, z)$ for $z \in \{x_\lambda, x'_\lambda\}$ is obtained by sampling $(\mathsf{sk}, \mathsf{ek}) \leftarrow \mathsf{HSS.Gen}(1^\lambda)$, sampling $(s_0, s_1) \leftarrow \mathsf{HSS.Enc}(\mathsf{sk}, z)$, and outputting (ek, s_b).*

3.3 Learning Parity with Noise

Our constructions rely on variants of the Learning Parity with Noise (LPN) assumption [BFKL93] over either $\mathbb{F}_2$ or a large finite field $\mathbb{F}$. Unlike the LWE assumption, in LPN over $\mathbb{F}$ the noise is assumed to have a small Hamming weight. Concretely, the noise is a random field element in a small fraction of the coordinates and 0 elsewhere. Similar assumptions have been previously used in the context of secure arithmetic computation [NP06, IPS09, ADI+17, DGN+17, GNN17]. Unlike most of these works, the flavors of LPN on which we rely do not require the underlying code to have an algebraic structure and are thus not susceptible to algebraic (list-) decoding attacks.

Definition 6 (LPN). *Let $\mathcal{D}(\mathcal{R}) = \{\mathcal{D}_{k,q}(\mathcal{R})\}_{k,q \in \mathbb{N}}$ denote a family of distributions over a ring $\mathcal{R}$, such that for any $k, q \in \mathbb{N}$, $\mathsf{Im}(\mathcal{D}_{k,q}(\mathcal{R})) \subseteq \mathcal{R}^q$. Let $\mathbf{C}$ be a probabilistic code generation algorithm such that $\mathbf{C}(k, q, \mathcal{R})$ outputs a matrix $A \in \mathcal{R}^{k \times q}$. For dimension $k = k(\lambda)$, number of samples (or block length) $q = q(\lambda)$, and ring $\mathcal{R} = \mathcal{R}(\lambda)$, the $(\mathcal{D}, \mathbf{C}, \mathcal{R})$-$\mathsf{LPN}(k, q)$ assumption states that*

$$\{(A, \boldsymbol{b}) \mid A \xleftarrow{\$} \mathbf{C}(k, q, \mathcal{R}), \boldsymbol{e} \xleftarrow{\$} \mathcal{D}_{k,q}(\mathcal{R}), \boldsymbol{s} \xleftarrow{\$} \mathbb{F}^k, \boldsymbol{b} \leftarrow \boldsymbol{s} \cdot A + \boldsymbol{e}\}$$
$$\overset{\mathrm{c}}{\approx} \{(A, \boldsymbol{b}) \mid A \xleftarrow{\$} \mathbf{C}(k, q, \mathcal{R}), \boldsymbol{b} \xleftarrow{\$} \mathcal{R}^q\}$$

Here and in the following, all parameters are functions of the security parameter λ and computational indistinguishability is defined with respect to λ.

When $\mathcal{R} = \mathbb{F}_2$ and $\mathcal{D}$ is the Bernoulli distribution over $\mathbb{F}_2^q$, where each coordinate is 1 with probability r and 0 otherwise, this corresponds to the standard binary LPN assumption.

Note that the search LPN problem, of finding the vector can be reduced to the decisional LPN assumption as defined above above when the code generator $\mathbf{C}$ outputs a uniform matrix A [BFKL93, AIK09]. However, this is less relevant for us as we are mainly interested in efficient variants with more structured codes. See [DI14] for further discussion of search-to-decision reductions in the general case.

3.3.1 Example: LPN with Fixed Weight Noise

For a finite field $\mathbb{F}$, we denote by $\mathcal{HW}_r(\mathbb{F})$ the distribution of uniform, weight r vectors over $\mathbb{F}$; that is, a sample from $\mathcal{HW}_r(\mathbb{F})$ is a uniformly random nonzero field element in r random positions, and zero elsewhere. The $(\mathsf{Ber}_r(\mathbb{F})^q, \mathbf{C}, \mathbb{F}) - \mathsf{LPN}(k, q)$ assumption corresponds to the standard (non-binary, fixed-weight) LPN assumption over a field $\mathbb{F}$ with code generator $\mathbf{C}$, dimension k, number of samples (or block length) q, and noise rate r.

When the block length q and noise rate r are such that k random coordinates will be all noiseless with non-negligible probability (e.g., when r is constant and $q = \Omega(k^2)$), LPN can be broken via Gaussian elimination (cf. [AG11]). This attack does not apply to our constructions, which typically have $q = O(k)$.

Definition 7 (dual LPN). *Let $\mathcal{D}(\mathcal{R})$ and $\mathbf{C}$ be as in Definition 6, $n, n' \in \mathbb{N}$ with $n' > n$, and define $\mathbf{C}^{\perp}(n', n, \mathcal{R}) = \{B \in \mathcal{R}^{n' \times n} : A \cdot B = 0, A \in \mathbf{C}(n' - n, n', \mathcal{R}), \mathsf{rank}(B) = n\}$.*

For $n = n(\lambda), n' = n'(\lambda)$ and $\mathcal{R} = \mathcal{R}(\lambda)$, the $(\mathcal{D}, \mathbf{C}, \mathcal{R})$-$\mathsf{dual\text{-}LPN}(n', n)$ assumption states that

$$\{(H, \boldsymbol{b}) \mid H \xleftarrow{\$} \mathbf{C}^{\perp}(n', n, \mathcal{R}), \boldsymbol{e} \xleftarrow{\$} \mathcal{D}(\mathcal{R}), \boldsymbol{b} \leftarrow \boldsymbol{e} \cdot H\}$$
$$\overset{c}{\approx} \{(H, \boldsymbol{b}) \mid H \xleftarrow{\$} \mathbf{C}^{\perp}(n', n, \mathcal{R}), \boldsymbol{b} \xleftarrow{\$} \mathcal{R}^n\}$$

The search version of the dual LPN problem is also known as syndrome decoding. The decision version defined above is equivalent to primal variant of LPN from Definition 6 with dimension $k = n' - n$ and number of samples $q = n'$. This follows from the simple fact that $(\boldsymbol{s} \cdot A + \boldsymbol{e}) \cdot H = \boldsymbol{s} \cdot A \cdot H + \boldsymbol{e} \cdot H = \boldsymbol{e} \cdot H$, when H is the parity-check matrix of A.

Remark 8. For any code generation algorithm $\mathbf{C}$ where $\mathsf{dual\text{-}LPN}$ is hard, it must hold that for $H \xleftarrow{\$} \mathbf{C}^{\perp}(n', n', \mathcal{R})$, H is full rank with overwhelming probability. If that was not the case, then we could easily distinguish $\boldsymbol{e} \cdot H$ from uniform due to a linear relation between some of its outputs.

Remark 9. As a concrete example of the actual flavor of the dual-LPN assumption we will use, our construction of silent OT from Section 5 relies on the dual-LPN assumption of Definition 6 with respect to a random linear code over the field $\mathbb{F}_2$. For deriving our concrete parameters, we choose a *regular* error distribution of weight t, where a length-n' error vector has t non-zero coordinates spread across weight-1 blocks of length n'/t. This is known as the regular-LPN or *regular syndrome decoding* problem. When $n \geq 2^{16}$ and $n' = 4n$, a fixed-weight noise of $t \approx 32$ suffices to achieve 80-bit security against the best known attacks on this flavor of LPN, which all take time exponential in $(n'/n) \cdot t$. We will also consider alternative choices of linear codes (such as LDPC codes or quasi-cyclic codes) to improve the concrete computational efficiency in our estimates; such codes still lead to plausible variants of LPN and do not significantly improve known attacks compared with random codes.

4 Pseudorandom Correlation Generators

In this section we put forward a general notion of pseudorandom correlation generator (PCG) and study some of its limitations, capabilities, and relation with other primitives. We start with our formal definition of PCG in Section 4.1. We then prove in Section 4.2 that a simpler and more natural simulation-based definition of PCG, that would suffice for *all* applications, is not realizable. As a second-best alternative, we show in Section 4.3 that PCGs can be used as a drop-in replacement for correlated randomness in every protocol that meets a slightly stronger security requirement, which is indeed met by natural MPC protocols in the correlated randomness model. Finally in Section 4.4 we show a two-way relation between PCGs for a useful class of "low-degree correlations" and HSS for low-degree polynomials as defined in Section 3.2.

4.1 Defining Pseudorandom Correlation Generators

At a high level, a pseudorandom correlation generator (PCG) for some relation takes as input a pair of short, correlated seeds and outputs long correlated pseudorandom strings, where the expansion procedure is deterministic and can be applied locally.

For correctness we require that the expanded output of a PCG is indistinguishable from truly random correlated strings.

For security it would be natural and straightforward to require that we can securely replace long correlated strings by short correlated seeds in any secure protocol execution. Unfortunately, as shown in the following section, this security requirement would be impossible to meet. Therefore, we will introduce (and subsequently prove useful) an indistinguishability based security notion. Namely, we require that an adversary given access to one of the short seeds k_σ, cannot distinguish the pseudorandom string $R_{1-\sigma}$ from a pseudorandom string that is chosen at random conditioned on (R_0, R_1) being correlated (where $R_\sigma = \mathsf{PCG}(\mathsf{k}_\sigma)$). In other words, an adversary given access to a short seed cannot learn more about the other party's pseudorandom string than what is obvious given access to its own pseudorandom string.

In order to formally define pseudorandom correlations, we first introduce the concept of a *correlation generator* as a PPT algorithm outputting correlated elements.

Definition 10 (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 $n \in \mathsf{poly}(\lambda)$.*

In order to define security, we require the notion of a reverse-sampleable correlation generator introduced in the following.

Definition 11 (Reverse-sampleable Correlation Generator). *Let $\mathcal{C}$ be a correlation generator. We say $\mathcal{C}$ is* reverse sampleable *if there exists a PPT algorithm* 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]. Note though that we do not enforce perfect correctness.

Definition 12 (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 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:*

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

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

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

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

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

Note that the above definition is trivial to achieve in general: We can let PCG.Gen on input 1^λ return $(R_0, R_1) \leftarrow \mathcal{C}(1^\lambda)$, and simply define Expand to be the identity. Typically, we will be interested in non-trivial constructions of PCGs, in which the seed size is significantly shorter than the output size. A pseudorandom generator with image in $\{0,1\}^n$ is a simple example for an expanding PCG for the equality correlation $\{(R, R) \mid R \in \{0,1\}^n\}$. In the following we will be interested in constructing PCGs for a much broader class of correlations, like OT correlations, OLE correlations and (authenticated) Beaver triples.

Remark 13 (PCG with Setup). We sometimes consider an additional algorithm PCG.Setup to sample a secret key, public parameters and a share of evaluation keys (or a subset of the mentioned), which can be reused throughout several instances. More precisely: On input 1^λ, PCG.Setup returns a tuple $(\mathsf{pp}, \mathsf{sk}, \{\mathsf{ek}_\sigma\}_{\sigma\in\{0,1\}})$, PCG.Gen receives the secret key sk as additional input (always assumed to include the public parameters pp), and PCG.Expand receives the public parameters pp and the respective evaluation key share ek_σ as additional inputs.

Remark 14 (PCG in the Multi-Party Setting). We also consider multi-party PCGs for reverse sampleable multi-correlation generators $\mathcal{C}^M$ which on input 1^λ outputs elements in $(\{0,1\}^n)^M$. In this case, $\mathsf{PCG.Gen}(1^\lambda)$ returns a M-tuple $(\mathsf{k}_1, \ldots, \mathsf{k}_M)$. Correctness is defined accordingly and security required against any subset of colluding parties. More precisely: For any $T \subset \{1, \ldots, M\}$, we require the following two distributions to be computationally indistinguishable:

$$\begin{aligned}
&\{(\{\mathsf{k}_j\}_{j\in T}, \{R_i\}_{i\notin T}) \,|(\mathsf{k}_1, \ldots, \mathsf{k}_M) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda), \\
&\qquad\qquad \forall i \notin T\colon R_i \leftarrow \mathsf{PCG.Expand}(i, \mathsf{k}_i)\} \quad \text{and} \\
&\{(\{\mathsf{k}_j\}_{j\in T}, \{R_i\}_{i\notin T}) \,|(\mathsf{k}_1, \ldots, \mathsf{k}_M) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda), \\
&\qquad\qquad \forall j \in T\colon R_j \leftarrow \mathsf{PCG.Expand}(j, \mathsf{k}_j), \\
&\qquad\qquad \{R_i\}_{i\notin T} \xleftarrow{\$} \mathsf{RSample}(T, \{R_j\}_{j\in T})\}
\end{aligned}$$

where RSample is the reverse sampling algorithm corresponding to the multi-correlation $\mathcal{C}^M$.

4.2 Impossibility of a Simulation-Based Definition

A natural and useful alternative to the security definition we gave in Section 4, is the following: In any secure protocol (say against semi-honest adversaries), one can replace sampling a pair of strings from the correlation $\mathcal{C}$ by generating a pair of seeds (which are later expanded) using a PCG for $\mathcal{C}$ without compromising security. Unfortunately, as sketched in [GI99], a non-trivial PCG construction cannot satisfy such a simulation-based definition. Consider the simple protocol, where P_0 samples a pair $(R_0, R_1) \leftarrow \mathcal{C}(1^\lambda)$ and sends R_1 to P_1, who simply outputs R_1. This protocol obviously realizes the protocol dictated by $\mathcal{C}$, with one-sided security against P_1. But, if P_0 instead generates $(\mathsf{k}_0, \mathsf{k}_1)$ according to the seed generation algorithm of the PCG and sends k_1 to P_1, a possible simulator runs into the following problem. Simulating the above protocol given only the output R_1 corresponds to finding a short seed k_1 that can be (deterministically) expanded to R_1. If the entropy in the second output of $\mathcal{C}$ exceeds the seed-length $|\mathsf{k}_1|$, such a compression violates correctness, as it could be used to distinguish R_1 from a string that is indeed chosen via $\mathcal{C}$.

In the following, we present a formal and more general version of the above argument for ruling out a simulation-based definition for non-trivial correlations. Our negative result is based on a lower bound given by Hubáček and Wichs [HW15]. There, the notion of Yao incompressibility entropy, the computational equivalent to Shannon entropy, is employed to establish a lower bound on the required communication in a secure protocol with long outputs. More precisely, Yao incompressibility entropy [HLR07, Yao82] is a measure on how well outputs of a distribution can be compressed on average, when the compressing and decompressing algorithms are required to be efficient. For example, a pseudorandom bit string of length ℓ has Yao incompressibility entropy ℓ.

Definition 15 (Yao Incompressiblity Entropy [HLR07] (simplified)). *Let $\ell = \ell(\lambda) \in \mathbb{N}$. A probability ensemble $X = \{X_\lambda\}$ has* Yao incompressibility entropy at least ℓ*, if for every pair of polynomial sized circuit-ensembles $C = \{C_\lambda\}$, $D = \{D_\lambda\}$ where C has output bit-length at most $\ell - 1$, there exists a negligible function $\mathsf{negl}\colon \mathbb{N} \to \mathbb{R}^+$ such that for every sufficiently large positive integer λ we have*

$$\Pr\left[x \leftarrow X\colon D(C(x)) = x\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

One of the main results of [HW15] is that the communication in a secure protocol has to at least meet the Yao incompressibility entropy of the output, when the adversary is allowed to fix the random coins of the corrupted party. Applying this result rules out meaningful PCG instantiations of a simulation-based security definition.

Theorem 16 (Impossiblity of Simulation-Based Definition for Non-Trivial PCGs). *Let $\mathcal{C}$ be a reverse-sampleable correlation generator, where the Yao incompressibility entropy of the output is ℓ. Then, for every pseudorandom correlation generator* $\mathsf{PCG} = (\mathsf{PCG.Gen}, \mathsf{PCG.Expand})$ *satisfying simulation-based security, the output of the seed generation* PCG.Gen *algorithm must at least have bit-length ℓ.*

Proof. Let $\mathsf{PCG} = (\mathsf{PCG.Gen}, \mathsf{PCG.Expand})$ be a pseudorandom correlation generator for $\mathcal{C}$ that satisfies simulation-based security. Then, in particular, the following protocol Π_{PCG} has to satisfy one-sided security against P_1: Party P_0 runs $(k_0, k_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda)$ and sends k_1 to P_1. Finally, P_1 outputs $R_1 \leftarrow \mathsf{PCG.Expand}(1, k_1)$.

Let ℓ_1 be the Yao incompressibility entropy of the output of $\mathcal{C}^1(1^\lambda) := \{R_1 \mid (R_0, R_1) \leftarrow \mathcal{C}(1^\lambda)\}$. Further, let

$$\mathcal{C}^1_{\mathsf{PCG}}(1^\lambda) := \{R_1 \mid (\mathsf{k}_0, \mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda), R_1 \leftarrow \mathsf{PCG.Expand}(1, \mathsf{k}_1)\}.$$

By correctness of the PCG, the output of $\mathcal{C}^1_{\mathsf{PCG}}$ (and therefore the output of the protocol Π_{PCG}) must meet the Yao incompressibility entropy ℓ_1, as an efficient pair of compressor and decompressor could be used as a distinguisher between $\mathcal{C}^1$ and $\mathcal{C}^1_{\mathsf{PCG}}$.

By [HW15, Theorem 5], for any protocol between two parties P_0 and P_1 with one-sided security against "honest-but-deterministic"[11] P_1, where P_1 has no input, it holds: *If the Yao incompressibility entropy of the output of P_1 is ℓ_1, then the communication complexity from P_0 to P_1 must be at least ℓ_1 bits.*

Therefore, as seed expansion is deterministic, the bit-length $|k_1|$ of the seed of the second party must be at least ℓ_1. Reversing the roles of P_0 and P_1 together with additivity of Yao incompressibility entropy yields the required.

For the special case, where $\mathcal{C}$ outputs pairs of identical random strings (R, R), Gilboa and Ishai [GI99] sketched why pseudorandom pads cannot securely substitute perfectly-random pads returned by $\mathcal{C}$ in every secure protocol. We obtain this result as a straightforward corollary of Theorem 16.

Corollary 17. *Let $\mathcal{C}$ be the correlation generator that on input 1^λ draws a string $R \xleftarrow{\$} \{0,1\}^\ell$ uniformly at random and returns (R, R). Then, there exists no pseudorandom generator with seed length strictly less than ℓ, such that sampling a seed (and later expanding via the pseudorandom generator) can securely replace sampling uniformly at random from $\{0,1\}^\ell$ in every protocol.*

Extension to deterministic functionalities. The above negative result gives a general counterexample for randomized functionalities. We add to this by showing that our weaker definition of PCG cannot replace correlated randomness in general, even in protocols that only realize *deterministic* functionalities. We prove this using public-key type assumptions, since we will use a correlation that produces the public key of a *messy* cryptosystem, where public keys statistically hide the message. We show that if instantiated with a PCG for $\mathcal{C}$ that produces a *real* public key, we completely break privacy of a (rather contrived) protocol.

Theorem 18. *Suppose that the DDH, QR or LWE assumption holds, and let $\mathcal{F}^{\mathcal{C}}_{\mathsf{corr}}$ be a sampling functionality for a correlation $\mathcal{C}$. Then there exists a reverse-sampleable correlation $\mathcal{C}$, a secure PCG for $\mathcal{C}$, and a protocol π for some deterministic functionality $\mathcal{F}$ such that: (i) π securely realizes $\mathcal{F}$ in the $\mathcal{F}^{\mathcal{C}}_{\mathsf{corr}}$-hybrid model with malicious, statistical security; and (ii) π is passively insecure when $\mathcal{F}^{\mathcal{C}}_{\mathsf{corr}}$ is replaced by $\mathcal{F}^{\mathsf{PCG.Gen}}_{\mathsf{corr}}$.*

Proof. Let $\mathcal{F}$ be a trivial functionality which takes private inputs (x_0, x_1) and outputs zero. Consider a public-key encryption scheme that, in addition to the usual algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, supports a *messy mode* of key generation, Gen^*, where public keys output by Gen^* are computationally indistinguishable from a real public key, however, ciphertexts produced with a messy public key are not decryptable, and *statistically hide* the message. This can be constructed from a range of assumptions including DDH, QR or LWE with standard methods, see for instance [PVW08].

Define the reverse-sampleable distribution to output $(pk, pk) \xleftarrow{\$} \mathcal{C}(1^\lambda)$, where $(pk, sk) \xleftarrow{\$} \mathsf{Gen}^*(1^\lambda)$. In the protocol π, party P_i sends $\mathsf{Enc}_{pk}(x_i)$ to the other party and outputs zero. This is easily seen to be statistically secure by the messy property of the encryption scheme. We now construct a secure PCG for $\mathcal{C}$. $\mathsf{PCG.Gen}$ samples a real key pair $(pk, sk) \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$ and outputs $(\mathsf{k}_0, \mathsf{k}_1)$ where $\mathsf{k}_0 = (pk, sk)$ and $\mathsf{k}_1 = pk$. $\mathsf{PCG.Expand}$ for either party simply outputs pk. This is a (computationally) secure PCG, due to the indistinguishability of the two modes of key generation. Nevertheless, the protocol π is now completely insecure when using PCG instead of $\mathcal{F}^{\mathcal{C}}_{\mathsf{corr}}$, since P_0 can decrypt P_1's input with the secret key.

[11] A "honest-but-deterministic" adversary has to behave according to the protocol, but is allowed to fix its random coins.

4.3 Applying PCGs in Protocols with Correlated Randomness

In this section we show that one *can* use PCGs in a "plug-and-play" fashion in protocols consuming correlated randomness sampled by a given functionality. More precisely, we show that PCGs can be directly applied to any protocol using a weaker form of correlated randomness, where corrupted parties can influence their outputs.

A simple example is random OT, where the weaker functionality we can realize allows a corrupt sender/receiver to choose its outputs, then the other party's outputs are sampled at random correspondingly. When using OT in an MPC protocol, the OT is typically implemented from random OT by masking the actual OT inputs with fresh random OT outputs. Allowing a corrupt party to choose its own OT outputs does not affect the security of these protocols, since (intuitively) this can only weaken security for the corrupt party and not for honest parties. More generally, it turns out that many practical MPC protocols, including those based on preprocessed multiplication triples for arithmetic circuits [BDOZ11, DPSZ12] and binary circuits [NNOB12, WRK17a, WRK17b], use this kind of corruptible, correlated randomness, since it is often easier to design a protocol that realizes this.

More formally, the randomness is modelled by the functionality $\mathcal{F}_{\mathsf{corr}*}^{\mathcal{C}}$ (Fig. 1), where a corrupted party may first *choose* its own output, and then the honest party's output is computed with the reverse sampling algorithm for $\mathcal{C}$. As we show in the following, PCGs can be used to securely realize $\mathcal{F}_{\mathsf{corr}*}^{\mathcal{C}}$, opening up many important applications at no extra cost.

To realize $\mathcal{F}_{\mathsf{corr}*}^{\mathcal{C}}$, we use a simple protocol, $\Pi_{\mathsf{corr}*}^{\mathcal{C}}$, that calls $\mathcal{F}_{\mathsf{corr}}^{\mathsf{PCG.Gen}}$ so that each party obtains a seed k_σ, which is then expanded to get the output $\mathsf{PCG.Expand}(\sigma, \mathsf{k}_\sigma)$.

Functionality $\mathcal{F}_{\mathsf{corr}*}^{\mathcal{C}}$

On input 1^λ, the functionality does as follows:

- If no parties are corrupt, sample $(R_0, R_1) \xleftarrow{\$} \mathcal{C}(1^\lambda)$.
- Otherwise, if P_σ is corrupt, wait to receive $R_\sigma \in \{0,1\}^{n_\sigma}$ from $\mathcal{A}$, then sample $R_{1-\sigma} \xleftarrow{\$} \mathsf{RSample}(\sigma, R_\sigma)$.

The functionality outputs R_0 to P_0 and R_1 to P_1, and then halts.

Fig. 1. Corruptible correlated randomness functionality for a reverse-sampleable correlation generator, $\mathcal{C}$

Theorem 19. *Let* $\mathsf{PCG} = (\mathsf{PCG.Gen}, \mathsf{PCG.Expand})$ *be a secure PCG for a reverse-sampleable correlation generator,* $\mathcal{C}$*. Then the protocol* $\Pi_{\mathsf{corr}*}$ *securely realizes the* $\mathcal{F}_{\mathsf{corr}*}^{\mathcal{C}}$ *functionality against a static, malicious adversary.*

Proof. Let $\mathcal{A}$ be a static adversary against the protocol π. We construct a simulator Sim, which interacts with $\mathcal{A}$ and $\mathcal{F}_{\mathsf{corr}*}^{\mathcal{C}}$ to produce a view for $\mathcal{A}$ that is indistinguishable from a real execution of the protocol. When both parties are corrupted, the simulator just runs $\mathcal{A}$ internally and security is straightforward. Similarly, when both parties are honest, simulation is trivial and indistinguishability follows from the correctness of PCG. Now suppose that only P_σ is corrupted, for $\sigma \in \{0,1\}$. On receiving the input 1^λ, Sim samples a pair of seeds $(\mathsf{k}_0, \mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda)$, then sends k_σ to $\mathcal{A}$ as its output of $\mathcal{F}_{\mathsf{corr}}^{\mathsf{PCG.Gen}}$, computes $R_\sigma \leftarrow \mathsf{PCG.Expand}(\sigma, \mathsf{k}_\sigma)$ and sends this to $\mathcal{F}_{\mathsf{corr}*}^{\mathcal{C}}$. Notice that in the ideal execution, the view of the distinguisher consists of the seed k_σ and the honest party's output $R_{1-\sigma}$, which is computed by $\mathcal{F}_{\mathsf{corr}*}^{\mathcal{C}}$ as $R_{1-\sigma} \xleftarrow{\$} \mathsf{RSample}(\sigma, R_\sigma)$. The only difference in the real execution, is that there the honest party's output is computed with $\mathsf{PCG.Expand}(1-\sigma, \mathsf{k}_{1-\sigma})$. These two views are computationally indistinguishable, due to the security property of PCG.

4.4 Relation Between PCGs and HSS

In this section we elaborate on the two-way relation between pseudorandom correlation generators and homomorphic secret sharing (HSS) schemes.

First, we show how to generically construct a PCG for additive correlations by combining a pseudorandom generator with a suitable HSS scheme. If the correlation is a degree-d polynomial and the PRG has degree d', then the HSS scheme must support evaluation of degree dd' polynomials. This can be viewed as a step forward towards the concretely efficient PCG constructions given in Section 7.

Second, in the other direction, we show that a PCG for degree-d additive correlations, for constant d, implies (secret-key) HSS for degree-d multivariate polynomials. By the results of [BGI16a], this has implications for secure multi-party computation. For details we refer to Section 6.1.

4.4.1 Generic Construction of PCG for Additive Correlations from HSS

Our high-level strategy is as follows: We combine a standard pseudorandom generator (PRG), expanding a short seed into a long pseudorandom string, with a suitable HSS scheme, which allows to *locally* compute the target correlation on shares of a random input. More precisely, we consider the special case of additive correlations, where R_0, R_1 are uniformly distributed subject to $R_0 + R_1 = f(X)$ for a random input X and fixed function f. Now, consider an HSS scheme with additive reconstruction for f. Recall that given *shares of the input* X an HSS allows to locally evaluate f on the shares, such that the respective outputs add up to $f(X)$.

This gives rise to the following PCG construction: During key generation a short seed k is shared between the players (as HSS shares). For expansion, the players can then locally evaluate $f(\mathsf{PRG}(\mathsf{k}))$ via the HSS operations. By the correctness of the HSS that indeed gives outputs R_0, R_1 with $R_0 + R_1 = f(X)$, where $X = \mathsf{PRG}(\mathsf{k})$. In this section we formally prove that the described construction meets the PCG requirements.

Note that the challenge lies in actually instantiating the described approach efficiently. This is due to the fact that known efficient HSS constructions only apply to limited classes of functions, for instance functions admitting small branching programs. It is therefore crucial to carefully select the underlying PRG and HSS. We will elaborate on how we address these challenges in Section 7.

To formalize the above outline, we first give a generalized definition of a pseudorandom generator, then formally define additive correlations corresponding to a function, before presenting the construction.

Let $\mathcal{R}$ be a ring and $\ell, n \in \mathbb{N}$. We consider distributions $\mathcal{D}^\ell$ over a ring $\mathcal{R}^\ell$ and write $X \xleftarrow{\$} \mathcal{D}^\ell(\mathcal{R})$ or simply $X \xleftarrow{\$} \mathcal{D}^\ell$ (if $\mathcal{R}$ is clear from the context) to denote sampling from $\mathcal{R}^\ell$ via $\mathcal{D}^\ell$. Note that the following definition of $\mathcal{D}^\ell$-pseudorandom generator coincides with the standard definition of a PRG, if we choose $\mathcal{D}^\ell(\mathcal{R}) = \mathcal{U}^\ell(\mathcal{R})$. We use this more general notion of a PRG, as for our PRG instantiation from LPN the seed is not chosen uniformly at random.

Definition 20 ($\mathcal{D}^\ell$-Pseudorandom Generator). *Let $\mathcal{R}$ be a ring (parametrized implicitly by λ) and let $\mathcal{D}^\ell$ be a distribution on $\mathcal{R}^\ell$. We say* $\mathsf{PRG}\colon \mathcal{R}^\ell \to \mathcal{R}^n$ *is a* $\mathcal{D}^\ell$-pseudorandom generator (PRG)*, if the following two distributions are computationally indistinguishable:*

$$\left\{Y \mid X \xleftarrow{\$} \mathcal{D}^\ell(\mathcal{R}), Y := \mathsf{PRG}(X)\right\} \text{ and } \left\{Y \mid Y \xleftarrow{\$} \mathcal{U}^n(\mathcal{R})\right\}.$$

We will consider *additive correlations* corresponding to a family of functions $\mathcal{F}$. Such a correlation is generated by outputting an additive secret-sharing of a function from $f \in \mathcal{F}$ applied to a source of randomness.

– $\mathsf{PCG.Setup}(1^\lambda)$: Sample and output $(\mathsf{sk}, \{\mathsf{ek}_\sigma\}_{\sigma\in\{0,1\}}) \leftarrow \mathsf{HSS.Gen}(1^\lambda)$.
– $\mathsf{PCG.Gen}(\mathsf{sk})$: Sample $r \leftarrow \mathcal{D}^\ell$ and output $(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{HSS.Share}(\mathsf{sk}, r)$.
– $\mathsf{PCG.Expand}(\sigma, \mathsf{ek}_\sigma, \mathsf{k}_\sigma, f)$: Output $R_\sigma \leftarrow \mathsf{HSS.Eval}(\sigma, \mathsf{ek}_\sigma, \mathsf{k}_\sigma, f \circ \mathsf{PRG})$.

Fig. 2. PCG for correlation $\mathcal{C}_\mathcal{F}$. Here, PRG is a $\mathcal{D}^\ell$-PRG and $\mathsf{HSS} = (\mathsf{HSS.Gen}, \mathsf{HSS.Share}, \mathsf{HSS.Eval})$ an HSS for the family of functions $\mathcal{F}_\mathsf{HSS} := \{f \circ \mathsf{PRG}\colon r \mapsto f(\mathsf{PRG}(r)) \mid f \in \mathcal{F}\}$.

Definition 21 (Correlation Generators for Additive Correlations). *Let $\mathcal{R}$ be a ring. Let $n, m \in \mathbb{N}$ and $\mathcal{F} \subseteq \{f\colon \mathcal{R}^n \to \mathcal{R}^m\}$ be a family of functions. Then we define a* correlation generator $\mathcal{C}_\mathcal{F}$ for $\mathcal{F}$ *as follows: On input 1^λ and $f \in \mathcal{F}$ the correlation generator $\mathcal{C}_\mathcal{F}$ samples $X \xleftarrow{\$} \mathcal{U}^n(\mathcal{R})$, and returns a pair $(R_0, R_1) \in \mathcal{R}^m \times \mathcal{R}^m$, which is distributed uniformly at random conditioned on $R_0 + R_1 = f(X)$.*

Note that $\mathcal{C}_\mathcal{F}$ is reverse-sampleable for any family of functions $\mathcal{F}$, as given a function $f \in \mathcal{F}$ and a share R_σ, one can draw an input $X \xleftarrow{\$} \mathcal{U}^n(\mathcal{R})$ and set $R_{1-\sigma} := R_\sigma - f(X)$. Further, note that it is straightforward to includes shares of the inputs in the correlation by considering the family $\mathcal{F}' := \{f'\colon \mathcal{R}^n \to \mathcal{R}^{n+m}, X \mapsto (X, f(X)) \mid f \in \mathcal{F}\}$.

Definition 22 (HSS satisfying Pseudorandomness of Outputs). *We say an HSS $\mathsf{HSS} = (\mathsf{HSS.Gen}, \mathsf{HSS.Share}, \mathsf{HSS.Eval})$ for a function family $\mathcal{F} := \{f\colon \mathcal{R}^n \to \mathcal{R}^m\}$ satisfies* pseudorandomness of outputs*, if for all $f\colon \mathcal{R}^n \to \mathcal{R}^m \in \mathcal{F}$, $(\mathsf{sk}, \{\mathsf{ek}_\sigma\}_{\sigma\in\{0,1\}}) \leftarrow \mathsf{HSS.Gen}(1^\lambda)$, $X \xleftarrow{\$} \mathcal{U}^n(\mathcal{R})$, $(\mathsf{k}_0, \mathsf{k}_1) \xleftarrow{\$} \mathsf{Share}(\mathsf{sk}, X)$, and $\sigma \in \{0,1\}$ the output $R_\sigma \xleftarrow{\$} \mathsf{HSS.Eval}(\sigma, \mathsf{ek}_\sigma, \mathsf{k}_\sigma, f)$ is distributed computationally close to uniformly at random over the output space.*

Note that if $f(\mathcal{U}^n(\mathcal{R}))$ is close to being uniformly random on $\mathcal{R}^m$, this property follows from the security of HSS.

Theorem 23. (PCG for Additive Correlations from HSS). *Let $\mathcal{R}$ be a ring and $n, m, \ell \in \mathbb{N}$. Let $\mathcal{F} \subseteq \{f\colon \mathcal{R}^n \to \mathcal{R}^m\}$ be a family of functions. Let PRG be a $\mathcal{D}^\ell$-PRG and $\mathsf{HSS} = (\mathsf{HSS.Gen}, \mathsf{HSS.Share}, \mathsf{HSS.Eval})$ an HSS with overhead O_HSS[12] for the family of functions $\mathcal{F}_\mathsf{HSS} := \{f \circ \mathsf{PRG}\colon \mathcal{R}^\ell \to \mathcal{R}^m, r \mapsto f(\mathsf{PRG}(r)) \mid f \in \mathcal{F}\}$ that further satisfies pseudorandomness of outputs. Then, $\mathsf{PCG} = (\mathsf{PCG.Setup}, \mathsf{PCG.Gen}, \mathsf{PCG.Expand})$ as defined in Figure 2 is a PCG for the correlation generator $\mathcal{C}_\mathcal{F}$ with key-length upper bounded by $\ell \cdot O_\mathsf{HSS}$.*

Proof. Correctness. Let $f \in \mathcal{F}$. We have

$$
\begin{aligned}
&\{(R_0, R_1) \mid (\mathsf{k}_0, \mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda), R_\sigma \leftarrow \mathsf{PCG.Expand}(\sigma, \mathsf{k}_\sigma) \text{ for } \sigma \in \{0,1\}\} \\
\overset{c}{\approx} &\{(R_0, R_1) \mid (\mathsf{sk}, \{\mathsf{ek}_\sigma\}_{\sigma\in\{0,1\}}) \leftarrow \mathsf{HSS.Gen}(1^\lambda), r \xleftarrow{\$} \mathcal{D}^\ell(\mathcal{R}), (\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{HSS.Share}(\mathsf{sk}, r), \\
&\qquad R_0 \leftarrow \mathsf{HSS.Eval}(0, \mathsf{ek}_0, \mathsf{k}_0, f \circ \mathsf{PRG}), R_1 := f(\mathsf{PRG}(r)) - R_0\} \\
\overset{c}{\approx} &\{(R_0, R_1) \mid r \xleftarrow{\$} \mathcal{D}^\ell(\mathcal{R}), R_0 \leftarrow \mathcal{R}^m, R_1 := f(\mathsf{PRG}(r)) - R_0\} \\
\overset{c}{\approx} &\{(R_0, R_1) \mid X \xleftarrow{\$} \mathcal{U}^n(\mathcal{R}), R_0 \leftarrow \mathcal{R}^m, R_1 := f(X) - R_0\}
\end{aligned}
$$

as required, where the first transition follow by correctness of HSS, the second by pseudorandomness of outputs of HSS and the last by pseudorandomness of PRG.

Security. Let $\sigma \in \{0,1\}$. We have

$$
\begin{aligned}
&\{(\mathsf{k}_{1-\sigma}, R_\sigma) \mid (\mathsf{k}_0, \mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda), R_\sigma \leftarrow \mathsf{PCG.Expand}(\sigma, \mathsf{k}_\sigma)\} \\
\overset{c}{\approx} &\{(\mathsf{k}_{1-\sigma}, R_\sigma) \mid (\mathsf{sk}, \{\mathsf{ek}_\sigma\}_{\sigma\in\{0,1\}}) \leftarrow \mathsf{HSS.Gen}(1^\lambda), r \xleftarrow{\$} \mathcal{D}^\ell(\mathcal{R}), (\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{HSS.Share}(\mathsf{sk}, r), \\
&\qquad R_{1-\sigma} \leftarrow \mathsf{HSS.Eval}(1-\sigma, \mathsf{ek}_{1-\sigma}, \mathsf{k}_{1-\sigma}, f \circ \mathsf{PRG}), R_\sigma := f(\mathsf{PRG}(r)) - R_{1-\sigma}\} \\
\overset{c}{\approx} &\{(\mathsf{k}_{1-\sigma}, R_\sigma) \mid (\mathsf{sk}, \{\mathsf{ek}_\sigma\}_{\sigma\in\{0,1\}}) \leftarrow \mathsf{HSS.Gen}(1^\lambda), r \xleftarrow{\$} \mathcal{D}^\ell(\mathcal{R}), r' \xleftarrow{\$} \mathcal{D}^\ell(\mathcal{R}),
\end{aligned}
$$

[12] We say a HSS has *overhead* O_HSS, if for every input the share size does not exceed O_HSS times the input size.

$$
\begin{aligned}
&(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{HSS.Share}(\mathsf{sk}, r'), R_{1-\sigma} \leftarrow \mathsf{HSS.Eval}(1-\sigma, \mathsf{ek}_{1-\sigma}, \mathsf{k}_{1-\sigma}, f \circ \mathsf{PRG}),\\
&R_\sigma := f(\mathsf{PRG}(r)) - R_{1-\sigma}\}\\
\overset{c}{\approx} \{(\mathsf{k}_{1-\sigma}, R_\sigma) \mid &(\mathsf{sk}, \{\mathsf{ek}_\sigma\}_{\sigma\in\{0,1\}}) \leftarrow \mathsf{HSS.Gen}(1^\lambda), X \xleftarrow{\$} \mathcal{U}^n(\mathcal{R}), r' \xleftarrow{\$} \mathcal{D}^\ell(\mathcal{R}),\\
&(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{HSS.Share}(\mathsf{sk}, r'), R_{1-\sigma} \leftarrow \mathsf{HSS.Eval}(1-\sigma, \mathsf{ek}_{1-\sigma}, \mathsf{k}_{1-\sigma}, f \circ \mathsf{PRG}),\\
&R_\sigma := f(X) - R_{1-\sigma}\},
\end{aligned}
$$

where the first transition follows by correctness of HSS, the second transition by security of HSS and the last by pseudorandomness of PRG.

4.4.2 Generic Construction of HSS from a PCG

We observe that there is also a connection in the reverse direction. Namely, given a PCG for general, additive degree-d correlations for a constant d, we show how to construct a homomorphic secret sharing scheme for degree-d multivariate polynomials, a primitive which is interesting in its own right. Consider two parties who wish to compute shares of $P(\boldsymbol{x})$ for some public multivariate polynomial P, given shares of $\boldsymbol{x}$. Let $\otimes_d \boldsymbol{x}$ denote the degree-d tensor product $\boldsymbol{x} \otimes \cdots \otimes \boldsymbol{x}$. Given a PCG for the additive, degree-d correlation $(\boldsymbol{r}, \otimes_2 \boldsymbol{r}, \cdots, \otimes_d \boldsymbol{r})$, we construct a (secret-key) homomorphic secret sharing scheme $\mathsf{HSS} = (\mathsf{HSS.Gen}, \mathsf{HSS.Share}, \mathsf{HSS.Eval})$ for P as follows.

- $\mathsf{HSS.Share}(\boldsymbol{x})$: generate PCG keys $(\mathsf{k}_0, \mathsf{k}_1)$ which expand to shares of $(\boldsymbol{r}, \otimes_2 \boldsymbol{r}, \cdots, \otimes_d \boldsymbol{r})$, set $\boldsymbol{x}' \leftarrow \boldsymbol{x} + \boldsymbol{r}$, and give to each party P_σ a share $s_\sigma = (\mathsf{k}_\sigma, \boldsymbol{x}')$.
- $\mathsf{HSS.Eval}(\sigma, s_b, P)$: On input party index $\sigma \in \{0, 1\}$, share s_σ of a size-n input, and a degree-d multivariate polynomial P, compute a share P'_σ of the polynomial P' satisfying $P'(X) = P(X - \boldsymbol{r})$. Note that the coefficients of P' are public degree $\leq d$ polynomials in $\boldsymbol{r}$, hence shares of the coefficients can be locally computed given shares of the monomials $\boldsymbol{r}, \cdots, \otimes_d \boldsymbol{r}$. Output $P'_\sigma(\boldsymbol{x}')$.

Correctness follows immediately by inspection, and security reduces to the security of the underlying PCG.

5 Silent Oblivious Transfer Extension From LPN

In this section we present a protocol for silent OT extension, which allows to generate n instances of random OT with sublinear communication complexity. To this end, we first show how to tweak the construction of Boyle et al. [BCGI18] to give correlated OT. Combining this observation with the OT extension technique of Ishai et al. [IKNP03] we obtain a PCG for random OT. Finally, we show how to use the protocol of Doerner and shelat [Ds17] for secure computation of the seed, giving sublinear OT extension.

5.1 Subfield Vector-OLE

Here, we introduce the notion of subfield vector oblivious linear evaluation (sVOLE), and show that sVOLE for $\mathbb{F}_q$ over subfield $\mathbb{F}_p \subset \mathbb{F}_q$ gives 1-out-of-p correlated OT. More precisely, a single big instance of sVOLE will give many 1-out-of-p OTs at once. Our construction of sVOLE comes with two additional advantages: It enjoys lower computational costs, because matrix multiplications are performed with a matrix over $\mathbb{F}_p$, and for $p = 2$ we can reduce security to the better-studied binary LPN problem, instead of its arithmetic variant over larger fields.

Subfield VOLE is a form of vector oblivious linear evaluation (VOLE) over $\mathbb{F}_q$, which computes $\boldsymbol{w} = \boldsymbol{u}x + \boldsymbol{v}$, where the vector $\boldsymbol{u}$ is restricted to lie over a subfield $\mathbb{F}_p \subset \mathbb{F}_q$, for $q = p^r$ (and we multiply $\boldsymbol{u}$ with $x \in \mathbb{F}_q$ component-wise, by viewing x as a vector over $\mathbb{F}_p$). It outputs $(\boldsymbol{u}, \boldsymbol{v})$ to the sender and $(x, \boldsymbol{w})$ to the receiver.

The construction in Fig. 3 uses the function $\mathsf{spread}_n(S, \boldsymbol{y})$, which expands a set $S = (s_1, \ldots, s_{|S|}) \subset [n]$ and a vector $\boldsymbol{y} \in \mathbb{F}_p^{|S|}$ into the vector $\boldsymbol{\mu} \in \mathbb{F}_p^n$, where $\mu_{s_i} = y_i$ for $i = 1, \ldots, |S|$, and $\mu_j = 0$ for $j \in [n] \setminus S$. It is a generalization of the VOLE generator from [BCGI18], which follows from the case $p = q$.

Construction G_{sVOLE}

PARAMETERS:

- Security parameter 1^λ, integers $n' > n$, $q = p^r$, and noise weight t.
- A code generation algorithm $\mathbf{C}$ and $H_{n',n} \xleftarrow{\$} \mathbf{C}(n', n, \mathbb{F}_p)$.
- A multi-point FSS scheme $(\mathsf{MPFSS.Gen}, \mathsf{MPFSS.FullEval})$.

CORRELATION: Output $(\boldsymbol{u}, \boldsymbol{v})$ and $(x, \boldsymbol{w})$, where $x \leftarrow \mathbb{F}_q$, $\boldsymbol{u} \xleftarrow{\$} \mathbb{F}_p^n$, $\boldsymbol{v} \xleftarrow{\$} \mathbb{F}_q^n$ and $\boldsymbol{w} = \boldsymbol{u}x + \boldsymbol{v}$.

GEN: On input 1^λ:

1. Pick a random size-t subset S of $[n']$, sorted in increasing order.
2. Pick a random vector $\boldsymbol{y} \in (\mathbb{F}_p^*)^t$ and $x \xleftarrow{\$} \mathbb{F}_q$.
3. Compute $(K_0^{\mathsf{fss}}, K_1^{\mathsf{fss}}) \xleftarrow{\$} \mathsf{MPFSS.Gen}(1^\lambda, f_{S,x\cdot\boldsymbol{y}})$.
4. Let $\mathsf{k}_0 \leftarrow (m, n, K_0^{\mathsf{fss}}, S, \boldsymbol{y})$ and $\mathsf{k}_1 \leftarrow (m, n, K_1^{\mathsf{fss}}, x)$.
5. Output $(\mathsf{k}_0, \mathsf{k}_1)$.

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

1. If $\sigma = 0$: parse k_0 as $(m, n, K_0^{\mathsf{fss}}, S, \boldsymbol{y})$. Set $\boldsymbol{\mu} \leftarrow \mathsf{spread}_{n'}(S, \boldsymbol{y})$ in $\mathbb{F}_p^{n'}$. Compute $\boldsymbol{v}_0 \leftarrow \mathsf{MPFSS.FullEval}(0, K_0^{\mathsf{fss}})$ in $\mathbb{F}_q^{n'}$. Output $(\boldsymbol{u}, \boldsymbol{v}) \leftarrow (\boldsymbol{\mu} \cdot H_{n',n}, -\boldsymbol{v}_0 \cdot H_{n',n})$.
2. If $\sigma = 1$: parse k_1 as $(m, n, K_1^{\mathsf{fss}}, x)$. Compute $\boldsymbol{v}_1 \leftarrow \mathsf{MPFSS.FullEval}(1, K_1^{\mathsf{fss}})$ in $\mathbb{F}_q^{n'}$, and output $(x, \boldsymbol{w} \leftarrow \boldsymbol{v}_1 \cdot H_{n',n})$.

Fig. 3. PCG for subfield vector-OLE

Theorem 24. *Suppose the $(\mathcal{HW}_t, \mathbf{C}, \mathbb{F}_p)$-dual-LPN$(n', n)$ assumption holds, and that* MPFSS *is a secure multi-point FSS scheme. Then the construction G_{sVOLE} (Fig. 3) is a secure PCG for the subfield vector-OLE correlation.*

Proof. First let $\sigma = 0$. Here, in the real distribution, the adversary is given a key $\mathsf{k}_0 = (m, n, K_0^{\mathsf{fss}}, S, \boldsymbol{y})$, where $(\mathsf{k}_0, \mathsf{k}_1) \xleftarrow{\$} G_{\mathsf{sVOLE}}.\mathsf{Gen}(1^\lambda)$, as well as the expanded output $R_1 = (x, \boldsymbol{w}) \xleftarrow{\$} G_{\mathsf{sVOLE}}.\mathsf{Expand}(\mathsf{k}_1)$. We need to show that this is indistinguishable from the ideal distribution, where $R_1 \xleftarrow{\$} \mathsf{RSample}(0, R_0)$.

Recall that $\mathsf{RSample}(0, R_0)$ proceeds by sampling $x \xleftarrow{\$} \mathbb{F}_q$ and outputing $(x, \boldsymbol{w} = \boldsymbol{u}x + \boldsymbol{v})$. In the real distribution, x is also uniformly random, and from the correctness of MPFSS we have that

$$\boldsymbol{w} = \boldsymbol{v}_1 \cdot H_{n',n} = (\boldsymbol{v}_0 + x \cdot \mathsf{spread}_{n'}(S, \boldsymbol{y})) \cdot H_{n',n} = \boldsymbol{v} + x \cdot \boldsymbol{u}$$

which is identically distributed to the ideal distribution.

Next consider the case of $\sigma = 1$. We use the following sequence of games.

Game G_0. This is the real distribution, where the adversary gets $\mathsf{k}_1 = (m, n, K_1^{\mathsf{fss}}, x)$ and $R_0 = (\boldsymbol{u}, \boldsymbol{v}) \xleftarrow{\$} G_{\mathsf{sVOLE}}.\mathsf{Expand}(0, \mathsf{k}_0)$, that is, $\boldsymbol{u} = \boldsymbol{\mu} \cdot H_{n',n}$ for $\boldsymbol{\mu} \xleftarrow{\$} \mathcal{HW}_{t,n'}(\mathbb{F}_p)$ and $\boldsymbol{v} = \boldsymbol{v}_0 \cdot H_{n',n} = (\boldsymbol{v}_1 + x \cdot \boldsymbol{\mu}) \cdot H_{n',n}$.

Game G_1. Here, we compute K_1^{fss} using the MPFSS simulator $\mathsf{Sim}(1^\lambda, \cdots)$, and $\boldsymbol{v} = (\boldsymbol{v}_1 + x \cdot \boldsymbol{\mu}) \cdot H_{n',n}$. This is indistinguishable from G_0, by the security of the MPFSS.

Game G_2. Finally, here we compute $\boldsymbol{u} \xleftarrow{\$} \mathbb{F}_p^n$ instead of $\boldsymbol{\mu} \cdot H_{n',n}$, and let $\boldsymbol{v} = \boldsymbol{v}_1 \cdot H_{n',n} + x \cdot \boldsymbol{u}$. Notice that since K_1^{fss} is independent of $\boldsymbol{\mu}$, any adversary distinguishing G_1 and G_2 can be used to attack $(\mathcal{HW}_t, \mathbf{C}, \mathbb{F}_p)$-dual-LPN$(n', n)$.

Game G_2 is identical to the ideal distribution, so this completes the proof of the security property.

Finally, we show the correctness property, namely, that the outputs $(R_0, R_1) = (\boldsymbol{u}, \boldsymbol{v}, x, \boldsymbol{w})$ are computationally indistinguishable from outputs of $\mathcal{D}(1^\lambda)$. By the same reasoning as security for $\sigma = 0$, R_1 is already identically distributed to the output of $\mathsf{RSample}(0, R_0)$, so we write $\boldsymbol{w} = \boldsymbol{u}x + \boldsymbol{v}$. Denoting the uniform distribution on $\mathbb{F}_p^n$ by U_p^n, we then use the following sequence of hops:

$$
\begin{aligned}
(\boldsymbol{u}, \boldsymbol{v}, x, \boldsymbol{w}) &= (\boldsymbol{\mu} \cdot H_{n',n}, (\boldsymbol{v}_1 + x \cdot \boldsymbol{\mu}) \cdot H_{n',n}, x, \boldsymbol{u}x + \boldsymbol{v}) \\
&\overset{\text{c}}{\approx} (\boldsymbol{\mu} \cdot H_{n',n}, (U_q^{n'} + x \cdot \boldsymbol{\mu}) \cdot H_{n',n}, x, \boldsymbol{u}x + \boldsymbol{v}) && (3) \\
&\overset{\text{s}}{\approx} (\boldsymbol{\mu} \cdot H_{n',n}, U_q^n, x, \boldsymbol{u}x + \boldsymbol{v}) && (4) \\
&\overset{\text{c}}{\approx} (U_p^n, U_q^n, x, \boldsymbol{u}x + \boldsymbol{v}) && (5) \\
&\equiv \mathcal{D}(1^\lambda)
\end{aligned}
$$

where (3) follows from the pseudorandomness of the MPFSS outputs (cf. Remark 2). Hop (4) holds because the LPN assumption implies from Remark 8 that $H_{n',n}$ must be full-rank with overwhelming probability, so preserves uniformity when multiplying by a uniform vector. Finally, (5) also holds due to the pseudorandomness of the LPN assumption.

5.1.1 Application to Correlated OT

Subfield VOLE immediately gives a PCG for *correlated OT* (or Δ-OT). This is a batch of 1-out-of-2 OTs where the sender's strings are of the form $(w_i, w_i \oplus \Delta)$ for some fixed string Δ, and is the main building block in practical MPC protocols such as TinyOT [NNOB12] and authenticated garbling [WRK17a, WRK17b].

To obtain correlated OT, we run subfield VOLE with $p = 2$ and $q = 2^r$, so the VOLE sender obtains $u_i \in \mathbb{F}_2, v_i \in \mathbb{F}_{2^r}$, while the VOLE receiver gets $x \in \mathbb{F}_{2^r}$ and $w_i = x \cdot u_i + v_i$, for $i = 1, \ldots, n$. Now switching the roles of sender and receiver, the VOLE sender can be seen as an OT receiver with choice bit u_i and string v_i. This gives us a correlated OT, since the OT sender (formerly VOLE receiver) can compute the strings $(w_i, w_i + x)$, and we have $v_i = w_i$ if $u_i = 0$ and $v_i = w_i + x$ if $u_i = 1$.

5.1.2 Application to Matrix Multiplication

Our construction for subfield VOLE can alternatively be seen as a PCG for *tensor product*: writing $x \in \mathbb{F}_q$ as $\boldsymbol{x} = (x_1, \ldots, x_r) \in \mathbb{F}_p^r$, and $\boldsymbol{u} = (u_1, \ldots, u_n) \in \mathbb{F}_p^n$, sVOLE computes secret shares of $\boldsymbol{x} \otimes \boldsymbol{u}$, that is, $x_i \cdot u_j$ for every $(i, j) \in [r] \times [n]$. This allows evaluation of secret-shared tensor products in 2-PC, which can in turn be used for matrix multiplication.

The seed size scales linearly in r, but this still improves upon the naive way of using r PCGs for VOLE over $\mathbb{F}_p$; the latter approach (with the VOLE from [BCGI18]) has seed size $O(rt \cdot (\lambda \log n + \log p))$ bits, whereas we reduce this to $O(t \cdot (\lambda \log n + r \log p))$ bits, saving at least a $\log n$ factor when $\log p = O(\lambda)$.

5.2 PCG for Random Oblivious Transfer

In Fig. 4, we use the G_{sVOLE} PCG to construct a PCG for the random oblivious transfer correlation. Given the above observation that subfield VOLE implies correlated OT, this is straightforward, as we can apply the OT extension technique of Ishai et al. [IKNP03], which converts

Construction G_{OT}

PARAMETERS:

- Security parameter 1^λ, integers n, $q = p^r = \lambda^{\omega(1)}$.
- An $\mathbb{F}_p$-correlation-robust function $\mathsf{H} : \{0,1\}^\lambda \times \mathbb{F}_q \to \{0,1\}^\lambda$.
- The subfield-VOLE PCG $(G_{\mathsf{sVOLE}}.\mathsf{Gen}, G_{\mathsf{sVOLE}}.\mathsf{Expand})$

CORRELATION: Outputs $(R_0, R_1) = \big(\{(u_i, w_{i,u_i})\}_{i\in[n]}, \{w_{i,j}\}_{i\in[n],j\in[p]}\big)$, where $w_{i,j} \xleftarrow{\$} \{0,1\}^\lambda$ and $u_i \xleftarrow{\$} \{1,\ldots,p\}$, for $i \in [n], j \in [p]$.

GEN: On input 1^λ, output $(\mathsf{k}_0, \mathsf{k}_1) \leftarrow G_{\mathsf{sVOLE}}.\mathsf{Gen}(1^\lambda, n, p, q)$.

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

1. If $\sigma = 0$: compute $(\boldsymbol{u}, \boldsymbol{v}') \leftarrow G_{\mathsf{sVOLE}}.\mathsf{Expand}(\sigma, \mathsf{k}_\sigma)$, where $\boldsymbol{u} \in \mathbb{F}_p^n, \boldsymbol{v}' \in \mathbb{F}_q^n$. Compute
$$v_i \leftarrow \mathsf{H}(i, v_i') \quad \text{for } i = 1,\ldots,n$$
and output (u_i, v_i).
2. If $\sigma = 1$: compute $(x, \boldsymbol{w}') \leftarrow G_{\mathsf{sVOLE}}.\mathsf{Expand}(\sigma, \mathsf{k}_\sigma)$, where $x \in \mathbb{F}_q, \boldsymbol{w}' \in \mathbb{F}_q^n$. Compute
$$w_{i,j} \leftarrow \mathsf{H}(i, w_i' - j \cdot x) \quad \text{for } i = 1,\ldots,n, \forall j \in \mathbb{F}_p$$
and output $\{w_{i,j}\}_{i,j}$.

Fig. 4. PCG for n sets of 1-out-of-p random OT

correlated OTs into random OTs using a suitable hash function. We extend this in a natural way to generate 1-out-of-p random OTs using subfield VOLE over $\mathbb{F}_p$. Note that for security when applying the hash function, we now need $q = \lambda^{\omega(1)}$.

We use the following generalization of a correlation robust function over $\mathbb{F}_p$. As recently shown in [GKWY19], this can be instantiated with fixed-key AES modeled as a random permutation when $p = 2$.

Definition 25 ($\mathbb{F}_p$-correlation robust function). *Let $n = \mathsf{poly}(\lambda)$ and $t_1,\ldots,t_n,x$ be uniformly sampled from $\mathbb{F}_p^r$, where $p^r = \lambda^{\omega(1)}$. Then, $\mathsf{H} : \{0,1\}^\lambda \times \mathbb{F}_p^r \to \{0,1\}^\lambda$ is* $\mathbb{F}_p$-correlation robust *if the distribution*

$$\big(t_1,\ldots,t_n, \{\mathsf{H}(1, t_1 - j\cdot x),\ldots,\mathsf{H}(n, t_n - j\cdot x)\}_{j\in\mathbb{F}_p\setminus\{0\}}\big)$$

is computationally indistinguishable from uniform on $\mathbb{F}_p^{rn} \times \{0,1\}^{\lambda(p-1)n}$.

Theorem 26. *Suppose that H is an $\mathbb{F}_p$-correlation robust hash function and G_{sVOLE} is a secure PCG. Then the silent OT construction (Fig. 4) is a secure PCG for the random 1-out-of-p OT correlation.*

Proof. We start by showing the correctness property. First, from the correctness of G_{sVOLE} we have

$$v_i = \mathsf{H}(i, v_i') = \mathsf{H}(i, w_i' - u_i \cdot x) = w_{i,u_i}$$

as required. Indistinguishability of the outputs from OT_p^n follows first from indistinguishability of the VOLE outputs (u_i, v_i', x, w_i'), and secondly by a standard reduction to the $\mathbb{F}_p$-correlation robustness property of H.

$$u_i, v_i, w_{i,j} = u_i, \mathsf{H}(i, w_i' - u_i \cdot x) \overset{\mathrm{c}}{\approx} (U, \mathsf{H}(i,))$$

We now consider the security property, for the case $\sigma = 0$. Here, the real distribution consists of the seed k_0 and the sender's outputs $w_{i,j}$, for $i = 1,\ldots,n$ and $j \in \mathbb{F}_p$. From correctness we have

that for $\boldsymbol{u}, \boldsymbol{v}' \leftarrow G_{\mathsf{sVOLE}}.\mathsf{Expand}(0, \mathsf{k}_0)$, it holds that $\mathsf{H}(i, v'_i) = w_{i,u_i}$. From the security property of G_{sVOLE}, we can replace all the sender's outputs $w_{i,j}$, except for w_{i,u_i}, with ones computed using uniform values x, w'_i, instead of from $G_{\mathsf{sVOLE}}.\mathsf{Expand}$. These are then indistinguishable from uniform under the $\mathbb{F}_p$-correlation robustness of H.

When $\sigma = 1$, the real distribution contains the seed k_1 and the receiver's outputs u_i, v_i. As before, from the correctness property we have that $v_i = w_{i,u_i}$, where $w_{i,j}$ is computed from k_1 via $G_{\mathsf{sVOLE}}.\mathsf{Expand}$ and the hash function. We only need to show that u_i is uniform, which follows directly from the security property of G_{sVOLE} when $\sigma = 1$.

5.3 From a PCG to Silent OT Extension

To construct an OT extension protocol, we can use 2-PC to securely compute the Gen algorithm of G_{OT}, and then have each party locally expand its output using $G_{\mathsf{OT}}.\mathsf{Expand}$. Applying Theorem 19 from Section 4.3, this realizes a *corruptible* form of the ideal functionality for random oblivious transfer, where corrupt parties may influence their random outputs.

To do this efficiently with semi-honest security, we use the black-box protocol of Doerner and shelat [Ds17] (also used in [BCGI18]) for setting up distributed point function keys. For a single point function of domain size n, this requires $O(\log n)$ OTs on $O(\lambda)$-bit strings, giving $O(t \log n)$ OTs for a multi-bit point function. Implementing each OT with (non-silent) OT extension [IKNP03] costs $O(\lambda)$ bits of communication, plus a setup phase of λ base OTs. Putting this together, we obtain the following.

Theorem 27. *Suppose the $(\mathcal{HW}_t, \mathbf{C}, \mathbb{F}_p)$-dual-$\mathsf{LPN}(n', n)$ assumption holds, and an $\mathbb{F}_p$-correlation robust hash function exists. Then there is a protocol that uses $O(\lambda)$ 1-out-of-2 OTs to realize n instances of random 1-out-of-p OT with semi-honest security, using $O(t\lambda \log n) + \mathsf{poly}(\lambda)$ bits of communication.*

We remark that this gives OT with *sublinear communication* when $t = o(n/(\lambda \log n))$, which translates to an instance of LPN with noise rate $1/\omega(\lambda \log n)$. If the matrix $H_{n',n}$ in G_{sVOLE} is uniformly random, the computational complexity is dominated by $O(n' \cdot n)$ arithmetic operations; using more structured matrices based on LDPC codes or quasi-cyclic codes, we get respective costs of $O(n')$ or $\tilde{O}(n')$ arithmetic and PRG operations.

5.3.1 Concrete Efficiency

In Section A of the Appendix, we analyze these costs more concretely and give a breakdown of the communication complexity, as well as some approximate runtime estimates based on the cost of the main operations. For example, for $n \leq 2^{22}$ OTs, the PCG seed size is under 10kB and requires less than 30kB of communication to create with the distributed setup procedure. After setup, we estimate that these seeds can be expanded into 16MB of OTs on 128-bit strings at a rate of around 1 million per second, or 2 million per second when expanding to 1MB, using a single core of a CPU on a modern laptop. When including the distributed setup procedure, in these two cases we get an amortized communication complexity of just 2.6 and 0.2 bits per OT, respectively.

5.4 Applications of Silent OT Extension

Protocols generating (pseudo)random OT have a wide range of applications. In this section we show how our silent OT extension can plugged in to obtain a batch oblivious pseudorandom function (OPRF) with under 1 bit of communication per OPRF evaluation on a random input, which is useful for private set intersection protocols. Further, we sketch how to construct a NIZK proof system in the preprocessing model, where the setup cost is independent of both the number and size of statements.

Batch oblivious PRF. Our random 1-out-of-p OT generator can even be used when p is exponentially large, provided the sender only needs to obtain polynomially many outputs. This type of random OT can be viewed as a form of batch, one-time oblivious PRF: each string $w_{i,j}$ output by the sender can be seen as a PRF evaluation $F(k_i, j)$, where k_i is a key implicitly defined by the sender's randomness. The receiver learns a random u_i and the value $w_{i,u_i} = F(k_i, u_i)$. We can also allow the receiver to choose its evaluation point u_i, with an additional $\lambda = \log p$ bits of communication. In [KKRT16], this type of OPRF was used to improve the efficiency of private set intersection protocols. Applying our PCG, we get a highly efficient batch related key OPRF usable in PSI, where each OPRF evaluation at a chosen point costs around λ bits of communication. This reduces the overall communication in the PSI protocol from [KKRT16] by around a factor of two.

Resuable NIZKs from LPN. In [BCGI18], it was shown how to use a PCG for vector-OLE to build a reusable NIZK in the preprocessing model. NIZK in the preprocessing model relaxes the standard NIZK definition by allowing the prover and the verifier to interact during a preprocessing phase, to generate a respective proving key and verification key. The construction from the vector-OLE PCG obtained a reusable preprocessing NIZK, where the preprocessing cost is *independent* of the number of theorems to be proven, if this is a priori bounded, under an arithmetic version of LPN.

In Section A of the Appendix, we observe that we can obtain an alternative NIZK construction by using our PCG for random OT, which brings two main advantages:

- It only requires the standard LPN assumption over $\mathbb{F}_2$, while the NIZK of [BCGI18] must rely on a generalization of LPN to exponentially large fields.
- The preprocessing phase is independent of both the number of theorems to be proven, and the size of these theorems. In comparison, the preprocessing phase in [BCGI18] is independent of the number of theorems to be proven, but grows linearly with a bound on the size of each statement. This is because we use OT instead of VOLE, so are no longer restricted to a batch setting where the same query must be reused for many statements.

On the down side, our OT-based NIZK protocols do not enjoy some of the efficiency features of the VOLE-based constructions from [BCGI18]. The latter support NIZK for an NP-relation represented by an *arithmetic* circuit of size s over $\mathbb{F}$, where the online computation of both the prover and the verifier consists of $O(s)$ arithmetic operations, and with $O(1/|\mathbb{F}|)$ soundness error. We do not know how to achieve this using the current OT-based approach.

6 PCG for Constant-Degree Correlations from LPN

In this section, we describe a pseudorandom correlation generator for arbitrary constant-degree correlations, from the dual LPN assumption over large fields. We first describe the construction for the case of bilinear correlations. More precisely, we consider the following type of additive correlations: the party P_σ receives pseudorandom vectors $(\boldsymbol{x}_\sigma, \boldsymbol{z}_\sigma)$ such that $B(\boldsymbol{x}_0, \boldsymbol{x}_1) = \boldsymbol{z}_0 + \boldsymbol{z}_1$, where B is a bilinear function. We note that this type of correlation generalizes naturally to the setting where the entries $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$ are additively shared between the parties (instead of being respectively known to one party), see Section C.

Theorem 28. *Suppose the* $(\mathcal{HW}_t, \mathbf{C}, \mathbb{F}_p)$-dual-LPN$(n', n)$ *assumption holds, and that* MPFSS *is a secure multi-point FSS scheme. Then the construction* G_{bil} *(Fig. 5) is a secure PCG for general bilinear correlations.*

Correctness follows by inspection, using the correctness of the MPFSS, and the bilinearity of the tensor product, and the security analysis is essentially indentical to the analysis of the dual vector-OLE generator described in [BCGI18] (see also Section 5.1).

Construction G_{bil}

PARAMETERS: $1^\lambda, n, n', t, p \in \mathbb{N}$, where $n' > n$. A code generation algorithm $\mathbf{C}$ and $H_{n',n} \xleftarrow{\$} \mathbf{C}(n', n, \mathbb{F}_p)$. A bilinear function $B_{\boldsymbol{c}} : (\boldsymbol{\alpha}, \boldsymbol{\beta}) \to \boldsymbol{c} \cdot (\boldsymbol{\alpha} \otimes \boldsymbol{\beta})^\intercal$, where $\otimes$ denotes the tensor product.

Gen: On input 1^λ:

1. Pick two random size-t subsets (S_0, S_1) of $[n']$, sorted in increasing order.
2. Pick two random vector $(\boldsymbol{y}_0, \boldsymbol{y}_1) \in (\mathbb{F}_p^t)^2$.
3. Compute $(K_0^{\mathsf{fss}}, K_1^{\mathsf{fss}}) \xleftarrow{\$} \mathsf{MPFSS.Gen}(1^\lambda, f_{S_0 \times S_1, \boldsymbol{y}_0 \otimes \boldsymbol{y}_1})$.
4. Let $\mathsf{k}_0 \leftarrow (n, K_0^{\mathsf{fss}}, S_0, \boldsymbol{y}_0)$ and $\mathsf{k}_1 \leftarrow (n, K_1^{\mathsf{fss}}, S_1, \boldsymbol{y}_1)$.
5. Output $(\mathsf{k}_0, \mathsf{k}_1)$.

Expand: On input $(\sigma, \mathsf{k}_\sigma)$, parse k_σ as $(n, K_\sigma^{\mathsf{fss}}, S_\sigma, \boldsymbol{y}_\sigma)$. Set $\boldsymbol{\mu}_\sigma \leftarrow \mathsf{spread}_{n'}(S_\sigma, \boldsymbol{y}_\sigma)$ in $\mathbb{F}_p^{n'}$ and $\boldsymbol{x}_\sigma \leftarrow \boldsymbol{\mu}_\sigma \cdot H_{n',n}$. Compute $\boldsymbol{v}_\sigma \leftarrow \mathsf{MPFSS.FullEval}(\sigma, K_\sigma^{\mathsf{fss}})$ in $\mathbb{F}_p^{(n')^2}$ and set $\boldsymbol{z}_\sigma \leftarrow -\boldsymbol{c} \cdot (\boldsymbol{v}_\sigma \cdot (H_{n',n} \otimes H_{n',n}))^\intercal$. Output $(\boldsymbol{x}_\sigma, \boldsymbol{z}_\sigma)$.

Fig. 5. PCG for Bilinear Correlations

Efficiency. Instantiating the MPFSS 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 $\tilde{O}(t^2)$ field elements over a large field ($\log_2 |\mathbb{F}| = O(\lambda)$). Expanding the seed involves $(tn')^2$ PRG evaluations and $O(n \cdot n')^2 = O(n^4)$ arithmetic operations.

Generalization. The scheme G_{bil} immediately generalizes to a PCG for arbitrary constant-degree polynomials,[13] where the size of the shares grows as $\tilde{O}(t^d)$ and the computational complexity is $\tilde{O}((tn')^d + (nn')^d)$. It allows two parties to locally compute, given the shares, additive shares of $(\boldsymbol{r}, P(\boldsymbol{r}))$, where $\boldsymbol{r}$ is pseudorandom (under LPN) and P is a degree-d multivariate polynomial over $\mathbb{F}$.

To see this, notice that we can replace $\boldsymbol{y}_0 \otimes \boldsymbol{y}_1$ in **Gen** with $\otimes_d \boldsymbol{y} = \boldsymbol{y} \otimes \cdots \otimes \boldsymbol{y}$, where $\otimes_d \boldsymbol{y}$ denotes the tensor product of $\boldsymbol{y}$ with itself d times (that is, the list of all degree-d monomials of $\boldsymbol{y}$). The parties can then compute shares of all degree-d terms in $P(\boldsymbol{r})$ for a random $\boldsymbol{r}$; to obtain shares of $\boldsymbol{r}$ and the lower-degree terms, we extend the MPFSS values to include $(\boldsymbol{y}, \boldsymbol{y} \otimes \boldsymbol{y}, \cdots, \otimes_{d-1} \boldsymbol{y})$ as well as $\otimes_d \boldsymbol{y}$.

Corollary 29. *Suppose the $(\mathcal{HW}_t, \mathbf{C}, \mathbb{F}_p)$-dual-LPN$(n', n)$ assumption holds, and that* MPFSS *is a secure multi-point FSS scheme. Then there exists a secure PCG for general constant-degree correlations, with share size $\tilde{O}(t^d)$ and computational complexity $O((n \cdot n')^d)$.*

In particular, using $n' = O(n)$, we get:

Corollary 30. *Assuming the standard LPN assumption over $\mathbb{F}_p$ with noise rate $r = o(n^{1/d-1})$ and linear number of samples, there exists a PCG for general degree-d polynomials, with sublinear share size (in the output size n) and polynomial computation.*

6.1 HSS and Secure Computation for Constant-Degree Polynomials from LPN

We observe that by Section 4.4 (Generic Construction of HSS from PCG), the above construction directly gives rise to a homomorphic secret sharing scheme for degree-d multivariate polynomials. Therefore, we get:

[13] In fact, assuming that dual-LPN has $2^{O(t)}$ security (which is in line with the best known attacks), t can be taken as small as $\omega(\log \lambda)$, in which case the degree $d(\lambda)$ of the polynomial can be larger, up to $O(\log \lambda / \log \log \lambda)$. The shares are still of polynomial size $\tilde{O}(t^d)$, although the computational cost $O((n \cdot n')^d)$ is slightly superpolynomial.

Corollary 31. *Suppose the* $(\mathcal{HW}_t, \mathbf{C}, \mathbb{F}_p)$-dual-LPN$(n', n)$ *assumption holds, and that* MPFSS *is a secure multi-point FSS scheme. Then there exists a secure HSS for general degree-d multivariate polynomials over* $\mathbb{F}$*, with share size* $n + \tilde{O}(t^d)$ *and computational complexity* $O((n \cdot n')^d)$*.*

Plugging this new HSS construction into the result of [BGI16a], we immediately obtain new results regarding secure computation from the LPN assumption:

Corollary 32. *Suppose the* $(\mathcal{HW}_t, \mathbf{C}, \mathbb{F}_p)$-dual-LPN$(n', n)$ *assumption holds, and that* MPFSS *is a secure multi-point FSS scheme. Then there exists a 2-party secure computation protocol with semi-honest security for general degree-d multivariate polynomials over* $\mathbb{F}$*, with communication* $\tilde{O}(n + t^d)$ *and computational complexity* $O((n \cdot n')^d)$*.*

In particular, applying the above corollary to layered circuits, we obtain a generic secure two-party protocol from LPN with communication smaller than the circuit size:

Corollary 33. *Suppose the* $(\mathcal{HW}_t, \mathbf{C}, \mathbb{F}_p)$-dual-LPN$(n', n)$ *assumption holds, and that* MPFSS *is a secure multi-point FSS scheme. Then for any constant c, there exists a 2-party secure computation protocol with semi-honest security for arbitrary layered circuits of size s, with total communication bounded by* s/c*, and computational complexity bounded by* $s \cdot \lambda^{2^{O(c)}}$*.*

7 PCG Constructions from Groups and Lattices

In this section we give PCG constructions for a range of correlations, starting from the generic construction of Section 4.4. In particular, we will describe PCGs for the generation of bilinear correlations from groups, and PCGs for so-called authenticated Beaver triples from lattices.

We start this section by presenting two specialized low-degree PRG constructions that will serve as crucial building blocks for our high-end constructions. Note that a straightforward choice like Goldreich's low-degree PRG [Gol00, MST03] turns out to be not suitable. (For more discussion we refer to Section 7.3.)

In Section 7.3 we deviate from the generic construction by splitting up the evaluation of the PRG and the evaluation of the function f itself, which is captured in the notion of *compressible* HSS.

To give a high-level idea, note that one can roughly think of the underlying HSS scheme to consist of two levels, where multiplication of a level-1 share with a level-2 share yields a level-2 share. The idea now is to start with *compressed* level-1 and level-2 shares, which can be expanded to long pseudorandom level-1 and level-2 shares, on which subsequently f is evaluated via the HSS itself. In other words, the PRG is evaluated *on the shares directly* and not via the HSS operation (see also Section 7.2).

In Section 7.4 we follow the generic construction of Section 4.4 more closely, building on the MQ-based PRG described in the following and several variants of lattice-based HSS schemes.

7.1 Pseudorandom Generators from MQ and LPN

Let $\mathcal{R}$ be a ring and $n \in \mathbb{N}$. By $\mathcal{U}^n(\mathcal{R})$ we denote the uniform distribution on $\mathcal{R}^n$. Recall that we consider distributions $\mathcal{D}^\ell$ over a ring $\mathcal{R}^\ell$ and write $X \leftarrow \mathcal{D}^\ell(\mathcal{R})$ or simply $X \leftarrow \mathcal{D}^\ell$ (if $\mathcal{R}$ is clear from the context) to denote sampling from $\mathcal{R}^\ell$ via $\mathcal{D}^\ell$. Further, for a matrix distribution $\mathcal{M}$ over a ring $\mathcal{R}^{n\times m}$ we write $\mathcal{M}(n, m, \mathcal{R})$ to make the parameters explicit.

Remark 34 (PRG from MQ). Let $\mathcal{R}$ be a ring, and $\ell, n \in \mathbb{N}$. Let $\mathcal{M}$ be a distribution over $\mathcal{R}^{\ell^2\times n}$ and $\mathbf{M} \xleftarrow{\$} \mathcal{M}(\ell^2, n, \mathcal{R})$. We assume that for an appropriate choice of parameters

$$\mathsf{PRG}_{\mathsf{MQ}} \colon \mathcal{R}^\ell \to \mathcal{R}^n, r \mapsto \mathbf{M}^\top \cdot (r \otimes r)$$

is a PRG. We say $\mathcal{M}(\ell^2, n, \mathcal{R})$ has *sparsity* ρ, if for every matrix $\mathbf{M}$ in the image of $\mathcal{M}(\ell^2, n, \mathcal{R})$, the number of non-zero entries in any column of $\mathbf{M}$ is at most ρ.

Note that if we choose $\mathcal{M}(\ell^2, n, \mathcal{R}) = \mathcal{U}^{\ell^2 \times n}(\mathcal{R})$, the above assumption equals the MQ assumption of [MI88, Wol05, AHI+17]. While multivariate public-key cryptography has a long history of schemes being built then broken, we stress that the MQ assumption itself (which states that it is infeasible to solve a random system of quadratic equations) is believed to be a conservative assumption (in particular, the pseudorandomness of the MQ-based PRG reduces to the conjectured *one-wayness* of solving a random system of quadratic assumptions [BGP06]), and underlies the security of plausible and well-studied primitives in minicrypt (such as signatures scheme, or the stream cipher QUAD [BGP06]). Existing attacks on multivariate public-key cryptosystems all exploited the fact that the security of these systems did not in fact reduce to the MQ assumption. Furthermore, variants of MQ with a sparse matrix were considered several time as a natural optimization of MQ-based schemes [BCJ07, LLY08], and the resistance of the variant with sparse matrix against classical attacks was analyzed in [BCJ07, DyY07].

Remark 35 ($\mathcal{D}^\ell(\mathcal{R})$-PRG from LPN). Let $\mathcal{R}$ be a ring, and $\ell, k, c, \tau, n \in \mathbb{N}$, such that $\ell = \tau ck$. Let $\mathcal{M}$ be a distribution on $\mathcal{R}^{\tau^c \times n}$ and $\mathbf{M} \xleftarrow{\$} \mathcal{M}(\tau^c, n, \mathcal{R})$. Let

$$\mathsf{PRG}_{\mathsf{LPN}} \colon (\mathcal{R}^\tau)^{ck} \to \mathcal{R}^n, (r_1, \ldots, r_{ck}) \mapsto \mathbf{M}^\top \cdot \sum_{i=0}^{k-1} r_{1+i\cdot c} \otimes \cdots \otimes r_{c+i\cdot c},$$

where $r_i \in \{0,1\}^\tau$ for all $1 \le i \le ck$. We assume that for appropriate choice of parameters $\mathsf{PRG}_{\mathsf{LPN}}$ is a $\mathcal{D}^\ell(\mathcal{R})$-PRG, where $\mathcal{D}^\ell$ is the distribution returning vectors that have *exactly* one non-zero entry (chosen uniformly at random in $\mathcal{R}\backslash\{0\}$) in every block of length τ.

Note that $\sum_{i=0}^{k-1} r_{1+i\cdot c} \otimes \cdots \otimes r_{c+i\cdot c}$ yields a random vector in $\mathcal{R}^{\tau^c}$ with exactly k non-zero entries. Therefore, the above assumption corresponds to the $(\mathsf{Ber}_k(\mathcal{R})^{\tau^c}, \mathcal{M}, \mathcal{R})$-dual-LPN$(\tau^c, n)$ assumption, where $\mathsf{Ber}_k(\mathcal{R})^{\tau^c}$ is the distribution returning vectors in $\mathcal{R}^{\tau^c}$ with exactly k non-zero entries.

7.2 Semi-Generic PCG Construction from Compressible HSS

As mentioned before, while our group-based constructions described in the following section build upon the generic approach, they exploit the specific structure of the underlying HSS to achieve better parameters: the generic construction would require a degree-4 HSS already to generate bilinear correlations using the LPN-based PRG. In contrast, our constructions achieve the same expressivity starting only from degree-2 HSS, building upon their homomorphic properties. The essence of this approach is captured in the following corollary. For a more formal treatment of compressible HSS we refer to Section C.3.

Corollary 36 (Informal). *Let $\mathcal{R}$ be a ring and $n, m, \tau, c, k \in \mathbb{N}$. Let $\mathcal{F} \subseteq \{f \colon \mathcal{R}^n \to \mathcal{R}^m\}$ be a family of functions. Let $\mathsf{HSS} = (\mathsf{HSS.Gen}, \mathsf{HSS.Share}, \mathsf{HSS.Eval})$ be an homomorphic secret sharing scheme for $\mathcal{F}$ with overhead O_{HSS} that supports degree-c compression of the shares (i.e. compressed shares can be decompressed via degree-c homomorphic operations preserving the respective share-level). Then, under the $(\mathsf{Ber}_k(\mathcal{R})^{\tau^c}, \mathcal{M}, \mathcal{R})$-dual-LPN$(\tau^c, n)$ assumption, there exists a PCG for the correlation generator $\mathcal{C}_{\mathcal{F}}$ with key-length upper bounded by $\tau ck \cdot O_{\mathsf{HSS}}$.*

7.3 Group-Based PCG for Bilinear Correlations: an Overview

In this section, we describe a construction of a PCG for general bilinear correlations, building upon the group-based HSS scheme of [BGI16a, BGI17, BCG+17]. Since our construction is quite involved, and the full description would not fit in the body of this paper, we only provide a high-level overview here; the detailed construction is given in Appendices C, D, and E. The high-level

idea of our construction is best explained by the compressible HSS abstraction, outlined in the "Overall Methodology" section of the introduction. To construct a PCG for bilinear correlations, we rely on the group-based HSS of [BGI16a, BGI17, BCG+17]. While this HSS can support evaluation of arbitrary branching programs, this comes at a very high computational cost, which makes it concretely impractical. Note that the full version of [BCG+17] describes a construction of PCG (which is called "cryptographic capsule" in their terminology) building upon the group-based HSS together with Goldreich's low-degree PRG [Gol00, MST03]. While the authors did not provide concrete efficiency estimations for this construction, our rough calculations show that it is entirely impractical: Goldreich's PRG requires very large seeds (at least 2^{14} bit long to achieve a stretch of only $n^{1.45}$ according to the recent study of [CDM+18]), and the HSS must be employed in a parameter regime where encoding *each bit* of the seed requires $O(\lambda)$ ElGamal ciphertext (e.g. 160 ciphertexts, when targetting 80 bits of security). Furthermore, the group-based HSS has an inverse failure probability per output which scales superlinearly with the encoding size. We estimate that an optimized implementation of their scheme would require a few hours of runtime to generate each output, and the PCG seed remains larger than the total amount of correlation generated (in fact, larger than the cost of a naive interactive generation of the material) for any feasible value of n.

Our improvement stems from two key observations. First, when the HSS is restricted to evaluating only *bilinear* function (as opposed to general branching programs), the encodings in the group-based HSS can be considerably smaller, a single ElGamal ciphertext per bit of the seed (this was observed in [BCG+17]). This in itself would not suffice, since generating the bilinear correlation requires first evaluating a PRG, then computing a bilinear function on top of that, which requires HSS of degree at least 4 (since a PRG must have degree at least 2 in its inputs). Our second key observation is that by relying on the BGN cryptosystem instead of ElGamal (which requires pairings but allows homomorphic evaluation of degree-3 functions on the ciphertext), extending the HSS-encoded seed into long pseudorandom encodings can be done directly on the encoding, without requiring any of the HSS operations. More precisely, the group-based HSS requires two types of encodings: a level-1 encoding of a vector $\boldsymbol{x}$, which is essentially a bitwise ElGamal encryption (when restricting our attention to HSS for degree-2 functions), and a level-2 encoding of a vector $\boldsymbol{y}$, which is essentially a pair of additive shares of both $\boldsymbol{y}$ and $\mathsf{sk} \cdot \boldsymbol{y}$, where sk is the ElGamal secrey key; the HSS operations allows the parties to locally compute shares of $B(\boldsymbol{x}, \boldsymbol{y})$ from these encodings, for any bilinear function B.

Using the BGN encryption scheme [BGN05], the level-1 encoding of a pseudorandom vector $\boldsymbol{x}$ of length n can be locally compressed by relying on the LPN-based PRG described in Section 4.4: the compressed encoding contains $2k$ component-wise BGN encryptions of random length-$\sqrt{n}$ unit vectors $\boldsymbol{u}_i, \boldsymbol{v}_i$, for a total size of $2k\sqrt{n}$ BGN ciphertexts, where k is a parameter of ther underlying LPN assumption which denotes the number of noisy coordinates in the error vector. Then, the parties can homomorphically compute BGN encryptions of $\boldsymbol{x} = M \cdot \sum_{i=1}^{k} \boldsymbol{u}_i \otimes \boldsymbol{v}_i$, where M is a public code matrix; security follows from the dual LPN assumption with code matrix M by observing that $\sum_{i=1}^{k} \boldsymbol{u}_i \otimes \boldsymbol{v}_i$ is just a uniformly random k-sparse vector. At the same time, using two parallel instances of the recent LPN-based PCG for vector-OLE of [BCGI18] allows to efficiently compress shares of $\boldsymbol{y}$ and $\mathsf{sk} \cdot \boldsymbol{y}$, where $\boldsymbol{y}$ is a pseudorandom vector and sk is a shared value, to only $O(\lambda k \log n)$ bits. Hence, we get a highly optimized PCG for bilinear functions by distributing compressed shares of level-1 and level-2 encodings of pseudorandom vectors to the parties, from which they can locally expand the compressed shares and apply the HSS operation to obtain shares of $B(\boldsymbol{x}, \boldsymbol{y})$.

In Appendix C, we formally introduce the construction following the above informal overview. In Appendix D, we discuss many optimizations that can be applied to the scheme, in particular by relying on new variants of the LPN assumption, which we introduce and analyze in the same section. Eventually, in Appendix E, we provide detailed concrete efficiency estimations for our PCG, for generating OLE correlations over small fields, together with further optimizations

tailored to this setting, and a concrete analysis of the resistance of our new assumption to a variety of attacks. Note that our PCG are already useful for generating OT correlations, since they are *programmable*, and therefore allow to generate multiparty correlations from pairwise correlations (unlike our silent OT extension protocol) – see Section 8 for the details.

A downside of the group-based HSS is that it only guarantees an imperfect correctness for the output, with an inverse-polynomial failure probability. To use the generated material in a subsequent protocols, the parties must therefore first *sanitize* the output, converting the faulty correlations into non-faulty correlations. This is non-trivial, since the position of the faulty outputs cannot be simply revealed, as it depends on secret information that should not be leaked. We apply the punctured OT strategy of [BGI17] to describe an efficient sanitization procedure, and we provide detailed concrete estimates of its efficiency in our setting. The strategy only requires adding a negligible amount of material to the PCG seed, and requires less than 3 bits of amortized communication per sanitized correlation. Eventually, building upon our new protocol for silent OT extension from Section 5, we devise an entirely new sanitization procedure, which is much more efficient and is compatible with efficient distributed seed generation.

7.3.1 Concrete Efficiency

Based on our our calculations in Appendices C, D, and E, we estimate that our group-based PCG for bilinear correlations should generate correlations at a rate of a few hundred milliseconds per (sanitized) output (e.g. about 200ms for generating one OLE correlation over a small ring, amortizing over $n > 2^{20}$ outputs), with a seed size reaching its breakeven point (where the size in bits of the seed is smaller than the number of outputs generated) for target number of correlations around 2^{24}. Since this is a fairly low number (preprocessing phases in standard MPC protocols must already generate billions of correlated values for securely evaluating moderately large functions), we believe that our result is already of practical interest. We stress again that our estimates are based on counting the number of operations and estimating the cost of each operation using benchmarks; the actual running time of an implementation might be somewhat higher due to other costs such as cache misses.

7.4 PCG from Lattices

In this section we consider constructing PCGs from lattices for generating a broader range of correlations. One example of useful correlations are authenticated Beaver triples which are used to achieve fast online computation time in multi-party protocols like [DPSZ12]. PCGs based on lattices can replace the preprocessing phase to yield protocols with very fast online time, where the players can exchange a short seed at any point of time and then expand their respective seeds silently before engaging in a secure computation. Thus, even though PCGs from lattices come with an expensive setup and slower expansion time, they are useful to obtain protocols with better overall complexity.

We focus on the use-case of generating 2-party shares of authenticated Beaver triples, that is additive shares of tuples $(a, b, ab), (a\alpha, b\alpha, ab\alpha)$, where α is a MAC-key for authentication (not known to any party in the plain). In the following we describe different lattice-based PCG constructions for generating such shared triples and provide efficiency estimates in Figure 2. For details we refer to Section F in the Appendix.

We follow the high-level approach of Section 4.4 based on homomorphic secret sharing (HSS), where the parties first jointly generate a shared PRG seed and for expansion disjointly evaluate $f_\alpha \circ \mathsf{PRG}$ on the shares, where $f_\alpha \colon (a, b) \mapsto (a, b, ab, a\alpha, b\alpha, ab\alpha)$. We instantiate the HSS scheme required in different ways. As underlying encryption scheme we use the BGV encryption scheme [BGV12]. For the most efficient instantiation we use the pseudorandom generator $\mathsf{PRG}_{\mathsf{MQ}} \colon \mathbb{Z}_p^\ell \to \mathbb{Z}_p^n$ from Remark 34 with $n = \ell^2/24$, where we choose a matrix with sparsity $\rho = 100$. Note that the choice of ρ is somewhat arbitrary and should be taken with some care, but to our knowledge

Underlying HSS	|key|	|triples|	setup	expansion	exp./triple
[BGV12]	3 GB	17 GB	≈ 20 s	8.0 h	0.16 ms
[BGV12] (iterative)	3 GB	1.6 MB/it.	≈ 20 s	10 s/it.	0.57 ms
[BGV12] (w/ packing)	6 MB	1.1 GB	< 0.1 s	900 h	280 ms
[BGV12, BKS19]	3 GB	17 GB	≈ 20 s	7.6 h	0.15 ms

Table 2. Overview of estimated efficiency of lattice-based approach to generate authenticated Beaver triples. The numbers provided are time estimates for joint seed generation (with security against semi-honest adversaries) and expansion. The numbers are based on [CS16], for [BGV12] supporting depth-4 homomorphic operations (note that depth-3 would suffice) and plaintext space modulus $p \approx 2^{128}$. As underlying encryption scheme we consider [BGV12] with plaintext space $\mathcal{R}_p \cong \mathbb{Z}_p^N$ and ciphertext space $\mathcal{R}_q^2$, where $\mathcal{R} := \mathbb{Z}[X]/(X^N + 1)$. The runtime estimates are based on NFLLib [ABG+16] with $\log q \approx 744$ and $N = 2^{14}$. Our results: We can expand correlated seeds of size '|key|' to 128-bit authenticated multiplication triples of total size '|triples|'. As PRG we employ $\mathsf{PRG}_{\mathsf{MQ}} \colon \mathbb{Z}_p^\ell \to \mathbb{Z}_p^{\ell^2/24}$, based on the $\mathcal{M}(\ell^2, \ell^2/24, \mathcal{R}_p)$-MQ assumption with sparsity $\rho = 100$ (here, $\ell = 2^9$ for all rows w/o packing and $\ell = N$ for the row w/ packing). The number of (maximal) obtained triples is $\ell^2 \cdot N/24$ for the rows with naive ciphertext packing (including the iterative version) and $N^2/24$ for the row with smart packing. Setup requires communication of roughly size |key| per party. We ignore small contributions like setting up the public key and generating suitable shares of the MAC key α, as computation and communication are dominated by generation and distribution of encryptions of the PRG seeds.

does not give rise to any attacks: While algebraic attacks either do not profit from sparsity at all (like the Groebner basis attack) or not significantly (like the XL attack), SAT solvers, which indeed heavily take advantage of sparsity, are still far from feasible for our choice of parameters.

In [DHRW16, BKS19] it is shown how to construct an HSS directly from a somewhat homomorphic encryption scheme, when the underlying encryption scheme additionally supports distributed decryption (i.e. decryption with additive shares of the secret key yields additive shares of the plaintext). We give a more detailed explanation in Theorem 55 in the Appendix. Our first PCG is based on this construction (instantiated with the encryption scheme of Brakerski et al. [BGV12]). On a high level, the resulting PCG is as follows.

Seed generation. The key generation algorithm of the underlying encryption scheme is called and additive shares of the secret key generated. Further, a MAC key $\alpha \in \mathbb{Z}_p$ and PRG seeds $r_a, r_b \xleftarrow{\$} \mathbb{Z}_p^\ell$ are chosen uniformly at random and encrypted (componentwise). The parties each obtain the public key, their respective share of the secret key and *all* encryptions as PCG seed. Public key and secret key shares of the encryption scheme and the encryption of the MAC key α can be reused across many instances.

Expansion. Both parties homomorphically evaluate $F_\alpha(\mathsf{PRG}(r_a), \mathsf{PRG}(r_b))$ on the ciphertexts (via the homomorphic operations of the underlying encryption scheme), where $F_\alpha \colon \mathbb{Z}_p^n \times \mathbb{Z}_p^n \to (\mathbb{Z}_p^n)^6$ corresponds to evaluating f_α componentwise on each of the n input tuples. Finally, each party decrypts the result with their respective share of the secret key.

The described approach would allow expanding 2ℓ ciphertexts into n shares of authenticated Beaver triples (over $\mathbb{Z}_p$). Additionally, we employ naive ciphertext packing [SV14]: Instead of encrypting a single $\mathbb{Z}_p$-element at a time one can 'pack' N $\mathbb{Z}_p$-elements into each ciphertext (where $N = 2^{14}$ is the dimension of the plaintext space over $\mathbb{Z}$). This way, starting with 2ℓ ciphertexts, expansion yields $N \cdot n$ shared authenticated Beaver triples. For more details we refer to Section F in the Appendix.

The second approach is based on the observation that not all authenticated Beaver triples have to be computed at once, employing the sparsity of the MQ-matrix. This is particularly desirable, as it allows computing correlations incrementally, only when needed. Note though that the given iterative approach is somewhat limited, as it is still restricted to sub-quadratic stretch and allows only to compute N Beaver triples at a time (due to naive ciphertext packing).

The third approach uses ciphertext packing more smartly, by letting the different 'slots' of the ciphertexts interact with each other. Here, starting with a single ciphertext (holding 2^{14}

plaintexts), we achieve almost quadratic stretch to $2^{28}/24$ triples. The better expansion rate (allowing to go from N to $\approx N^2$ instead of from ℓN to $\approx \ell^2 N$ triples), comes at a cost: Due to the high computational costs introduced by key switching during matrix multiplication, this approach seems currently impractical.

The last approach replaces the homomorphic multiplication with the MAC key α on ciphertexts with the more efficient multiplication operation of the HSS by Boyle et al. [BKS19]. More precisely, during seed generation secret shares of the MAC key α times the secret key sk are generated (instead of an encryption of α). Then, the last level of multiplication (i.e. to obtain $a\alpha, b\alpha, ab\alpha$) can be replaced by a distributed decryption with the shares of $\alpha \cdot \mathsf{sk}$, saving 3 homomorphic multiplications per triple generated. Note that this construction is compatible with the 'iterative' and 'packed' versions of BGV described. It does not seem competitive to use an HSS solely based on [BKS19], as, when handling more than one homomorphic multiplication, their scheme has to account for the plaintext magnitude, which in the described setting would lead to significantly larger parameters.

8 Multi-Party PCG for Bilinear Correlations

In this section, we construct *multi-party* PCGs for a useful class of bilinear correlations, capturing M-party OT, M-party vector OLE, M-party Beaver triples, and more. Incorporating into appropriate existing secure computation protocols, this yields secure M-party protocols for corresponding computations, with short correlated randomness, whose online execution requires only lightweight information theoretic operations and communication that scales *linearly* in the number of parties M.

Our construction approach provides a semi-generic transformation from any PCG for the corresponding 2-party bilinear correlation that satisfies an additional "programmability" property. Roughly, this property requires a way of "reusing" inputs across instances without compromising security. The M-party construction will leverage this structure by executing $M(M-1)$ pairwise instances of the underlying 2-party PCG, for all the "cross-terms."

We obtain M-party PCGs for various bilinear correlations by identifying corresponding 2-party PCG constructions that satisfy the required programmability notion. In particular:

- M-party VOLE: From lightweight DPF and LPN, leveraging the 2-party VOLE generator of [BCGI18].
- M-party OT / Beaver triple: From group-based or lattice-based HSS, leveraging our 2-party PCGs from the previous sections.

Interestingly, the lightweight 2-party OT PCG from Section 5 does not seem to support programmability in the necessary manner, since the resulting sender message pairs are implicitly defined as a function of the receiver's bit selections.

We begin by defining the class of bilinear correlations, and the PGC programmability property to which our transformation applies.

Definition 37 (Simple Bilinear Correlation: 2-party). *A 2-party correlation $\mathcal{C}$ is a* simple bilinear relation *if there exists Abelian groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ and a bilinear map $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ for which $\mathcal{C}$ is a distribution over $(\mathbb{G}_1, \mathbb{G}_T) \times (\mathbb{G}_2 \times \mathbb{G}_T)$ of the form*

$$\mathcal{C} = \{((a,c),(b,d)) \mid a \leftarrow \mathbb{G}_1, b \leftarrow \mathbb{G}_2, c \leftarrow \mathbb{G}_T, d = e(a,b) + c\}\,.$$

Note that the groups $\mathbb{G}$ and map e are implicitly parametrized by λ.

This captures, for example, Vector OLE (with $\mathbb{G}_1 = \mathbb{G}_T = \mathbb{F}^n$ and $e\colon \mathbb{F}^n \times \mathbb{F} \to \mathbb{F}^n$ by $e(\boldsymbol{u}, x) = x\boldsymbol{u}$), n-OLE (with $\mathbb{G}_1 = \mathbb{G}_2 = \mathbb{G}_T = \mathbb{F}^n$ and $e\colon \mathbb{F}^n \times \mathbb{F}^n \to \mathbb{F}^n$ by $e(\boldsymbol{u}, \boldsymbol{y}) = \boldsymbol{u} * \boldsymbol{v}$ componentwise multiplication), and String OT and n-OT as special cases for $\mathbb{F} = \mathbb{F}_2$.

Definition 38 (Simple Bilinear Correlation: M-party). *For simple bilinear 2-party correlation $\mathcal{C}_2$ specified by $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ we define the corresponding M-party correlation $\mathcal{C}_M$ by*

$$\mathcal{C}_M = \left\{ (a_i, b_i, c_i)_{i\in[M]} \;\middle|\; \begin{array}{c} a_i \xleftarrow{\$} \mathbb{G}_1, b_i \xleftarrow{\$} \mathbb{G}_2 \;\forall i \in [M],\; c_i \xleftarrow{\$} \mathbb{G}_T \;\forall i \in [M-1], \\ c_M = e\left(\sum_{i=1}^{M} a_i, \sum_{i=1}^{M} b_i\right) - \sum_{i=1}^{M-1} c_i \end{array} \right\}$$

Example 39. Useful specific examples:

- $\mathcal{C}_{M\text{-}\mathsf{VOLE}}$: Each party holds random $(\boldsymbol{u}_i, x_i, \boldsymbol{v}_i) \in \mathbb{F}^n \times \mathbb{F} \times \mathbb{F}^n$ s.t. $(\sum x_i)(\sum \boldsymbol{u}_i) = (\sum \boldsymbol{v}_i)$
- $\mathcal{C}_{M\text{-}\mathsf{OLE}}$: Each party holds random $(\boldsymbol{u}_i, \boldsymbol{v}_i, \boldsymbol{w}_i) \in \mathbb{F}^n \times \mathbb{F}^n \times \mathbb{F}^n$ s.t. $(\sum \boldsymbol{u}_i) * (\sum \boldsymbol{v}_i) = (\sum \boldsymbol{w}_i)$ (componentwise)

We consider 2-party PCGs that support the following notion of *programmability*: loosely, that allow a party to "reuse" a piece of his input (either $a \in \mathbb{G}_1$ or $b \in \mathbb{G}_2$) in multiple instances of the 2-party correlation, while maintaining security.

Definition 40 (Programmability). *We will say that a PCG $\mathsf{PCG} = (\mathsf{PCG.Gen}, \mathsf{PCG.Expand})$ for simple bilinear 2-party correlation $\mathcal{C}_2$ (specified by $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$) support* reusable inputs *if $\mathsf{PCG.Gen}(1^\lambda)$ takes additional random inputs $a', b' \in \{0,1\}^\star$ such that:*

- ***Programmability.*** *There exist public efficiently computable functions f_a, f_b for which*

$$\Pr\left[\begin{array}{l} a', b' \leftarrow \$, (\mathsf{k}_0, \mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda, a', b') \\ (a, c) \leftarrow \mathsf{PCG.Expand}(0, \mathsf{k}_0), \\ (b, d) \leftarrow \mathsf{PCG.Expand}(1, \mathsf{k}_1) \end{array} : \begin{array}{l} a = f_a(a') \\ b = f_b(b') \end{array}\right] \geq 1 - \mathsf{negl}(\lambda).$$

- ***Security.*** *The distributions*

$$\left\{ (\mathsf{k}_1, b', f_a(a')) \;\middle|\; \begin{array}{l} (a', b') \leftarrow \$ \\ (\mathsf{k}_0, \mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda, a', b') \end{array} \right\} \quad \textit{and}$$

$$\left\{ (\mathsf{k}_1, b', f_a(\tilde{a})) \;\middle|\; \begin{array}{l} (a', b') \leftarrow \$, \tilde{a} \leftarrow \$ \\ (\mathsf{k}_0, \mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen}(1^\lambda, a', b') \end{array} \right\}$$

  are computationally close. A symmetric requirement holds for $b', \tilde{b}$.

In Appendix G, we present and analyze the following general transformation from any *programmable* 2-party PCG for a simple bilinear correlation to a M-party PCG for the corresponding multi-party correlation.

Theorem 41 (Multi-party Simple Bilinear PCG). *Let $\mathsf{PCG}_2 = (\mathsf{PCG}_2\mathsf{.Gen}, \mathsf{PCG}_2\mathsf{.Expand})$ be a programmable PCG for simple bilinear 2-party correlation $\mathcal{C}_2$ (specified by $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$) with key sizes $s_0(\lambda), s_1(\lambda)$. Then there exists a PCG $\mathsf{PCG}_M = (\mathsf{PCG}_M\mathsf{.Gen}, \mathsf{PCG}_M\mathsf{.Expand})$ for the corresponding M-party correlation $\mathcal{C}_M$ with the following properties.*

- $\mathsf{PCG}_M\mathsf{.Gen}(1^\lambda)$ *runs $M(M-1)$ executions of $\mathsf{PCG}_2\mathsf{.Gen}$; each output key k_i, $i \in [M]$, has size $(M-1)(s_0(\lambda) + s_1(\lambda) + \lambda)$ bits.*
- $\mathsf{PCG}_M\mathsf{.Expand}(i, \mathsf{k}_i)$ *runs $2(M-1)$ executions of $\mathsf{PCG}_2\mathsf{.Expand}$ and makes $(M-1)$ evaluations of a pseudorandom generator.*

Acknowledgements. We would like to thank Peter Rindal and Melissa Rossi for helpful discussions and pointers, and the anonymous Crypto 2019 reviewers for their comments.

E. Boyle, N. Gilboa, and Y. Ishai supported by ERC Project NTSC (742754). E. Boyle additionally supported by ISF grant 1861/16 and AFOSR Award FA9550-17-1-0069. G. Couteau supported by ERC Project PREP-CRYPTO (724307). N. Gilboa additionally supported by ISF grant 1638/15 and a grant by the BGU Cyber Center. Y. Ishai additionally supported by ISF grant 1709/14, NSF-BSF grant 2015782, and a grant from the Ministry of Science and Technology, Israel and Department of Science and Technology, Government of India. L. Kohl supported by ERC Project PREP-CRYPTO (724307), by DFG grant HO 4534/2-2 and by a DAAD scholarship. This work was done in part while visiting the FACT Center at IDC Herzliya, Israel. P. Scholl supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No 731583 (SODA), and the Danish Independent Research Council under Grant-ID DFF-6108-00169 (FoCC).

References

ABD+16. C. Aguilar, O. Blazy, J.-C. Deneuville, P. Gaborit, and G. Zémor. Efficient encryption from random quasi-cyclic codes. Cryptology ePrint Archive, Report 2016/1194, 2016. http://eprint.iacr.org/2016/1194.

ABG+16. C. Aguilar Melchor, J. Barrier, S. Guelton, A. Guinet, M.-O. Killijian, and T. Lepoint. NFLlib: NTT-based fast lattice library. In *CT-RSA 2016*, LNCS. Springer, 2016.

ADI+17. B. Applebaum, I. Damgård, Y. Ishai, M. Nielsen, and L. Zichron. Secure arithmetic computation with constant computational overhead. LNCS. Springer, 2017.

AFS03. D. Augot, M. Finiasz, and N. Sendrier. A fast provably secure cryptographic hash function. Cryptology ePrint Archive, Report 2003/230, 2003. http://eprint.iacr.org/2003/230.

AG10. S. Arora and R. Ge. Learning parities with structured noise. In *Electronic Colloquium on Computational Complexity (ECCC)*, page 66, 2010.

AG11. S. Arora and R. Ge. New algorithms for learning in presence of errors. In *ICALP 2011, Part I*, LNCS. Springer, July 2011.

AHI11. B. Applebaum, D. Harnik, and Y. Ishai. Semantic security under related-key attacks and applications. In *ICS 2011*. Tsinghua University Press, January 2011.

AHI+17. B. Applebaum, N. Haramaty, Y. Ishai, E. Kushilevitz, and V. Vaikuntanathan. Low-complexity cryptographic hash functions. 2017.

AIK09. B. Applebaum, Y. Ishai, and E. Kushilevitz. Cryptography with constant input locality. *Journal of Cryptology*, (4), October 2009.

Ale03. M. Alekhnovich. More on average case vs approximation complexity. In *44th FOCS*. IEEE Computer Society Press, October 2003.

ALSZ13. G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *ACM CCS 13*. ACM Press, November 2013.

Bar04. M. Bardet. *Étude des systèmes algébriques surdéterminés. Applications aux codes correcteurs et à la cryptographie*. PhD thesis, Université Pierre et Marie Curie-Paris VI, 2004.

BCG+17. E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, and M. Orrù. Homomorphic secret sharing: Optimizations and applications. In *ACM CCS 17*. ACM Press, 2017.

BCG+19. E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In *Advances in Cryptology - CRYPTO 2019, 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019. Proceedings*, 2019.

BCGI18. E. Boyle, G. Couteau, N. Gilboa, and Y. Ishai. Compressing vector OLE. In *ACM CCS 18*. ACM Press, 2018.

BCJ07. G. V. Bard, N. T. Courtois, and C. Jefferson. Efficient methods for conversion and solution of sparse systems of low-degree multivariate polynomials over GF(2) via SAT-Solvers. Cryptology ePrint Archive, Report 2007/024, 2007. http://eprint.iacr.org/2007/024.

BDOZ11. R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT 2011*, LNCS. Springer, May 2011.

Bea91. D. Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, *Lecture Notes in Computer Science* 576, pages 420–432. Springer, 1991.

Bea96. D. Beaver. Correlated pseudorandomness and the complexity of private computations. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 479–488, 1996.

BFKL93. A. Blum, M. L. Furst, M. J. Kearns, and R. J. Lipton. Cryptographic primitives based on hard learning problems. In *Advances in Cryptology - CRYPTO '93, 13th Annual International Cryptology Conference, Santa Barbara, California, USA, August 22-26, 1993, Proceedings*, pages 278–291, 1993.

BFSY05. M. Bardet, J. Faugere, B. Salvy, and B. Yang. Asymptotic behaviour of the index of regularity of quadratic semi-regular polynomial systems. In *The Effective Methods in Algebraic Geometry Conference (MEGA'05)(P. Gianni, ed.)*, pages 1–14. Citeseer, 2005.

BGI15. E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing. LNCS. Springer, 2015.

BGI16a. E. Boyle, N. Gilboa, and Y. Ishai. Breaking the circuit size barrier for secure computation under DDH. In *CRYPTO 2016, Part I*, LNCS. Springer, August 2016.

BGI16b. E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing: Improvements and extensions. In *ACM CCS 16*. ACM Press, 2016.

BGI17. E. Boyle, N. Gilboa, and Y. Ishai. Group-based secure computation: Optimizing rounds, communication, and computation. LNCS. Springer, 2017.

BGI+18. E. Boyle, N. Gilboa, Y. Ishai, H. Lin, and S. Tessaro. Foundations of homomorphic secret sharing. In *9th Innovations in Theoretical Computer Science Conference, ITCS 2018, January 11-14, 2018, Cambridge, MA, USA*, pages 21:1–21:21, 2018.

BGN05. D. Boneh, E.-J. Goh, and K. Nissim. Evaluating 2-DNF formulas on ciphertexts. In *TCC 2005*, LNCS. Springer, February 2005.

BGP06. C. Berbain, H. Gilbert, and J. Patarin. QUAD: A practical stream cipher with provable security. In *EUROCRYPT 2006*, LNCS. Springer, May / June 2006.

BGV12. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *ITCS 2012*. ACM, January 2012.

BH74. J. R. Bunch and J. E. Hopcroft. Triangular factorization and inversion by fast matrix multiplication. *Mathematics of Computation*, 28(125):231–236, 1974.

BKS19. E. Boyle, L. Kohl, and P. Scholl. Homomorphic secret sharing from lattices without fhe. In *EUROCRYPT '19*, 2019. https://eprint.iacr.org/2019/129.

Bor57. J. L. Bordewijk. Inter-reciprocity applied to electrical networks. *Applied Scientific Research, Section A*, 6(1):1–74, 1957.

CCK+18. M. Chen, C. Cheng, P. Kuo, W. Li, and B. Yang. Multiplying boolean polynomials with frobenius partitions in additive fast fourier transform. *CoRR*, abs/1803.11301, 2018.

CDI05. R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *TCC 2005*, LNCS. Springer, February 2005.

CDI+18. M. Chase, Y. Dodis, Y. Ishai, D. Kraschewski, T. Liu, R. Ostrovsky, and V. Vaikuntanathan. Reusable non-interactive secure computation. *IACR Cryptology ePrint Archive*, 2018:940, 2018.

CDM+18. G. Couteau, A. Dupin, P. Méaux, M. Rossi, and Y. Rotella. On the concrete security of Goldreich's pseudorandom generator. LNCS. Springer, December 2018.

CDN01. R. Cramer, I. Damgård, and J. B. Nielsen. Multiparty computation from threshold homomorphic encryption. In *Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 6-10, 2001, Proceeding*, pages 280–299, 2001.

Cou19. G. Couteau. A note on the communication complexity of multiparty computation in the correlated randomness model. In *Advances in Cryptology - EUROCRYPT*. Springer, 2019.

CS16. A. Costache and N. P. Smart. Which ring based somewhat homomorphic encryption scheme is best? In *CT-RSA 2016*, LNCS. Springer, 2016.

CW90. D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of symbolic computation*, 9(3):251–280, 1990.

DGN+17. N. Döttling, S. Ghosh, J. B. Nielsen, T. Nilges, and R. Trifiletti. TinyOLE: Efficient actively secure two-party computation from oblivious linear function evaluation. In *ACM CCS 17*. ACM Press, 2017.

DHRW16. Y. Dodis, S. Halevi, R. D. Rothblum, and D. Wichs. Spooky encryption and its applications. LNCS. Springer, August 2016.

DI14. E. Druk and Y. Ishai. Linear-time encodable codes meeting the gilbert-varshamov bound and their cryptographic applications. In *ITCS 2014*. ACM, 2014.

DKK18. I. Dinur, N. Keller, and O. Klein. An optimal distributed discrete log protocol with applications to homomorphic secret sharing. In *CRYPTO*, 2018.

DKS+17. G. Dessouky, F. Koushanfar, A.-R. Sadeghi, T. Schneider, S. Zeitouni, and M. Zohner. Pushing the communication barrier in secure computation using lookup tables. In *NDSS 2017*, 2017.

DNNR17. I. Damgård, J. B. Nielsen, M. Nielsen, and S. Ranellucci. The TinyTable protocol for 2-party secure computation, or: Gate-scrambling revisited. LNCS. Springer, 2017.

DPSZ12. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO 2012*, LNCS. Springer, August 2012.

DRRT18. D. Demmler, P. Rindal, M. Rosulek, and N. Trieu. PIR-PSI: Scaling private contact discovery. In *PETS '18*, 2018.

Ds17. J. Doerner and a. shelat. Scaling ORAM for secure computation. In *ACM CCS 17*. ACM Press, 2017.

DyY07. J. Ding and B. yin Yang. Multivariates polynomials for hashing. Cryptology ePrint Archive, Report 2007/137, 2007. http://eprint.iacr.org/2007/137.

FH96. M. K. Franklin and S. Haber. Joint encryption and message-efficient secure computation. *J. Cryptology*, 9(4):217–232, 1996.

FIPR05. M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. In *TCC 2005*, LNCS. Springer, February 2005.

Fis74. P. C. Fischer. Further schemes for combining matrix algorithms. In *International Colloquium on Automata, Languages, and Programming*, pages 428–436. Springer, 1974.

Fre10. D. M. Freeman. Converting pairing-based cryptosystems from composite-order groups to prime-order groups. In *EUROCRYPT 2010*, LNCS. Springer, May 2010.

Frö85. R. Fröberg. An inequality for hilbert series of graded algebras. *Mathematica Scandinavica*, 56(2):117–144, 1985.

Gal62. R. Gallager. Low-density parity-check codes. *IRE Transactions on information theory*, 8(1):21–28, 1962.

GGM86. O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, (4), October 1986.

GI99. N. Gilboa and Y. Ishai. Compressing cryptographic resources. In *CRYPTO'99*, LNCS. Springer, August 1999.

GI14. N. Gilboa and Y. Ishai. Distributed point functions and their applications. In *EUROCRYPT 2014*, LNCS. Springer, 2014.

GIK+15. S. Garg, Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Cryptography with one-way communication. In *CRYPTO 2015, Part II*, LNCS. Springer, August 2015.

GKWY19. C. Guo, J. Katz, X. Wang, and Y. Yu. Efficient and secure multiparty computation from fixed-key block ciphers. Cryptology ePrint Archive, Report 2019/074, 2019. https://eprint.iacr.org/2019/074.

GMW87. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *19th ACM STOC*. ACM Press, May 1987.

GNN17. S. Ghosh, J. B. Nielsen, and T. Nilges. Maliciously secure oblivious linear function evaluation with constant overhead. LNCS. Springer, December 2017.

Gol00. O. Goldreich. Candidate one-way functions based on expander graphs. Cryptology ePrint Archive, Report 2000/063, 2000. http://eprint.iacr.org/2000/063.

HIJ+16. S. Halevi, Y. Ishai, A. Jain, E. Kushilevitz, and T. Rabin. Secure multiparty computation with general interaction patterns. ACM, 2016.

HKL+12. S. Heyse, E. Kiltz, V. Lyubashevsky, C. Paar, and K. Pietrzak. Lapin: An efficient authentication protocol based on ring-LPN. In *FSE 2012*, pages 346–365, 2012.

HLR07. C.-Y. Hsiao, C.-J. Lu, and L. Reyzin. Conditional computational entropy, or toward separating pseudoentropy from compressibility. In *EUROCRYPT 2007*, LNCS. Springer, May 2007.

HOSS18. C. Hazay, E. Orsini, P. Scholl, and E. Soria-Vazquez. TinyKeys: A new approach to efficient multiparty computation. LNCS. Springer, 2018.

HS14. S. Halevi and V. Shoup. Algorithms in HElib. In *CRYPTO 2014, Part I*, LNCS. Springer, August 2014.

HS18. S. Halevi and V. Shoup. Faster homomorphic linear transformations in HElib. LNCS. Springer, 2018.

HW15. P. Hubacek and D. Wichs. On the communication complexity of secure function evaluation with long output. In *ITCS 2015*. ACM, 2015.

IKM+13. Y. Ishai, E. Kushilevitz, S. Meldgaard, C. Orlandi, and A. Paskin-Cherniavsky. On the power of correlated randomness in secure computation. In *TCC 2013*, LNCS. Springer, March 2013.

IKNP03. Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *CRYPTO 2003*, LNCS. Springer, August 2003.

IKOS04. Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Batch codes and their applications. In *36th ACM STOC*. ACM Press, June 2004.

IKOS07. Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Zero-knowledge from secure multiparty computation. In *39th ACM STOC*. ACM Press, June 2007.

IKOS08. Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Cryptography with constant computational overhead. In *40th ACM STOC*. ACM Press, May 2008.

IPS08. Y. Ishai, M. Prabhakaran, and A. Sahai. Founding cryptography on oblivious transfer - efficiently. In *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, pages 572–591, 2008.

IPS09. Y. Ishai, M. Prabhakaran, and A. Sahai. Secure arithmetic computation with no honest majority. In *TCC 2009*, LNCS. Springer, March 2009.

Kap04. I. Kaporin. The aggregation and cancellation techniques as a practical tool for faster matrix multiplication. *Theoretical Computer Science*, 315(2-3):469–510, 2004.

Kil88. J. Kilian. Founding cryptography on oblivious transfer. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 20–31, 1988.

KK13. V. Kolesnikov and R. Kumaresan. Improved OT extension for transferring short secrets. In *CRYPTO 2013, Part II*, LNCS. Springer, August 2013.

KKRT16. V. Kolesnikov, R. Kumaresan, M. Rosulek, and N. Trieu. Efficient batched oblivious PRF with applications to private set intersection. In *ACM CCS 16*. ACM Press, 2016.

KMO90. J. Kilian, S. Micali, and R. Ostrovsky. Minimum resource zero-knowledge proofs (extended abstract). In *CRYPTO'89*, LNCS. Springer, August 1990.

KOR+17. M. Keller, E. Orsini, D. Rotaru, P. Scholl, E. Soria-Vazquez, and S. Vivek. Faster secure multi-party computation of AES and DES using lookup tables. In *ACNS 17*, LNCS. Springer, 2017.

KPR18. M. Keller, V. Pastro, and D. Rotaru. Overdrive: Making SPDZ great again. LNCS. Springer, 2018.

KRRW18. J. Katz, S. Ranellucci, M. Rosulek, and X. Wang. Optimizing authenticated garbling for faster secure two-party computation. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III*, pages 365–391, 2018.

KS12. K. Kobayashi and T. Shibuya. Generalization of lu's linear time encoding algorithm for ldpc codes. In *Information Theory and its Applications (ISITA), 2012 International Symposium on*, pages 16–20. IEEE, 2012.

LG12. F. Le Gall. Faster algorithms for rectangular matrix multiplication. In *Foundations of Computer Science (FOCS), 2012 IEEE 53rd Annual Symposium on*, pages 514–523. IEEE, 2012.

LJKS+16. C. Löndahl, T. Johansson, M. Koochak Shooshtari, M. Ahmadian-Attari, and M. R. Aref. Squaring attacks on mceliece public-key cryptosystems using quasi-cyclic codes of even dimension. *Des. Codes Cryptography*, 80(2):359–377, August 2016.

LLY08. F.-H. Liu, C.-J. Lu, and B.-Y. Yang. Secure PRNGs from specialized polynomial maps over any. 2008.

LM10. J. Lu and J. M. Moura. Linear time encoding of ldpc codes. *IEEE Transactions on Information Theory*, 56(1):233–249, 2010.

LPR10. V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT 2010*, LNCS. Springer, May 2010.

MBD+18. C. A. Melchor, O. Blazy, J. Deneuville, P. Gaborit, and G. Zémor. Efficient encryption from random quasi-cyclic codes. *IEEE Trans. Information Theory*, 64(5):3927–3943, 2018.

MI88. T. Matsumoto and H. Imai. Public quadratic polynominal-tuples for efficient signature-verification and message-encryption. In *EUROCRYPT'88*, LNCS. Springer, May 1988.

MN96. D. J. MacKay and R. M. Neal. Near shannon limit performance of low density parity check codes. *Electronics letters*, 32(18):1645, 1996.

MST03. E. Mossel, A. Shpilka, and L. Trevisan. On e-biased generators in NC0. In *44th FOCS*. IEEE Computer Society Press, October 2003.

MTSB12. R. Misoczki, J.-P. Tillich, N. Sendrier, and P. S. L. M. Barreto. MDPC-McEliece: New McEliece variants from moderate density parity-check codes. Cryptology ePrint Archive, Report 2012/409, 2012. http://eprint.iacr.org/2012/409.

NNOB12. J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In *CRYPTO 2012*, LNCS. Springer, August 2012.

NP01. M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *12th SODA*. ACM-SIAM, January 2001.

NP06. M. Naor and B. Pinkas. Oblivious polynomial evaluation. *SIAM J. Comput.*, 35(5):1254–1281, 2006.

Pan18. V. Y. Pan. Fast feasible and unfeasible matrix multiplication. *arXiv preprint arXiv:1804.04102*, 2018.

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

PsV06. R. Pass, a. shelat, and V. Vaikuntanathan. Construction of a non-malleable encryption scheme from any semantically secure one. In *CRYPTO 2006*, LNCS. Springer, August 2006.

PVW08. C. Peikert, V. Vaikuntanathan, and B. Waters. A framework for efficient and composable oblivious transfer. In *CRYPTO 2008*, LNCS. Springer, August 2008.

Sch18. P. Scholl. Extending oblivious transfer with low communication via key-homomorphic PRFs. LNCS. Springer, 2018.

Spi96. D. A. Spielman. Linear-time encodable and decodable error-correcting codes. *IEEE Transactions on Information Theory*, 42(6):1723–1731, 1996.

Str69. V. Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

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

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

Wol05. C. Wolf. Multivariate quadratic polynomials in public key cryptography. Cryptology ePrint Archive, Report 2005/393, 2005. http://eprint.iacr.org/2005/393.

WRK17a. X. Wang, S. Ranellucci, and J. Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *ACM CCS 17*. ACM Press, 2017.

WRK17b. X. Wang, S. Ranellucci, and J. Katz. Global-scale secure multiparty computation. In *ACM CCS 17*. ACM Press, 2017.

Yao82. A. C.-C. Yao. Theory and applications of trapdoor functions (extended abstract). In *23rd FOCS*. IEEE Computer Society Press, November 1982.

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

Appendix

A Details on Silent OT Extension

A.1 Application to Reusable NIZK From LPN

We sketch in this section an application of the silent OT extension from subfield vector-OLE given in Section 5 to non-interactive zero-knowledge proofs in the preprocessing model. NIZK in the preprocessing model relaxes the standard NIZK definition by allowing the prover and the verifier to interact during a preprocessing phase, to generate respective proving key and verification key. It is well known that preprocessing NIZKs can be constructed from any oblivious transfer [KMO90, PsV06, IKOS07, GIK+15]; in fact, these works even imply the existence of *information-theoretic* preprocessing NIZKs in the OT-hybrid model. However, in all these protocols, the computation and communication of the preprocessing grow with the size (and the number) of the theorems to be proven; in other words, the preprocessing implies an a-priori bound on the size and number of statements to be proven later.

A.1.1 Reusable Preprocessing NIZK from Silent OT

Plugging our random OT PCG given in Section 5 into the construction of preprocessing NIZK from OT, we readily obtain an improved preprocessing NIZK: following a one-time preprocessing phase whose communication and computation grow only with the *soundness parameter*[14] of the proofs to be sent in the online phase. After this preprocessing phase, the prover and the verifier can locally, without further interaction, extend the preprocessed material into an arbitrary polynomial number of (pseudo)random OTs, leading to a preprocessing NIZK with preprocessing cost *independent* of the size and number of theorems to be proven. In other word, the same preprocessing material can be reused for an a-priori arbitrary number of proofs. (Adaptive, multi-theorem) zero-knowledge and (adaptive, reusable) soundness readily follow from the security properties of the silent OT extension, and soundness holds even if the prover is allowed to get the answer of the verifier for an arbitrary polynomial number of proofs before trying to prove a false statement.

Using our construction of silent OT extension from LPN and correlation-robust assumption, this readily implies the existence of a reusable preprocessing NIZK from the same assumptions. Compared to the recent reusable preprocessing NIZK of [BCGI18], our preprocessing NIZK has two advantages:

- It only requires the standard LPN assumption over $\mathbb{F}_2$ (together with correlation-robust hash functions), while the NIZK of [BCGI18] must rely on a generalization of LPN to exponentially large fields;
- The preprocessing phase is independent of both the number of theorems to be proven, and the size of these theorems. In comparison the preprocessing phase in [BCGI18] is independent of the number of theorems to be proven, but grows linearly with a bound on the size of each statement.

A.1.2 Removing the Correlation-Robust Hash Functions

Our silent OT extension of Section 5 relies on a correlation-robust hash function. Intuitively, this comes from the use of the classical IKNP strategy for OT extension [IKNP03]: to generate m random OTs of strings of length ℓ, the sender and the receiver exchange their roles and perform

[14] We say that a NIZK has soundness parameter t if the probability for the prover to cause the verifier to accept a proof for an invalid statement is at most 2^{-k}.

λ executions of a length-m OT protocol (λ is a security parameter): that is, the receiver with selection bits $\boldsymbol{b} = (b_i)_{i \le m}$ plays the role of the sender with inputs $(\boldsymbol{x}^j, \boldsymbol{x}^j + \boldsymbol{b})_{j \le \lambda}$, where $\boldsymbol{x}^j$ is a random string of length m, and the sender plays the role of the receiver with a random selection vector $\boldsymbol{s}$ of length λ. This allows the sender to reconstruct $\boldsymbol{k}_i = \boldsymbol{x}_i + b_i \boldsymbol{s}$, where $\boldsymbol{x}_i$ is the length-λ vector of the i-th coordinates of the vectors $\boldsymbol{x}^j$. Then, to execute an OT protocol with length ℓ-inputs (s_i^0, s_i^1) from the sender and b_i from the verifier, the sender sends $(z_i^0, z_i^1) = (s_i^0 \oplus H(\boldsymbol{k}_i), s_i^1 \oplus H(\boldsymbol{k}_i - \boldsymbol{s}))$. Note that the verifier knows $\boldsymbol{x}_i = \boldsymbol{k}_i - b_i \boldsymbol{s}$, hence he can reconstruct $s_i^{b_i} = z_i^{b_i} \oplus H(\boldsymbol{x}_i)$; sender security follows from the correlation-robustness of the hash function.

In [AHI11], it was observed that a similar construction can be obtained by replacing the correlation-robust hash function by an encryption scheme semantically secure against related-key attack for the class of linear functions under general group. An RKA-secure scheme maintains it's semantic security even if the keys satisfy a known (in fact, adaptively chosen by the adversary) relation. In the above construction, the sender can send $(z_i^0, z_i^1) = (\mathsf{Enc}(s_i^0, \boldsymbol{k}_i), \mathsf{Enc}(s_i^1, \boldsymbol{k}_i - \boldsymbol{s}))$, where Enc is semantically secure against RKA attacks for linear functions: the sender security reduces to the hardness of distinguishing $\mathsf{Enc}(s_i^{1-b_i}, \boldsymbol{k}_i - (1-b_i)\boldsymbol{s}) = \mathsf{Enc}(s_i^{1-b_i}, \boldsymbol{x}_i + (-1)^{b_i}\boldsymbol{s})$ from $\mathsf{Enc}(0^\ell, \boldsymbol{x}_i + (-1)^{b_i}\boldsymbol{s})$, even given the keys $\boldsymbol{x}_i$ for each i, which follows from the semantic security of Enc against RKA attacks. Furthermore, [AHI11] provides in particular a construction of an encryption scheme semantically secure against RKA-attacks for linear functions, from the standard LPN assumption over $\mathbb{F}_2$. Plugging their scheme as a replacement for the correlation-robust hash function in our construction of preprocessing NIZK from silent OT, we obtain a reusable preprocessing NIZK *solely* under the standard LPN assumption.

Theorem 42. *Under the standard LPN assumption over $\mathbb{F}_2$, with dimension n, number of samples $\mathsf{poly}(n)$, and slightly sub-constant noise rate $1/\omega(\lambda \log n)$, there exists a reusable preprocessing NIZK proof system for NP which satisfies adaptive multi-theorem zero-knowledge and adaptive reusable soundness, where the communication and the computation of the preprocessing phase depend solely on the soundness error of the proof system, but are independent of the size and number of statements to be proven.*

We note that the flavor of LPN used in our construction is a "minicrypt-style" assumption: it belongs to a parameter regime which is not known to imply the existence of public key encryption (which is only known from LPN with smaller noise rate $O(1/\sqrt{n})$). Furthermore, the exact flavor of LPN that we need is in fact slightly weaker than the one given in the theorem above. Our NIZK can be based on the assumption that the following two variants of LPN are secure:

- LPN over $\mathbb{F}_2$ with dimension n, *very small number of samples* $n + o(n)$, and slightly sub-constant noise rate $1/\omega(\lambda \log n)$ (this assumption underlies our silent OT extension), and
- LPN over $\mathbb{F}_2$ with dimension n, polynomial number of samples $\mathsf{poly}(n)$, and *constant noise rate* $O(1)$ (this assumption underlies the RKA-secure symmetric-key encryption scheme of [AHI11]).

A.1.3 Replacing 1-out-of-2 OT with Rabin OT

While the construction described above relies on 1-out-of-2 oblivious transfer, it is also possible to build preprocessing NIZKs directly from the original Rabin OT primitive (where the sender has a single input, and the receiver learns it with probability 1/2) [GIK+15]. We note that our silent OT extension can also be used to generate Rabin-type OT correlations, with a factor 2 of savings compared to the 1-out-of-2 silent OT extension protocol.

A.2 Efficiency Analysis

To improve the efficiency, we modify the construction to use the *regular* version of the syndrome decoding (or dual-LPN) problem, where the error vector $\boldsymbol{e}$ is divided into t equally-spaced blocks,

each of weight one. This means the MPFSS scheme can be built by concatenating t DPFs of size n'/t, instead of XORing t DPFs of size n'.

In Tables 3 and 4, we present the communication complexity and estimated runtimes of our Silent OT extension protocol for various choices of parameters. We chose our parameters as in [BCGI18], so that the best known attacks against the regular syndrome decoding problem cost at least 2^{80} operations. We always use $n' = 4n$, and values of the noise-weight t as shown in the table. We estimated runtimes based on the costs of the main operations in the Doerner-shelat [Ds17] protocol for running a distributed DPF setup with semi-honest security, and our own estimates for the cost of matrix multiplication. These are based on a single core of an i7-7600 CPU @2.8GHz with SSE and AES-NI instructions.

Communication complexity. When using the DPF setup protocol of [Ds17], for a single DPF of size N we need $\log N$ OTs[15] on λ-bit strings, except the last OT which has length 2λ. With the regular LPN variant, we use t DPFs of size $N = n'/t$, with $n' = 4n$. Using OT extension, an OT on ℓ-bit strings requires $\lambda + 2\ell$ bits of communication [ALSZ13], giving a total of $t(\log(n'/t) + 1)(2\lambda + 1)$ bits of communication for these. The base OTs for OT extension can be implemented with the Naor-Pinkas protocol [NP01] based on DDH over a 256-bit elliptic curve group, at a cost of sending 1024 bits per OT.

This gives rough overall costs as follows, shown in detail in Table 3.
$(\log(N) - 1) \cdot 3\lambda + 5\lambda = 3\lambda \cdot \log(N) + 2\lambda$

- *Seed size:* $\approx t \cdot (\lambda \cdot \log(n'/t) + 2\lambda)$ bits
- *Base OTs (one-time cost):* $1024 \cdot \lambda$ bits
- *Distributed setup:* $\approx 2 \cdot \lambda \cdot t \cdot (\log(n'/t) + 1)$ bits

Encoding method. It seems likely that when n is large the dominating cost is the $\mathbb{F}_{2^\lambda}^{n'} \times \mathbb{F}_2^{n' \times n}$ matrix multiplication. We consider two different encoding methods, both suggested in [BCGI18].

- *Quasi-cyclic codes.* H is the parity-check matrix of a random quasi-cyclic code. Multiplication by H can be computed as a length n'/n inner product over $\mathbb{Z}_2[X]/(X^n - 1)$, for which we estimate costs using a state-of-the-art implementation of fast binary polynomial multiplication [CCK+18]. Security reduces to the quasi-cyclic syndrome decoding problem; note that this requires n to be prime to avoid attacks expoiting the quasi-cyclic structure [LJKS+16].
- *LDPC codes.* H is the *transpose* of the generator matrix for an LDPC (low-density parity check) code, which is defined by a random parity check matrix with constant sparsity d. This implies that we need LPN to be hard for d-local codes, which is the same assumption that was conjectured in [Ale03]. We use $d = 10$, as suggested in [ADI+17]. Multiplication by H is essentially the transpose of an LDPC encoding, which can be done in $O(n)$ operations by "transposing" a linear-time LDPC encoding algorithm.

We have not implemented the LDPC encoding algorithm, but instead tested a dummy algorithm which performs the same operation count, with a worst-case access pattern where each operation reads from a random component of the input. As can be seen in Table 4, this becomes much slower beyond inputs of size 2^{20} bits, which no longer fit in the L1 cache (128KiB). However, we expect that carefully implementing the algorithm could lead to performance improvements.

Other costs. The other costs involved are a DPF full-domain evaluation, and the cost of distributed setup for the DPFs. We ignore the cost of the OTs in the DPF setup procedure, since for our parameters we always need under 1000 OTs, which take under 1ms using modern OT extension implementations. To simplify estimates, we therefore assume the DPF cost in setup is the same as one full-domain evaluation, and estimate this using the implementation from [DRRT18],

[15] In [Ds17], $2 \log N$ OTs are needed, but in our case all the OTs in one direction can be avoided, since one party knows the secret point.

which reports a throughput of 2.6 billion bits/s for full-domain evaluation. Since our DPFs output 128 bits instead of one, we scale this down to get around 20 million evaluations per second.

n	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}
t	39	34	32	31	30	29	28
Seed size (kB)	5.65	6.02	6.69	7.97	8.67	9.31	9.89
Comp. ratio	11.6	43.6	157	526	1930	7210	27150
Base OT comms. (kB)	16.38	16.38	16.38	16.38	16.38	16.38	16.38
Setup comms. (kB)	18.10	19.04	20.99	24.80	26.88	28.77	30.46
Bits per OT (exc. base)	35.34	9.30	2.56	0.76	0.21	0.05	0.01

Table 3. Communication complexity of silent OT for various parameter sets, all with $n' = 4n$. Compression ratio is λn divided by the seed size in bits, with $\lambda = 128$.

n	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}
Quasi-cyclic	11.3	12.8	53.8	238	1113	5212	21090
LDPC transpose	0.3	1.18	9.9	74.3	646	4460	47960
MPFSS full eval	0.8	3.2	12.9	51.6	207	826	3300
Total time (quasi-cyclic)	12.9	19.2	79.6	341	1527	6864	27690
Total time (LDPC)	1.9	7.58	35.7	178	1060	6112	54560
Throughput, QC (million/s)	0.32	0.85	0.82	0.77	0.69	0.61	0.60
Throughput, LDPC (million/s)	2.16	2.16	1.84	1.47	0.99	0.69	0.31

Table 4. Estimated runtimes (ms) and throughput for n silent OTs. Based on (total time) $\approx 2\times$ (MPFSS full eval) + (encoding time)

B One-Time Truth Table Generator

We define the authenticated, masked truth table correlation, $\mathcal{C}_{\mathsf{TT}}$ for a lookup table $T : [n] \to \{0,1\}^m$ as follows. $\mathcal{C}_{\mathsf{TT}}$ first samples MAC key shares $\alpha_0, \alpha_1 \xleftarrow{\$} \mathbb{F}_{2^\lambda}$ and a mask $s \xleftarrow{\$} [n]$, then computes $\alpha = \alpha_0 + \alpha_1$, $y_i = T(s + i \bmod n)$ and $\gamma_i = y_i \cdot \alpha$ in $\mathbb{F}_{2^\lambda}$, for $i \in [n]$, viewing the outputs of T as elements of the field. It outputs

$$(R_0, R_1) = \big((\alpha_\sigma, \{y_i^\sigma, \gamma_i^\sigma\}_{i\in[n]})\big)_{\sigma\in\{0,1\}}$$

where $y_i^\sigma \in \{0,1\}^m, \gamma_i^\sigma \in \mathbb{F}_{2^\lambda}$ are sampled at random such that $y_i^0 + y_i^1 = y_i$ and $\gamma_i^0 + \gamma_i^1 = \gamma_i$.

Our starting point is the observation from [KOR+17] that this correlation can be generated locally, given secret-shares of a random unit vector. This is because, if $s \in [n]$ is the random mask, and $\boldsymbol{e}_s \in \{0,1\}^n$ is the s-th unit vector, then we have

$$T(i + s \bmod n) = \sum_{j=1}^{n} \boldsymbol{e}_s[j] \cdot T(i + j \bmod n)$$

and this can be computed locally given additive shares of $\boldsymbol{e}_s$, since T is public.

Given a DPF for the function with domain $[n]$ that maps s to 1 and is zero elsewhere, two parties can compute shares of $\boldsymbol{e}_s$ by simply evaluating the DPF on every input in $[n]$. This already allows us to compress a simple, semi-honest version of $\mathcal{C}_{\mathsf{TT}}$ without MACs. We can extend this to also create the secret-shared MACs *at no extra cost*, just by choosing the DPF to

map s to $(1\|\alpha)$, where α is a random MAC key. We remark that, as described in Section 2, this can be seen as an instance of the DPF-based HSS scheme that is implicit in our subfield-VOLE PCG.

The complete construction is given in Fig. 6. Instantiating DPF with [BGI16b], the total PCG seed size is $\log n \cdot (\lambda + 2) + 3\lambda + 1$ bits, and note that this is independent of the precise lookup table, or even the length of its values. Compared with the cost of naively storing a 1-bit output one-time truth table with MACs of $n \cdot (\lambda + 1) + \lambda$ bits from [DNNR17], we obtain storage savings of over 20x for a length-256 table (as in the AES S-box), and this increases to almost 80x for a table of size 1024. The computational cost of expanding the entire correlation is n calls to $\mathsf{DPF.FullEval}$, however, when using this for 2-PC only a single entry of the table is needed, and this can be computed on-the-fly with just 1 call to $\mathsf{FullEval}$, for a cost of $O(n)$ PRG evaluations.

Construction G_{TT}

PARAMETERS: A lookup table $T : [n] \to \{0,1\}^m$, and a distributed point function $\mathsf{DPF} = (\mathsf{DPF.Gen}, \mathsf{DPF.FullEval})$.

Gen: On input 1^λ:

1. Pick a random index $s \xleftarrow{\$} \{1, \ldots, n\}$.
2. Sample $\alpha_0, \alpha_1 \xleftarrow{\$} \mathbb{F}_{2^\lambda}$ and let $\alpha = \alpha_0 \oplus \alpha_1$.
3. Compute $(K_0^{\mathsf{fss}}, K_1^{\mathsf{fss}}) \xleftarrow{\$} \mathsf{DPF.Gen}(1^\lambda, f_{s,1\|\alpha})$.
4. Let $\mathsf{k}_0 \leftarrow (K_0^{\mathsf{fss}}, \alpha_0)$ and $\mathsf{k}_1 \leftarrow (K_1^{\mathsf{fss}}, \alpha_1)$.
5. Output $(\mathsf{k}_0, \mathsf{k}_1)$.

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

1. Parse k_σ as $(K_\sigma^{\mathsf{fss}}, \alpha_\sigma)$.
2. Compute $\boldsymbol{v}^\sigma \leftarrow \mathsf{DPF.FullEval}(\sigma, K_\sigma^{\mathsf{fss}})$ in $\{0,1\}^{n\cdot(\lambda+1)}$.
3. For each $i \in [n]$, write $v_i^\sigma = (b_i^\sigma \| c_i^\sigma) \in \{0,1\} \times \mathbb{F}_{2^\lambda}$. Compute, for $j \in [n]$:

$$y_j^\sigma = \bigoplus_{i=1}^{n} b_i^\sigma \cdot T(i + j \bmod n) \in \{0,1\}^m, \quad \gamma_j^\sigma = \bigoplus_{i=1}^{n} c_i^\sigma \cdot T(i + j \bmod n) \in \mathbb{F}_{2^\lambda}$$

and output $(\alpha_\sigma, \{y_j^\sigma, \gamma_j^\sigma\}_{j=1}^n)$.

Fig. 6. PCG for authenticated, one-time truth table correlations

The following theorem can be proven with a reduction to MPFSS, similarly to (and simpler than) the proof of Theorem 24 for the subfield-VOLE generator.

Theorem 43. *Let* DPF *be a secure distributed point function. Then construction* G_{TT} *in Fig. 6 is a secure PCG for the correlation* G_{TT}.

B.1 Application to Sublinear-Communication MPC in the Preprocessing Model

Secure computation in the correlated randomness model typically requires communicating $O(s)$ values in the online phase, where s is the circuit size. In a recent paper, Couteau [Cou19] showed that this is not inherent: given access to a trusted source of (polynomially many) large one-time truth tables correlations, N parties can securely evaluate arbitrary *layered* (boolean or arithmetic) circuits (whose nodes can be partitioned into layers such that any edge connects adjacent layers – such circuits capture a variety of circuits that arise in practice) with information-theoretic security and *sublinear* communication $O(s/\log\log s)$. A downside of this protocol, that strongly limits its partical implications, is that it requires a large amount of preprocessing material: to securely evaluate the circuit in the online phase, the parties need to generate and store $O(s^2 \log\log s)$ bits of preprocessed material (i.e., $O(s/\log\log s)$ one-time truth tables of

size $O(s)$ each). Using our PCG for truth table correlations, this can be compressed to a quasi-linear amount $O(\lambda s \log s / \log\log s)$ of preprocessing material, assuming only one-way functions. The preprocessing material can then be *locally* expanded by the parties, without any interaction, into $O(s^2 \log\log s)$ bits of (pseudo)random one-time truth tables.

C Group-Based PCG for Bilinear Correlations

In this section, we exhibit a construction of PCG from the (external) DDH assumption over pairing-friendly elliptic curves, for the class of (additive) bilinear correlations. This construction builds upon the group-based HSS developed in [BGI16a, BGI17, BCG+17]. Note that all these constructions satisfy an imperfect correctness notion, where correctness is only guaranteed to hold except with probability δ, and the evaluation algorithm is allowed to run in time polynomial in $1/\delta$ (it is called a "Las Vegas" HSS in [BGI16a, BGI17, BCG+17]). Our construction of group-based PCG will inherit this imperfect correctness. However, the parties can detect when a given output has a high risk of being incorrect; knowing the location of the faulty outputs allows them to use efficient techniques (such as punctured OT [BGI17] or leakage-absorbing pads [BCG+17]) to securely delete them in a *sanitization phase*. Hence, we denote PCG with imperfect correctness and detectable failures *sanitizable pseudorandom correlation generators*.

Our full construction is somewhat technical. Because of the inverse polynomial failure probability, our group-based PCG does not directly fit into the definition of PCG given in Section 4. To simplify the presentation, we therefore first introduce the notion of *sanitizable bilinear correlation generators*, which are PCG for the class of bilinear correlations with an inverse polynomial failure probability whose incorrect outputs can be *detected* efficiently. Then, to proceed with the construction, we introduce an intermediate notion: the notion of *compressible* HSS. Informally, a compressible HSS is an HSS schemes in which the encodings of the inputs can be compressed to a small string when they come from an appropriate distribution. This captures the fact that our construction will build upon the specific homomorphic properties of the group-based HSS of [BGI16a, BGI17, BCG+17] to show that the encodings of (pseudo)random inputs can be efficiently compressed. Then, we show that a compressible Las Vegas degree-2 HSS for bilinear correlations can be used to construct a sanitizable bilinear correlation generator, assuming the learning parity with noise (LPN) assumption. Afterward, we proceed with a description of a compressible Las Vegas degree-2 HSS for bilinear correlations, which builds upon the homomorphic properties of the group-based Las Vegas HSS of [BGI16a, BGI17, BCG+17] when instantiated over a pairing-friendly elliptic curves, and upon the VOLE generator of [BCGI18]. Eventually, we discuss optimizations of our construction.

Notations. For vectors $\boldsymbol{x}_i$ over a multiplicative group, we denote by $\prod_i \boldsymbol{x}_i$ their component-wise product. Given a multiplicative group $\mathbb{G}$ of order q, a length-n vector $\boldsymbol{g} = (g_1, \cdots, g_n) \in \mathbb{G}^n$, and a length-$n$ vector of exponents $\boldsymbol{x} = (x_1, \cdots, x_n) \in \mathbb{Z}_q^n$, we denote $\boldsymbol{x} \bullet \boldsymbol{g}$ the "scalar product in the exponent": $\boldsymbol{x} \bullet \boldsymbol{g} = \prod_{i=1}^n g_i^{x_i}$; for any $g \in \mathbb{G}$, $g^{\boldsymbol{x}}$ denotes $(g^{x_1}, \cdots, g^{x_n})$.

A Note on Additive Bilinear Correlations. The general notion of additive correlations which we consider in this paper are those in which the parties receive shares of some input $\boldsymbol{x}$, as well as shares of $C(\boldsymbol{x})$ for some correlation C. However, when restricting our attention to bilinear correlations, it is more convenient to consider that each party will know *in the clear* one of two inputs, $\boldsymbol{x}$ and $\boldsymbol{y}$, as well as shares of a bilinear function $B(\boldsymbol{x}, \boldsymbol{y})$. Indeed, letting the parties know part of the output in the clear will allow for several non-trivial optimizations of the construction. Furthermore, in the specific case of bilinear correlation, it turns out that this is without loss of generality: the general case (where the inputs are shared as well) can be obtained in a blackbox way using two parallel calls to the PCG for these restricted forms of bilinear correlations. We demonstrate this with the construction below, where BCG_s denotes a

pseudorandom correlation generator for general bilinear correlations, and BCG denotes a PCG for restricted bilinear correlations.

- $\mathsf{BCG}_s.\mathsf{Setup}(1^\lambda)$ outputs $(\mathsf{pp}, \mathsf{sk}) \xleftarrow{\$} \mathsf{BCG.Setup}(1^\lambda)$;
- $\mathsf{BCG}_s.\mathsf{Gen}(\mathsf{sk}, B)$ runs twice $\mathsf{BCG.Gen}(\mathsf{sk}, B)$, and outputs $((\mathsf{k}_0, \mathsf{k}'_0), (\mathsf{k}_1, \mathsf{k}'_1))$;
- $\mathsf{BCG}_s.\mathsf{Expand}(\mathsf{pp}, \sigma, (\mathsf{k}_\sigma, \mathsf{k}'_\sigma))$ runs twice $\mathsf{BCG.Expand}$, and outputs $\boldsymbol{z}_\sigma = ((1-\sigma)\boldsymbol{x}_\sigma + \sigma\boldsymbol{x}'_\sigma, \sigma\boldsymbol{x}_\sigma + (1-\sigma)\boldsymbol{x}'_\sigma, \boldsymbol{y}_\sigma + \boldsymbol{y}'_\sigma + B(\boldsymbol{x}_\sigma, \boldsymbol{x}'_\sigma))$.

Then, it holds that

$$\begin{aligned}\boldsymbol{z}_0 + \boldsymbol{z}_1 &= (\boldsymbol{x}_0 + \boldsymbol{x}'_1, \boldsymbol{x}'_0 + \boldsymbol{x}_1, B(\boldsymbol{x}_0, \boldsymbol{x}_\mathbf{1}) + B(\boldsymbol{x}'_0, \boldsymbol{x}'_1) + B(\boldsymbol{x}_0, \boldsymbol{x}'_0) + B(\boldsymbol{x}_1, \boldsymbol{x}'_1)) \\ &= (\boldsymbol{x}_0 + \boldsymbol{x}'_1, \boldsymbol{x}'_0 + \boldsymbol{x}_1, B(\boldsymbol{x}_0 + \boldsymbol{x}'_1, \boldsymbol{x}'_0 + \boldsymbol{x}_1)) \\ &= (\boldsymbol{x}, \boldsymbol{x}', B(\boldsymbol{x}, \boldsymbol{x}')), \text{ denoting } \boldsymbol{x} = \boldsymbol{x}_0 + \boldsymbol{x}'_1, \boldsymbol{x}' = \boldsymbol{x}'_0 + \boldsymbol{x}_1.\end{aligned}$$

C.1 Sanitizable Bilinear Correlation Generator

Since our group-based construction will be inherently limited to generating *bilinear* correlations, with an inverse polynomial failure probability, we provide a self-contained formal definition of *sanitizable bilinear correlation generators*. A sanitizable BCG is a PCG for bilinear correlations, where correctness holds except with some inverse polynomial probability. To capture the fact that in our construction, a part of the seed generation can be reused accross several instantiations, we add to the definition a Setup algorithm, which produces the reusable part of the seed (which contains public and secret parameters).

Definition 44 (Sanitizable Bilinear Correlation Generator). *A (δ-failure) sanitizable bilinear correlation generator over a ring $\mathcal{R}$ is a triple of algorithms* $(\mathsf{BCG.Setup}, \mathsf{BCG.Gen}, \mathsf{BCG.Expand})$ *with the following syntax:*

- $\mathsf{BCG.Setup}(1^\lambda)$ *is a PPT algorithm that given a security parameter λ, outputs public parameters* pp *and a secret key* sk*;*
- $\mathsf{BCG.Gen}(\mathsf{sk}, B)$ *is a PPT algorithm that given secret key* sk *and a bilinear map $B : \mathcal{R}^n \times \mathcal{R}^n \mapsto \mathcal{R}^m$, outputs a pair of seeds $(\mathsf{k}_0, \mathsf{k}_1)$;*
- $\mathsf{BCG.Expand}(\mathsf{pp}, \sigma, \mathsf{k}_\sigma, \delta)$ *is an algorithm running in time polynomial in λ and $1/\delta$ that, given party index $\sigma \in \{0,1\}$, a seed k_σ, and a failure bound δ, outputs a pair of vectors $(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{R}^n \times \mathcal{R}^m$, as well as a list of m confidence flags $\gamma_{\sigma,i} \in \{\bot, \top\}$ for $i = 1$ to m to indicate full confidence ($\top$) or a possibility of failure ($\bot$) for any given output.*

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

- **δ-Correctness.** *For every $i \le m$ and every polynomial p, there is a negligible ν such that for every positive integer λ, bilinear function $B : \mathcal{R}^n \times \mathcal{R}^n \mapsto \mathcal{R}^m$, and failure bound $\delta > 0$, where $|B|, 1/\delta \le p(\lambda)$, we have:*

$$\Pr[(\gamma_{0,i} = \bot) \wedge (\gamma_{1,i} = \bot)] \le \delta + \nu(\lambda),$$

and

$$\Pr[((\gamma_{0,i} = \top) \vee (\gamma_{1,i} = \top)) \wedge y_{0,i} + y_{1,i} \neq B(\boldsymbol{x}_0, \boldsymbol{x}_1)_i] \le \nu(\lambda),$$

where the probability is taken over

$$(\mathsf{pp}, \mathsf{sk}) \xleftarrow{\$} \mathsf{BCG.Setup}(1^\lambda), (\mathsf{k}_0, \mathsf{k}_1) \xleftarrow{\$} \mathsf{BCG.Gen}(\mathsf{sk}, B)$$

and where we denote $(\boldsymbol{x}_0, \boldsymbol{y}_0) \leftarrow \mathsf{BCG.Expand}(0, \mathsf{k}_0)$, and $(\boldsymbol{x}_1, \boldsymbol{y}_1) \leftarrow \mathsf{BCG.Expand}(1, \mathsf{k}_1)$.

– **Security.** *For any $\sigma \in \{0,1\}$ and any (stateful, nonuniform) polynomial-time adversary $\mathcal{A}$, it holds that*

$$\Pr\left[\begin{array}{ll}(\mathsf{pp},\mathsf{sk}) \xleftarrow{\$} \mathsf{BCG.Setup}(1^\lambda), B \leftarrow \mathcal{A}(\mathsf{pp}), & \\ (\mathsf{k}_0,\mathsf{k}_1) \xleftarrow{\$} \mathsf{BCG.Gen}(\mathsf{sk},B), & : \mathcal{A}(\boldsymbol{x}_\sigma, \mathsf{k}_{1-\sigma}) = 1 \\ (\boldsymbol{x_\sigma}, \boldsymbol{y_\sigma}, \boldsymbol{\gamma_\sigma}) \leftarrow \mathsf{BCG.Expand}(\mathsf{pp},\sigma,\mathsf{k}_\sigma) & \end{array}\right]$$

$$\approx \Pr\left[\begin{array}{ll}(\mathsf{pp},\mathsf{sk}) \xleftarrow{\$} \mathsf{BCG.Setup}(1^\lambda), B \leftarrow \mathcal{A}(\mathsf{pp}), & \\ (\mathsf{k}_0,\mathsf{k}_1) \xleftarrow{\$} \mathsf{BCG.Gen}(\mathsf{sk},B), & : \mathcal{A}(\boldsymbol{x}_\sigma, \mathsf{k}_{1-\sigma}) = 1 \\ \boldsymbol{x_\sigma} \xleftarrow{\$} \mathcal{R}^n, \boldsymbol{\gamma_\sigma} \xleftarrow{\$} \mathsf{Ber}_\delta(\{\bot,\top\})^m & \end{array}\right].$$

C.2 Las Vegas HSS

We recall below the definition of δ-failure HSS (with a Las Vegas correctness guarantee), adapted from [BCG+17].

Definition 45 (Las Vegas Homomorphic Secret Sharing). *A (2-party, secret-key, Las Vegas δ-failure)* Degree-d Homomorphic Secret Sharing (HSS) *scheme over a ring $(\mathcal{R},+,\cdot)$ is a triple of PPT algorithms $\mathsf{HSS} = (\mathsf{HSS.Gen}, \mathsf{HSS.Share}, \mathsf{HSS.Eval})$ with the following syntax:*

- $\mathsf{HSS.Gen}(1^\lambda)$*: On input a security parameter 1^λ, the key generation algorithm outputs a secret key sk and an evaluation key ek.*
- $\mathsf{HSS.Share}(\mathsf{sk}, x)$*: Given secret key sk and secret input value $x \in \mathcal{R}^n$, the sharing algorithm outputs a pair of shares (s_0, s_1). We assume that the input length n is included in each of (s_0, s_1).*
- $\mathsf{HSS.Eval}(\sigma, \mathsf{ek}, s_b, P, \delta)$*: On input party index $\sigma \in \{0,1\}$, evaluation key ek_σ, share s_σ of a size-n input, degree-d arithmetic circuit P with n input bits and m output bits, and a failure bound δ, the homomorphic evaluation algorithm outputs $y_b \in \mathcal{R}^m$, constituting party b's share over $\mathcal{R}$ of an output $y \in \mathcal{R}^m$, as well as a confidence flag $\gamma_b \in \{\bot,\top\}$ to indicate full confidence ($\top$) or a possibility of failure ($\bot$).*

The algorithms $(\mathsf{HSS.Gen}, \mathsf{HSS.Share}, \mathsf{HSS.Eval})$ should satisfy the following correctness and security requirements:

- **Correctness:**
  For every polynomial p there is a negligible ν such that for every sufficiently large integer λ, input $\boldsymbol{x} \in \mathcal{R}^n$, degree-$d$ arithmetic circuit P with input length n, and failure bound $\delta > 0$, where $|P|, 1/\delta \leq p(\lambda)$, we have:

  $$\Pr[(\gamma_0 = \bot) \wedge (\gamma_1 = \bot)] \leq \delta + \nu(\lambda),$$

  and

  $$\Pr[((\gamma_0 = \top) \vee (\gamma_1 = \top)) \wedge \boldsymbol{y}_0 + \boldsymbol{y}_1 \neq P(\boldsymbol{x})] \leq \nu(\lambda),$$

  where probability is taken over

  $$\begin{array}{l}(\mathsf{sk},\mathsf{ek}) \leftarrow \mathsf{HSS.Gen}(1^\lambda); (s_0,s_1) \leftarrow \mathsf{HSS.Share}(\mathsf{sk},\boldsymbol{x}); \\ (\boldsymbol{y}_\sigma, \gamma_\sigma) \leftarrow \mathsf{HSS.Eval}(\sigma,\mathsf{ek},s_\sigma,P,\delta),\ \sigma \in \{0,1\}.\end{array}$$

- **Security:** *For any $\sigma \in \{0,1\}$, any pair of inputs x, x' of the same length, the distribution ensembles $C_\sigma(\lambda, x)$ and $C_\sigma(\lambda, x')$ are computationally indistinguishable, where $C_\sigma(\lambda, y)$ for $y \in \{x, x'\}$ is obtained by sampling $(\mathsf{sk},\mathsf{ek}) \leftarrow \mathsf{HSS.Gen}(1^\lambda)$, sampling $(s_0, s_1) \leftarrow \mathsf{HSS.Enc}(\mathsf{sk}, y)$, and outputting (ek, s_σ).*

C.3 $(\mathcal{D}, \mathsf{comp})$-Compressible HSS

In this section, we introduce a variant of homomorphic secret sharing, called *compressible HSS for a family of distributions* $\mathcal{D}$ (CHSS). Intuitively, a CHSS allows to share inputs sampled from a distribution $\mathcal{D}_n$, and guarantees that the size of each share is upper-bounded by $\mathsf{comp}(\lambda, n)$, where comp is called the *compression ratio* of the CHSS.

Definition 46 ($(\mathcal{D}, \mathsf{comp})$-Compressible HSS). *A (2-party, secret-key, degree-d) compressible homomorphic secret sharing over a ring $\mathcal{R}$ for a family of distributions $\mathcal{D}(\mathcal{R}) = \{\mathcal{D}_n(\mathcal{R})\}_{n\in\mathbb{N}}$ (such that $\mathsf{Im}(\mathcal{D}_n(\mathcal{R})) \subseteq \mathcal{R}^n$) with compression ratio comp, or $(\mathcal{D}(\mathcal{R}), \mathsf{comp})$-CHSS, is a (2-party, secret-key, degree-d) homomorphic secret sharing over $\mathcal{R}$ whose correctness is relaxed as follows:*

- **Relaxed Correctness.** *For every sufficiently large positive integers λ, n and degree-d arithmetic circuit P with input length n, we have:*

$$\Pr[\boldsymbol{y}_0 + \boldsymbol{y}_1 \neq P(\boldsymbol{x})] \leq \mathsf{negl}(\lambda),$$

where probability is taken over

$$(\mathsf{sk}, \mathsf{ek}) \leftarrow \mathsf{HSS.Gen}(1^\lambda);\ \boldsymbol{x} \leftarrow \mathcal{D}(\mathcal{R})_n;\ (s_0, s_1) \xleftarrow{\$} \mathsf{HSS.Share}(\mathsf{sk}, \boldsymbol{x});$$
$$\boldsymbol{y}_b \leftarrow \mathsf{HSS.Eval}(b, \mathsf{ek}, s_b, P),\ b \in \{0,1\},$$

and which satisfies an additional compressibility *property:*

- **Compressibility.** *A $(\mathcal{D}, \mathsf{comp})$-CHSS is* compressible with compression ratio comp *if for every sufficiently large integers λ, n, every input $\boldsymbol{x} \in \mathsf{Im}(\mathcal{D}_n)$, every $(\mathsf{sk}, \mathsf{ek})$ in the image of $\mathsf{HSS.Gen}(1^\lambda)$, and every (s_0, s_1) in the image of $\mathsf{HSS.Share}(\mathsf{sk}, \boldsymbol{x})$, it holds that $|s_\sigma| \leq \mathsf{comp}(\lambda, n)$ for $\sigma = 0, 1$.*

C.4 BCG from LPN and Degree-2 CHSS

Let $\mathcal{R}$ be a ring. Let $\mathcal{D}(\mathcal{R}) = \{\mathcal{D}(\mathcal{R})_n\}_{n\in\mathbb{N}}$ be a family of efficiently sampleable distributions over $\mathcal{R}^n$. Let comp be a compression ratio. Let $\mathsf{HSS} = (\mathsf{HSS.Gen}, \mathsf{HSS.Share}, \mathsf{HSS.Eval})$ be a (2-party secret-key) degree-2 $(\mathcal{D}(\mathcal{R}) \times \mathcal{D}(\mathcal{R}), \mathsf{comp})$-CHSS. We describe below a construction of a bilinear correlation generator over $\mathcal{R}$. Note that any bilinear function $B : \mathcal{R}^n \times \mathcal{R}^n \mapsto \mathcal{R}^m$ can be fully described by a list of m matrices $B_1 \cdots B_m$ with $B_i \in \mathcal{R}^{n\times n}$ such that for any inputs $(\boldsymbol{x}, \boldsymbol{x}') \in \mathcal{R}^n \times \mathcal{R}^n$, $B(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\intercal B_i \boldsymbol{x}')_{i\leq m}$. Let α be a positive constant.

- $\mathsf{BCG.Setup}(1^\lambda)$: output $(\mathsf{pp} = \mathsf{ek}, \mathsf{sk}) \xleftarrow{\$} \mathsf{HSS.Gen}(1^\lambda)$.
- $\mathsf{BCG.Gen}(\mathsf{sk}, B)$: let n denote the input size of B and let $\ell \leftarrow \alpha n$. Let $M \leftarrow \mathbf{C}(\ell, \ell - n, \mathcal{R})$ be the encoding matrix of a linear code, and let $N \in \mathcal{R}^{n\times\ell}$ denote its parity check matrix (*i.e.*, N is the matrix over $\mathcal{R}^{n\times\ell}$ which satisfies $NM = 0$). Pick two vectors $(\boldsymbol{z}_0, \boldsymbol{z}_1) \leftarrow \mathcal{D}_\ell(\mathcal{R}) \times \mathcal{D}_\ell(\mathcal{R})$ and let r_0 (resp. r_1) denote the random coin used to sample $\boldsymbol{z}_0$ (resp. $\boldsymbol{z}_1$). Run $(s_0, s_1) \leftarrow \mathsf{HSS.Share}(\mathsf{sk}, (\boldsymbol{z}_0, \boldsymbol{z}_1))$. Output $\mathsf{k}_\sigma \leftarrow (r_\sigma, s_\sigma)$ for $\sigma = 0, 1$.
- $\mathsf{BCG.Expand}(\mathsf{pp}, \sigma, \mathsf{k}_\sigma)$: parse k_σ as (r_σ, s_σ) and reconstruct $\boldsymbol{z}_\sigma$ from r_σ (using the sampling procedure of $\mathcal{D}_n(\mathcal{R})$). Define the bilinear function $P : \mathcal{R}^\ell \times \mathcal{R}^\ell \mapsto \mathcal{R}^m$ as follows: on input $(\boldsymbol{x}, \boldsymbol{x}') \in \mathcal{R}^\ell \times \mathcal{R}^\ell$, P outputs $(\boldsymbol{x}^\intercal \cdot B_i' \cdot \boldsymbol{x}')_{i\leq m}$, where B_i' is defined as $B_i' \leftarrow N^\intercal B_i N$. Set $\boldsymbol{x}_\sigma \leftarrow N\boldsymbol{z}_\sigma$. Compute $\boldsymbol{y}_\sigma \leftarrow \mathsf{HSS.Eval}(\sigma, \mathsf{pp}, s_\sigma, P)$, and output $(\boldsymbol{x}_\sigma, \boldsymbol{y}_\sigma)$.

Theorem 47. *Assuming the $(\mathcal{D}(\mathcal{R}), \mathbf{C})$-$\mathsf{LPN}(\ell, \ell - n)$ assumption, the above construction is a bilinear correlation generator with seed size upper-bounded by $2 \cdot \mathsf{comp}(\lambda, \ell)$.*

C.5 Proof of Theorem 47

C.5.1 Correctness.

As $(\boldsymbol{z}_0, \boldsymbol{z}_1)$ is sampled from $\mathcal{D}_\ell(\mathcal{R}) \times \mathcal{D}_\ell(\mathcal{R})$, the relaxed correctness property of the CHSS applies, and we get:

$$\begin{aligned}\boldsymbol{y}_0 + \boldsymbol{y}_1 &= \mathsf{HSS.Eval}(0, \mathsf{ek}, s_0, P) + \mathsf{HSS.Eval}(1, \mathsf{ek}, s_1, P) \\ &= (\boldsymbol{z}_0^\intercal B_i' \boldsymbol{z}_1)_{i \le m} = ((N\boldsymbol{z}_0)^\intercal B_i (N\boldsymbol{z}_1))_{i \le m} \\ &= (\boldsymbol{x}_0^\intercal B_i \boldsymbol{x}_1)_{i \le m} = B(\boldsymbol{x}_0, \boldsymbol{x}_1).\end{aligned}$$

C.5.2 Security.

Let $\mathcal{A}$ be a (stateful, nonuniform) PPT adversary, and let σ be a bit. We proceed through a sequence of game.

- **Game** G_0**.** In this game, we run $(\mathsf{pp}, \mathsf{sk}) \xleftarrow{\$} \mathsf{BCG.Setup}(1^\lambda)$, set $B \leftarrow \mathcal{A}(\mathsf{pp})$, $(\mathsf{k}_0, \mathsf{k}_1) \xleftarrow{\$} \mathsf{BCG.Gen}(\mathsf{sk}, B)$, and
$$(\boldsymbol{x}_\sigma, \boldsymbol{y}_\sigma) \leftarrow \mathsf{BCG.Expand}(\mathsf{pp}, \sigma, \mathsf{k}_\sigma).$$
This corresponds to the first experiment in the security definition of bilinear correlation generators. Let b_0 be the output of $\mathcal{A}(\boldsymbol{x}_\sigma, \mathsf{k}_{1-\sigma})$.
- **Game** G_1**.** In this game, we modify the execution of $\mathsf{BCG.Gen}$ as follows: instead of computing $(s_0, s_1) \leftarrow \mathsf{HSS.Share}(\mathsf{sk}, (\boldsymbol{z}_0, \boldsymbol{z}_1))$, we set $(s_0, s_1) \leftarrow \mathsf{HSS.Share}(\mathsf{sk}, 0^{2\ell})$. Let b_1 denote the output of $\mathcal{A}$ in this game. By the security property of the CHSS, the distribution of $(\mathsf{ek}, s_{1-\sigma})$ in this game is computationally indistinguishable from the distribution of $(\mathsf{ek}, s_{1-\sigma})$ in G_0, hence we have $\Pr[b_0 \neq b_1] = \mathsf{negl}(\lambda)$.
- **Game** G_2**.** In this game, we modify the execution of $\mathsf{BCG.Expand}$ as follows: instead of computing $\boldsymbol{x}_\sigma$ as $N\boldsymbol{z}_\sigma$, we pick $\boldsymbol{x}_\sigma \xleftarrow{\$} \mathcal{R}^n$. Note that $\mathsf{k}_{1-\sigma}$ does not depend on $\boldsymbol{z}_\sigma$, hence distinguishing between the games G_2 and G_1 amounts to distinguishing $N\boldsymbol{z}_\sigma$ from a random vector over $\mathcal{R}^n$, where $\boldsymbol{z}_\sigma$ is drawn from $\mathcal{D}_\ell(\mathcal{R})$. Note also that $N\boldsymbol{z}_\sigma = N(M\boldsymbol{a} + \boldsymbol{z}_\sigma)$ for any vector $\boldsymbol{a} \in \mathcal{R}^n$ (as $NM = 0$). Therefore, this amounts to distinguishing $M\boldsymbol{a} + \boldsymbol{z}_\sigma$ from random, for an arbitrary secret vector $\boldsymbol{a} \in \mathcal{R}^n$, and a noise vector $\boldsymbol{z}_\sigma$ sampled from $\mathcal{D}_\ell(\mathcal{R})$, which is infeasible under the $(\mathcal{D}(\mathcal{R}), \mathbf{C}) - \mathsf{LPN}(\ell, \ell - n)$ assumption. Therefore, denoting b_2 the output of $\mathcal{A}$ in G_2, we have $\Pr[b_1 \neq b_2] = \mathsf{negl}(\lambda)$.
- **Game** G_3**.** In this game, we revert the change made in G_1 and compute again (s_0, s_1) as $\mathsf{HSS.Share}(\mathsf{sk}, (\boldsymbol{z}_0, \boldsymbol{z}_1))$. Let b_3 denote the output of $\mathcal{A}$ in this game. By the security property of the CHSS, we have $\Pr[b_2 \neq b_3] = \mathsf{negl}(\lambda)$. Furthermore, this game is exactly the second experiment in the security definition of a BCG, which concludes the proof.

C.5.3 Efficiency.

The seed size of the BCG is $|\mathsf{k}_\sigma| = |r_\sigma| + |s_\sigma|$. By the compressibility of the CHSS, $|s_\sigma| \le \mathsf{comp}(\lambda, \ell)$. Furthermore, by the restricted correctness of the CHSS, it must hold that $|r_\sigma| \le |s_\sigma| \le \mathsf{comp}(\lambda, \ell)$ (otherwise, using $\mathsf{HSS.Share}$ and $\mathsf{HSS.Eval}$ would allow to compress samples from $\mathcal{D}_\ell(\mathcal{R})$ below their amount of entropy, which is impossible). Hence, we get $|\mathsf{k}_\sigma| \le 2 \cdot \mathsf{comp}(\lambda, \ell)$.

C.5.4 Sanitizable BCG from LPN and Compressible Las Vegas HSS.

Given a compressible Las Vegas HSS, one immediately gets a sanitizable BCG under the LPN assumption, using the above construction. The security analysis of the construction is almost identical, with the failure probability of Las Vegas HSS translating directly to the failure probability of BCG outputs. Note that the standard formulation of Las Vegas HSS considers a global

failure probability for the entire vector output, and outputs a single flag in $\{\top, \bot\}$ to indicate correctness of the output, while our definition of sanitizable BCG requires outputting a different flag for each output (with the goal of later removing faulty outputs while keeping correct ones). This is essentially a syntactic difference: existing construction of Las Vegas HSS can easily be modified to output a flag for each output bit. In the formal construction of sanitizable BCG from Las Vegas HSS, it suffices to apply the Las Vegas HSS independently for each functions f_i outputting the ith bit of the target bilinear correlation, to get independent failure flags for each output bit.

C.6 Group-Based HSS for Bilinear Functions

In this section, we provide an overview of the group-based homomorphic secret sharing scheme first introduced in [BGI16a], and subsequently optimized in [BGI17, BCG+17, DKK18]. When restricted to bilinear functions, as observed in [BCG+17, Section 4.6], the scheme can be considerably simplified and optimized. Below, we briefly recall the HSS scheme of [BGI16a], taking into account the optimizations for bilinear functions of [BCG+17]. The scheme allows to compute bilinear functions over any ring $\mathbb{Z}_t$ of polynomial size; it relies on a group $\mathbb{G}$ where the discrete log is conjectured to be hard.

C.6.1 Encoding $\mathbb{Z}_q$ Elements.

Let q be a large prime, and let $\mathbb{G}$ be a hard-discrete-log group of order q. Let g denote a generator of $\mathbb{G}$. For any $x \in \mathbb{Z}_q$, we consider the following 3 types of two-party encodings:

Level 1: "Encryption." For $x \in \mathbb{Z}_q$, we let $[x]$ denote g^x, and $[\![x]\!]_s$ denote $([r], [r \cdot s + x])$ for a uniformly random $r \in \mathbb{Z}_q$, which corresponds to an ElGamal encryption of x with a secret key $s \in \mathbb{Z}_q$. All level-1 encodings are known to both parties. We let $\mathsf{sk} \leftarrow (s, -1)$.

Level 2: "Additive shares." Let $\langle x \rangle$ denote a pair of shares $x_0, x_1 \in \mathbb{Z}_q$ such that $x_0 = x_1 + x$, where each share is held by a different party. We let $\langle\!\langle x \rangle\!\rangle_s$ denote $(\langle -s \cdot x \rangle, \langle x \rangle) = \mathsf{sk} \cdot \langle x \rangle \in (\mathbb{Z}_q^2)^2$, namely each party holds one share of $\langle -s \cdot x \rangle$ and one share of $\langle x \rangle$. Note that both types of encodings are additively homomorphic over $\mathbb{Z}_q$, namely given encodings of x and x' the parties can locally compute a valid encoding of $x + x'$.

Level 3: "Multiplicative shares." Let $\{x\}$ denote a pair of shares $x_0, x_1 \in \mathbb{G}$ such that the difference between their discrete logarithms is x. That is, $x_0 = x_1 \cdot g^x$.

C.6.2 Operations on Encodings.

All types of encodings allow to locally evaluate linear functions (they are additively homomorphic). To evaluate degree-2 polynomials, we define a multiplication algorithm Mult which, given a level 1 encoding of an inpuy x and a level 2 encoding of an input y, outputs additive shares of xy. To this end, we consider the following two types of operations, performed locally by the two parties:

1. $\mathsf{Pair}([\![x]\!]_s, \langle\!\langle y \rangle\!\rangle_s) \mapsto \{xy\}$. This pairing operation exploits the fact that the decryption of an ElGamal ciphertext $[\![x]\!]_s$ is computed as a scalar product in the exponent: $g^x = \mathsf{sk} \bullet [\![x]\!]_s$. Therefore, Pair computes $\langle\!\langle y \rangle\!\rangle_s \bullet [\![x]\!]_s = \langle y \rangle \bullet (\mathsf{sk} \bullet [\![x]\!]_s) = \{xy\}$. Note that we consider ElGamal ciphertexts for the sake of concreteness only; any encryption scheme whose decryption follows a similar "scalar product in the exponent" structure would suffice.
2. $\mathsf{Convert}(\{z\}, \delta) \mapsto \langle z \rangle$, with failure bound δ. The implementation of $\mathsf{Convert}$ is also given an upper bound M on the "payload" z ($M = 1$ by default), and its expected running time grows linearly with M/δ. We omit M from the following notation.

The Convert algorithm works as follows. Each party, on input $h \in \mathbb{G}$, outputs $i \bmod t$, with i the minimal integer $i \geq 0$ such that $h \cdot g^i$ is "distinguished," where roughly a δ-fraction of the group elements are distinguished. Distinguished elements were picked in [BGI16a] by applying a pseudo-random function to the description of the group element. An optimized conversion procedure from [BGI17, BCG+17] applies the heuristic of defining a group element to be distinguished if its bit-representation starts with a 1 followed by $d \approx \log_2(M/\delta)$ leading 0's. Note that this heuristic only affects the running time and not security, and thus it can be validated empirically. Correctness of Convert holds if no group element *between* the two shares $\{z\} \in \mathbb{G}^2$ is distinguished. Finally, Convert signals that there is a potential failure if there is a distinguished point in the "danger zone." Namely, Party $b = 0$ (resp., $b = 1$) raises a potential error flag if $h \cdot g^{-i}$ (resp., $h \cdot g^{i-1}$) is distinguished for some $i = 1, \ldots, M$.

The Convert algorithm requires an upper bound M on the multiplicatively shared exponent, and runs in time proportional to M. Concretely, this means that we only consider inputs coming from a set $\mathbb{Z}_t$ where t is polynomial, so that the product between any two (linear combination of) inputs is polynomially bounded. Note that all operations are performed over $\mathbb{Z}_q$, where q is an exponentially large prime; as all inputs are lower than t, which is polynomial, no modular reduction ever occurs, hence the computation of i is performed over the integers; therefore, the parties obtain additive shares of the product over $\mathbb{Z}_t$ after the final (local) reduction of i modulo t. We refer the reader to [BCG+17] for a detailed analysis and further optimizations of the Convert procedure.

Given the Pair and Convert algorithms, the multiplication algorithm Mult sequentially executes these two operations: $\mathsf{Mult}(\llbracket x \rrbracket_c, \langle\!\langle y \rangle\!\rangle_c, \delta) \mapsto \langle xy \rangle$, with error δ. Note that the output of the procedure is not a level 1 or a level 2 encoding. Therefore, this δ-failure HSS allows to evaluate arbitrary bilinear functions B on vectors $(\boldsymbol{x}, \boldsymbol{y})$ (encodings of level 1 and 2, as well as additive shares, being additively homomorphic) but does not generalize immediately to functions of higher degree; generalizations to branching programs are presented in [BGI16a,BGI17,BCG+17], but are several orders of magnitude less efficient.

C.6.3 Improved Conversion.

In a recent paper [DKK18], Dinur *et al.* designed an improved distributed discrete logarithm procedure. Their elegant algorithm is based on a clever random walk with steps of varying length, that bears some resemblance with Pollard's kangaroo method (but requires a considerably more complex and mathematically involved analysis). The improved algorithm requires T multiplications to achieve a failure probability of $O(M/T^2)$, where the constant hidden in the $O(\cdot)$ notation is always upper bounded by $2^{10.2}$, and in practice approximately equal to 400 (the authors provide a table with optimal parameters for various choices of T, that gives the exact constant), improving over the $O(M/T)$ failure probability of the previous method. Their paper also proves the optimality of this algorithm. In this work, we will rely on their improved conversion algorithm, and refer the reader to [DKK18] for further details on this procedure.

C.7 The BGN-EG Cryptosystem

The Boneh-Goh-Nissim cryptosystem (BGN) was introduced in [BGN05]. It is a variant of the ElGamal cryptosystem over composite-order pairing-friendly elliptic curve, which allows to homomorphically compute any degree-2 polynomial on encrypted plaintexts, provided that the output is of polynomial size (as decryption requires computing a discrete logarithm). An adaptation of BGN to prime-order pairing-friendly elliptic curves was introduced by Freeman in [Fre10]. As it suffices for our purpose, we will rely in this work on a simplified variant of Freeman's cryptosystem, where encryption in any of the pairing-friendly groups will exactly be ElGamal encryption, and where ciphertexts obtained through homomorphic operations are not rerandomized.

Let $\mathsf{BilinearGen}$ denote a PPT algorithm which, on input 1^λ, outputs a prime q, the description of three cyclic groups $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t)$, elements $(g_1, g_2) \in \mathbb{G}_1 \times \mathbb{G}_2$, and a map e such that

- the cyclic groups have the same order $q = q(\lambda)$;
- the map $e : \mathbb{G}_1 \times \mathbb{G}_2 \mapsto \mathbb{G}_t$ is an efficiently computable non-degenerate bilinear map, *i.e.*, $\forall (u, v) \in \mathbb{G}_1 \times \mathbb{G}_2$ and $(a, b) \in \mathbb{Z}_p$, $e(u^a, v^b) = e(u, v)^{ab}$;
- g_i generates $\mathbb{G}_i$ for $i \in \{1, 2\}$ (and $g_t \leftarrow e(g_1, g_2)$ generates $\mathbb{G}_t$).

Note that this captures groups equipped with an *asymmetric* pairing. To simplify notations, given vectors $\boldsymbol{x} \in \mathbb{G}_1^n, \boldsymbol{y} \in \mathbb{G}_2^n$, we will write $e(\boldsymbol{x}, \boldsymbol{y})$ to denote $(e(x_i, y_j))_{i,j \leq n}$. When it happens that the components of the vector are vectors themselves (e.g. if $\boldsymbol{x}, \boldsymbol{y}$ are vectors of ciphertexts, each ciphertext consisting of several group elements), we apply the notation recursively: $e(\boldsymbol{x}, \boldsymbol{y}) = (e(x_i, y_j))_{i,j \leq n}$ and for every i, j, $e(x_i, y_j) = (e(x_{i,i'}, y_{j,j'}))_{i',j'}$.

We outline below our simplified variant of Freeman's adaptation of the BGN cryptosystem, which we denote $\mathsf{BGN\text{-}EG}$.

- $\mathsf{BGN\text{-}EG.Setup}(1^\lambda)$: output $\mathsf{pp} = (q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, g_1, g_2, e) \xleftarrow{\$} \mathsf{BilinearGen}(1^\lambda)$.
- $\mathsf{BGN\text{-}EG.KeyGen}(\mathsf{pp})$: pick $(s_1, s_2) \xleftarrow{\$} \mathbb{Z}_q^2$, compute $(h_1, h_2) \leftarrow (g_1^{s_1}, g_2^{s_2})$. Set $\mathsf{sk}_i \leftarrow (s_i, -1)$ for $i = 1, 2$, and $\mathsf{sk}_t = (s_1 \cdot s_2, -s_1, -s_2, 1)$. Output $\mathsf{pk} \leftarrow (\mathsf{pp}, h_1, h_2)$ and $\mathsf{sk} \leftarrow (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{sk}_t)$.
- $\mathsf{BGN\text{-}EG.Enc}_i(\mathsf{pk}, m; r)$: on input the public key pk, a message $m \in \mathbb{Z}_q$, and a random coin $r \in \mathbb{Z}_q$, output $\boldsymbol{c}_i \leftarrow (g_i^r, h_i^r g_i^m)$ (this corresponds to encryption over $\mathbb{G}_i$).
- $\mathsf{BGN\text{-}EG.Dec}_i(\mathsf{sk}, \boldsymbol{c}_i)$: on input the secret key sk and a ciphertext $\boldsymbol{c}_i$, compute $c \leftarrow \mathsf{sk}_i \bullet \boldsymbol{c}_i$ and output $m \leftarrow \mathsf{dlog}_{g_i}(c)$.

Correctness follows by inspection; security reduces to the decisional Diffie-Hellman assumption (DDH) in $\mathbb{G}_1$ and $\mathbb{G}_2$. It is easy to see that the scheme is additively homomorphic over each of $\mathbb{G}_1$ and $\mathbb{G}_2$. Furthermore, given an encryption $\boldsymbol{c}_1$ of a message m_1 over $\mathbb{G}_1$ and an encryption $\boldsymbol{c}_2$ of a message m_2 over $\mathbb{G}_2$, one can homomorphically construct an encryption of $m_1 m_2$ over $\mathbb{G}_t$ as follows: compute $\boldsymbol{c}_t \leftarrow e(\boldsymbol{c}_1, \boldsymbol{c}_2)$. The resulting ciphertext has four components and remains additively homomorphic over $\mathbb{G}_t$ (addition of plaintext is computed by component-wise multiplication). Decryption of a ciphertext over $\mathbb{G}_t$ is performed by computing $c \leftarrow \mathsf{sk}_t \bullet \boldsymbol{c}_t$, and outputting $m \leftarrow \mathsf{dlog}_{g_t}(c)$, where $g_t = e(g_1, g_2)$.

C.8 Compressible HSS from BGN-EG

We now show how the group-based HSS of [BGI16a, BGI17, BCG+17] can be modified to get a compressible HSS. In this section, we will consider compressible HSS over a small (polynomial-size) ring $\mathbb{Z}_t$ for inputs drawn from the Bernouilli distribution $\mathsf{Ber}_r(\mathbb{Z}_t)$ for some rate r (see Section 3.3); that is, we will consider inputs with a sparse structure, and show how level 1 and level 2 encodings of such inputs can be compressed. Let $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t)$ denote three cyclic groups of order q, and let $e : \mathbb{G}_1 \times \mathbb{G}_2 \mapsto \mathbb{G}_t$ denote a pairing. Observe that the ciphertexts in the target group $\mathbb{G}_t$ of $\mathsf{BGN\text{-}EG}$ have a suitable structure to be used as level-1 encodings in the HSS scheme, as decryption of a ciphertext over $\mathbb{G}_t$ is performed via a scalar product in the exponent.

C.8.1 Modifying the DDH-Based HSS.

We modify the HSS scheme of the previous section as follows: a level 1 encoding $[\![x]\!]_{\mathsf{sk}}$ of x is a $\mathsf{BGN\text{-}EG}$ encryption of x in the target group $\mathbb{G}_t$. A level 2 encoding $\langle\!\langle y \rangle\!\rangle_{\mathsf{sk}}$ of a message y is a 4-tuple $\mathsf{sk}_t \cdot \langle y \rangle$. The Pair algorithm, on input a level 1 encoding $[\![x]\!]_{\mathsf{sk}}$ and level 2 shares $\langle\!\langle y \rangle\!\rangle_{\mathsf{sk}}$, outputs $\langle\!\langle y \rangle\!\rangle_{\mathsf{sk}} \bullet [\![x]\!]_{\mathsf{sk}}$; a level 3 encoding of a message z is a multiplicative share over $\mathbb{G}_t$ of g_t^z; it follows that $\mathsf{Pair}([\![x]\!]_{\mathsf{sk}}, \langle\!\langle y \rangle\!\rangle_{\mathsf{sk}}) = \{xy\}$. Then, the parties can apply the optimized $\mathsf{Convert}$ procedure of [DKK18] to obtain additive shares of xy. The security of this modified HSS scheme immediatly reduces to the DDH assumption in $\mathbb{G}_1$ and $\mathbb{G}_2$, by the same security analysis as for the HSS of [BGI16a] (note that, because we restrict our attention to HSS for

bilinear function, we do not need to use circularly-secure variants of ElGamal to get security under DDH; a circularly-secure scheme is needed in [BGI16a] because encryptions of the secret key must be released to allow for the evaluation of more complex functions).

C.8.2 Compressing Level 1 Encodings under BGN-EG.

The modified scheme suggests a natural strategy to reduce the size of level 1 encodings when the input is sparse, by exploiting the homomorphic properties of BGN-EG ciphertexts. Let $\boldsymbol{m}$ be a length-ℓ vector over $\mathbb{Z}_q$, which has at most k non-zero coordinates. We assume ℓ to be a square for simplicity. The compression method works as follows: arrange the coordinates of $\boldsymbol{m}$ in a $\sqrt{\ell} \times \sqrt{\ell}$ matrix M. Decompose M into $\sum_{i=1}^{k} M_i$, where each matrix M_i has a single non-zero coordinate. For $i = 1$ to k, let $(u_i, v_i) \in [\sqrt{\ell}] \times [\sqrt{\ell}]$ denote the coordinate of the non-zero entry of M_i.

Pick $2\sqrt{\ell}$ elements $(\alpha_{i,j}, \beta_{i,j})_{j \leq \sqrt{\ell}}$ of $\mathbb{Z}_q$ as follows: for each pair $(u, v) \neq (u_i, v_i)$, set $\alpha_{i,u} = \beta_{i,v} = 0$, and set $\alpha_{i,u_i} = 1$, $\beta_{i,v_i} = M_i|_{u_i,v_i}$. Observe that, by construction, it holds that $M_i|_{u,v} = \alpha_{i,u} \cdot \beta_{i,v}$ for any pair $(u, v) \in [\sqrt{\ell}] \times [\sqrt{\ell}]$. Therefore, we have for any $(u, v) \in [\sqrt{\ell}] \times [\sqrt{\ell}]$:

$$M|_{u,v} = \sum_{i=1}^{k} \alpha_{i,u} \cdot \beta_{i,v}.$$

This shows that for any vector $\boldsymbol{m}$ with at most k non-zero entries, one can create a level 1 encoding of $\boldsymbol{m}$ as follows: compute the $(\alpha_{i,u}, \beta_{i,u})_{i \leq k, u \leq \sqrt{\ell}}$ as above, and set the encoding of $\boldsymbol{m}$ to be $k \cdot \sqrt{\ell}$ BGN-EG encryptions of $(\alpha_{i,u})_{i,u}$ over $\mathbb{G}_1$, and $k \cdot \sqrt{\ell}$ BGN-EG encryptions of $(\beta_{i,u})_{i,u}$ over $\mathbb{G}_2$. Using the homomorphic properties of BGN-EG, any party can then locally reconstruct a BGN-EG encryption of $\boldsymbol{m} = (\sum_{i=1}^{k} \alpha_{i,u} \cdot \beta_{i,v})_{u,v}$ over $\mathbb{G}_t$. The size of the compressed encoding is $2k \cdot \sqrt{\ell} \cdot (|\mathbb{G}_1| + |\mathbb{G}_2|)$. Note that this strategy corresponds exactly to using the LPN-based PRG introduced in Section 4.4 to stretch the encoded seed.

C.8.3 Compressing Level 2 Encodings using FSS.

We now turn our attention to level 2 encodings $\langle\!\langle \boldsymbol{m} \rangle\!\rangle_{\mathsf{sk}} = (\langle s_1 s_2 \cdot \boldsymbol{m} \rangle, \langle -s_1 \cdot \boldsymbol{m} \rangle, \langle -s_2 \cdot \boldsymbol{m} \rangle, \langle \boldsymbol{m} \rangle)$. Note that if $\boldsymbol{m}$ is a sparse vector, so are $s_1 s_2 \cdot \boldsymbol{m}$, $-s_1 \cdot \boldsymbol{m}$, and $-s_2 \cdot \boldsymbol{m}$. We therefore focus on compressing additive shares of arbitrary sparse vectors over $\mathbb{Z}_q$. As was observed recently in [BCGI18], this type of correlation can be efficiently compressed using function secret sharing for multi-point functions (MPFSS). We elaborate below.

Let $\mathsf{MPFSS} = (\mathsf{MPFSS.Gen}, \mathsf{MPFSS.Eval}, \mathsf{MPFSS.FullEval})$ be a multi-point function secret sharing. On input a vector $\boldsymbol{m} \in \mathbb{Z}_q^{\ell}$ with $\mathsf{HW}(\boldsymbol{m}) \leq k$, let $\mathsf{proj}_{\boldsymbol{m}} : [\ell] \mapsto \mathbb{Z}_q$ be the function which, on input $i \in [\ell]$, outputs the i'th coordinate of $\boldsymbol{m}$. Note that $\mathsf{proj}_{\boldsymbol{m}}$ is an (ℓ, k)-multi-point function over $\mathbb{Z}_q$. To generate compressed additive shares of $\boldsymbol{m}$, compute $(K_0, K_1) \xleftarrow{\$} \mathsf{MPFSS.Gen}(1^{\lambda}, \mathsf{proj}_{\boldsymbol{m}})$. To decompress the string, each party with input K_σ computes $\mathsf{MPFSS.FullEval}(\sigma, K_\sigma)$, obtaining additive shares of $(\mathsf{proj}_{\boldsymbol{m}}(i))_{i \in [\ell]} = \boldsymbol{m}$. Correctness immediately follows from the correctness of the underlying MPFSS. Regarding security, we must show that for any pair $(\boldsymbol{m}, \boldsymbol{m'})$ (with $\mathsf{HW}(\boldsymbol{m}), \mathsf{HW}(\boldsymbol{m'}) \leq k$) and any $\sigma \in \{0, 1\}$, the distribution of $(\mathsf{ek}_\sigma, s_\sigma)$ obtained by sampling $(\mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{HSS.Gen}(1^{\lambda})$, sampling $(s_0, s_1) \leftarrow \mathsf{HSS.Enc}(\mathsf{sk}, \boldsymbol{m})$ or $(s_0, s_1) \leftarrow \mathsf{HSS.Enc}(\mathsf{sk}, \boldsymbol{m'})$, and outputting $(\mathsf{ek}_\sigma, s_\sigma)$, are computationally indistinguishable. This immediately follows from the fact that, by the MPFSS security, there is a simulator which (given $\mathsf{Leak}(\mathsf{proj}_{\boldsymbol{m}}) = \mathsf{Leak}(\mathsf{proj}_{\boldsymbol{m'}})$, and no further information about $\boldsymbol{m}, \boldsymbol{m'}$) can output a simulated key $s_\sigma = K_\sigma$ whose distribution is indistinguishable from an honestly generated key.

Using the PRG-based MPFSS of [BGI16b, BCGI18], the size of a compressed encoding using this method is equal to $k \cdot (\lceil \log \ell \rceil \cdot (\lambda + 2) + \lambda + \lceil \log q \rceil)$. We refer the reader to [BCGI18] for further discussions on this method and optimizations of MPFSS tailored to this application.

C.8.4 Putting the Pieces Together.

Combining the above techniques, we get

Theorem 48. *Assuming the DDH assumption over pairing-friendly elliptic curves, for any integer t of polynomial size and any integer k, there exists a $(\mathsf{Ber}_{k/\ell}(\mathbb{Z}_t), \mathsf{comp}(\lambda, \ell, k))$-compressible (Las Vegas, secret-key, degree-2) CHSS, with*

$$\mathsf{comp}(\lambda, \ell, k) = k \cdot \sqrt{\ell} \cdot \mathsf{poly}(\lambda).$$

For self-containement, we describe below the full CHSS. It has two sharing algorithms, corresponding to level 1 and level 2 shares respectively. The evaluation procedure allows to compute (shares of) any bilinear function $B(\boldsymbol{x}, \boldsymbol{y})$ where $\boldsymbol{x}$ is level-1-shared and $\boldsymbol{y}$ is level-2-shares between the parties.

- $\mathsf{HSS.Gen}(1^\lambda)$: run $\mathsf{pp} = (q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, g_1, g_2, e) \xleftarrow{\$} \mathsf{BGN\text{-}EG.Setup}(1^\lambda)$, as well as $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{BGN\text{-}EG.KeyGen}(\mathsf{pp})$. Compute $g \leftarrow e(g_1, g_2)$. Output $\mathsf{ek} \leftarrow (\mathsf{pp}, \mathsf{pk}, g)$ and sk.
- $\mathsf{HSS.Share}_1(\mathsf{sk}, \boldsymbol{m})$: let k denote an upper bound on the sparsity of $\boldsymbol{m}$, and let ℓ denote the length of $\boldsymbol{m}$. Let $(\boldsymbol{\alpha}_i, \boldsymbol{\beta}_i)_{i \le k}$ denote the decomposition of $\boldsymbol{m}$ in $2k$ length-$\sqrt{\ell}$ vectors, as described in Section C.8. Compute, for $i = 1$ to k, $(\boldsymbol{c}_i, \boldsymbol{d}_i) \xleftarrow{\$} (\mathsf{BGN\text{-}EG.Enc}_1(\mathsf{pk}, \boldsymbol{\alpha}_i), \mathsf{BGN\text{-}EG.Enc}_2(\mathsf{pk}, \boldsymbol{\beta}_i))$ and output $\mathsf{share}_0 = \mathsf{share}_1 = (\boldsymbol{c}_i, \boldsymbol{d}_i)_{i \le k}$.
- $\mathsf{HSS.Share}_2(\mathsf{sk}, \boldsymbol{m})$: let k denote an upper bound on the sparsity of $\boldsymbol{m}$, and let ℓ denote the length of $\boldsymbol{m}$. Parse sk as $(\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{sk}_t)$ and parse sk_i as $(s_i, -1)$ for $i = 1, 2$. Let $\mathbf{proj}_{\boldsymbol{m}, \mathsf{sk}} \leftarrow (\mathsf{proj}_{s_1 s_2 \boldsymbol{m}}, \mathsf{proj}_{-s_1 \boldsymbol{m}}, \mathsf{proj}_{-s_2 \boldsymbol{m}}, \mathsf{proj}_{\boldsymbol{m}})$, where proj is defined as in Section C.8.3. Compute

  $$(\boldsymbol{K}_0, \boldsymbol{K}_1) \leftarrow \mathsf{MPFSS.Gen}(1^\lambda, \mathbf{proj}_{\boldsymbol{m}, \mathsf{sk}}).$$

  Return $\mathsf{share}'_0 \leftarrow \boldsymbol{K}_0$ and $\mathsf{share}'_1 \leftarrow \boldsymbol{K}_1$.
- $\mathsf{HSS.Eval}(\sigma, \mathsf{ek}, \mathsf{share}_\sigma, \mathsf{share}'_\sigma, B, \delta)$: On input party index $\sigma \in \{0, 1\}$, evaluation key ek, level 1 share share_σ of a size-ℓ input, level 2 share share'_σ of a size-ℓ input, a bilinear function $B : \mathbb{Z}_t^\ell \times \mathbb{Z}_t^\ell \mapsto \mathbb{Z}_t^m$ with $B(\boldsymbol{x}, \boldsymbol{y}) = (\sum_{i,j} b_{i,j,\theta} \cdot x_i y_j)_{\theta \le m}$, and failure probability bound $\delta > 0$:
  - Parse share_σ as $(\boldsymbol{c}_i, \boldsymbol{d}_i)_{i \le k}$, and compute ℓ ciphertexts $(e_1, \cdots, e_\ell) \leftarrow \prod_{i=1}^k e(\boldsymbol{c}_i, \boldsymbol{d}_i)$.
  - Parse share'_σ as $\boldsymbol{K}_\sigma$. Compute $\boldsymbol{K}'_\sigma \leftarrow \mathsf{MPFSS.FullEval}(\sigma, \boldsymbol{K}_\sigma)$. Note that $\boldsymbol{K}'_\sigma$ is a vector of ℓ length-4 vectors $\boldsymbol{K}'_{\sigma,i}$.
  - For every $(i, j) \in [\ell]^2$, $r_{i,j} \leftarrow \mathsf{Pair}(e_i, \boldsymbol{K}'_{\sigma,j}) = \boldsymbol{K}'_{\sigma,j} \bullet e_i$.
  - For $\theta = 1$ to m, let $h_\theta \leftarrow (b_{i,j,\theta})_{i,j} \bullet (r_{i,j})_{i,j}$.
  - Output $(\mathsf{Convert}(h_i, \delta/m))_{i \le m}$, where $\mathsf{Convert}$ is run in base g over $\mathbb{G}_t$.

Plugging the above HSS in our LPN-based construction of BCG from compressible HSS, we get:

Theorem 49. *Assuming the DDH assumption over pairing-friendly elliptic curves and the $\mathsf{LPN}((\alpha - 1) \cdot n, \alpha \cdot n, k/(\alpha n))$ assumption over an integer ring $\mathbb{Z}_t$ of polynomial size, for a positive constant $\alpha > 1$, there exists a δ-failure SBCG with seed size $k \cdot (\alpha \cdot n)^{1/2} \cdot \mathsf{poly}(\lambda)$.*

D Optimizing the Group-Based PCG

While the compressible HSS described in Section C leads to a (sanitizable) PCG for bilinear correlations under the LPN assumption, a direct instantiation from the above group-based CHSS and LPN would remain computationally heavy. Below, we outline several optimizations to reduce the cost of group-based PCG, which significantly reduce the computation and size of the PCG seeds.

D.1 General Optimizations

In the construction of BCG from CHSS of Section C.4, two vectors $\boldsymbol{z}_0$ and $\boldsymbol{z}_1$ are sampled and shared between the parties using $\mathsf{HSS.Share}$, and each party gets to know one of these vectors. Using the group-based CHSS, this means that one party knows both $\boldsymbol{z}_0$ and the level 1 encoding of $\boldsymbol{z}_0$. Therefore, we can trivially observe that revealing the random coins of this level 1 encoding to this party does not compromise the security of the BCG (as $\mathsf{BGN\text{-}EG}$ is only used to encrypt vectors that he knows in the clear anyway). Knowing the $\mathsf{BGN\text{-}EG}$ random coins of the level 1 encoding allows the party to perform the decompression procedure without computing pairings. To illustrate, consider the task of homomorphically multiplying two ciphertexts $(g_1^{r_1}, h_1^{r_1} g_1^{m_1}) \in \mathbb{G}_1^2$ and $(g_2^{r_2}, h_2^{r_2} g_2^{m_2}) \in \mathbb{G}_2^2$. The party who knows (m_1, r_1, m_2, r_2) can simply have precomputed $e(g_1, g_2), e(g_1, h_2), e(h_1, g_2)$, and $e(h_1, h_2)$ in a one-time setup phase, and obtain each term directly by computing exponentiations over $\mathbb{G}_t$ (e.g., $e(g_1^{r_1}, g_2^{r_2})$ is computed as $e(g_1, g_2)^{r_1 r_2}$, using the precomputed $e(g_1, g_2)$ and the random coins r_1, r_2 known in the clear). Therefore, for this party, the cost of decompressing a level 1 encoding (still counting all previous optimizations) is reduced from computing 4ℓ *pairings* to computing 4ℓ *exponentiations over* $\mathbb{G}_t$. Such exponentiations are typically up to an order of magnitude less costly than pairing computations.

It is relatively straightforward to share equally the benefits of this optimization between the two parties: instead of computing the BCG seeds as a (compressed) level 1 encoding of $\boldsymbol{z}_0$ and a (compressed) level 2 encoding of $\boldsymbol{z}_1$, we break each vector $\boldsymbol{z}_i$ in two equal length parts $(\boldsymbol{z}_i^0, \boldsymbol{z}_i^1)$. Then, the seed is now computed as level 1 encodings of both $\boldsymbol{z}_0^0$ and $\boldsymbol{z}_1^1$, and level 2 encodings of both $\boldsymbol{z}_0^1$ and $\boldsymbol{z}_1^0$, and each party P_j (for $j = 0, 1$) receives the random coins of one of the two level 1 encodings of half-vectors, and can therefore apply the above optimization to this half-vector. This reduces the number of pairings computed by each party by a factor 2 (in exchange for computing exponentiations over $\mathbb{G}_t$), which results in an improvement of almost a factor two. Note that in a scenario where the parties have very different computational power (e.g. a client and a server), it is better to use the asymmetric version, where the computationally weak devices computes only exponentiations instead of pairings.

We can further reduce the size of level 1 encodings by applying a heuristic PRG-based optimization which was described in [BGI16a]: the algorithm $\mathsf{HSS.Share}_1$ picks a random PRG seed k for a PRG G, and parse $G(\mathsf{k})$ as a sequence of $\sqrt{\ell}$ group elements over $\mathbb{G}_1$ and $\sqrt{\ell}$ group elements over $\mathbb{G}_2$. Each group element is then interpreted as the first component of an ElGamal ciphertext. Given a plaintext m and a first component c, the second component of an encryption of m over $\mathbb{G}_i$ is computed as $c^{s_i} \cdot g_i^m$. $\mathsf{HSS.Share}_1$ outputs k together with the list of all second components of ElGamal ciphertexts, which results overall in a factor 2 compression. Note, however, that this optimization cannot be directly cumulated with the previous optimization, as $\mathsf{HSS.Share}_1$ cannot reveal the corresponding random coins to one of the parties.

However, there is another straightforward optimization which does not conflict the optimization based on revealing the random coins to a party: the coins and the plaintexts can be generated as the output of a PRG G on a short seed k, and the party who gets to learn the coins and the plaintexts can be given k instead of the list of all plaintexts and random coins, reducing the storage overhead for this party.

The security of our group-based BCG reduces to the standard LPN assumption, where the linear code is generated uniformly at random, and the noise is a random sparse vector. LPN has a long history in cryptography, and it is a common practice to consider variants of this basic assumption to improve the efficiency of LPN-based primitives (several variants of LPN are by now standard and well-established assumptions; a typical example is Alekhnovich's assumption [Ale03], which underlies his famous LPN-based cryptosystem). Two standard tweaks that can be applied to LPN are the following:

– replacing the random linear code by a code with good properties (e.g. efficient encoding) to speed-up the computation of noisy codewords;

– modifying the noise distribution, to adapt it to the constraints of the application.

Below, we analyze both types of modifications of the LPN assumption, showing how using variants of LPN (some well-established, some less standard) leads to improved efficiency guarantees for the group-based BCG.

D.2 Optimizing the Noise Distribution

In this section, we discuss alternative choice of distributions from which to sample the noise vector, which are better suited for the encoding algorithms of the group-based BCG.

D.2.1 Regular Syndrome Decoding.

The regular syndrome decoding assumption is a variant of the LPN assumption which was introduced in [AFS03] as the assumption underlying the security of a candidate for the SHA-3 competition, and has been studied at length (see [HOSS18] for a recent survey about the cryptanalysis of the RSD assumption and a detailed discussion about its security). Roughly, it states that LPN remains hard, even if the sparse noise vector is *regular*, meaning that it is divided into k blocks of size n/k each, each block containing a single random 1, and zeroes everywhere else.

It was shown in [BCGI18] that the RSD can be used to improve the computational effiency of the FSS-based encoding compression method (which correspond to our level-2 encodings): with the LPN-based instantiation, the cost of $\mathsf{MPFSS.FullEval}$ is equal to k times the cost of the $\mathsf{FullEval}$ algorithm of an FSS for point functions, over a domain of size n. The RSD-based instantiation reduces this cost to k times the cost of the $\mathsf{FullEval}$ algorithm of an FSS for point functions, over a domain of size n/k. We refer the reader to [BCGI18] for further details.

Here, we observe that RSD can also be used to reduce the *size* of a level-1 encoding. Given the block structure of the noise pattern, one can simply apply the compression procedure of Section C.8 separately to each of the k blocks. As each block has length n/k and contains a single 1, its encoding has size $2\sqrt{n/k}$. Therefore, the total size of the encoding is $k \cdot (2\sqrt{n/k}) = 2\sqrt{n \cdot k}$, as opposed to $2\sqrt{n} \cdot k$ with LPN.

D.2.2 Learning Parity with Tensored Noise.

We now describe a more agressive choice of pattern, which leads to even better efficiency. Note that unlike LPN or RSD, the assumption that we introduce in this paragraph, although natural, is new. While we will analyze its resistance to standard attacks, further cryptanalysis is required to gain confidence in its security. We stress, however, that our assumption as strong connections to (an can be seen as a natural strenghtening of) well-studied assumptions from the literature; hence, it appears very plausible.

We focus on compressing level 1 shares in group-based HSS, since they form the bottleneck of our scheme. Let $\otimes$ denote the tensor product between vectors – that is, for length-n vectors $\boldsymbol{a} = (a_i)_i, \boldsymbol{b} = (b_i)_i$, $\boldsymbol{a} \otimes \boldsymbol{b}$ denotes the length-n^2 vector with coordinates $a_i b_j$. We suggest the following family of distributions for integers k, n (we assume n to be a perfect square for simplicity):

$$\mathcal{D}^{\otimes}_{k,n}(\mathcal{R}) = \left\{ \boldsymbol{x} \in \mathcal{R}^n \mid (\boldsymbol{a}, \boldsymbol{b}) \xleftarrow{\$} \left(\mathsf{Ber}_{k/\sqrt{n}}(\mathcal{R})^{\sqrt{n}}\right)^2, \boldsymbol{x} \leftarrow \boldsymbol{a} \otimes \boldsymbol{b} \right\}.$$

That is, a sample from $\mathcal{D}^{\otimes}_{k,n}(\mathcal{R})$ is the tensor product between two length-$\sqrt{n}$ vectors drawn from the Bernouilli distribution with rate $k/\sqrt{n}$. Such a sample has k^2 non-zero coordinates on average. Compressing level 1 shares of such samples is straightfoward: the compressed level 1 share simply contains a $\mathsf{BGN\text{-}EG}$ encryption of $\boldsymbol{a}$ over $\mathbb{G}_1$ and of $\boldsymbol{b}$ over $\mathbb{G}_2$. The sample can be

reconstructed from the homomorphic properties of BGN-EG by computing the tensor product $\boldsymbol{a} \otimes \boldsymbol{b}$ over $\mathbb{G}_t$. This amounts to a total of $2\sqrt{n}$ BGN-EG ciphertexts, improving over the method of the previous section by a factor of $\sqrt{k}$. Note, in addition, that this alternative noise distribution also reduces the number of pairings to be computed by a comparable factor – as it turns out, in concrete efficiency estimations, the number of pairings is one of the main computational bottleneck. Therefore, this variants also strongly improves the computational efficiency of the BCG.

It remains to discuss the security of LPN with noise vectors drawn from this distribution. Unlike for standard Bernouilli noise, there are specific attacks which are known to apply as soon as the noise distribution satisfies a low-degree relation (in the present situation, the noise distribution satisfies a degree-2 relation). To our knowledge, the only attacks which specifically exploit low-degree relations in the noise distribution are those of Arora and Ge [AG10], which provide a polynomial-time algorithm solving LPN instances using N^d samples, where d is the degree of the relation and N is the dimension. Since we consider much more limited number of samples ($O(N)$), this attack does not apply in our setting. However, the algebraic structure of the noise also implies that, unlike standard LPN, this assumption is sensitive to algebraic attacks (e.g. Gröbner basis attacks); therefore, we will take into account algebraic attacks in addition to standard attacks on LPN when estimating the concrete security level offered by this new assumption.

Relation to MQ and LPN. If the vector $\boldsymbol{x}$ had been sampled from the standard Bernouilli distribution, the above assumption would be exactly LPN. Furthermore, if the vecto $\boldsymbol{x}$ had been computed as the tensor product of a *uniformly random* (as opposed to sparse) vector $\boldsymbol{a}$ with itself, this assumption would exactly be the multivariate quadratic (MQ) assumption, a well-studied assumption [MI88, Wol05, AHI+17] that asserts that it is computationally infeasible to sove a random system of quadratic equations. Hence, the LPN with tensored noise assumption can be seen as a natural strengthening of both the MQ assumption and the LPN assumption.

Cryptanalysis. Since it shares both the algebraic structure of the MQ assumption, and the sparsity of the LPN assumption, the new assumption introduced above is sensitive to the standard cryptanalytic attacks that apply to each of these assumptions. We provide an overview of the existing attacks below.

- The most powerful attacks on LPN (in the setting of a limited number of samples and a low noise rate, which is the case here) are the Gaussian elimination attack (which attempts to guess a noise-free subset of the coordinates of the vector, and use Gaussian elimination to invert the corresponding system of equation), the low-weight parity-check attack [Zic17] (which uses the existence of low-weight codewords in the dual code to establish the existence of a distinguisher), and the information set decoding (ISD) attack (which uses variants of Prange's algorithm [Pra62] to solve the corresponding syndrom decoding problem). We refer the reader to [BCGI18] for a more detailed overview of these attacks, and bounds on their computational efficiency. We note that the structure of the noise pattern in our LPN variant leads to better alternative than the random choice to pick candidate noise-free coordinates of the vector; we will takes this observation into account when evaluating the cost of the above attacks. Furthermore, note that the low-weight parity-check attack only applies to LPN over exponentially large fields (see [Zic17]). Over $\mathbb{F}_2$, this attack provably fails.
- The most powerful attacks on MQ are the algebraic attacks, such as Gröbner basis attacks. These attacks exploit the algebraic structure of the system of quadratic equations to generate many new equations, until sufficient information was gathered to solve the system efficiently. We note that algebraic attacks such that XL and Gröbner basis attacks do not provide ways to take advantage of the sparsity of the noise pattern.

Note that we could have alternatively used noise vectors constructed as tensor product between vectors sampled from the uniform distribution, leading to an MQ-based construction.

However, bounding the number of nonzero coefficients (here, by k^2) is crucial for the efficiency of the Convert operation, which relies on an expensive "distributed discrete logarithm" where the parties must compute a number of multiplications over $\mathbb{G}_t$ polynomial in the number of nonzero coefficients of the input vectors.

D.3 Changing the Code Matrix

In this section, we describe alternative choices of code matrices which lead to better computational efficiency for the Expand procedure.

D.3.1 Faster Expansion using LDPC Codes.

Constructing BCG from degree-2 CHSS essentially boils down to using the deg-2 CHSS to evaluate $N^\intercal B_i N$, where B_i is the matrix of the ith output for a target bilinear correlation, and N is the parity check matrix of an LPN-friendly code. However, this general method has a downside: in many concrete scenarios, the matrices B_i of the target correlation will be highly sparse (for example, for building (pseudo)random oblivious transfers, the corresponding matrices B_i have a single nonzero entry). Yet, N cannot be a sparse matrix: using a code whose parity check matrix is sparse would render LPN insecure, since this corresponds to decoding an LDPC code, which can be done in polynomial time. Therefore, the matrices $N^\intercal B_i N$ will be dense even though each B_i might be sparse. This makes the computation wastefull: the computation of $\boldsymbol{z}_0^\intercal N^\intercal B_i N \boldsymbol{z}_1$ requires $O(n \cdot \ell) = O(\alpha \cdot n^2)$ operations (recall that this is for computing each output of the correlation), which becomes quickly inefficient for a large target output size n.

We suggest an alternative approach to bring this computational cost from quadratic to linear. The main idea is to set N to be the matrix of a low-density parity check code [Gal62, MN96] (LDPC) instead of a random code. LDPC codes are particularly attractive in our scenario, for two main reasons.

1. First, the conjectured intractability of LPN instances obtained from the parity check matrix of an LDPC code (which is a sparse matrix) is a well-established standard assumption, similar to the assumption made in the seminal work of Alekhnovich [Ale03] on public-key encryption from LPN.
2. Second, LDPC admit a linear-time, data oblivious encoding algorithm over arbitrary fields [LM10, KS12], with very good concrete efficiency: encoding with a binary LDPC whose parity-check matrix has ℓ rows requires at most $4 \cdot \ell \cdot (\bar{k} - 1)$ XORs, where $\bar{k}$ denotes the row weight of the parity-check matrix (LPN with highly sparse matrices being conjectured to be intractable, the value $\bar{k}$ can be taken to be very small, say, lower than 10, in practice; see e.g. [ADI+17]).

Of course, LDPC codes are expanding, while we need a compressing matrix; therefore, the right approach here is to use the transpose $M^\intercal$ of the matrix M of an LDPC code. It might not be obvious at first sight that the existence of a linear-time algorithm for computing $\boldsymbol{x} \mapsto M\boldsymbol{x}$ implies the existence of a linear-time algorithm for computing $\boldsymbol{x} \mapsto M^\intercal \boldsymbol{x}$. However, this is the case for linear mappings computed by linear operations: for any such mapping, there is a circuit of the same size computing the transposed mapping [Bor57, IKOS08], which essentially consists in reversing the computation while interchanging XORs and fan-out operations.

The second idea is to use the fact that level 1 and level 2 shares of the CHSS of the previous section directly support homomorphic evaluation of linear functions. Combining the two ideas gives rise to the following approach for speeding up the Expand procedure of a BCG built out of this CHSS:

1. Set N to be the transpose of the encoding matrix of an LDPC code whose parity-check matrix has low row-weight (say, at most 10).

2. Given a level 1 encoding $[\![\boldsymbol{z}_0]\!]_{\mathsf{sk}}$ of a vector $\boldsymbol{z}_0$ (i.e., a BGN-EG encryption of $\boldsymbol{z}_0$ over $\mathbb{G}_t$) and a level 2 encoding $\langle\!\langle \boldsymbol{z}_1 \rangle\!\rangle_{\mathsf{sk}}$ of a vector $\boldsymbol{z}_1$ (i.e., $\mathsf{sk}_t \bullet \langle \boldsymbol{z}_1 \rangle$), homomorphically compute $[\![N\boldsymbol{z}_0]\!]_{\mathsf{sk}}$ and $\langle\!\langle N\boldsymbol{z}_1 \rangle\!\rangle_{\mathsf{sk}}$ using the transposed version [Bor57] of the linear encoding algorithm of [LM10]. For example, if $\mathcal{R} = \mathbb{F}_2$ and $\bar{k} = 5$, computing $[\![N\boldsymbol{z}_0]\!]_{\mathsf{sk}}$ requires $64 \cdot \ell$ multiplications over $\mathbb{G}_t$ (16 for each of the 4 components of a BGN-EG ciphertext). The cost of computing $\langle\!\langle N\boldsymbol{z}_1 \rangle\!\rangle_{\mathsf{sk}}$ is negligible in comparison (it involves only additions over $\mathbb{Z}_q$, which are cheap).
3. Apply the Mult algorithm on $[\![N\boldsymbol{z}_0]\!]_{\mathsf{sk}}$ and $\langle\!\langle N\boldsymbol{z}_1 \rangle\!\rangle_{\mathsf{sk}}$ to compute additive shares of the value $(N\boldsymbol{z}_0)^\intercal B_i (N\boldsymbol{z}_1)$ for each of the matrices B_i. Note that this Mult procedure can now take advantage of the sparsity of the B_i.

The above alternative method brings down the cost of computing each output of the Expand procedure from $O(\alpha \cdot n^2)$ to $O(\alpha \cdot n)$ in the (classical) situation where the B_i are sparse. As we will see later in our efficiency estimations for the concrete goal of computing OT correlations, this easily amounts to an improvement by several orders of magnitude of the cost of this step.

D.3.2 Alternative Choices of Code Matrices.

The choice of LDPC-based codes above is motivated by efficiency considerations. However, alternative choices of code matrices can be envisioned, which lead to different tradeoffs between the strength of the assumption and the computational efficiency of the matrix-vector multiplication. (The list given below was taken from [BCGI18])

- *MDPC Codes.* A more conservative variant of the above is to rely on MDPC codes (medium-density parity-check codes), where the parity-check matrix has row weight $O(\sqrt{n})$ (instead of constant). MDPC codes have been thoroughly studied, since they are used in optimized variants of the famous McEliece cryptosystem [MTSB12].
- *Quasi-Cyclic Codes.* A second alternative option is to rely on quasi-cyclic codes, which admit fast (albeit superlinear) encoding algorithms. Quasi-cyclic codes have been recently used to construct optimized variants of the LPN-based cryptosystem of Alekhnovich and the code-based cryptosystem of McEliece [ABD+16, MBD+18].
- *Druk-Ishai Codes.* Another possibility is to rely on the linear-time encodable codes developped by Druk and Ishai in [DI14]. Their construction of linear-time encodable code is essentially a concatenation of good a linear encoding and its transpose, intertwined with random local mixing. This design strategy leads to codes satisfying the combinatorial properties of random linear codes (e.g. meeting the Gilbert-Varshamov bound) and do not support efficient decoding, while having a fast (linear-time) encoding algorithm; this makes it a strong candidate in our scenario.
- *Other Codes.* Many other alternatives can be envisioned: since we do not require the code to have structure, or decoding algorithms. Therefore, any sufficiently good heuristic mixing strategy (e.g. a strategy based on expander graphs, such as the approach developped by Spielman in [Spi96]) will likely lead to a secure LPN instance in our setting.

E Group-Based Silent OT/OLE Extension

In this section, we focus on the task of using PCG to generate (random) oblivious transfer correlations. After motivating the question of computing random OTs with sublinear communication, which we believe to be a task of fundamental interest, we describe various optimizations of PCG tailored to this application, perform extensive efficiency estimations, and analyze techniques to mitigate the inverse-polynomial failure probability of the group-based PCG of the previous section for this application.

E.1 Generating ROT-Correlations with Group-Based PCG

In this section, we study the application of the techniques we developped for the specific (but fundamental) case of random OT correlations. We provide an extensive efficiency analysis of the construction.

E.1.1 Random Oblivious Transfer.

A random (bit) oblivious transfer (ROT) is a two party protocol between a sender and a receiver, both with no input. The sender gets as output two uniformly random bits (r_0, r_1), and the receiver gets (b, r_b), where b is a random bit. Given black-box access to an ROT primitive, any multiparty functionality can be securely evaluated with information-theoretic security (in particular, implementing OT from ROT requires exchanging 3 bits), making ROT a core primitive for multiparty computation. One can easily observe that an equivalent formulation of the ROT functionality is the following: the sender and the receiver both get a respective random bit (s, r), as well as additive shares (over $\mathbb{F}_2$) of their product sr. We focus below on the latter formulation. As ROTs correspond to a degree-2 correlation, they can be generated using a PCG.

E.1.2 Optimized Group-Based PCG for ROT Correlations.

We provide below a self-contained description of the group-based sanitizable PCG of the previous section, tailored to ROT correlations, using all optimizations that apply in our setting. The PCG is parametrized with three integers (α, k, n), where α is a parameter of the underlying LPN assumption, n is the target output length, and k is a sparsity parameter. For simplicity, we assume that $\alpha \cdot n$ is a perfect square. To balance the benefits of optimizations based on revealing the ElGamal random coins to one of the parties, the parties (P_0, P_1) will execute two parallel instances of the algorithms Gen and Expand below, exchanging their roles in each instance. To optimize for efficiency, we use an LDPC code matrix, and rely on the LPN with tensored noise assumption, discussed in Section D.

- **Output.** The two parties get respective outputs $(\boldsymbol{x}, \boldsymbol{w}_0) \in \mathbb{F}_2^{2n}$ and $(\boldsymbol{y}, \boldsymbol{w}_1) \in \mathbb{F}_2^{2n}$ such that $\boldsymbol{w}_0 + \boldsymbol{w}_1 = \boldsymbol{x} \cdot \boldsymbol{y}$, where $\cdot$ denotes the component-wise product over $\mathbb{F}_2$.
- Setup. Sets $\mathsf{pp}' = (q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, g_1, g_2, e) \xleftarrow{\$} \mathsf{BGN\text{-}EG.Setup}(1^\lambda)$, and compute $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{BGN\text{-}EG.KeyGen}(\mathsf{pp}')$. Compute $g_t \leftarrow e(g_1, g_2)$. Output $\mathsf{pp} \leftarrow (\mathsf{pp}', \mathsf{pk}, g_t)$ and sk. Let

$$\mathsf{MPFSS} = (\mathsf{MPFSS.Gen}, \mathsf{MPFSS.Eval}, \mathsf{MPFSS.FullEval})$$

  be a multi-point function secret sharing over $\mathbb{Z}_q$, and let G be a PRG.
- Gen. Set $\ell \leftarrow \alpha \cdot n$. Pick a random seed k_0, and parse $G(\mathsf{k}_0)$ as (a representation of) a pair of (pseudo)random sparse vectors $(\boldsymbol{a}, \boldsymbol{b}) \in (\mathbb{Z}_q^{\sqrt{\ell}})^2$, each with exactly k entries equal to 1 and $\sqrt{\ell} - k$ entries equal to 0, together with $2\sqrt{\ell}$ (pseudo)random coins $(\boldsymbol{r}_1, \boldsymbol{r}_2) \in (\mathbb{Z}_q^{\sqrt{\ell}})^2$. Set $\boldsymbol{c}_1 \leftarrow \mathsf{BGN\text{-}EG.Enc}_1(\mathsf{pk}, \boldsymbol{a}; \boldsymbol{r}_1), \boldsymbol{c}_2 \leftarrow \mathsf{BGN\text{-}EG.Enc}_2(\mathsf{pk}, \boldsymbol{b}; \boldsymbol{r}_2)$. Pick another random seed k_1, and parse the string $G(\mathsf{k}_1)$ as a (representation of) a (pseudo)random vector $\boldsymbol{d}_0 \in \mathbb{Z}_q^\ell$ with exactly k coordinates equal to 1, and $\ell - k$ coordinates equal to 0. We denote $\mathsf{proj}_{\boldsymbol{x}}$ (for any vector $\boldsymbol{x} \in \mathbb{Z}_q^n$) the multi-point function which, on input $i \leq n$, outputs the i'th coordinate of $\boldsymbol{x}$. Parse sk as $(\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{sk}_t)$ and parse sk_i as $(s_i, -1)$ for $i = 1, 2$. Let $\mathbf{proj}_{\boldsymbol{m}, \mathsf{sk}} \leftarrow (\mathsf{proj}_{s_1 s_2 \boldsymbol{m}}, \mathsf{proj}_{-s_1 \boldsymbol{m}}, \mathsf{proj}_{-s_2 \boldsymbol{m}}, \mathsf{proj}_{\boldsymbol{m}})$. Compute $(\boldsymbol{K}_0, \boldsymbol{K}_1) \leftarrow \mathsf{MPFSS.Gen}(1^\lambda, \mathbf{proj}_{\boldsymbol{d}_0, \mathsf{sk}})$. The output of P_0 is $(\mathsf{k}_0, \boldsymbol{K}_0)$, and the output of P_1 is $(\mathsf{k}_1, \boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{K}_1)$.
- Expand. Let $\boldsymbol{d}_1 \leftarrow \boldsymbol{a} \otimes \boldsymbol{b}$. P_0 and P_1 expand their respective FSS keys $\boldsymbol{K}_0$ and $\boldsymbol{K}_1$ using $\mathsf{MPFSS.FullEval}$, getting $\langle \mathsf{sk}_t \cdot \boldsymbol{d}_0 \rangle$ (i.e., additive shares over $\mathbb{Z}_q$ of $\mathsf{sk}_t \cdot \boldsymbol{d}_0$).
  - P_1 computes $\boldsymbol{c}_t \leftarrow e(\boldsymbol{c}_1, \boldsymbol{c}_2) \in \mathbb{G}_t^{4\ell}$.

- P_0 expands k_0 into $(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{r}_1, \boldsymbol{r}_2)$ using G, and computes $\boldsymbol{c}_t$ directly from $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{r}_1, \boldsymbol{r}_2$ as

$$\begin{aligned}
\boldsymbol{c}_{t,0} &\leftarrow e(g_1, g_2)^{\boldsymbol{r}_1 \otimes \boldsymbol{r}_2} \\
\boldsymbol{c}_{t,1} &\leftarrow e(g_1, h_2)^{\boldsymbol{r}_1 \otimes \boldsymbol{r}_2} \cdot e(g_1, g_2)^{\boldsymbol{r}_1 \otimes \boldsymbol{b}} \\
\boldsymbol{c}_{t,2} &\leftarrow e(h_1, g_2)^{\boldsymbol{r}_1 \otimes \boldsymbol{r}_2} \cdot e(g_1, g_2)^{\boldsymbol{r}_2 \otimes \boldsymbol{a}} \\
\boldsymbol{c}_{t,3} &\leftarrow e(h_1, h_2)^{\boldsymbol{r}_1 \otimes \boldsymbol{r}_2} \cdot e(h_1, g_2)^{\boldsymbol{r}_1 \otimes \boldsymbol{b}} \cdot e(g_1, h_2)^{\boldsymbol{r}_2 \otimes \boldsymbol{a}} \cdot e(g_1, g_2)^{\boldsymbol{d}_0}
\end{aligned}$$

  Note that all pairings involved can be precomputed in a one-time setup phase.
- Let $N \in \mathbb{Z}_q^{\ell \times n}$ be the transpose of the matrix of a binary LDPC code (each entry of N is viewed as a 0 or a 1 over $\mathbb{Z}_q$). Both parties locally compute $(\langle \mathsf{sk}_t \cdot d'_i \rangle)_{i \le n} \leftarrow \langle \mathsf{sk}_t \cdot N \boldsymbol{d}_0 \rangle$ and homomorphically multiply the plaintext of $\boldsymbol{c}_t$ by N, obtaining a list of n ciphertexts $(c'_i)_{i \le n}$ over $\mathbb{G}_t^4$. Note that this multiplication by N can be done by homomorphically evaluating on $\boldsymbol{c}_t$ the "transposed version" of the circuit of [LM10], which contains only XOR gates, interpreting all XOR gates as a modular addition over $\mathbb{Z}_q$ (as no modular reduction occurs, the parties will obtain the correct result by locally reducing their final additive shares modulo 2).
- P_0 locally sets $\boldsymbol{x} \leftarrow N \cdot \boldsymbol{d}_0 \bmod 2$, and P_1 expands k_1 into $\boldsymbol{d}_1$ and locally sets $\boldsymbol{y} \leftarrow N \cdot \boldsymbol{d}_1 \bmod 2$.
- For $i = 1$ to n, the parties compute the Mult algorithm by sequentially evaluating $\mathsf{Pair}(c'_i, \langle \mathsf{sk}_t \cdot d'_i \rangle) = \{g_t^{z_i}\}$, where z_i is an integer value satisfying $z_i = x_i \cdot y_i \bmod 2$, and $\mathsf{Convert}(\{g_t^{x_i \cdot y_i}\}, \delta) = \langle z_i \rangle$ (where the equality holds except with some failure probability δ independently for each share). Finally, the parties locally convert these integer shares into $\mathbb{F}_2$-shares by reducing $\langle z_i \rangle$ modulo 2, getting $\langle x_i \cdot y_i \rangle \bmod 2$. We denote by $\boldsymbol{w}_\sigma$ the length-n vector of shares obtained by party P_σ.

E.2 Efficiency Estimations

We now estimate the concrete efficiency of the group-based bilinear correlation generator, using all the optimizations described in the previous section, for generating a large number of (bit) ROT correlations. Cost estimates for other useful types of correlations (e.g. Beaver triples) can be easily extrapolated from our detailed estimations. We note that, although we will carefully evaluate the resistance to known attacks of the assumptions underlying some of our optimizations, these assumptions remain new and deserve further exploration. The efficiency estimations given in the upcoming sections are based on the concrete parameters designed so that known attacks take at least 2^{80} steps to break the underlying assumptions; if improved cryptanalytic methods are discovered, our efficiency estimations should be revised accordingly.

E.2.1 Choices of Curve.

We estimate the costs of sanitizable PCG for random OT using the benchmark numbers of the Miracl library.[16] All benchmarks are executed on one core of a 2.4 GHz Intel i5 520M processor. We consider two pairing-friendly elliptic curves for type-3 pairings:

– An MNT curve with a 160-bit modulus q and an embedding degree equal to 6. A $\mathbb{G}_1$ (resp. $\mathbb{G}_2, \mathbb{G}_t$) element is 160-bit (resp. 320-bit , 960-bit) long. The Miracl documentation reports the following timings over this curve: 1,9ms for a pairing, and 0,24ms for an exponentiation over $\mathbb{G}_t = \mathbb{F}_{q^6}$ (which approximately corresponds to $6 * 160 * 1.5 = 1440$ multiplications over $\mathbb{G}_t$). This curve is conjectured to provide 80 bits of security and is a good choice of curve to minimize the seed size (as each group element is only 160 bits long).

[16] https://libraries.docs.miracl.com/miracl-explained/benchmarks

– A Cocks-Pinch curve with a 512-bit modulus q and an embedding degree equal to 2. A $\mathbb{G}_1$ or $\mathbb{G}_2$ (resp. $\mathbb{G}_t$) element is 512-bit (resp. 1024-bit) long. The Miracl documentation reports the following timings over this curve: 1,14ms for a pairing, and 0,12ms for an exponentiation over $\mathbb{G}_t = \mathbb{F}_{q^2}$ (which approximately corresponds to $2 * 512 * 1.5 = 1536$ multiplications over $\mathbb{G}_t$). This curve is conjectured to provide 80 bits of security and is a good choice of curve to minimize the computational overhead (as pairings and exponentiations are respectively 40% and 50% less expensive than over an MNT curve, for the same security level).

E.2.2 Matrix Multiplication and Gaussian Elimination.

Most attacks on the LPN assumption and its variants require computing matrix multiplications, and solving systems of linear equations, for large matrices and systems. To estimate properly the efficiency of these attacks, we overview in this section the state-of-the-art regarding matrix multiplication and resolution of linear system.

It is well known, and was first shown in [BH74], that solving linear systems of equations is reducible to matrix multiplication: if $n \times n$ matrix multiplication can be computed in $O(n^\omega)$, then solving $A\boldsymbol{x} = \boldsymbol{y}$ for an $n \times n$ matrix A can be done in time $O(n^\omega)$. More precisely, if $M(n)$ denotes the time for multiplying two $n \times n$ matrices, then solving $A\boldsymbol{x} = \boldsymbol{y}$ for an $n \times n$ matrix A requires less than $2 \cdot M(n)$ operations. Therefore, we focus in this overview on the cost of multiplying two matrices. We denote by ω the smallest possible value such that the time for multiplying two $n \times n$ matrices is in $O(n^\omega)$.

Matrix multiplication in subcubic time was believed to be impossible, until the seminal paper of Strassen in 1969 [Str69], which described an algorithm using $n^{\log_2(7)}$ multiplications. This result was followed from decades of improvement, culminating with the Coppersmith-Winograd algorithm [CW90] which established $\omega \leq 2.375477$, followed by subsequent improvements, with the current record being held by Le Gall's algorithm [LG12] ($\omega \leq 2.3728639$). However, all the results starting with the work of Coppersmith-Winograd suffer from the *curse of recursion*: they involve complex nested recursive calls, which make the constant term of the algorithms prohibitively large – so large that no such method has ever be implemented as of today, nor is believed to provide any concrete speedup for matrices of any realistic size [Pan18]. All known implementations of subcubic matrix multiplication algorithms rely on the original algorithm of Strassen, or it's improvement by Winograd (which reduces the number of additions, but leaves the asymptotic complexity unchanged) [Fis74].

It is hard to estimate what exactly is the current optimal algorithm for *feasible* matrix multiplication, since it largely depends on the specific features of the problem at hand. To our knowledge, the smallest matrix multiplication exponent for which matrix multiplication has feasible constants is 2.7760 [Kap04]. The algorithm of Kaporin [Kap04] was implemented and compared with the algorithms of Winograd and Strassen (which have exponents $\log_2(7) \approx 2.807$) for $n \times n$ matrices of size up to $n = 2 \cdot 10^5$. At these sizes, Kaporin's algorithm had a running time within a factor 1.05 to that of Strassen and Winograd's algorithm. It follows that a sufficiently good upper bound on the running time of a matrix multiplication algorithm is given by the cost $n^{\log_2(7)}$ of Winograd's algorithm (note that over $\mathbb{F}_2$, where additions cost as much as multiplications, the number of bit operations for Winograd's algorithm is $5 \cdot n^{\log_2(7)}$).

While the previous results establish the best known running time for multiplying two $n \times n$ arbitrary matrices, our optimized group-based PCG involves *structured* matrices – typically, LDPC matrices, which have a sparse parity-check matrix. Multiplying an LDPC matrix with an arbitrary matrix can be done faster than through general matrix multiplication: the linear-time LDPC encoding algorithm of [LM10] allows to compute an LDPC encoding $A\boldsymbol{x}$ of a vector $\boldsymbol{x}$ with code matrix A in time $4d \cdot n$, where d is the maximum number of non-zero elements in the parity check matrix of A. This directly lead to a matrix multiplication algorithm in time $4d \cdot n^2$. While it is not entirely clear wether the linear system solver from matrix multiplication of [BH74] preserves the LDPC structure of the matrix through the algorithm (since the algorithm involves

recursive calls to the matrix multiplication functionality on blocks of A), it is plausible that it can be modified to maintain this structure. Therefore, we will conservatively assume below, when estimating the resistance of our scheme to known attacks, that the cost of solving a system of linear equations is lower bounded by $4d \cdot n^2$.

E.2.3 Parameters for the LPN with Tensored Noise Assumption.

We discuss choices of parameters for the LPN with tensored noise assumption, by evaluating the efficiency of known attacks on this assumption. We consider LPN with tensored noise with k^2 noisy coordinates, dimension $(\alpha - 1)n$, and number of samples αn. Note that we use LPN with tensored noise over $\mathbb{F}_2$, therefore the low-weight parity-check attack does *not* apply to our setting (see [Zic17]).

- **Gröbner Basis Attack.** We first evaluate the resistance of LPN with tensored noise against Gröbner basis attacks. These attacks to not directly depend on the number of noisy coordinates, but mainly on the compression factor (recall that we extend $\sqrt{\alpha n}$ bits to αn bits, before compressing the output to n bits). Hence, these attacks will allow us to determine the optimal value of α for a given n.
  As is always done to study the complexity of Gröbner basis attacks, we conjecture that the corresponding system of quadratic equations is semi-regular. This conjecture is supported by the fact that the proportion of semi-regular systems goes to 1 when the number of unknown grows [Bar04, Frö85]. By [BFSY05, Theorem 1], the average complexity of a Gröbner basis attack is given by
  $$\binom{t}{d_{\mathsf{reg}}}^{\omega},$$
  where $t = \sqrt{\alpha n}$ is the number of unknowns in the system, ω is the matrix multiplication exponent (which we assume equl to 2.8), and d_{reg} is the regularity degree, which was shown in [BFSY05] to be upper bounded by
  $$-n + \frac{t}{2} + \frac{t}{2}\sqrt{2(n/t)^2 - 10n/t - 1 + 2(n/t+2)\sqrt{(n/t)\cdot(n/t+2)}}\,.$$
  Solving the above under the condition that the Gröbner basis attacks requires at least 2^{80} operations, we get the following:
  - For $n < 2^{26}$, the regularity degree d_{reg} is equal to 3, and the smallest value of α which allows to stretch $\sqrt{\alpha n}$ bits to n bits is $\alpha = 24$.
  - For $n \geq 2^{26}$, the regularity degree d_{reg} is equal to 2, and the smallest value of α which allows to stretch $\sqrt{\alpha n}$ bits to n bits is $\alpha = 16$.
- **Gaussian Elimination Attack.** For the standard LPN assumption with k noisy coordinates, the Gaussian elimination attack requires on average $(1/(1-k/(\alpha n)))^{(\alpha-1)n}$ iterations, where the adversary must invert an $((\alpha-1)n) \times ((\alpha-1)n)$ matrix, which takes time at least $((\alpha-1)n)^{2.8}$ using Strassen's matrix multiplication algorithm. However, since we use an LDPC code, which admits a linear-time encoding algorithm, the linear system can be solved more efficiently: the linear-time encoding algorithm of [LM10] gives matrix-vector multiplication in time $(2\alpha-1)\cdot d \cdot n$, where d is the sparsity parameter of the parity-check matrix of the code, which we denote d; hence, multiplying and LDPC code matrix with an arbitrary square matrix takes time at most $\alpha(2\alpha-1)\cdot d \cdot n^2$. Using the reduction from matrix multiplication to solving systems of linear equations [BH74], this leads to an algorithm for solving the system in time lower-bounded by $\alpha(2\alpha-1)\cdot d \cdot n^2$ (see the discussion in the previous section).
  For LPN with tensored noise, which involves a noise vector with k^2 noisy coordinates, there is a better way to sample the candidate noise-free subvector: as the vector is of the form $\boldsymbol{x} = \boldsymbol{a} \otimes \boldsymbol{a}$ for a k-sparse vector $\boldsymbol{a}$, it suffices to divide $\boldsymbol{x}$ into $\sqrt{\alpha n}$ blocks of $\sqrt{\alpha n}$ coordinates

each. Note that each block is either equal to $\boldsymbol{a}$ or to the all-0 vector. Hence, the adversary can simply guess $(\alpha-1)n/\sqrt{\alpha n}$ noise-free blocks (there are $\sqrt{\alpha n}-k$ such blocks), which requires on average $(1/(1-k/\sqrt{\alpha n}))^{(\alpha-1)n/\sqrt{\alpha n}}$ iterations. Therefore, a lower bound on the bit-security of the LPN with tensored noise instance with respect to the Gaussian elimination attack is given as

$$\log_2\left(\left(\frac{1}{1-k/\sqrt{\alpha n}}\right)^{((\alpha-1)n)/\sqrt{\alpha n}}\cdot\alpha(2\alpha-1)\cdot d\cdot n^2\right).$$

– **Parity-Check Attack.** A noisy codeword can be distinguished from random by multiplying it with a parity-check vector. Over $\mathbb{F}_2$, a random vector will pass the parity-check with probability exactly 1/2. However, a noisy codeword will pass the check with probability 1 conditioned on all the noisy coordinates corresponding to zero-entries of the parity-check vector, and with probability 1/2 otherwise. Therefore, parity-check vectors allows for a non-trivial distinguishing advantage.
Since the dual code has distance at most $(\alpha-1)n$ in our setting, there always exists as parity check vector with at most $(\alpha-1)n+1$ non-zero coordinates, which is obtained by writing the dual matrix in systematic form, and using it to encode an arbitrary low-weight vector. This way, up to $\alpha n-(\alpha-1)n-1=n-1$ coordinates are guaranteed to be zero; the remaining $(\alpha-1)n+1$ coordinates being over $\mathbb{F}_2$ in our setting, they will contain on average $((\alpha-1)n+1)/2$ zeroes. Therefore, we assume that the adversary can compute at no cost (since they can be preprocessed given only the code matrix) a list of parity-check vectors of weight $((\alpha-1)n+1)/2$. Under the heuristic that the k^2 noisy coordinates are randomly spread over the noise vector, a parity-check vector will have only zeroes in positions corresponding to the noisy coordinates with probability

$$\left(\frac{\alpha n-((\alpha-1)n+1)/2}{\alpha n}\right)^{k^2}=\left(\frac{\alpha n+n-1}{2\alpha n}\right)^{k^2}.$$

After each iteration, the adversary must compute a parity-check with a vector of weight $((\alpha-1)n+1)/2$, which costs him $((\alpha-1)n+1)/2$ arithmetic operations. Therefore, the (logarithm of the) average computational cost of the parity check attack is given by

$$\log_2\left(\frac{(\alpha-1)n+1}{2}\cdot\left(\frac{\alpha n}{\alpha n+n-1}\right)^{k^2}\right).$$

Finally, observe that the heuristic assumption that the noise is randomly spread is clearly false here: since the noise vector was computed as a tensor product between low-weight vectors, it satisfies a block structure, where each block is of length $\sqrt{\alpha n}$, k blocks are noisy and the remaining are noise-free, and the noise pattern is the same accross the noisy blocks. However, to exploit this known structure, the adversary would have to guess the position of noise-free blocks, and then to find a parity check vector with a large number of non-zero coordinates belonging to these noise-free blocks. While this can be done in a preprocessing phase, it requires a prohibitive amount of preprocessing: we estimated that, in our range of parameters, the adversary would need to compute at least 2^{200} operations when preprocessing the parity-check vectors to get a noticeable speedup in the online attack. Therefore, we do not consider speedups from preprocessing parity-check vectors in our estimations, but note that finding imoproved preprocessing techniques is a possible direction that deserves further study toward getting a better cryptanalysis of our assumption using parity-check attacks.

– **Information Set Decoding Attack.** We now turn our attention to the ISD attack. Many variants of the attack have been developed in the past years, and the asymptotic costs of these attacks are often non-trivial to estimate. However, in our parametter setting, the noise rate $k^2/(\alpha n)$ is small, and the advantages of the variants of the original algorithm of

Prange [Pra62] vanish in this situation, as shown in the analysis of [TS16]. We will therefore focus on bounding the cost of the original algorithm of Prange; since we will find this attack to have much worst performances than the Gaussian elimination attack, this leaves a large security gap. To make conservative estimates, we evaluate the cost of the attack against a *standard* LPN instance with k noisy coordinates; that is, we make the assumption that due to their structure, the k^2 noisy coordinates of LPN with tensored noise do not provide additional security compared to k (random) noisy coordinates. We rely on the detailed concrete efficiency analysis of ISD given in [HOSS18], which shows that the bit-security of the LPN instance with respect to Prange's algorithm is upper-bounded by

$$\log_2 \left(\frac{\binom{\alpha n}{k}}{\binom{n-1}{k}} \cdot 8d \cdot (n-1)^2 \right),$$

where $4d \cdot (n-1)^2$ denotes our conservative estimate of the cost of solving a linear system of equations given by an LDPC code with parity-check matrix of row-weight d (see the discussion in the previous section).

E.2.4 Optimal Parameters.

We use the above analysis to choose optimal parameters to achieve 80 bits of security against Gröbner basis attacks, Gaussian elimination, parity checks, and ISD. The optimal parameters are represented on Table 5.

n	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}	2^{28}	2^{30}
α	24	24	24	24	24	16	16	16
k	25	23	20	17	14	12	9	8

Table 5. Optimal parameters k and α for various choices of n. We consider the cost of attacking the LPN with tensored noise assumption, instantiated with an LDPC code matrix, using either Gaussian elimination, Gröbner basis attacks, ISD, or parity-check attacks. We set the sparsity parameter of the parity-check matrix to $d = 10$.

E.2.5 Size of the Seeds.

We estimate below the size of the seeds stored by P_0 and P_1, using the MPFSS of [BCGI18]. For concreteness, we consider an AES-based implementation of the PRGs, and instantiate the elliptic curve with the two candidates described above. The PRG seeds $\mathsf{k}_0, \mathsf{k}_1$ are λ-bit long, while $|\boldsymbol{K}_0| = |\boldsymbol{K}_1| = k \cdot (\lceil \log \ell \rceil \cdot (\lambda + 2) + \lambda + \lceil \log q \rceil)$ with $\ell = \alpha \cdot n$. Each $\boldsymbol{c}_i$ for $i = 1, 2$ is of size $2 \cdot \sqrt{\ell} \cdot |\mathbb{G}_i|$. Therefore:

- The total size of P_0's seed is $\lambda + k \cdot (\lceil \log \ell \rceil \cdot (\lambda + 2) + \lambda + \lceil \log q \rceil)$ bits.
- The total size of P_1's seed is $2 \cdot \sqrt{\ell} \cdot (|\mathbb{G}_1| + |\mathbb{G}_2|) + \lambda + k \cdot (\lceil \log \ell \rceil \cdot (\lambda + 2) + \lambda + \lceil \log q \rceil)$ bits.

Concretely, consider a target number of 2^{27} ROTs, which are computed using two parallel instances of the above PCG for $n = 2^{26}$ where P_0 and P_1 exchange their roles in the two instances (to balance the storage load), and set $\alpha = 2$ (a smaller α would further reduce the storage, but at the cost of increasing k, which has an impact on the computational efficiency of the Convert procedure). Then, using the MNT curve, each party stores a seed of length 3.76 Megabytes, which on average amounts to 4.3 ROT produced per bit of the seed. Using the Cocks-Pinch curve, the seed size increases to 8 Megabytes (2 ROT produced per bit stored). We represent on Table 6 the seed size (in Megabytes) for generating n ROT correlations, for various values of n.

n	2^{17}	2^{19}	2^{21}	2^{23}	2^{25}	2^{27}	2^{29}	2^{31}
MNT Curve	0.16	0.30	0.59	1.16	2.31	3.76	7.50	15.0
ROT/bit (MNT)	0.10	0.21	0.42	0.86	1.73	4.25	8.52	17.1
Cocks-Pinck Curve	0.33	0.63	1.24	2.47	4.91	8.01	16.0	32.0
ROT/bit (Cocks-Pinch)	0.05	0.10	0.20	0.41	0.81	2.00	4.00	8.00

Table 6. Seed size for generating n ROT correlations. We consider a balanced version of the group-based PCG, where the parties obtain seed of the same length by exchanging their roles in two parallel PCG instances. We set λ to 128, and choose the values of α and k according to Table 5. We provide the size (in Megabytes) of the seeds using both an MNT curve and a Cocks-Pinch curve, whose parameters are described at the beginning of Section E.2. The row ROT/bit (curve) denotes the number of ROT produced per bit of the seed stored, i.e., the bit size of the seed divided by n, using the curve curve.

E.2.6 Efficiency of Expand.

We now estimate the computational cost of each step of the Expand procedure. In this section, we assume that the column-weight d of the parity-check matrix of N is upper bounded by 10, which was shown to be reasonable choice in [ADI+17]. With this parameters, computing the map $\boldsymbol{m} \mapsto N \cdot \boldsymbol{m}$ requires $10 \cdot (2\alpha - 1)n$ XORs using the linear-time algorithm of [LM10, KS12] (see e.g. [BCGI18]). For each of the two parallel executions of Expand, the parties do the following (we ignore some costs which are several orders of magnitude smaller than all other costs, e.g. computing $\boldsymbol{x} \leftarrow N \cdot \boldsymbol{d}_0$ and $\boldsymbol{y} \leftarrow N \cdot \boldsymbol{d}_1$):

1. Both parties execute 4 instances of MPFSS.FullEval.
2. P_0 computes 4ℓ pairings (to build $\boldsymbol{c}_t$ from $\boldsymbol{c}_1, \boldsymbol{c}_2$).
3. P_1 computes 9ℓ exponentiations over $\mathbb{G}_t$ (to build $\boldsymbol{c}_t$ from $(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{r}_1, \boldsymbol{r}_2)$).
4. Both parties compute $40 \cdot (2\alpha - 1)n$ multiplications over $\mathbb{G}_t$ (to build $(c'_i)_{i \le n}$ from $\boldsymbol{c}_t$ and N; the XORs are evaluated as homomorphic additions over $\mathbb{Z}_q$, and each such homomorphic addition requires the term-by-term product of two ciphertexts, hence 4 multiplications over $\mathbb{G}_t$).
5. The parties execute n distributed discrete logarithms (the Convert algorithm). The efficiency of this step depends on the average size of the values z_i such that $\mathsf{Pair}(c'_i, \langle \mathsf{sk}_t \cdot d'_i \rangle) = \{g_t^{z_i}\}$. Concretely, each z_i is the product of the i'th coordinates of $N \cdot \boldsymbol{d}_0$ and $N \cdot \boldsymbol{d}_1$. Assuming that N has average row-weight $\ell/2$, and since $\boldsymbol{d}_0$ has k non-zero entries (all equal to 1) and $\boldsymbol{d}_1 = \boldsymbol{a} \otimes \boldsymbol{b}$ has k^2 non-zero entries (all equal to 1), the average size of z_i is $k^3/4$. Using the optimized DDLOG of [DKK18] with average payload $k^3/4$, the failure probability δ of the output is approximately $\delta = B(T) \cdot k^3/(4 \cdot T^2)$, where T is the number of multiplications over $\mathbb{G}_t$ performed by each party, and $B(T)$ is a value below $2^{10.4}$ (optimal values of B for various T are given in Table 1 of [DKK18]). We represent on Table 7 the failure δ obtained for various output lengths n and number of multiplications T, using the values $B(T)$ from Table 1 of [DKK18].

n	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}	2^{28}	2^{30}
$T = 2^{13}$	$2^{-5.7}$	$2^{-6.0}$	$2^{-6.6}$	$2^{-7.3}$	$2^{-8.2}$	$2^{-8.8}$	$2^{-10.1}$	$2^{-10.6}$
$T = 2^{14}$	$2^{-7.6}$	$2^{-8.0}$	$2^{-8.6}$	$2^{-9.3}$	$2^{-10.1}$	$2^{-10.8}$	$2^{-12.0}$	$2^{-12.5}$
$T = 2^{15}$	$2^{-9.5}$	$2^{-9.9}$	$2^{-10.5}$	$2^{-11.2}$	$2^{-12.0}$	$2^{-12.7}$	$2^{-14.0}$	$2^{-14.5}$
$T = 2^{16}$	$2^{-11.5}$	$2^{-11.9}$	$2^{-12.5}$	$2^{-13.2}$	$2^{-14.0}$	$2^{-14.7}$	$2^{-15.9}$	$2^{-16.4}$
$T = 2^{17}$	$2^{-13.8}$	$2^{-14.0}$	$2^{-14.6}$	$2^{-15.3}$	$2^{-16.2}$	$2^{-16.8}$	$2^{-18.1}$	$2^{-18.6}$
$T = 2^{18}$	$2^{-15.6}$	$2^{-16.0}$	$2^{-16.6}$	$2^{-17.3}$	$2^{-18.1}$	$2^{-18.8}$	$2^{-20.0}$	$2^{-20.5}$

Table 7. Failure probability δ for various choices of n, T and an average size $k^3/4$ of the exponent. The value $B(T)$ is chosen according to Table 1 of [DKK18]. We choose the optimal value of k for each value of n according to Table 5.

We now estimate the average running time of each of the 5 steps outlined above, for both MNT and Cocks-Pinch elliptic curves, using the benchmark values of the Miracl library (which are obtained on one core of a 2.4 GHz Intel i5 520M processor).

1. Using the analysis of [BCGI18] for an evaluation of MPFSS.FullEval on one core of a standard laptop, using either $\alpha = 16$ or $\alpha = 24$, the running time of this step per ROT produced is bounded above by $1\mu s$. Note that we use a rough upper bound on the estimations of [BCGI18]; the average cost per ROT of this step being essentially negligible compared to the total average cost, these imprecise estimates have no significant impact on the overall estimated running time.
2. A pairing computation takes 1.9ms on an MNT curve, hence the average cost of step 2 is $4 * 1.9\ell/n = 7.6\alpha$, which amounts to 182.4ms per ROT for $\alpha = 24$, and $121.6ms$ per ROT for $\alpha = 16$. On a Cocks-Pinch curve, where a pairing takes 1.14ms, the cost is reduced to 107ms per ROT with $\alpha = 24$ (resp. $71.7ms$ with $\alpha = 16$). Note that only one of the parties performs this step, hence the cost per party is reduced by a factor 2 when balancing over two parallel instanced of PCG where the parties exchange their roles.
3. An exponentiation over $\mathbb{G}_t$ takes 0.24ms on an MNT curve (resp. 0.12ms on a Cocks-Pinch curve), hence the average cost of step 3 is $9 * 0.24\ell/n = 51.8$ms using $\alpha = 24$, or $34.6ms$ with $\alpha = 16$ over an MNT curve (resp. 25.9ms or 17.3ms over a Cocks-Pinch curve). As only P_1 executes this step, the same observation as in the previous step applies, and the average cost per party can be reduced by a factor 2.
4. An exponentiation amounts to about 1440 multiplications over $\mathbb{G}_t$ for the MNT curve, and to about 1536 multiplications over $\mathbb{G}_t$ for the Cocks-Pinch curve. Therefore, the average cost of step 4 is $(1/1440) * 64 * 0.24\ell/n = 0.26$ms or 0.17ms using $\alpha = 24, 16$ for the MNT curve, and 0.12ms, 0.08ms for the Cocks-Pinch curve.
5. This step requires T multiplications over $\mathbb{G}_t$ per party, where the choice of T determines the failure probability of each ROT (see Table 7). The estimated cost per ROT (in millisecond) of this step for both the MNT and the Cocks-Pinch curve, for various values of T, is represented on Table 8.

T	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}
MNT	1.4ms	2.7ms	5.5ms	10.9ms	21.8ms	43.7ms
CP	0.6ms	1.3ms	2.6ms	5.1ms	10.2ms	20.4ms

Table 8. Running time in milliseconds of the distributed discrete logarithm for various choices of T and an average size $k^3/4$ of the exponent. The value $B(T)$ is chosen according to Table 1 of [DKK18]. We choose the optimal value of k for each value of n according to Table 5. The running time of exponentiations is taken from the benchmark of the Miracl library, ran on one core of a 2.4 GHz Intel i5 520M processor. MNT denotes a 160-bit MNT curve, and CP denotes a 512-bit Cocks-pinch curve.

In Table 9, we represent the total running time per parties (in millisecond) when averaged over two parallel executions of the PCG (e.g. for a target output size $n = 2^{21}$, the parties execute two parallel instances of PCG with output size 2^{20}, exchanging their roles in each instance). We set $\alpha = 2$. We illustrate the cost on an instance for concreteness: to compute 2^{23} ROT, setting $\alpha = 2$ and $T = 2^{15}$ gives an average running time of 15ms per party over an MNT curve, where each ROT is correct except with probability $1 - 2^{-9.4}$ (hence, one out of 676 ROT will be faulty on average). The storage overhead (i.e., the size of the seed stored by each party) amounts to 0.35 bits per ROT.

T	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}
MNT, $\alpha = 24$	118.8ms	120.1ms	122.8ms	128.3ms	139.2ms	161.1ms
MNT, $\alpha = 16$	96.5ms	97.8ms	100.5ms	106ms	116.9ms	138.8ms
CP, $\alpha = 24$	84.8ms	86.1ms	88.8ms	94.3ms	99.4ms	109.6ms
CP, $\alpha = 16$	62.9ms	64.2ms	66.9ms	72.4ms	77.5ms	87.7ms

Table 9. Average running time per ROT (in milliseconds) of the entire Expand procedure, on one core of a 2.4 GHz Intel i5 520M processor, using the benchmark of the Miracl library. MNT denotes a 160-bit MNT curve, and CP denotes a 512-bit Cocks-pinch curve.

E.3 Extension to OLE over Larger Rings

Our optimized group-based PCG for OT correlations can be readily used to generate OLE correlations over larger (polynomial size) rings. Since the dominant cost in the seed size comes from the $O(\sqrt{n})$ BGN ciphertexts, increasing the ring size (which means increasing the size of the encrypted plaintexts) does not significantly change the seed size.

However, generalizing to larger rings comes at a cost in terms of computational efficiency. While the number of pairing operations, which is the main efficiency bottleneck for generating OT correlations, remains the same, the cost of the distributed discrete logarithm procedure scales linearly with the increased field size. More precisely, suppose two parties want to generate an OLE correlation over a ring $\mathbb{Z}_t$, for some small polynomial t, with a failure probability of δ per output. Executing a distributed discrete logarithm on a value of average size N requires $T(\delta)$ steps, where $T(\delta)$ is chosen such that $\delta = B \cdot N/T^2$, with $B \approx 400$ [DKK18]. In the group-based PCG for OLE correlations, the target value z for the distributed discrete logarithm is of the form $x \cdot y$, where x is a value computed as the inner product between an arbitrary vector and a weight-k^2 random vector, and y is a value computed as the inner product between an arbitrary vector and a weight-k random vector (where all vectors are over $\mathbb{Z}_t$). Therefore, the average size of z is $N = k^3 \cdot (p-1)^2/4$. Hence, the number of steps T to be performed grows as

$$T = (p-1) \cdot \sqrt{\frac{B \cdot k^3}{4\delta}}.$$

Let us fix for example a target failure probability of $\delta \approx 2^{-15}$ per output, and a target number $n = 2^{26}$ of OLE correlations (hence $k = 12$ by Table 5). Using an MNT curve (using the Miracl benchmark to estimate the cost of operations) and solving the above equation for T gives an estimated running time of $(t-1) \cdot 12.5$ milliseconds over $\mathbb{Z}_t$ for the distributed discrete logarithm procedure. For the same n, k and using $\alpha = 16$, the estimated running time of expand, ignoring the distributed discrete logarithm, is about 95 ms. Therefore, the estimated running time to compute OLE correlations over $\mathbb{Z}_t$, for $n = 2^{26}$ and with failure probability 2^{-15} per output, grows as

$$95 + (t-1) \cdot 12.5 \text{ milliseconds.}$$

E.4 Sanitizing the Output: the Punctured OT Approach

While the previous sections gives reasonable estimates of the cost of generating n ROT with a group-based PCG, the ROT computed this way will be faulty, due to the inverse-polynomial failure probability of the group-based PCG. After computing n instances of ROT correlations using group-based PCG, it remains for the parties to securely remove on average $\delta \cdot n$ faulty outputs (where δ is the failure probability of the PCG). This failure probability comes from the imperfect correctness of the distributed discrete logarithm procedure. The DDLOG protocols of [BGI16a] and [DKK18] both allow a selected party to detect whether there is a risk of error at a mild cost (e.g. with the scheme of [DKK18], detecting failures requires computing an additional

$O(M)$ number of multiplications, where M is the bound on the input, independently of the failure probability). Unfortunately, we cannot simply let the selected party notify his opponent about the faulty outputs to remove them: knowing that a failure risk was detected would leak informations to this party. [17] To overcome this issue, the works of [BGI16a] and [BCG+17] developped different strategies, which we briefly outline below.

- Punctured OT [BGI16a]. To allow for secure reconstruction of faulty outputs, the work of [BGI16a] suggest to use a simple erasure-correction code, and to let one of the parties obliviously recover all outputs at positions where he did not detect a failure. By the properties of the erasure code, the recovered outputs suffice to reconstruct the entire output. The core observation of [BGI16a] is that an all-but-few random OT can be constructed efficiently: there is a protocol which securely realizes $(n-t)$-out-of-n random OT, with a communication proportional to t and $\log n$ only. This protocol relies on a puncturable pseudorandom function.
- Leakage-Absorbing Pads [BCG+17]. Alternatively, the work of [BCG+17] suggests to add some number of level-2 shares of random bits to the BCG seed, and develop methods to perform the computations on values masked with these random bits. This allows to square the leakage probability, as a failure will leak information only if two failures occured with respect to the same random mask. Asymptotically, this strategy has costs comparable to punctured OT, but leads to a considerably simpler distributed seed generation. However, the technique can only be used to mask bit inputs, and does not seem to generalize to larger integers.

Below, we apply the punctured OT approach [BGI16a], which we estimated to be more efficient in our scenario. We provide a detailed overview of an optimized application of this approach to our scenario. To implement this approach, we need two ingredient:

- an efficient punctured OT protocol, and
- a randomized linear code with good erasure-correction properties.

Punctured OT protocols can be built from t-puncturable pseudorandom functions, together with an appropriate two-party computation protocol. We recall the definition of puncturable PRFs below, and describe an optimized algorithm to puncture a PRF at t points efficiently.

E.4.1 Puncturable Pseudorandom Function.

We first recall the definition of puncturable pseudorandom functions (pPRFs), as they are the main primitive involved in the efficient realization of a punctured OT protocol, and describe an improved construction of a t-pPRF from the GGM PRF.

Definition 50 (t-Puncturable Pseudorandom Function). *A puncturable pseudorandom function with key space $\mathcal{K}$, domain $\mathcal{X}$, and range $\mathcal{Y}$, is a pseudorandom function F with an additional punctured key space $\mathcal{K}_p$ and three probabilistic polynomial-time algorithms ($F.\mathsf{KeyGen}$, $F.\mathsf{Puncture}$, $F.\mathsf{Eval}$) such that*

- *$F.\mathsf{KeyGen}(1^\lambda)$ outputs a random key $K \in \mathcal{K}$,*
- *$F.\mathsf{Puncture}(K, x)$, on input a ley $K \in \mathcal{K}$, and a subset $S \subset \mathcal{X}$ of size t, outputs a punctured key $K\{S\} \in \mathcal{K}_p$,*
- *$F.\mathsf{Eval}(K\{S\}, x)$, on input a key $K\{S\}$ punctured at all points in S, and a point x, outputs $F(K, x)$ if $x \notin S$, and $\perp$ otherwise,*

such that no probabilistic polynomial-time adversary wins the experiment Exp-s-pPRF *represented on Figure 7 with non-negligible advantage over the random guess.*

[17] Denoting $\boldsymbol{x}$ and $\boldsymbol{y}$ the inputs of the parties, and N_i the ith line of the matrix N, notifying a party of a risk of failure for the output i leaks to this party the value $(N_i\boldsymbol{x}) \cdot (N_i\boldsymbol{y})$ over the integers (with high probability). While $(N_i\boldsymbol{x}) \cdot (N_i\boldsymbol{y})$ mod 2 is pseudorandom under LPN and leaks nothing about $\boldsymbol{x}, \boldsymbol{y}$, this is not the case for the product over the integer, and this leakage cannot be allowed in general.

Experiment Exp-s-pPRF

Setup Phase. The adversary $\mathcal{A}$ sends a size-t subset $S^* \in \mathcal{X}$ to the challenger. When it receives S^*, the challenger picks $K \xleftarrow{\$} F.\mathsf{KeyGen}(1^\lambda)$ and a random bit $b \xleftarrow{\$} \{0,1\}$.

Challenge Phase. The challenger sends $K\{S^*\} \leftarrow F.\mathsf{Puncture}(K, S^*)$ to $\mathcal{A}$. If $b = 0$, the challenger additionaly sends $(F(K,x))_{x \in S^*}$ to $\mathcal{A}$; otherwise, if $b = 1$, the challenger picks t random values ($y_x \xleftarrow{\$} \mathcal{Y}$ for every $x \in S^*$) and sends them to $\mathcal{A}$.

Fig. 7. Selective security game for puncturable pseudorandom functions. At the end of the experiment, $\mathcal{A}$ sends a guess b' and wins if $b' = b$.

A pPRF can be constructed from any length-doubling pseudorandom generator, using the celebrated GGM construction [GGM86]. It proceeds as follows: On input a key K and a point x, set $K^{(0)} \leftarrow K$ and perform the following iterative evaluation procedure: for $i = 1$ to $\ell \leftarrow \log |x|$, compute $(K_0^{(i)}, K_1^{(i)}) \leftarrow G(K^{(i-1)})$, and set $K^{(i)} \leftarrow K_{x_i}^{(i)}$. Output $K^{(\ell)}$. This procedure creates a complete binary tree with edges labeled by keys; the output of the PRF on an input x is the key labeling the leaf at the end of the path defined by x from the root of the tree.

- $F.\mathsf{KeyGen}(1^\lambda)$: output a random seed for G.
- $F.\mathsf{Puncture}(K, z)$: on input a key $K \in \{0,1\}^k$ and a point x, apply the above procedure and return $K\{x\} = (K_{1-x_1}^{(1)}, \ldots, K_{1-x_\ell}^{(\ell)})$.
- $F.\mathsf{Eval}(K\{x\}, x')$, on input a punctured key $K\{x\}$ and a point x, if $x = x'$, output $\perp$. Otherwise, parse $K\{x\}$ as $(K_{1-x_1}^{(1)}, \ldots, K_{1-x_\ell}^{(\ell)})$ and start the iterative evaluation procedure from the first $K_{1-x_i}^{(i)}$ such that $x'_i = 1 - x_i$.

To obtain a t-puncturable PRF with input domain $[n]$, one can simply run t instances of the above puncturable PRF and set the output of the PRF to be the bitwise xor of the output of each instance. With this construction, the length of a key punctured at t points is $t\lambda \log n$, where λ is the seed size of the PRG. However, in the context of using a pPRF to design a punctured OT, we observe that we can do better. Intuitively, to obtain a t-puncturable PRF out of the GGM PRF, it suffices to define a key punctured at a subset S of leaves to be the smallest set of intermediate PRG values that allows to reconstruct all leaf values indexed by $[n] \setminus S$, and does not allow to reconstruct the leaf values indexed by S. We represent on Figure 8 a labelling algorithm which finds the indices of such a subset of the keys. The correctness of the algorithm follows easily by inspection; with a little more effort, one can also show that this algorithm is optimal (i.e., it produces the smallest possible punctured key satisfying the constraints). The worst-case scenario is easily seen to happen when all the punctured leaves are regularly spaces, with a distance of n/t between every two punctured leaves. This observation allows to upper bound the length of a key punctured at t points by $t\lambda \log(n/t)$, improving over the cost $t\lambda \log n$ of the naive approach.

E.4.2 Punctured OT.

A t-out-of-n oblivious transfer (OT) protocol involves a sender, with a database $D = (d_1, \ldots, d_n)$, and a receiver holding a subset $S \subset [n]$ of size $|S| = t$. The receiver should learn all entries $(d_i)_{i \in S}$, without learning the entries indexed by $[n] \setminus S$, while the sender should not learn which entries the receiver got. Standard t-out-of-n OT protocols involve $O(\lambda \cdot (t + n))$ bits of communication. In [BGI17], the authors observed that when t is very close to n ($n - t = o(n)$), this primitive can be implemented more efficiently, using only $n + o(n)$ bits of communication. We outline the construction below; it relies on a general two-party computation protocol (modeled as an oracle Π, which can be implemented with a trusted setup or any standard protocol, such as Yao's protocol) and a t-puncturable PRF F with domain $[n]$.

Algorithm **Puncture-Label**

Input. A complete binary tree T with n leaves (indexed by $[n]$), and a size-t subset S of $[n]$. We denote by $s_1 < s_2 < \cdots < s_t$ the indices of the leaves in S.
Output. A labelling L_t of all nodes of T, such that all nodes of $[n] \setminus S$, and only them, belong to a subtree of T whose root belongs to L_t.
Procedure. The labelling proceeds in t steps. Given a leave x and a subtree T' of T which contains x, we denote by $\mathsf{Label}(x, T')$ the procedure which outputs all nodes of T' which have their parent node in P but are not in P themselves, where P denotes the path from the root of T' to x.

- In step 1, set $L_1 \leftarrow \mathsf{Label}(s_1, T)$.
- In step $i+1$, let T_{i+1} denote the smallest subtree of T which contains s_{i+1} and whose root belongs to L_i (T_{i+1} exists by construction), and let r_{i+1} denote its root. Set $L_{i+1} \leftarrow (L_i \setminus \{r_{i+1}\}) \cup \{\mathsf{Label}(s_{i+1}, T_{i+1})\}$.

After all steps are completed, output L_t.

Fig. 8. Labelling algorithm to compute the indices of a subset of keys in the GGM PRF construction which allows to reconstruct the output of the GGM PRF at all points except exactly t.

1. The parties invoke Π on a randomized functionality that, on input $S \subset [n]$ from the receiver, outputs a random PRF key K to the sender, and the key $K\{[n] \setminus S\}$ punctured at all points in $[n] \setminus S$ to the receiver.
2. For $i = 1$ to n, the sender computes and sends $d'_i \leftarrow d_i \oplus F(K, i)$.
3. The receiver outputs $(i, d'_i \oplus F.\mathsf{Eval}(K\{S\}, i))$ for all $i \in [n] \setminus S$.

Plugging in Yao's protocol for Π, and the GGM construction for F, this leads to a protocol with $n + (n-t) \cdot \log n \cdot \mathsf{poly}(\lambda)$ bits of communication, which is $n + o(n)$ when $n - t$ is sufficiently small, hence the result.

E.4.3 Erasure-Correcting Code.

The second ingredient we need is a randomized linear code with good erasure-correction properties. Such codes are common in the literature.

Lemma 51 (Lemma 5.2 from [BGI17]). *There is a randomized linear encoding function $E_r : \{0,1\}^n \mapsto \{0,1\}^{n+n/\lambda}$ that can correct a $1/\lambda^2$ rate of random erasures, with all but $n \cdot \mathsf{negl}(\lambda)$ probability.*

The encoding of an n-bit input x is obtained by appending n/λ bits $x'_1, \cdots, x'_{n/\lambda}$ to x, where each x'_i is the parity of a random subset of $\lambda^2/2 - 1$ bits of x. By a standard Chernoff bound, the probability that a given bit of x cannot be recovered (which happens when all subsets containing this bit contain an erasure) is bounded by $n \cdot 2^{\lambda/3}$, hence the result.

While the above lemma provides an asymptotic statement, in practice computing the parity bit of $O(\lambda^2)$ outputs with the PCG would be too expensive. Instead, we will seek to obtain a good tradeoff between the cost of computing the parity bits (we obtain better computational efficiency when each parity bit is the parity of few bits), and the total size of the encoding (we obtain a smaller size when each parity bit is the parity of many bits). Explicit choices of parameters a discussed in the section Efficiency Estimations below.

E.4.4 Putting the Pieces Together.

Given a punctured OT and an efficient erasure-correcting code, the following protocol gives an asymptotically good method to sanitize the faulty outputs of the group-based PCG: first, instead evaluating the target bilinear correlation B on the input with the PCG, the functionality is modified to evaluate $E(B(\cdot))$ instead, where E is the encoding function of the above code (note that E is linear, hence $E \circ B$ is a bilinear function).

Remark 52 (Preprocessing the Sanitizing Phase). It would be desirable, in our context, to preprocess the material needed to execute the sanitization phase (i.e., the punctured OT), so as to include the appropriate material directly in the seed of the PCG. However, the indices of the faulty outputs are not known when the PCG seed is generated. Nevertheless, the parties can preprocess the punctured OT on a uniformly random subset S of the appropriate size t (which adds $\lambda t \log n$ bits to the seed). In the sanitization phase, the receiver can first send a permutation σ of $[n]$ that maps all faulty outputs to S, and the sender applies σ to his output before executing the step 2. As S is random, this leaks nothing about which outputs failed. However, this increases the communication of the sanitization phase to $n \log n$, superlinear in the number n of ROT produced.

As observed in [BCG+17], the cost of the sanitization phase can be reduced to $O(n)$ by executing it on *blocks* of outputs, rather than on individual outputs. Observe that each output of the GGM PRF is a random-looking λ-bit key; hence, this key can be used to mask up to λ outputs directly (or any larger number of outputs by first feeding this key to a PRG). The parties partition $[n]$ into $N = O(n/\log n)$ blocks (indexed by $[N]$), each containing n/N outputs, for some tradeoff parameter N. As there are t faults in total, at most t blocks contain a faulty output. The parties execute a punctured OT on a random size-t subset S of $[N]$. In the sanitization phase, the receiver sends a permutation σ of $[N]$ that maps the indices of all blocks containing at least one faulty output to S. Note that exchanging this permutation requires $O(N \log N) = O(n)$ bits of communication. Note that the erasure code must also be applied directly at the block level.

E.4.5 Efficiency Estimations.

We provide an estimation of the cost of using punctured OT to mitigate the leakage. Consider the task of computing 2^{26} ROTs using the group-based PCG, setting $\alpha \leftarrow 16$ and $T \leftarrow 2^{16}$. The probability of a failure event for each individual output with these parameters is $2^{-14.7}$, and these events are independent of each other. By a standard Chernoff bound, for any ε, the probability that the number of failures exceeds $n \cdot 2^{-14.7} \cdot (1+\varepsilon) = 2^{11.3}(1+\varepsilon)$ is bounded by

$$\Pr[X \geq 2^{11.3}(1+\varepsilon)] \leq \left[\frac{e^{\varepsilon}}{(1+\varepsilon)^{1+\varepsilon}}\right]^{2^{11.3}},$$

solving for the smallest ε such that the above quantity is bounded above by 2^{-80} gives $\varepsilon \approx 0.217$. Therefore, except with probability $\approx 2^{-80}$, the total number of failures will be bounded by $t = 1.217 \cdot 2^{11.3} < 2^{11.6}$.

We set the number of blocks N to be $n/128 = 2^{19}$ (hence each block contains 128 consecutive outputs). The size of a PRF key punctured at t points, for a GGM PRF over domain $[N]$, is upper bounded by $t\lambda \log(N/t) < 2^{11.6} \cdot 2^7 \cdot (19 - 11.6) < 2^{21.5}$ bits. As the seed of the group-based PCG for $n = 2^{26}$ and $\alpha = 16$ already has size $\approx 2^{25}$ bits (for a Cocks-Pinch elliptic curve), adding the punctured key to the PCG seed does only marginally increase its size.

We now turn our attention to the parameters of the erasure-correcting code. We set the number nb of blocks involved in a parity check to be an arbitrary small number, say, 5 (other choices of nb would lead to a different tradeoff between computation and communication). Suppose that we want to ensure that every punctured block can be reconstructed, except with global probability 2^{-40} (we stress that this is a statistical success probability, it has no impact on security). As there is at most $2^{11.6}$ punctured blocks (out of 2^{19} blocks in total), this means that every block must be involved in at least $-40/\log(5 \cdot (2^{11.6}/2^{19})) \approx 7.9$ parities to ensure a $1 - 2^{40}$ probability of successful reconstruction. By a standard Chernoff bound, this means that the number of parity-check blocks to be added to the output is approximately $7.9 * n/\mathsf{nb} \approx 1.58n$.

To apply the error-correcting code as part of the PCG Expand procedure, the parties executes the steps 1-5 of Expand as previously. Then, for each parity-check block to be computed, the parties retrieve their multiplicative shares of $(g_t^{\boldsymbol{b}_1}, \cdots, g_t^{\boldsymbol{b}_5})$, where $\boldsymbol{b}_1 \cdots \boldsymbol{b}_5$ correspond to the

$\mathsf{nb} = 5$ blocks of outputs involved in the parity check, homomorphically multiply their shares (getting multiplicative shares of $g_t^{\sum_{i=1}^5 \boldsymbol{b}_i}$), compute a DDLOG to recover additive shares of $\sum_{i=1}^5 \boldsymbol{b}_i$, and reduce their shares modulo 2. This adds to the total computation $1.58n$ DDLOGs on exponents of average size $5 \cdot (k^3/4)$ (the cost of the homomorphic multiplications is negligible compared to the other costs).

E.4.6 Running Time and Communication of the Sanitization.

With these parameters, using again the benchmark numbers of the Miracl library, the average running time for computing each *sanitized* ROT on one core of an Intel i5 processor over a 512-bit Cocks-Pinch curve ($n = 2^{26}, \alpha = 16, T = 2^{16}, \mathsf{nb} = 5$) is approximately 187ms. The communication involved in the punctured OT protocol consists in exchanging a permutation of $[N] = [n/128]$, and sending N blocks masked by a PRF output, for a total communication of $(1 + 1.58) \cdot (\log(n/128) + \lambda) \cdot n/128 < 3 \cdot 2^{26}$ bits, taking $n = 2^{26}$ and $\lambda = 128$ (hence on average 3 bits per sanitized ROT). We represent on Table 10 the result of similar calculations for other choices of T and $n \geq 2^{26}$, with $\alpha = 16$. The table summarizes the size of the punctured key (which is added to the seed of the PCG), the estimated running time, and the communication required to sanitize the output (per sanitized OT correlation produced).

Please note that the running time estimates given in the table do not correspond to measured running time of an actual implementation, but are estimated from the number of operations performed, using benchmark data for estimating the running time of each operation. Therefore, these estimates do not take into account additional costs resulting from, e.g., cache-misses, and should only be seen as providing a rough indication of the actual running time rather than an accurate estimation.

(T, n)	$(2^{16}, 2^{26})$	$(2^{16}, 2^{28})$	$(2^{17}, 2^{28})$	$(2^{16}, 2^{30})$	$(2^{17}, 2^{30})$	$(2^{18}, 2^{30})$
Punct. key size	$2^{21.5}$	$2^{22.4}$	$2^{20.7}$	$2^{23.9}$	$2^{22.4}$	$2^{20.7}$
Runtime (est.)	187ms	160ms	148ms	154ms	144ms	151ms
Comm. (bit/ROT)	3	2.6	2.2	2.5	2.2	2.0

Table 10. Punctured key size, estimated running time (for the total computation, including the seed extension plus the sanitization), and communication per sanitized ROT, for various choices of (T, n). We use $\alpha = 16$ and a Cocks-Pinch curve in the calculations. The security parameter λ is set to 128, and we ensure a statistical probability of $1 - 2^{-40}$ of producing at least n sanitized OT correlations. The failure probabilities corresponding to the choice of T are taken from Table 5, and the running time estimates for the non-sanitized OT correlations are taken from Table 9.

E.5 Sanitizing the Output: a New Approach

In this section, we observe that a much more efficient sanitization procedure can be obtained by relying on the silent OT extension protocol introduced in Section 5, demonstrating an interesting and surprising interplay between our techniques for building PCGs. The key idea is that, by choosing an appropriate failure bound per output, it can be ensured that all but a tiny number of *blocks* of output correlation contain a sufficiently small number of faulty outputs. Then, this tiny number of blocks with too many faults can be safely deleted by simply revealing their position (recall that the failures are detectable, hence one of the parties can know their position). If the number of such block is guaranteed to be sufficiently small with overwhelming probability, this only incurs a leakage of $O(1)$ bits of information about the seed of our group-based PCG. Then, the security of the construction is maintained under the non-standard but plausible assumption that the underlying assumption (LPN or a variant of it) is resilient to a constant amount of leakage. Eventually, since all remaining blocks are guaranteed to contain a sufficient number t of

non-faulty outputs, a t-out-of-m oblivious transfer protocol can be used to let one of the parties securely select the t non-faulty correlations. Using the silent OT extension protocol introduced in Section 5, these t-out-of-m oblivious transfers can be locally generated from a short seed (which can be added to the group-based seed with almost no overhead) by the parties. Compared to the punctured OT approach, this method incurs a slightly larger communication (about 4 bits per sanitized outputs, instead of 3 with punctured OT), but is computationally much more efficient, and requires much less preprocessing material (which, in particular, should considerably simplify the task of distributively generate the preprocessing material; note also that our silent OT extension admits a very efficient distributed setup based on the Doerner-shelat protocol). We elaborate below.

E.5.1 Concrete Instantiation.

Assume for simplicity that the parties want to generate $n = 2^{26}$ sanitized correlations. The PCG seed is made of the seed of our group-based PCG, together with two seeds for a 1-out-of-4 silent OT extension. To generate n sanitized correlations, the parties P_0 and P_1 first generate $2n$ faulty correlation (using e.g. two parallel instances of the group-based PCG with $n = 2^{26}$, exchanging their roles to balance the cost), setting the parameters so that each output has a probability bounded by 2^{-15} of being faulty (based on our previous estimates, this requires performing about $T = 72 \cdot 10^3$ multiplications for each distributed discrete logarithm, which takes about 12ms using an MNT curve). At the same time, the parties locally generate n random 1-out-of-4 OTs, using the silent OT extension protocol. We assume that P_0 has detected the position of the potentially faulty outputs.

To sanitize the output, the parties divide the $2n$ outputs into $n/2$ blocks of 4 outputs. P_0 first indicate to P_1 the position of all blocks that contain at least three faulty outputs; by a standard Chernoff bound, there will be at most 3 such blocks in total, except with probability bounded by $\min_{\ell>0}(1 + 2^{-15} \cdot (e^\ell - 1))^{n/2}/e^{4\ell} < 2^{-70}$ (note that this is a statistical security guarantee). Therefore, under the plausible assumption that LPN with tensored noise is resilient to 9 bits of *random* leakage, revealing the position of these blocks does not harm the security of the protocol. Both parties locally delete the corresponding block.

For all remaining blocks, which are guaranteed to contain at most 2 faulty outputs, the parties execute two 1-out-of-4 OT protocol using the preprocessed material from the silent OT extension, where P_1 plays the role of the sender. More precisely, for each 4-tuple of shares (u_0, u_1, u_2, u_3), P_1 locally generate new shares (v_0, v_1), and uses $(u_0 + v_0, u_1 + v_0, u_2 + v_0, u_3 + v_0)$ and $(u_0 + v_1, u_1 + v_1, u_2 + v_1, u_3 + v_1)$ as input to the two 1-out-of-4 OT instances. If the faulty outputs are at positions 0 and 1 (for example), P_0 will securely retrieve $(u_2 + v_0, u_3 + v_1)$, hence the two parties will now have shares of the non-faulty correlations (u_3, u_4). The protocol involves communicating 2 bits and 8 value per block, hence 4 values and 1/2 bit per sanitized correlation. In terms of computation, the main overhead comes from the need of computing twice more correlations in the first place, hence a factor-2 overhead compared with the non-sanitized group-based PCG. Note that this cost has not been optimized: both the communication and the computation of the sanitized PCG can be reduced by generating less faulty outputs, and fine-tuning the size of the blocks and the number of outputs to retrieve per block. We leave the optimization and the fine-tuning of this approach to future work.

F PCG from Lattices

In this section we give a lattice-based PCG construction for any family of polynomials of bounded degree over large finite fields, extending the results of the previous sections to more general correlations. As a use-case we consider the generation of authenticated Beaver triples, that is for the correlation

$$\{(a, b, ab, a\alpha, b\alpha, ab\alpha) \mid a, b \in \mathbb{Z}_p\}$$

for some fixed MAC $\alpha \in \mathbb{Z}_p$. We provide efficiency estimates for joint seed generation (with security against semi-honest adversaries) and silent expansion.

The assumptions we build on are the sparse MQ-assumption discussed in Section 7.1 and the ring-version of the learning with errors assumption, recalled in the following.

Definition 53 (Learning With Errors over Rings (RLWE)). *Let $N \in \mathbb{N}$ be a power of two $q \in \mathbb{N}$ with $q \geq 2$, $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ and $\mathcal{R}_q = \mathcal{R}/(q\mathcal{R})$. Let χ be an error distribution over $\mathcal{R}$. Let $s \leftarrow \chi$. The $\mathsf{RLWE}_{N,q,\chi}$-assumption states that the following two distributions over $\mathcal{R}_q^2$ are computationally indistinguishable:*

- *$\mathcal{O}_{\chi,s}$: Output (a, b) where $a \leftarrow \mathcal{R}_q, e \leftarrow \chi$ and $b = a \cdot s + e$*
- *U: Output $(a, u) \leftarrow \mathcal{R}_q^2$*

In the following we instantiate the generic construction from Section 4.4 with variants of RLWE-based homomorphic secret sharing schemes.

F.1 PCG from Somewhat Homomorphic Encryption

As observed in [DHRW16, BKS19], from a somewhat homomorphic encryption scheme which supports distributed decryption one can construct a homomorphic secret sharing scheme. In the following we give a semi-generic definition of properties the underlying encryption scheme has to satisfy. Note that the definition can be instantiated with a variety of lattice-based encryption schemes.

Definition 54 (Depth-d Somewhat Homomorphic Encryption w/ Distributed Decryption). *Let $\mathsf{PKE} := (\mathsf{PKE.Gen}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec})$ be an IND-CPA secure public-key encryption scheme. We say that PKE is a* secure depth-d public-key encryption scheme with distributed decryption *if it further satisfies the following properties:*

- Distributed decryption: *Let $\mathcal{R} := \mathbb{Z}[X]/(X^N+1)$, for N a power of two, $\kappa \in \mathbb{N}$ and the secret key space of PKE contained in $\mathcal{R}_q^\kappa$. We say PKE supports distributed decryption, if there exists an algorithm DDec such that for $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE.Gen}(1^\lambda)$, $\mathsf{sk}_0 \xleftarrow{\$} \mathcal{R}_q^\kappa$, $\mathsf{sk}_1 := \mathsf{sk} - \mathsf{sk}_0$, $m \in \mathcal{R}_p$, and $\boldsymbol{c} \xleftarrow{\$} \mathsf{Enc}(\mathsf{pk}, m)$ it holds $\mathsf{DDec}(\mathsf{sk}_0, \boldsymbol{c}) + \mathsf{DDec}(\mathsf{sk}_1, \boldsymbol{c}) = m$ with overwhelming probability.*
- Depth-d somewhat homomorphic encryption: *There exists a procedure $\mathsf{PKE.Eval}$ such that for any function $f\colon \mathcal{R}^n \to \mathcal{R}^m$ that can be evaluated by a circuit of depth at most d, for any $\lambda \in \mathbb{N}$, for any $(\mathsf{pk}, \mathsf{sk})$ in the image of $\mathsf{Gen}(1^\lambda)$, for all messages $m_1, \ldots, m_n \in \mathcal{R}_p$, for all ciphertexts $\boldsymbol{c}_1, \ldots \boldsymbol{c}_n$ in the image of $\mathsf{PKE.Enc}(\mathsf{pk}, m_1), \ldots, \mathsf{PKE.Enc}(\mathsf{pk}, m_n)$ and for any $\boldsymbol{c}$ in the image of $\mathsf{PKE.Eval}(f, (\boldsymbol{c}_1, \ldots, \boldsymbol{c}_n))$ it holds*

$$\mathsf{PKE.Dec}(\mathsf{sk}, \boldsymbol{c}) = f(m_1, \ldots, m_n).$$

  Notation. *We generalize $\mathsf{Alg} \in \{\mathsf{Enc}, \mathsf{DDec}, \mathsf{Eval}\}$ to vectors of inputs in a straightforward way: Alg is run independently on each entry of the vector (with independent random coins if Alg is randomized).*

Instantiating the generic construction of Section 4.4 with an HSS based on somewhat homomorphic encryption yields a PCG for any degree-d correlation (see Figure 9). As the following Theorem is a straightforward consequence of Theorem 23, we omit the proof.

Theorem 55. *Let $\mathcal{R}$ be a ring, $\ell, n, p, q, m \in \mathbb{N}$, PRG be a degree-c $\mathcal{D}^\ell$-PRG $\mathsf{PRG}\colon \mathcal{R}_p^\ell \to \mathcal{R}_p^n$ and $\mathsf{PKE} = (\mathsf{PKE.Gen}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec})$ be a depth-$\lceil \log cd \rceil$ somewhat homomorphic encryption scheme with message space $\mathcal{R}_p$ and secret key space contained in $\mathcal{R}_q^\kappa$. If PKE additionally support distributed decryption, then the PCG $\mathsf{PCG} = (\mathsf{PCG.Setup}, \mathsf{PCG.Gen}, \mathsf{PCG.Expand})$ from Figure 9 is a PCG for the family of functions $\mathcal{F} := \{f\colon \mathcal{R}_p^n \to \mathcal{R}_p^m \mid f \text{ is of degree at most } d\}$.*

$\mathsf{PCG.Gen}(1^\lambda)$:

- **Generate the encryption keys.** Generate keys $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE.Gen}(1^\lambda)$. Choose $\mathsf{sk}_0 \xleftarrow{\$} \mathcal{R}_q^\kappa$ and set $\mathsf{sk}_1 := \mathsf{sk} - \mathsf{sk}_0$.
- **Choose and encrypt a PRG-seed.** Choose $r \leftarrow \mathcal{D}^\ell(\mathcal{R}_p)$. Compute

$$\boldsymbol{c}^r = \mathsf{PKE.Enc}(\mathsf{pk}, r) \in (\mathcal{R}_q^\kappa)^\ell.$$

- Output $\mathsf{k}_0 := (\mathsf{sk}_0, \boldsymbol{c}^r)$, $\mathsf{k}_1 := (\mathsf{sk}_1, \boldsymbol{c}^r)$.

$\mathsf{PCG.Expand}(\sigma, \mathsf{k}_\sigma, f)$:

- Parse $\mathsf{k}_\sigma =: (\mathsf{sk}_\sigma, \boldsymbol{c}^r)$.
- **Evaluate $f \circ \mathsf{PRG}$ on the encrypted seed.** Compute

$$\boldsymbol{c}^Y \leftarrow \mathsf{PKE.Eval}(f \circ \mathsf{PRG}, \boldsymbol{c}^r) \in (\mathcal{R}_q^\kappa)^m.$$

- **Decrypt the result.** Decrypt and output

$$R_\sigma^Y \leftarrow \mathsf{PKE.DDec}(\mathsf{sk}_\sigma, \boldsymbol{c}^Y) \in \mathcal{R}_p^m.$$

Fig. 9. PCG for the family of degree-d functions from degree-c $\mathcal{D}^\ell$-PRG PRG and depth-$\lceil \log cd \rceil$ somewhat homomorphic encryption scheme PKE.

Remark 56. Note that the key generation of the PCG given in Figure 9 can be sourced out to the setup phase and the same secret key shares used across many instances.

Corollary 57. *Instantiating the PRG in the construction of Figure 9 with the degree-2 ρ-sparse PRG $\mathsf{PRG}_{\mathsf{MQ}} : \mathbb{Z}_p^\ell \to \mathbb{Z}_p^n$ from Definition 34 and the somewhat homomorphic encryption scheme with the BGV encryption scheme of Brakerski et al. [BGV12] (chosing parameters $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1), p, q$ and error distribution χ s.t. evaluation of at least degree-5 functions/ depth-3 circuits is supported), we obtain a PCG for the generation of authenticated Beaver triples, assuming ρ-sparse $\mathcal{M}(\ell^2, n, \mathcal{R}_p)$-MQ and $\mathsf{RLWE}_{N,q,\chi}$.*

F.2 Efficiency Estimates

Authenticated Beaver triples are used in multi-party protocols like [DPSZ12] to achieve fast online computation. Our PCG construction can be used as a plug-in to replace the preprocessing in such protocols by a short joint seed generation phase (with little communication) followed by a completely silent expansion phase. As in the described setting a large amount of Beaver triples has to be generated at once, lattice-based PCG constructions are of practical interest, despite the overhead introduced by encryption.

Generating many Beaver triples at once we can use ciphertext packing, as first observed by [SV14].

Remark 58 (Ciphertext packing, [SV14]). Let p be a prime and $N \in \mathbb{N}$ a power of 2, such that the polynomial $X^N + 1$ splits over $\mathbb{Z}_p$ into pairwise different degree-1 polynomials. If $\mathcal{R} := \mathbb{Z}[X]/(X^N + 1)$ (similar for general cyclotomic polynomials), this implies $\mathcal{R}_p \cong (\mathbb{Z}_p)^N$ and enables "packing" N plaintexts into one ciphertext (by encrypting $\Psi(z)$ for some $z \in \mathbb{Z}_p^N$, where $\Psi \colon (\mathbb{Z}_p)^N \to \mathcal{R}_p$). In the following we will refer to $\mathcal{R}_p$ as *coefficient representation*, and to $(\mathbb{Z}_p)^N$ as *CRT representation*.

Thus, each ciphertext has room to hold N encryptions. We first consider "naive" ciphertext packing: We start with ℓ encryptions of each N seeds $r \in \mathbb{Z}_p^\ell$, perform the expansion homomorphically on the ciphertexts (which corresponds to expanding feach of the N seeds in parallel). This gives an output of nN correlated tuples in total.

In the following we estimate efficiency of the PCG construction given in Corollary 57 with the above described ciphertext packing. We use the parameters given in [CS16] to support depth-4

homomorphic operations (as an upper bound) and plaintext space modulus $\approx 2^{128}$ listed in the following. Here, by T_M we denote the time required for multiplication over $\mathcal{R}_q$ and by T_C the time for multiplication of a $\mathbb{Z}_q$ element with an element in $\mathcal{R}_q$.

- *Dimension of $\mathcal{R}$ (over $\mathbb{Z}$):* $N \approx 13688$ (we use $N = 2^{14}$)
- *Ciphertext modulus:* $\log q \approx 750$ (we use $\log q = 744$)
- *Parameter for key switching:* $\log T \approx 140$
- *Cost of key switching:* $T_{\mathsf{KS}} \approx 2(\log q / \log T)T_C$
- *Cost of multiplication on ciphertexts:* $T_{\mathsf{Eval}} \approx 4T_M + T_{\mathsf{KS}}$
- *Cost of multiplication of constant with ciphertext:* $\approx 2T_C$
- *Cost of encryption:* $T_{\mathsf{Enc}} \approx 2(T_M + T_C)$
- *Cost of decryption:* $T_{\mathsf{Dec}} \approx 2T_M$

For MQ we use parameters $n = \ell^2/24$ and $\rho = 100$. Further, we set $\ell = c \cdot 2^9$ for $c \geq 1$. Later we will see that chosing $c = 1$ we surpass the breakeven point. In other words, $\ell = 2^9$ is the smallest choice where the total output size of the correlation generator exceeds the seed-length. Our runtime estimates are based on NFLLib [ABG+16]: A multiplication over $\mathcal{R}_q$ requires time ≈ 9.54 ms and a multiplication over $\mathcal{R}_q \times \mathbb{Z}_q$ requires time ≈ 0.55 ms. For an overview of estimated setup computation and communication complexity (i.e. time and communication required for jointly generating the seed) and estimated expansion times for the described PCG construction and variants we refer to Table 2 in the main body.

Distributed seed generation: We first describe the setup of the keys and MAC $\alpha \in \mathbb{Z}_p$, which can be reused across many instances. First, the parties jointly generate secret key shares $(\mathsf{sk}_0, \mathsf{sk}_1)$ and the corresponding public key pk, e.g. by generating secret keys according to a suitable distribution and exchanging shares as well as the corresponding public keys. Next, both parties choose a MAC share $\alpha_\sigma \xleftarrow{\$} \mathbb{Z}_p$ and define $\boldsymbol{\alpha}_\sigma \in \mathbb{Z}_p^N$ to be the vector of all α_σ entries. Next, the parties each compute and exchange $\boldsymbol{c}^{\Psi(\boldsymbol{\alpha}_\sigma)} := \mathsf{Enc}(\mathsf{pk}, \Psi(\boldsymbol{\alpha}_\sigma))$, and set $\boldsymbol{c}^{\Psi(\boldsymbol{\alpha})} := \boldsymbol{c}^{\Psi(\boldsymbol{\alpha}_0)} + \boldsymbol{c}^{\Psi(\boldsymbol{\alpha}_1)}$.

To generate encryptions of N seeds a and b in $\mathbb{Z}_p^\ell$, both parties repeat 2ℓ times: Sample an element $\mathcal{R}_p$ at random (corresponding to N random $\mathbb{Z}_p$ elements), and as for generating an encryption of the MAC key, exchange and add up the corresponding encryption.

As computation and communication is dominated by the last step, a rough estimate in the semi-honest setting are as follows: Generating 2ℓ encryptions takes about $c \cdot 20$ seconds of computation and exchanging 2ℓ ciphertexts (each of size $2N \log q$ bits) requires $c \cdot 3$ GB of communication (per party). We estimate that in the dishonest setting communication complexity would roughly double.

Expansion rate: We expand $2\ell N$ elements in $\mathbb{Z}_q$ to nN shared authenticated Beaver triples in $\mathbb{Z}_p$ (each consisting of 6 $\mathbb{Z}_p$ elements), which corresponds to expanding roughly $c \cdot 3$ GB of seed material to authenticated Beaver triples of total size $c^2 \cdot 17$ GB.

Computational efficiency of expansion: The computational costs add up as follows.

- *Expanding the seed:* The complexity to evaluate the PRG homomorphically on 2ℓ ciphertexts sums up to $2\ell^2$ ciphertext multiplications and $4n\rho$ multiplications of a constant with a ciphertext.
- *Computing the triples:* Evaluation of f_α requires $4n$ ciphertexted multiplications.
- *Obtaining the output shares:* To obtain the output we have to decrypt $n6$ ciphertexts.

Altogether, the costs sum up to

$$\approx 4n\rho T_C + 4(2\ell^2 + 7n)T_M + 2(\ell^2 + 2n)T_{\mathsf{KS}}.$$

This gives a total computation time of around $c^2 \cdot 8.0$ hours, which corresponds to an amortized computations time of roughly 0.16 ms per authenticated Beaver triple.

F.3 PCG with Iterative Expansion

As we choose a sparse matrix distribution (namely only $\rho = 100$ non-zero entries per row) to instantiate MQ, we can locally evaluate the PRG and therefore obtain a way to iteratively generate N Beaver triples at a time. This requires slightly more computational costs (assuming one wants to discard intermediary products), but allows to generate Beaver triples whenever needed instead of having to generate all at once. This can be achieved as follows: To generate shares of N Beaver triples, one needs to compute the scalar product of some i-th column of the $\mathcal{M}(l^2, n, \mathcal{R}_p)$-MQ matrix $\mathbf{M}$ with the vector of encryptions of $r \otimes r$, where r is some seed. As $\mathbf{M}$ is sparse by assumption, this requires only to compute a linear combination of ρ products $r_i \cdot r_j$ on encryptions. With the numbers from above to generate N triples this approach inherits computational costs for expansion of

$$(4 + 2\rho)T_{\mathsf{Eval}} + 4\rho T_C + 12T_M = 4\rho T_C + (28 + 8\rho)T_M + (4 + 2\rho)T_{\mathsf{KS}}.$$

This corresponds to a computation time of about 10 seconds per iteration (i.e. per N triples of total size 1.6 MB generated), which is amortized 0.57 seconds per triple. Note that this approach is still limited to a total expansion of nN triples.

F.4 PCG with Full Ciphertext Packing

Note that the above approach limits us to go from $\approx \ell N$ elements to $\approx \ell^2 N$ elements (where N is the degree of the plaintext space over $\mathbb{Z}_p$). As N is generally quite large that leads to a somewhat limited expansion. To overcome this we investigated into packing more smartly, which allows going from $\approx N$ elements to $\approx N^2$ elements. To do so we build on the techniques of [HS18]. Even though the approach does not look promising in terms of computation due to expensive key switching operations, we believe that it is an interesting direction for future research.

For simplicity assume that $\mathcal{R} := \mathbb{Z}[X]/(X^N + 1)$ such that $X^N + 1$ splits completely over $\mathbb{Z}_p$ and further, that the Galois group $G := \mathsf{Gal}(\mathbb{Q}(w)/\mathbb{Q}) = \{X \mapsto X^j : j \in \mathbb{Z}_m^\star\}$ is cyclic, where w is a primitive N-th root of unity. In this case the automorphisms $\alpha \in G$ act on $\mathbb{Z}_p^N$ by rotating the slots. Let $r = \Psi(\boldsymbol{r}) \in R_p$ be some packed plain text in coefficient representation corresponding to a vector of N plain texts $\boldsymbol{r} \in \mathbb{Z}_p^N$. As shown in [LPR10, HS18], applying an automorphism τ to a ciphertext $\boldsymbol{c} \xleftarrow{\$} \mathsf{Enc}(\mathsf{pk}, \Psi(\boldsymbol{r}))$ leads to an encryption of $\Psi(\tau(\boldsymbol{r}))$ which can be decrypted with $\tau(\mathsf{sk})$ (roughly, the reason is that applying an automorphism does not change the norm of the noise much and therefore one can decrypt with the shifted key to the shifted plaintext). Thus, by applying an automorphism and introducing a key-switching step, we can let plaintexts in different slots interact with each other and thereby achieve truly quadratic expansion.

Again, consider the BGV scheme with depth-4 homomorphic operations and plaintext space modulus $\approx 2^{128}$. (Note that in order to get exact numbers one would have to make a careful analysis on the the noise growth introduced by applying the automorphisms. As the numbers we use support up to depth-4 homomorphic operations, whereas we only need to compute a circuit of depth-3, we assume that the parameters also apply to the described setting for the following analysis.)

We apply the matrix multiplication technique of [HS14] and assume to be given $\mathbf{M} \xleftarrow{\$} \mathcal{M}(N^2, N^2/24, \mathcal{R}_q)$ accordingly in CRT-packed form. We can now start with a single ciphertext, i.e. $2N$ elements in $\mathbb{Z}_q$. In the following we provide the numbers for obtaining $\approx N^2$ authenticated Beaver triples. Additional costs are N key switching operations during seed generation, and N^2 key switching steps during matrix multiplication (because the matrix multiplication technique requires to permute each part of the vector). Note that one can alternatively only partly expand the ciphertext to generate less triples at a time and thereby saving in terms of computations (at

a time). We obtain the following estimated costs:

$$\begin{aligned} &2NT_{\mathsf{KS}} + 2NT_{\mathsf{Eval}} + 2N^2T_{\mathsf{KS}} + 2(N^2/24)\rho + 4T_{\mathsf{Eval}} + 6T_{\mathsf{Dec}} \\ &\quad = (8N+28)T_M + 2(N^2+2N+2)T_{\mathsf{KS}} + 2(N^2/24)\rho. \end{aligned}$$

This adds up to a total expansion time in the order of $c^2 \cdot 900$ hours and is therefore far from practical. We leave it as an open question to achieve truly quadratic extension at a reasonable time.

F.5 PCG from Somewhat Homomorphic Encryption with Nearly Linear Decryption

In this section we consider a hybrid of the HSS based on somewhat homomorphic encryption and the HSS of Boyle et. al [BKS19] based on encryption schemes which satisfy "nearly linear decryption". Roughly, the idea of [BKS19] is to replace multiplications of ciphertexts $\boldsymbol{c}^x$ and $\boldsymbol{c}^y$, by a distributed decryption of the ciphertext $\boldsymbol{c}^x$ with shares of $y \cdot \mathsf{sk}$, that is a distributed decryption of y times the secret key sk. This gives an improvement in terms of computation, as in practise distributed decryption can be an order of magnitude faster than homomorphic multiplication.

Our strategy is to replace the last multiplication with an encryption of $\psi(\boldsymbol{\alpha})$ by a distributed decryption of $\psi(\boldsymbol{\alpha})$ times the secret key.

In Figure 10 we present the construction for the generation of authenticated Beaver triples for a fixed MAC $\alpha \in \mathbb{Z}_p$ employing naive ciphertext packing. As [BKS19], we require the underlying scheme to additionally support nearly linear decryption. For details on a suitable choice of encryption scheme, we refer to [BKS19].

When choosing the parameters of the underlying encryption scheme, one needs to take into account the noise growth introduced by homomorphic multiplication, as the distributed decryption technique of [BKS19] requires $p/\|e\|_\infty \ll q$, where e is the noise term in the ciphertext. We estimate that taking the parameters for depth-4 BGV scheme is sufficient in practice. With the scheme of Figure 10 we can save $3n$ evaluation operations compared to the scheme solely based on somewhat homomorphic encryption, which results in a saving of about $c^2 \cdot 0.4$ hours. We conjecture that an additional efficiency improvement over the previous scheme can be achieved by choosing the parameters more carefully.

Alternatively, we could employ the techniques of the group-based section building on function secret sharing for multi-point functions to get a compact sharing of one of the expanded seeds (i.e. $(r \otimes r)$ for one seed r). This would require switching one PRG to $\mathsf{PRG}_{\mathsf{LPN}}$ to allow for a sparse seed. Note though that in this case the MAC α has to be given in encrypted form again (and the order of multiplication to be switched), due to the structure of [BKS19]: Their scheme only supports a multiplication of an input value (= encryption) with a memory value (= secret share), where the output again is a memory value. Thus, for multiplication of a degree 3-polynomial, two of the factors have to be given as encryptions. Also, this results in significantly larger computation times, as for evaluating the PRG a non-sparse matrix has to be multiplied with a non-sparse vector (as the automorphism Ψ does not preserve sparseness).

We do not switch to the construction of Boyle et. al [BKS19] completely for the evaluation of f_α, even though this would save another n evaluation operations, as to evaluate polynomials of degree-3 or higher their construction relies on the so-called *modulus-lifting* technique, which necessitates the choice of larger parameters for the underlying ring to ensure correctness.

G Multi-Party Simple Bilinear PCG

We begin with the proof of Theorem 41, providing the general transformation from any programmable 2-party PCG for simple bilinear correlation to a corresponding M-party PCG.

$\underline{\mathsf{PCG.Gen}(1^\lambda):}$

- **Generate the encryption keys.** Generate keys $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE.Gen}(1^\lambda)$. Choose $\mathsf{sk}_0 \xleftarrow{\$} \mathcal{R}_q^\kappa$ and set $\mathsf{sk}_1 := \mathsf{sk} - \mathsf{sk}_0$.
- **Generate a share of the MAC key.** Choose $\alpha \xleftarrow{\$} \mathbb{Z}_p$, define $\boldsymbol{\alpha} \in \mathbb{Z}_p^N$ to be the vector of all α entries. Choose $\boldsymbol{s}_0 \xleftarrow{\$} \mathcal{R}_q^\kappa$ and compute

$$\boldsymbol{s}_1 := \Psi(\boldsymbol{\alpha}) \cdot \mathsf{sk} - \boldsymbol{s}_0.$$

- **Choose and encrypt a PRG-seeds.** Choose $r_a, r_b \leftarrow R_p^\ell$. Compute and output

$$\boldsymbol{c}^{r_a} = \mathsf{PKE.Enc}(\mathsf{pk}, r_a), \boldsymbol{c}^{r_b} = \mathsf{PKE.Enc}(\mathsf{pk}, r_b) \in (\mathcal{R}_q^\kappa)^\ell.$$

- Output $\mathsf{k}_0 := (\mathsf{sk}_0, \boldsymbol{s}_0, \boldsymbol{c}^{r_a}, \boldsymbol{c}^{r_b})$, $\mathsf{k}_1 := (\mathsf{sk}_1, \boldsymbol{s}_1, \boldsymbol{c}^{r_a}, \boldsymbol{c}^{r_b})$.

$\underline{\mathsf{PCG.Expand}(\sigma, \mathsf{k}_\sigma, f):}$

- Parse $\mathsf{k}_\sigma =: (\mathsf{sk}_\sigma, \boldsymbol{s}_\sigma, \boldsymbol{c}^{r_a}, \boldsymbol{c}^{r_b})$.
- **Evaluate PRG homomorphically on the encrypted seed.** Compute

$$\boldsymbol{c}^a \leftarrow \mathsf{PKE.Eval}(\mathsf{PRG}, \boldsymbol{c}^{r_a}), \boldsymbol{c}^b \leftarrow \mathsf{PKE.Eval}(\mathsf{PRG}, \boldsymbol{c}^{r_b}) \in (\mathcal{R}_q^\kappa)^n.$$

- **Evaluate F_α homomorphically on the encrypted input.** Compute

$$\boldsymbol{c}^Y \leftarrow \mathsf{PKE.Eval}(F_\alpha, \boldsymbol{c}^a, \boldsymbol{c}^b) \in (\mathcal{R}_q^\kappa)^{6n}.$$

- **Obtain shares of Y via distributed decryption with sk_σ .** Compute

$$R_\sigma^Y \leftarrow \mathsf{PKE.DDec}(\boldsymbol{s}_\sigma, \boldsymbol{c}^Y) \in \mathcal{R}_p^{6n}.$$

- **Obtain the output shares over $\mathbb{Z}_p$.** Output

$$\boldsymbol{\Psi}^{-1}(R^{\boldsymbol{Y}}) \in \mathbb{Z}_p^{6nN}.$$

Fig. 10. PCG for authenticated Beaver triples with MAC α from degree-2 PRG $\mathsf{PRG}\colon \mathbb{Z}_p^\ell \to \mathbb{Z}_p^n$ and depth-2 somewhat homomorphic encryption scheme PKE with nearly linear decryption. Here, $F_\alpha\colon \mathbb{Z}_p^n \times \mathbb{Z}_p^n \to (\mathbb{Z}_p^n)^6, (a,b) \mapsto (a, b, a \circ b, a \circ \alpha, b \circ \alpha, a \circ b \circ \alpha)$ corresponds to evaluating f_α componentwise on each of the n input tuples ($\circ$ denotes the entrywise product). By $\boldsymbol{\psi}^{-1}$ we denote the map evaluating $\psi^{-1}\colon \mathcal{R}_p \to \mathbb{Z}_p^N$ componentwise.

Proof. We analyze the following M-party PCG construction:

- $\mathsf{PCG}_M.\mathsf{Gen}(1^\lambda)$:
  1. Sample random $a'_1, \ldots, a'_M \xleftarrow{\$} \{0,1\}^\lambda$, $b'_1, \ldots, b'_M \xleftarrow{\$} \{0,1\}^\lambda$ as specified by programmability property.
  2. For every $i \neq j \in [M]$: Run $\mathsf{k}_0^{ij}, \mathsf{k}_1^{ij} \leftarrow \mathsf{PCG}_2.\mathsf{Gen}(1^\lambda, a'_i, b'_j)$ and sample PRG seed $s^{ij} \xleftarrow{\$} \{0,1\}^\lambda$
  3. For each $i \in [M]$, output $\mathsf{k}_i = \left(\{s^{ij}\}_{j\neq i}, \{\mathsf{k}_0^{ij}\}_{j\neq i}, \{\mathsf{k}_1^{ji}\}_{j\neq i}\right)$
- $\mathsf{PCG}_M.\mathsf{Expand}(i, \mathsf{k}_i)$:
  1. For every $j \neq i$, compute

$$r_{ij} \leftarrow \mathsf{PRG}(s^{ij}),$$
$$(a_{ij}, c_{ij}) \leftarrow \mathsf{Expand}_2(0, \mathsf{k}_0^{ij}), (b_{ji}, d_{ji}) \leftarrow \mathsf{Expand}_2(1, \mathsf{k}_1^{ji})$$

  2. Output $A_i = a_{ij}$, $B_i = b_{ji}$ (same for all j) and $C_i = -\sum_{j\neq i} c_{ij} + \sum_{j\neq i} d_{ji} + e(A_i, B_i) + (-1)^{[i<j]} r_{ij}$, where $[i < j] = 1$ if $i < j$ and 0 if $j < i$.

Correctness: By assumption, each expanded output from $(\mathsf{PCG}_2.\mathsf{Gen}, \mathsf{PCG}_2.\mathsf{Expand})$ is computationally indistinguishable from $\mathcal{C}_2$. In particular, it must hold whp for all $i \neq j$: $e(A_i, B_j) = d_{ij} - c_{ij}$, where $(\mathsf{k}_0^{ij}, \mathsf{k}_1^{ij}) \leftarrow \mathsf{PCG}_2.\mathsf{Gen}(1^\lambda, a'_i, b'_j), (a_{ij}, c_{ij}) \leftarrow \mathsf{PCG}_2.\mathsf{Expand}(0, \mathsf{k}_0^{ij}), (b_{ij}, d_{ij} \leftarrow$

$\mathsf{PCG}_2.\mathsf{Expand}(1, \mathsf{k}_1^{ij})$. This means with overwhelming probability,

$$\begin{aligned} e\left(\sum_{i=1}^{M} A_i, \sum_{j=1}^{M} B_j\right) &= \sum_{i=1}^{M}\sum_{j=1}^{M} e\left(A_i, B_j\right) \\ &= \sum_{i=1}^{M} e\left(A_i, B_i\right) + \sum_{i=1}^{M}\sum_{j\neq i} e\left(A_i, B_j\right) \\ &= \sum_{i=1}^{M} e\left(A_i, B_i\right) + \sum_{i=1}^{M}\sum_{j\neq i}(d_{ij} - c_{ij}) = \sum_{i=1}^{k} C_i. \end{aligned}$$

(Note that for all $i \neq j$, r_{ij} is added and subtracted exactly once in the sum.) Further, by indistinguishability of the expanded outputs from $\mathsf{PCG}_2.\mathsf{Expand}$ to the target correlation $\mathcal{C}_2$, it holds that each (a_i, b_j) pair is pseudorandom. Thus, for M independent samples, $\sum a_i$ and $\sum b_i$ are jointly pseudorandom. Finally, from the pairwise pseudorandom offsets r_{ij} (independent of the a_i and b_i), it holds that the C_i are pseudorandom, up to the required constraint. Correctness follows.

Security. We now proceed to prove security of PCG_M. Let $T \subset [M]$ corrupted. We wish to show that given $\{\mathsf{k}_i\}_{i\in T}$, the expanded outputs of honest parties $(A_i, B_i, C_i)_{i\notin T}$ cannot be distinguished from an independent resampling, conditioned on the *expanded* values $(A_i, B_i, C_i)_{i\in T}$ of the corrupt seeds.

We first observe that due to the pariwise secret pseudorandom offsets $r^{ij} = \mathsf{PRG}(s^{ij})$, that even given $\{\mathsf{k}_i\}_{i\in T}$ and $(A_i, B_i)_{i\in T}$ the joint distribution of $(C_i)_{i\notin T}$ is indistinguishable from random, up to the preserved sum $\sum_{i\notin T} C_i$ as required.

It thus remains to show that given $\{\mathsf{k}_i\}_{i\in T}$, the expanded honest values $(A_i, B_i)_{i\notin T}$ are pseudorandom. By a hybrid argument, we may replace the values of honest A_i and B_i one at a time. It then suffices to adress an extreme case of this step, where all but one party $i \in [M]$ is corrupted.

We first treat A_i; the argument for B_i is symmetric. For any $i \in [M]$,

$$\begin{aligned} &\left\{ (\{\mathsf{k}_j\}_{j\neq i}, (A_i, B_i)) \;\middle|\; \begin{array}{l} (\mathsf{k}_1, \ldots, \mathsf{k}_M) \xleftarrow{\$} \mathsf{PCG}_M.\mathsf{Gen}(1^\lambda) \\ (A_i, B_i, C_i) \xleftarrow{\$} \mathsf{PCG}_M.\mathsf{Expand}(i, \mathsf{k}_i) \end{array} \right\} \\ &\equiv \left\{ \left(\{\mathsf{k}_1^{ij}\}_{j\neq i}, f_a(a_i'), X\right) \;\middle|\; \begin{array}{l} a_i' \leftarrow \$, b_j' \leftarrow \$\ \forall j \neq i \\ (\mathsf{k}_0^{ij}, \mathsf{k}_1^{ij}) \leftarrow \mathsf{PKE}_2.\mathsf{Gen}(1^\lambda, a_i', b_j') \\ X \leftarrow \mathsf{RestOfSeeds}(\{b_j'\}_{j\neq i}) \end{array} \right\}, \end{aligned}$$

where $\mathsf{RestOfSeeds}$ is an efficiently sampleable distribution that samples $b_i' \leftarrow \$, a_j' \leftarrow \$\ \forall j \neq i$, executes the remaining $(2M-1)(M-1)$ instances of $\mathsf{PCG}_2.\mathsf{Gen}(1^\lambda, a_\ell', b_j')$ and outputs

$$\left(\{\mathsf{k}_0^{j\ell}\}_{j\neq i, l\in[M]}, \{\mathsf{k}_1^{\ell j}\}_{j\neq i, l\neq i}, B_i = f_b(b_i')\right).$$

By a direct sequence of $(M-1)$ hybrids over $j \neq i \in [M]$ we may appeal to the security of $(\mathsf{PCG}_2.\mathsf{Gen}, \mathsf{PCG}_2.\mathsf{Expand})$ to iteratively replace the jth key k_1^{ij} generated by $(\mathsf{k}_0^{ij}, \mathsf{k}_1^{ij}) \leftarrow \mathsf{PCG}_2.\mathsf{Gen}(1^\lambda, a_i', b_j')$ with $\tilde{\mathsf{k}}_1^{ij}$ generated as $(\tilde{\mathsf{k}}_0^{ij}, \tilde{\mathsf{k}}_1^{ij}) \leftarrow \mathsf{PCG}_2.\mathsf{Gen}(1^\lambda, \tilde{a}_i, b_j')$, for independent $\tilde{a}_i \leftarrow \$$.

We thus obtain the above distribution is indistinguishable from

$$\approx \left\{ \left(\{\tilde{\mathsf{k}}_1^{ij}\}_{j\neq i}, f_a(a_i'), X\right) \;\middle|\; \begin{array}{l} a_i' \leftarrow \$, \tilde{a}_i \leftarrow \$, b_j' \leftarrow \$\ \forall j \neq i \\ (\mathsf{k}_0^{ij}, \mathsf{k}_1^{ij}) \leftarrow \mathsf{PKE}_2.\mathsf{Gen}(1^\lambda, a_i', b_j') \\ X \leftarrow \mathsf{RestOfSeeds}(\{b_j'\}_{j\neq i}) \end{array} \right\}.$$

However, in this case $A_i = f_a(a'_i)$ for $a'_i \leftarrow \$$ is completely independent of $\left(\{\tilde{\mathsf{k}}_1^{ij}\}_{j\neq i}, X\right)$, generated now as a function in only $\tilde{a}_i$ (not a'_i). The claim, and thus security of the construction, follows.

We now explore sample 2-party PCG constructions that support the necessary programmability.

Proposition 59 (Programmability of 2-party PCGs). *The following 2-party PCGs are* programmable*, as per Definition 40.*

- *The 2-party VOLE generator of [BCGI18], based on DPF and LPN.*
- *The PCGs for arbitrary simple 2-party bilinear correlations as constructed in this work from somewhat-homomorphic encryption or BGN.*

Proof. M-party VOLE. Recall the 2-party VOLE generator of [BCGI18] takes the following form (we describe their "dual" construction). The sender party receives a (short representation of) a sparse random vector $\boldsymbol{y}$ over the field $\mathbb{F}$, the receiver receives a field element $x \in \mathbb{F}$, and each receives an FSS share of a multi-point function corresponding to the product $x\boldsymbol{y}$. The scheme is parameterized by a public matrix H for which the dual-LPN problem is hard (with respect to sparse noise). The sender expands to output $(\boldsymbol{u}, \boldsymbol{v})$, and the receiver to $(x, \boldsymbol{w})$.

It was already observed in [BCGI18] that the construction was programmable with respect to the receiver's output x. We observe that a similar programmability holds also for the output $\boldsymbol{u}$ of the sender, where in particular $\boldsymbol{u}$ is the output of compressing the value $\boldsymbol{y}$ via the public matrix H. In both cases, the "programming information" ($a' = \boldsymbol{u}$ and $b' = x$) is anyway given to the respective party, so the required security notion is directly implied by standard PCG security.

General Simple Bilinear via HSS. One can support 2-party PCG of any simple bilinear correlation via our PCGs for degree-2 correlations (e.g., obtained from lattices and BGN) by giving each party a short seed a', b', respectively, as well as HSS shares of a' and b' that support homomorphic evaluation of individual PRG expansion $a = \mathsf{PRG}(a'), b = \mathsf{PRG}(b')$ and then the multiplication ab. This construction inherently supports programmability, by the initial values a', b'.