The Rise of Paillier: Homomorphic Secret Sharing and Public-Key Silent OT

Claudio Orlandi, Peter Scholl, and Sophia Yakoubov

Aarhus University

Abstract. We describe a simple method for solving the distributed discrete logarithm problem in Paillier groups, allowing two parties to locally convert multiplicative shares of a secret (in the exponent) into additive shares. Our algorithm is perfectly correct, unlike previous methods with an inverse polynomial error probability. We obtain the following applications and further results.

- **Homomorphic secret sharing.** We construct homomorphic secret sharing for branching programs with *negligible* correctness error and supporting *exponentially large* plaintexts, with security based on the decisional composite residuosity (DCR) assumption.
- **Correlated pseudorandomness.** Pseudorandom correlation functions (PCFs), recently introduced by Boyle et al. (FOCS 2020), allow two parties to obtain a practically unbounded quantity of correlated randomness, given a pair of short, correlated keys. We construct PCFs for the oblivious transfer (OT) and vector oblivious linear evaluation (VOLE) correlations, based on the quadratic residuosity (QR) or DCR assumptions, respectively. We also construct a pseudorandom correlation generator (for producing a bounded number of samples, all at once) for general degree-2 correlations including OLE, based on a combination of (DCR or QR) and the learning parity with noise assumptions.
- **Public-key silent OT/VOLE.** We upgrade our PCF constructions to have a *public-key setup*, where after independently posting a public key, each party can locally derive its PCF key. This allows completely *silent generation* of an arbitrary amount of OTs or VOLEs, without any interaction beyond a PKI, based on QR, DCR, a CRS and a random oracle. The public-key setup is based on a novel non-interactive vector OLE protocol, which can be seen as a variant of the Bellare-Micali oblivious transfer protocol.

Table of Contents

1 Introduction

Homomorphic secret sharing, or HSS, allows two parties to non-interactively perform computations on secret-shared private inputs. In contrast to homomorphic encryption, where a single party carries out the computation on encrypted data, HSS can be viewed as a distributed variant where several servers are each given a share of the inputs, and then (without further interaction) can homomorphically evaluate these to obtain a share of the desired output. Useful applications of HSS include succinct forms of secure multi-party computation [BGI16a, BGI17, BGMM20], private querying to public databases [GI14, BGI15, WYG$^+$17] and generating correlated randomness in secure computation protocols [BCGI18, BCG$^+$19]. In this work, we will focus on a strong flavour of HSS with *additive reconstruction*, meaning that the server's shares of the output can be simply added together (in an abelian group) to give the result of the computation.[1]

At Crypto 2016, Boyle, Gilboa and Ishai [BGI16a] constructed two-server HSS for the class of polynomial-size branching programs based on the decisional Diffie-Hellman (DDH) assumption. Branching programs are a class of computations that cover restricted classes of circuits such as NC^1 and logspace computations. One application of their construction is succinct secure computation protocols for these types of computation, where the communication complexity is proportional only to the input and output lengths [BGI17]. However, Boyle et al. also managed to achieve secure computation for general, leveled circuits with a communication cost that is *sublinear in the circuit size* by a logarithmic factor. Previously, breaking this circuit-size barrier was only known to be possible using fully homomorphic encryption, so this result positioned HSS as an alternative path towards secure computation with low communication.

At the heart of the DDH-based construction [BGI16a] is a *distributed discrete log* procedure, where two parties are given multiplicative shares of a secret g^x (for some fixed base g), and wish to locally convert these into *additive shares* of x. Their method of solving this unfortunately has an inverse polynomial probability ε of correctness error, which is expensive to keep small, since the workload in homomorphic evaluation scales with $O(1/\varepsilon)$.

Their original HSS construction has been extended in several works, including a simpler "public-key" style sharing phase [BGI17], a variant based on Paillier encryption [FGJS17], improved efficiency of the distributed discrete log step [BCG$^+$17, DKK18] and techniques for mitigating leakage that can arise from the non-negligible correctness error [BCG$^+$17].

Despite this progress, all these constructions still have the limitation of a non-negligible chance of an incorrect computation, which requires a large amount of extra work to keep small. In fact, Dinur et al. [DKK18] showed a conditional lower bound that solving the distributed discrete log protocol with correctness probability ε *requires* $\Omega(1/\sqrt{\varepsilon})$ computation, unless the discrete logarithm prob-

[1] This leads to a form of optimally succinct reconstruction that even fully homomorphic cannot achieve on its own.

lem in an interval can be solved more efficiently. They also gave a matching upper bound.

On the other hand, if we rely on the learning with errors (LWE) assumption, instead of discrete log- or factoring-based assumptions, it is possible to obtain HSS for arbitrary circuits [DHRW16, BGI+18], and with a *negligible* probability of correctness error, when using LWE with a superpolynomial modulus. This construction builds on fully homomorphic encryption [Gen09, BV11], and despite much recent progress, this still involves a significant computational overhead. When restricting computations to branching programs instead of circuits, and limiting the number of servers to two, there is a much specialized construction that reduces these costs [BKS19].

Pseudorandom Correlation Generators. A recent, promising application of techniques based on HSS is to build *pseudorandom correlation generators* (PCGs) [BCGI18, BCG+19], which are a way of expanding short, correlated seeds into a large amount of correlated randomness. This correlated randomness might be, for instance, a batch of oblivious transfers (OTs) on random inputs, which can be used to obtain cheap, information-theoretic protocols for secure computation of Boolean circuits. Other correlations can be used to securely compute arithmetic circuits over a ring R, for example, in oblivious linear evaluation (OLE), each sample has the form $(a, b), (x, ax + b)$ for random $a, b, x \in R$. Another example is vector oblivious linear evaluation (VOLE), which has the restriction that x is fixed for each sample of the correlation.

More concretely, a PCG is a pair of algorithms $(\mathsf{Gen}, \mathsf{Expand})$, where Gen outputs a pair of short, correlated seeds, while Expand takes one of these seeds and expands it into a longer output string. The security requirements are that the joint distribution of both outputs is indistinguishable from the desired correlation, and also that each seed preserves privacy of the other party's output.

While PCGs can be constructed from suitably expressive HSS [BCG+19], this requires homomorphic evaluation of a pseudorandom generator inside HSS and typically leads to poor concrete efficiency. Instead, the most promising constructions so far are based on variants of the learning parity with noise (LPN) assumption, and build upon practical constructions of HSS for point functions (or, function secret sharing) [GI14, BGI15, BGI16b]. Using LPN, we can obtain PCGs for the VOLE [BCGI18], OT [BCG+19] and OLE [BCG+19, BCG+20b] correlations, and these can even be concretely efficient when relying on structured variants of LPN such as ring-LPN, or using quasi-cyclic codes.

Pseudorandom Correlation Functions. Very recently, Boyle et al. [BCG+20a] extended the notion of PCG to a *pseudorandom correlation function* (PCF), which allows generating an unbounded number of correlated outputs in an on-the-fly manner, given a pair of correlated keys. This is similar to how a pseudorandom function extends the concept of a pseudorandom generator. While PCFs can be constructed in a generic but inefficient manner based on LWE, Boyle et al. also gave constructions based on new flavours of *variable-density* LPN assumptions. The practical security and efficiency of these constructions has yet to be determined, although their initial results suggest that the PCFs for the OT and VOLE correlations could be concretely efficient.

1.1 Our Contributions

In this work, we present new constructions of homomorphic secret sharing and pseudorandom correlation functions based on standard, number-theoretic assumptions related to factoring. At the heart of most of our constructions is a single, key technique, namely, an efficient algorithm for solving the distributed discrete logarithm problem when using the Paillier cryptosystem over $\mathbb{Z}_{N^2}^*$ (where N is an RSA modulus). Unlike previous algorithms [BGI16a, FGJS17, DKK18], which always incurred an inverse polynomial probability of error, our method is very simple and has *perfect correctness.*

Building on this technique, we obtain the following results.

Homomorphic Secret Sharing. We construct homomorphic secret sharing for branching programs with *negligible correctness error* and supporting computations on an *exponentially large* plaintext space. We present several variants. The first two are based on circular security assumptions (of Paillier encryption and of a Paillier-ElGamal hybrid, respectively); however, the second has the advantage of allowing for a public-key style setup. The third variant also allows for a public-key setup, and additionally relies solely on the decisional composite residuosity (DCR) assumption. However, it is less efficient.

This gives the first construction of negligible-error HSS for branching programs without relying on LWE with a superpolynomial modulus. Compared with previous constructions based on discrete log-type assumptions [BGI16a, BGI17], as well as the Paillier-based construction of Fazio et al. [FGJS17], we avoid their limitations of a $1/\mathsf{poly}$ correctness error and polynomial-sized plaintext space. We also obtain smaller share sizes and much better computational efficiency.

Pseudorandom Correlation Functions and Pseudorandom Correlation Generators. We construct PCFs for producing an arbitrary number of random instances of vector oblivious linear evaluation, based on the DCR assumption, and oblivious transfer, based on the quadratic residuosity (QR) assumption. These constructions are very simple and have relatively small key sizes, consisting of $O(1)$ and $O(\lambda)$ group elements, respectively (for security parameter λ). We also construct a weaker pseudorandom correlation generator (for producing a bounded number of samples, or, all at once) for general degree-2 correlations, when assuming $\{\text{DCR} \vee \text{QR}\} \wedge \text{LPN}$. Compared with a previous construction based on only LPN [BCG$^+$19], we reduce the key size by a factor $O(\lambda)$ and reduce the computational cost from quadratic to linear in the output length. This can also be upgraded to a PCF, when assuming a recent, variable-density version of LPN [BCG$^+$20a].

Public-key Silent OT and VOLE. We show how to upgrade our PCF constructions to have a *public-key setup*. After independently posting a public key, which uses a CRS, each party can use the other party's key, together with the private randomness for its own key, to derive a key for the PCF. Using their PCF keys, the parties can then silently compute an arbitrary quantity of OT or VOLE correlations, all *without any interaction* beyond the PKI. To our knowledge, these are the first such constructions that allow producing non-trivial correlations from a reusable public-key setup, without relying on lattice-based assumptions and homomorphic evaluation of PRGs inside multi-key FHE [DHRW16, BCG$^+$19]. Note that we assume both a CRS and the random oracle model.

The public-key PCF for OT can be plugged into an existing construction of two-round multi-party computation with an OT correlations setup [GIS18], and reduces the complexity of its setup phase. This leads to a passively secure, two-round MPC protocol based on the DCR and QR assumptions, which makes a black-box use of a PRG, and has a PKI setup that scales independently of the circuit size. This is in contrast to the *strong PKI* setup from [GIS18], where the size of each public key scales linearly with the circuit size.

1.2 Comparison with Previous Results

We now give a more detailed comparison of our results with those from previous work, and discuss some efficiency metrics.

Homomorphic Secret Sharing. As already mentioned, we avoid the $1/\mathsf{poly}$ correctness error and small message space associated with previous constructions based on DDH [BGI16a, BCG$^+$17, BGI17] or Paillier [FGJS17]. This brings us the additional benefit that the share size of our HSS is smaller, since in all these constructions, each share of an input x contains encryptions of $x \cdot d_i$, where d_i are the bits of the secret key. Since we support a large message space, we can instead choose d_i to be a much larger chunk of the secret key, so that each share only contains a *constant* number of group elements, instead of $O(\lambda)$. The smaller share size and improved share conversion step in our construction also give us much lower computational costs, since previous works require a workload that scales with $\Omega(1/\sqrt{\varepsilon})$, where ε is the correctness error probability [DKK18].

We can also compare our HSS with constructions based on LWE. Using LWE with a superpolynomial modulus, these can also support negligible error and a superpolynomial plaintext space [BKS19], and can even go beyond branching programs to evaluate general circuits [DHRW16, BGI$^+$18]. When restricted to branching programs or low-depth circuits, and using ring-LWE, homomorphic evaluation would likely be more efficient than our HSS, due to fast algorithms for polynomial arithmetic in ring-LWE, compared with exponentiations in Paillier. On the other hand, our scheme has much smaller shares, since ring-LWE ciphertexts with a superpolynomial modulus are orders of magnitude larger than Paillier ciphertexts (ranging from 100kB–3MB in [BKS19] vs under 1kB for Paillier).

Finally, we remark that LWE is a very different assumption to DCR. On the one hand, it can plausibly resist attacks by quantum computers; however, in a purely classical setting, factoring-based assumptions are arguably better studied than ring-LWE with a superpolynomial modulus.

PCFs and PCGs. Compared with PCGs for OT and VOLE based on LPN [BCGI18, BCG⁺19], our PCFs have the advantages of a public-key setup and the ability to incrementally produce an unbounded number of outputs (which comes with being a PCF and not just a PCG). In VOLE, our PCF has the additional benefit of much smaller keys, since each party's key only contains two Paillier group elements, compared with $\tilde{O}(\lambda^2)$ bits using LPN. The key size in our PCF for OT is around λ elements of $\mathbb{Z}_N$, which is comparable to the LPN-based PCG keys at typical security levels. The main drawback of our constructions is their computational efficiency, since our PCF for VOLE requires one exponentiation in $\mathbb{Z}_{N^2}^*$ to produce a single output in $\mathbb{Z}_N$, while the PCF for OT requires ≈ 128 exponentiations in $\mathbb{Z}_N^*$ to obtain one string-OT (at the 128-bit security level). Similarly, our PCG for OLE (based on both LPN and DCR) reduces the key size of previous LPN-based PCGs for OLE by a factor of $O(\lambda)$, at the cost of requiring exponentiations and limited to OLEs over $\mathbb{Z}_N$ or $\mathbb{Z}_2$, rather than more general rings or fields.

The recent PCFs for OT and VOLE from variable-density LPN [BCG⁺20a] overcome the PCG limitation of standard LPN-based constructions, although still do not have a public-key setup. They also come with much larger keys than their PCG counterparts, as well as our VOLE and OT constructions. Their computational efficiency has not yet been demonstrated, although they may be faster than our number-theoretic constructions due to being based on lightweight primitives like distributed point functions.

1.3 Overview of Techniques

We start by recalling the share conversion procedure used by Boyle et al. [BGI16a]. The basic idea of their scheme allows two parties to locally multiply secrets $x, y \in \mathbb{Z}$, where x is encrypted and y is secret shared, obtaining a secret sharing of the result $z = xy$. However, z is now *multiplicatively* (or, rather, *divisively*) shared; that is, the parties have group elements $g_0, g_1 \in \mathbb{G}$, such that $g_1 = g_0 \cdot g^z$. To continue evaluating a program, they would like to convert these into *linear* (*subtractive*) shares, so they can be used in another multiplication (with a ciphertext).

Boyle et al. described an ingenious protocol for converting such divisive shares to subtractive shares. To obtain subtractive shares of z, it is enough that the parties *agree upon* some distinguished element h that is not too far away from g_0, g_1 in terms of multiplications by g. If they find such an h, then party σ can compute the distance of g_σ from h by brute force: by multiplying h by g repeatedly, and seeing how many such multiplications it takes to get to g_σ. If we're guaranteed that h isn't too far away, this should not be too inefficient. Let

d_σ be the distance of g_σ from h — that is, $hg^{d_\sigma} = g_\sigma$. Then,

$$\begin{aligned} g_1 = g_0 \cdot g^z &\Leftrightarrow hg^{d_1} = hg^{d_0} g^z \\ &\Leftrightarrow g^{d_1} = g^{d_0} g^z \end{aligned}$$

We can conclude that $d_1 \equiv d_0 + z$ modulo the order of the subgroup generated by g, and if d_0, d_1 are small then these shares can be recovered efficiently.

The major challenge is agreeing upon a common point h. Boyle et al. did so by having the parties first fix a set of random, distinguished points in the group; party σ then finds the closest point in this set to g_σ. As long as both parties find the same point, this will lead to a correct share conversion. To make this process efficient, the distance t between successive points can't be too large, since the running time will be $O(t)$. However, there is then an inherent $\approx 1/t$ probability of failure, in case a point lies between the original two shares and they fail to agree.

This leads to a tradeoff between running time and correctness of the share conversion procedure. Dinur, Keller and Klein [DKK18] described an improved conversion algorithm, which achieves $1/t$ error probability in only $O(\sqrt{t})$ steps. On the negative side, they showed that any algorithm which beats this could be used to improve the cost of finding discrete logarithms in an interval, a well-studied problem that is believed to be hard.

Despite the correctness difficulty, this weaker form of HSS still suffices for many applications including sublinear secure two-party computation, with some additional work to correct for errors [BGI16a, BGI17].

Share Conversion in Paillier. By moving to a Paillier group ($\mathbb{Z}^*_{N^2}$), we can overcome the challenges of (a) agreeing on a distinguished point and (b) efficiently finding the distance of a multiplicative share from that point, *without requiring a correctness / efficiency tradeoff.*

The parties' multiplicative shares will still have the form g_0, g_1 such that $g_1 = g_0 \cdot g^z$; however, now we take $g = (1+N)$, which has order N in $\mathbb{Z}^*_{N^2}$. To find the distinguished point h, the parties simply *reduce their shares mod* N. Remarkably, this leads to the parties always agreeing upon the same value h, which is also guaranteed to be the *smallest* value in the coset $X = (g_0, g_0 \cdot g, \ldots, g_0 \cdot g^{N-1})$. To see that parties agree on h, notice that since $(1+N)^x \equiv 1 \pmod{N}$, we have $g_0 \equiv g_1 \pmod{N}$. To see that h lies in X, write $g_0 = (h + h'N)$ and suppose that $h = g_0(1+N)^s$ for some s. Then, since $(1+N)^s \equiv (1+sN) \pmod{N^2}$, we have $h \equiv (h + h'N)(1+sN) \pmod{N^2}$, which we can easily solve to get $s \equiv -h'h^{-1} \pmod{N}$.

Given this, party σ can compute the distance from their multiplicative shares to h *without the use of brute force.* They simply take $h/g_\sigma = (1+N)^{z_\sigma}$, and then take the discrete logarithm of this (exploiting the fact that discrete logs are easy with base $1+N$) to find their additive share z_σ satisfying $z_0 - z_1 = z \bmod N$.

As well as removing the correctness error, moving to Paillier groups has removed the limitation that messages must be small, since we can efficiently apply share conversion to shares of any message in $\mathbb{Z}_N$. We are still missing one step, however; to be able to continue the HSS computation, we need shares of z *over the integers*, rather than modulo N. Using a trick previously used in an LWE-based scheme [BKS19], if z is sufficiently smaller than N, with high probability subtractive shares of z modulo N are *already* valid shares over the integers. Assuming z to be much smaller than N is not very limiting, since we can still have, say, z around $\sqrt{N}$ and achieve both negligible failure probability and exponentially large plaintexts.

HSS Variants. We use the trick described above to get homomorphic secret sharing for branching programs. We subtractively share (digits of) the Paillier decryption key d (where $d \equiv 1 \pmod{N}$ and $d \equiv 0 \pmod{\phi(N)}$) between the two parties. Similarly to [FGJS17], we can use such a sharing of the secret key to go from a Paillier ciphertext to a divisive sharing of the underlying message x of the form $g_1 = g_0 \cdot (1+N)^x$. Once we have that, we can obtain a subtractive sharing of x, as described above. Given subtractive shares of d times some $y \in [N]$, we can similarly get a divisive sharing — and then a subtractive sharing — of xy. Using encryptions of digits of the key d, we can maintain the invariant that we always have subtractive shares of d times our intermediate values available, so we can continue the computation and multiply more encrypted values by our intermediate values.

There are two downsides to our Paillier-based HSS scheme: (1) it requires trusted setup (to distribute shares of the key d), and (2) since we use encryptions of digits of d, we need to assume the *circular security* of Paillier. We can avoid trusted setup by instead using Paillier-ElGamal encryption [CS02, DGS03, BCP03]. When using Paillier-ElGamal, once a modulus N is generated and published, the parties need only do a public-key style setup, where each party independently generates a secret/public key pair, and publishes its public key, following a previous ElGamal-based method [BGI17]. We can additionally avoid assuming circular security by using the Brakerski-Goldwasser scheme [BG10], which is *provably* circular-secure. The downside of using this scheme is much larger ciphertexts.

Pseudorandom Correlation Functions. Our pseudorandom correlation functions use techniques similar to our Paillier-based homomorphic secret sharing construction, with the difference that the Paillier decryption key d is known to one of the parties (whereas before, it was secret shared). Our PCF constructions also crucially rely on the fact that Paillier ciphertexts can be obliviously sampled; any element in $\mathbb{Z}^*_{N^2}$ is in fact a valid ciphertext! In our PCF for the VOLE correlation, one party knows d, the other party knows a value x, and dx is subtractively secret shared between the two. Given a random ciphertext, the party who knows d can decrypt it to learn a, and both parties can recover shares of xa using the trick from our HSS construction.

We get a PCF for OT from similar techniques, but using the Goldwasser-Micali bit-encryption scheme [GM82], which admits a simple distributed discrete log procedure (as also observed in [DGI+19]). We also construct the weaker notion of a PCG for OLE, by essentially generating many instances of the VOLE PCF setup, but compressing them in clever ways using the LPN assumption together with function secret sharing, inspired by previous PCG constructions [BCGI18]. This construction also generalizes in several ways, to give secret-shared degree-2 correlations over $\mathbb{Z}_N$ or $\mathbb{Z}_2$, and to give PCFs when relying on the variable-density LPN assumption [BCG+20a] instead of standard LPN.

Public-key Setup for PCFs. Our PCF for VOLE requires a setup where one party knows d, the other knows x, and both hold subtractive shares of dx. (Our PCF for OT uses a similar setup.) We show that such setup can be instantiated *non-interactively*; each party locally generates a secret/public key pair, and extracts the setup information it needs from its own key pair and the other party's public key.

The PCF setup itself can be seen as an OLE instance for values x and d. So, our public-key setup is based on a novel non-interactive vector-OLE protocol that can be seen as a variant of Bellare-Micali oblivious transfer [BM90] with a CRS. Their original non-interactive oblivious transfer protocol allows two parties to use a (non-reusable) PKI setup to non-interactively obtain an OT. In our Paillier-based variant, instead of only producing OT – and crucially thanks to our distributed discrete log procedure – we show how the parties can obtain a *vector OLE*, where the sender's input is x, the receiver's input are some values $a_1, \ldots, a_n$, and the parties end up with additive sharings of the product $x \cdot a_i$ for all i's. This suffices to generate the keys for our PCF constructions non-interactively.

2 Preliminaries

We work with Blum integers of the form $N = pq$, where p and q are safe primes.[2] We let $(N, p, q) \leftarrow \mathsf{GenPQ}(1^\lambda)$ be a randomized algorithm which, on input the security parameter λ, samples two such random primes p, q of length $\ell = \ell(\lambda)$, and outputs (N, p, q). In some of our constructions, $N = pq$ will be a public modulus generated by a trusted setup algorithm (such that no-one knows the factorization p and q), while in other cases the factorization will be known to one party.

2.1 Assumptions

In this paper, we leverage the following assumptions.

[2] A safe prime p is equal to $2p' + 1$ where p' is also prime. This is not actually required by all our constructions, but for simplicity we use the same group generation algorithm through the paper.

Assumption 1 (Decisional Composite Residuosity (DCR) Assumption)**.** *For* $(N, p, q) \leftarrow \mathsf{GenPQ}(1^\lambda)$*, let* $g_0 \leftarrow \mathbb{Z}^*_{N^2}$*, and* $g_1 = g_0^N \bmod N^2$*. For* $b \leftarrow \{0, 1\}$*, for all PPT algorithms* $\mathcal{A}$*,*

$$\Pr[\mathcal{A}(N, g_b) = b] \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

Assumption 2 (Quadratic Residuosity (QR) Assumption)**.** *For* $(N, p, q) \leftarrow \mathsf{GenPQ}(1^\lambda)$*, let* $g_0 \leftarrow \mathbb{Z}^*_N$*, and* $g_1 = g_0^2 \bmod N$*. For* $b \leftarrow \{0, 1\}$*, for all PPT algorithms* $\mathcal{A}$*,*

$$\Pr[\mathcal{A}(N, g_b) = b] \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

We also leverage a lemma from Brakerski and Goldwasser [BG10] that refers to the *interactive vector game*. We rephrase the lemma here in terms of Paillier groups only. Consider the decomposition $\mathbb{Z}^*_{N^2} = \mathbb{G}_R \times \mathbb{G}_M$, where $\mathbb{G}_R$ is the group of Nth residues modulo N^2 and $\mathbb{G}_M$ is the group of elements of orders that divide N. The DCR assumption (Assumption 1) states that a random element from $\mathbb{G}_R$ is indistinguishable from a random element in $\mathbb{Z}^*_{N^2}$. In the interactive vector game, the challenger samples a bit $b \leftarrow \{0, 1\}$, and $(g_1, \ldots, g_l) \leftarrow \mathbb{G}_R^l$ (for a parameter l). It sends $(g_1, \ldots, g_l)$ to the adversary. The adversary then makes adaptive queries of the form $(a_1, \ldots, a_l) \in \mathbb{G}_M^l$, to which the challenger responds by sampling r from $[N^2]$ and returns $(a_1^b g_1^r, \ldots, a_l^b g_l^r)$. Finally, the adversary returns a guess b' at the value of b.

Lemma 2.1 (Rephrased Lemma B.1 From [BG10]). *Assuming the DCR assumption, for all efficient adversaries* $\mathcal{A}$*, the probability that* $\mathcal{A}$ *guesses* b *correctly in the interactive vector game is at most negligibly greater than half.*

2.2 Encryption

KDM Security. In some of our constructions, we assume that (variants of) the Paillier encryption scheme are *key-dependent message* (KDM) secure. The definition we use is similar to the one given by Brakerski and Goldwasser [BG10], with the differences that we only consider one key pair, and do not consider adaptive adversary queries. (This makes for a weaker definition, and thus a milder assumption.)

Definition 2.2 (KDM Security). *An encryption scheme* $(\mathsf{ES.Gen}, \mathsf{ES.Enc}, \mathsf{ES.Dec})$ *is* KDM secure *over the set of programs* F *if for all security parameters* $\lambda \in \mathbb{N}$*, for all polynomial sets of fixed output length programs* $f_1, \ldots, f_\rho \in F$ *and for all PPT adversaries* $\mathcal{A}$*,*

$$\left| \Pr\left[\mathcal{A}(\mathsf{pk}, \mathsf{ct}_{\beta,1}, \ldots, \mathsf{ct}_{\beta,\rho}) = \beta \;\middle|\; \begin{array}{l} (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{ES.Gen}(1^\lambda), \\ x_{0,i} \leftarrow f_i(\mathsf{sk}) \textit{ for } i \in [\rho], \\ x_{1,i} \leftarrow 0^{|x_{0,i}|} \textit{ for } i \in [\rho], \\ \mathsf{ct}_{b,i} \leftarrow \mathsf{ES.Enc}(\mathsf{pk}, x_b) \textit{ for } b \in \{0,1\}, i \in [\rho], \\ \beta \leftarrow \{0,1\} \end{array} \right] - \frac{1}{2} \right| \leq \mathsf{negl}(\lambda).$$

Paillier Encryption. While there are many known variants of the Paillier cryptosystem [Pai99], we use the variant where the decryption key is an integer d such that raising any ciphertext to the power d gives $(1+N)^m \pmod{N^2}$, where m is the message and N the public modulus. Since it is easy to compute discrete logarithms with base $1+N$ in $\mathbb{Z}_{N^2}^*$, this gives an efficient decryption procedure. We describe the Paillier cryptosystem below; its security is based on the DCR assumption (Assumption 1).

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

1. Sample $(N, p, q) \leftarrow \mathsf{GenPQ}(1^\lambda)$.
2. Compute $d \in \mathbb{Z}$ such that $d \equiv 0 \pmod{\phi(N)}$ and $d \equiv 1 \pmod{N}$.
3. Output $\mathsf{pk} = N$, $\mathsf{sk} = d$.

$\mathsf{Paillier.Enc}(\mathsf{pk}, x)$:

1. Sample a random $r \leftarrow [N^2]$.
2. Output $\mathsf{ct} = r^N(1+N)^x \bmod N^2$.

Observe that, since $(1+N)^x \equiv 1 + xN \pmod{N^2}$, $(1+N)$ has order N in $\mathbb{Z}_{N^2}^*$. Additionally, observe that since the order of r in $\mathbb{Z}_{N^2}^*$ must divide $N\phi(N)$,

$$\begin{aligned}
\mathsf{ct}^d \pmod{N^2} &\equiv r^{Nd}(1+N)^{dx} && \pmod{N^2} \\
&\equiv r^{Nd \pmod{N\phi(N)}}(1+N)^{dx \pmod{N}} && \pmod{N^2} \\
&\equiv (1+N)^x && \pmod{N^2} \\
&\equiv 1 + xN.
\end{aligned}$$

$\mathsf{Paillier.Dec}(\mathsf{sk}, \mathsf{ct})$:

1. Output $x = \frac{(\mathsf{ct}^d \bmod N^2) - 1}{N}$.

We will also use the following fact, namely, that the encryption function is a bijection. In particular, this implies that a randomly chosen element of $\mathbb{Z}_{N^2}$ defines a valid ciphertext with overwhelming probability.

Proposition 2.3 ([Pai99]). *The following map is a bijection:*

$$\begin{aligned}
\mathbb{Z}_N \times \mathbb{Z}_N^* &\to \mathbb{Z}_{N^2}^* \\
(x, r) &\mapsto r^N(1+N)^x
\end{aligned}$$

In our homomorphic secret sharing constructions (Section 4), we use two other flavors of Paillier encryption: a Paillier-ElGamal hybrid, and the KDM-secure scheme due to Brakerski and Goldwasser [BG10]. In Section 5, we also use the Goldwasser–Micali cryptosystem [GM82].

2.3 Secret Sharing

We work with subtractive secret sharing. We let $\langle x \rangle_0^{(m)}, \langle x \rangle_1^{(m)}$ denote a subtractive sharing of x modulo m, such that $\langle x \rangle_1^{(m)} - \langle x \rangle_0^{(m)} \equiv x \pmod{m}$. If one share is chosen uniformly at random from $[m]$ (and the other is chosen to satisfy the equation above), each share alone perfectly hides x, while the two together allow the reconstruction of x.

Similarly, we let $\langle x\rangle_0^{(\mathbb{Z})}, \langle x\rangle_1^{(\mathbb{Z})}$ denote a subtractive sharing of x over the integers, such that $\langle x\rangle_1^{(\mathbb{Z})} - \langle x\rangle_0^{(\mathbb{Z})} = x$. For $x \in \{0, \ldots, m-1\}$, in order for each share alone to statistically hide x, $\langle x\rangle_1^{(\mathbb{Z})}$ can be chosen uniformly at random from the range $\{0, \ldots, m2^\kappa - 1\}$, where κ is the statistical security parameter. If $\langle x\rangle_0^{(\mathbb{Z})}$ is then defined as $\langle x\rangle_1^{(\mathbb{Z})} - x$, then it is within statistical distance $2^{-\kappa}$ of the uniform distribution.

3 Share Conversion for Paillier Encryption

Suppose two parties hold respective values g_0 and g_1 in $\mathbb{Z}_{N^2}^*$, such that $g_1 \equiv g_0(1+N)^x \pmod{N^2}$ for some $x \in \mathbb{Z}_N$. The parties wish to locally convert these *multiplicative* (or, rather, *divisive*) shares into *subtractive* shares of x.

We can view g_0 and g_1 as elements of the coset

$$X_{g_0} := \left\{g_0, g_0(1+N), g_0(1+N)^2, \cdots, g_0(1+N)^{N-1}\right\}.$$

If both parties can *agree upon* a distinct element of this set, say h, without communicating, then they can each calculate the distance (in terms of powers of $1+N$) between g_i and h to obtain a subtractive share of x. In particular, if they obtain $h = g_0(1+N)^z$ for some z, then P_1 can compute the discrete logarithm of $g_1/h = (1+N)^{x-z}$ and output $z_1 := x - z$, while P_0 uses $g_0/h = (1+N)^{-z}$ to get $z_0 := -z$, giving $z_1 - z_0 \equiv x \pmod{N}$.

To agree upon such a representative h, we have the parties compute the smallest element from X_{g_0}, defined by viewing elements of $\mathbb{Z}_{N^2}^*$ as integers in $\{0, \ldots, N^2-1\}$. Surprisingly, this can be done by simply computing $h = g_i \bmod N$. Since $(1+N)^x \equiv 1 \pmod{N}$, it is clear that this gives the same h for both g_0 and g_1. It remains to show that h lies in the same coset.

Proposition 3.1. *Let $g \in \mathbb{Z}_{N^2}^*$, $h = g \bmod N$ and $h' = \lfloor g/N \rfloor$. Then h can be written as $g(1+N)^{-z}$, where $z = h'h^{-1} \bmod N$.*

Proof. Suppose that we can write $h = g(1+N)^{-z} \bmod N^2$, for some $z \in \mathbb{Z}$. Since $g = h + h'N$, this is equivalent to

$$\begin{aligned} h &= (h + h'N)\cdot(1 - zN) \mod N^2 \\ &= h + (h' - zh)N \mod N^2 \end{aligned}$$

The above is satisfied if and only if $h'N \equiv zhN \pmod{N^2}$, or equivalently, $z \equiv h'h^{-1} \pmod{N}$. □

This gives us a direct way to solve the distributed discrete log problem, which we present in Algorithm 3.2. Instead of computing g_i/h and then finding the discrete logarithm with respect to $1+N$, we can simply compute z as in Proposition 3.1.

Algorithm 3.2: $\mathsf{DDLog}_N(g)$

```
1. Write $g = h + h'N$, where $h, h' < N$, using the division algorithm.
2. Output $z = h'h^{-1} \mod N$.
```

Lemma 3.3. *Let $g_0, g_1 \in \mathbb{Z}_{N^2}^*$ such that $g_1 = g_0(1+N)^x \bmod N^2$. If $z_b = \mathsf{DDLog}_N(g_b)$, then $z_1 - z_0 \equiv x \pmod N$.*

Proof. First, observe that since each g_i is in $\mathbb{Z}_{N^2}^*$, it must have an inverse modulo N, so DDLog will not fail.

From Proposition 3.1, each z_i satisfies $h \equiv g_i(1+N)^{-z_i} \pmod{N^2}$, where $h = g_0 \bmod N = g_1 \bmod N$. This gives

$$\begin{aligned} g_1(1+N)^{-z_1} &\equiv g_0(1+N)^{-z_0} && \pmod{N^2} \\ \Leftrightarrow (1+N)^{x-z_1} &\equiv (1+N)^{-z_0} && \pmod{N^2} \\ \Leftrightarrow x &\equiv z_1 - z_0 && \pmod{N}. \end{aligned}$$

□

Remark 3.4. We can alternatively interpret the share conversion procedure by viewing each input g_i as a Paillier ciphertext $g_i = (1+N)^{x_i} r^N$ for some (unknown) message x_i, and the *same randomness* r. Under this condition, share conversion allows each party to locally obtain a subtractive share of $x = x_1 - x_0$. Note that this does not violate the security of Paillier, because we were given two ciphertexts with the same randomness.

3.1 Using a Secret Shared Decryption Key

Getting g_0, g_1 such that $g_1 = g_0(1+N)^x$ given a Paillier encryption $g = (1+N)^x r^N$ of x can be done using subtractive shares (over the integers) $\langle d\rangle_0^{(\mathbb{Z})}, \langle d\rangle_1^{(\mathbb{Z})}$ of the Paillier decryption key d (where $\langle d\rangle_1^{(\mathbb{Z})} - \langle d\rangle_0^{(\mathbb{Z})} = d$, $d \equiv 1 \pmod N$, and $d \equiv 0 \pmod{\phi(N)}$).

Using our ciphertext $g = (1+N)^x r^N$, we compute

$$g_0 = g^{\langle d\rangle_0^{(\mathbb{Z})}} = (1+N)^{x\langle d\rangle_0^{(\mathbb{Z})}} (r^{\langle d\rangle_0^{(\mathbb{Z})}})^N \bmod N^2,$$

and

$$g_1 = g^{\langle d\rangle_1^{(\mathbb{Z})}} = (1+N)^{x\langle d\rangle_1^{(\mathbb{Z})}} (r^{\langle d\rangle_1^{(\mathbb{Z})}})^N \bmod N^2.$$

Since $d \equiv 0 \pmod{\phi(N)}$, it follows that $d = \langle d\rangle_1^{(\mathbb{Z})} - \langle d\rangle_0^{(\mathbb{Z})} \Rightarrow \langle d\rangle_0^{(\mathbb{Z})} \equiv \langle d\rangle_1^{(\mathbb{Z})} \pmod{\phi(N)}$ and therefore

$$r^{\langle d\rangle_1^{(\mathbb{Z})} N} \equiv r^{\langle d\rangle_0^{(\mathbb{Z})} N} \pmod{N^2}.$$

Then, as desired,

$$\begin{aligned}\frac{g_1}{g_0} &\equiv (1+N)^{x(\langle d\rangle_1^{(\mathbb{Z})} - \langle d\rangle_0^{(\mathbb{Z})})} \pmod{N^2} \\ &\equiv (1+N)^x \pmod{N^2}.\end{aligned}$$

Remark 3.5. If, instead of subtractive shares of d, we have shares of yd (that is, $\langle yd\rangle_0^{(\mathbb{Z})}, \langle yd\rangle_1^{(\mathbb{Z})}$ such that $\langle yd\rangle_1^{(\mathbb{Z})} - \langle yd\rangle_0^{(\mathbb{Z})} = yd$), we can use these as described above to get g_0, g_1 such that $g_1 \equiv g_0(1+N)^{xy} \pmod{N^2}$.

3.2 Getting Shares Over Integers

The previous sections describe how to use $g_1 = g_0(1+N)^x \bmod N^2$ to get subtractive shares of x over $\mathbb{Z}_N$. However, we often want subtractive shares of x over the integers. This can be done as long as x is sufficiently smaller than N.

Observe that, if $z_1 - z_0 \equiv x \pmod{N}$, then $z_1 - z_0 = x$ as long as $z_1 - x \geq 0$. There are only x values of z_1 such that this isn't true; so, for $x < N/2^\kappa$ and uniform choice of $z_1 \in \mathbb{Z}_N$, $z_1 - z_0 = x$ over $\mathbb{Z}$, except with probability $\leq 2^{-\kappa}$.

4 Homomorphic Secret Sharing

4.1 Definitions

We base our definitions of homomorphic secret sharing (HSS) on those given by Boyle *et al.* [BKS19].

Definition 4.1 (Homomorphic Secret Sharing). *A (2-party, public-key)* Homomorphic Secret Sharing (HSS) *scheme for a class of programs $\mathcal{P}$ over a ring R with input space $\mathcal{I} \subseteq R$ consists of PPT algorithms* $(\mathsf{HSS.Setup}, \mathsf{HSS.Input}, \mathsf{HSS.Eval})$ *with the following syntax:*

- $\mathsf{HSS.Setup}(1^\lambda) \to (\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1))$*: Given a security parameter 1^λ, the setup algorithm outputs a public key* pk *and a pair of evaluation keys* $(\mathsf{ek}_0, \mathsf{ek}_1)$*.*
- $\mathsf{HSS.Input}(\mathsf{pk}, x) \to (\mathsf{I}_0, \mathsf{I}_1)$*: Given public key* pk *and private input value $x \in \mathcal{I}$, the input algorithm outputs input information* $(\mathsf{I}_0, \mathsf{I}_1)$*.*
- $\mathsf{HSS.Eval}(\sigma, \mathsf{ek}_\sigma, (\mathsf{I}_\sigma^{(1)}, \ldots, \mathsf{I}_\sigma^{(\rho)}), P) \to y_\sigma$*: On input a party index $\sigma \in \{0,1\}$, evaluation key ek_σ, vector of ρ input values and a program $P \in \mathcal{P}$ with ρ input values, the homomorphic evaluation algorithm outputs $y_\sigma \in R$, which is party σ's share of an output $y \in R$.*

Note that, in the constructions we consider, we have $\mathsf{I}_0 = \mathsf{I}_1$*. We say that* $(\mathsf{HSS.Setup}, \mathsf{HSS.Input}, \mathsf{HSS.Eval})$ *is a* homomorphic secret sharing scheme *for the class of programs $\mathcal{P}$ if the following conditions hold:*

- ***Correctness.*** *For all security parameters $\lambda \in \mathbb{N}$, for all programs $P \in \mathcal{P}$, for all $x^{(1)}, \ldots, x^{(\rho)} \in \mathcal{I}$ (where $\mathcal{I}$ is the input space of P), for* $(\mathsf{pk}, \mathsf{ek}_0, \mathsf{ek}_1) \leftarrow \mathsf{HSS.Setup}(1^\lambda)$ *and for* $(\mathsf{I}_0^{(i)}, \mathsf{I}_1^{(i)}) \leftarrow \mathsf{HSS.Input}(\mathsf{pk}, x^{(i)})$*, we have*

```
$\underline{\mathsf{Exp}^{\mathsf{HSS,sec}}_{\mathcal{A},\sigma,b}(\lambda):}$
$(x_0, x_1, \mathsf{state}) \leftarrow \mathcal{A}(1^\lambda)$
$(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{HSS.Setup}(1^\lambda)$
$(\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{HSS.Input}(\mathsf{pk}, x_b)$
$b' \leftarrow \mathcal{A}(\mathsf{state}, \mathsf{pk}, \mathsf{ek}_\sigma, \mathsf{I}_\sigma)$
**return** $b'$
```

Fig. 1. Security of HSS.

$$\Pr\left[y_0 + y_1 = P(x^{(1)}, \ldots, x^{(\rho)})\right] \geq 1 - \mathsf{negl}(\lambda),$$

where

$$y_\sigma \leftarrow \mathsf{HSS.Eval}(\sigma, \mathsf{ek}_\sigma, (\mathsf{I}_\sigma^{(1)}, \ldots, \mathsf{I}_\sigma^{(\rho)}), P)$$

for $\sigma \in \{0,1\}$ *where the probability is taken over the random coins of* HSS.Setup*,* HSS.Input *and* HSS.Eval*.*

- ***Security.*** *For each* $\sigma \in \{0,1\}$ *and non-uniform adversary* $\mathcal{A}$ *(of size polynomial in the security parameter* λ*), it holds that*

$$\left|\Pr[\mathsf{Exp}^{\mathsf{HSS,sec}}_{\mathcal{A},\sigma,0}(\lambda) = 1] - \Pr[\mathsf{Exp}^{\mathsf{HSS,sec}}_{\mathcal{A},\sigma,1}(\lambda) = 1]\right| \leq \varepsilon(\lambda)$$

for all sufficiently large λ*, where* $\mathsf{Exp}^{\mathsf{HSS,sec}}_{\mathcal{A},\sigma,b}(\lambda)$ *for* $b \in \{0,1\}$ *is as defined in Fig. 1.*

Restricted Multiplication Straight-line Programs. Our HSS schemes support homomorphic evaluation for a class of programs called *Restricted Multiplication Straight-line (RMS)* programs [Cle91, BGI16a]. An RMS program is an arithmetic circuit, with the restriction that every multiplication must be between an input value and an intermediate value of the computation, called a *memory value.* This class of programs captures polynomial-size branching programs, which includes arbitrary logspace computations and NC1 circuits.

Definition 4.2 (RMS programs). *An RMS program consists of a magnitude bound* B_{msg} *and a sequence of instructions of the four types described below. The inputs to the program are initially provided as a set of* input values I_x*, for each input* $x \in \mathbb{Z}$*. We consider the class of programs where the absolute value of all* memory values *during the computation is bounded above by* B_{msg}*.*

- $\mathsf{ConvertInput}(\mathsf{I}_x) \rightarrow \mathsf{M}_x$*: Load an input* x *into memory.*
- $\mathsf{Add}(\mathsf{M}_x, \mathsf{M}_y) \rightarrow \mathsf{M}_z$*: Add two memory values, obtaining* $z = x + y$*.*
- $\mathsf{Add}(\mathsf{I}_x, \mathsf{I}_y) \rightarrow \mathsf{I}_z$*: Add two input values, obtaining* $z = x + y$*.*
- $\mathsf{Mul}(\mathsf{I}_x, \mathsf{M}_y) \rightarrow \mathsf{M}_z$*: Multiply a memory value by an input, obtaining* $z = x \cdot y$*.*
- $\mathsf{Output}(\mathsf{M}_x, n_{\mathsf{out}}) \rightarrow x \bmod n_{\mathsf{out}}$*: Output a memory value, reduced modulo* n_{out} *(for some* $n_{\mathsf{out}} \leq B_{\mathsf{msg}}$*).*

We additionally assume that each instruction is implicitly assigned a unique identifier $\mathsf{id} \in \{0,1\}^*$.

If at any step of execution the size of a memory value exceeds the bound B_{msg} *or becomes negative (i.e.* $z > B_{\mathsf{msg}}$ *or* $z < 0$*), the output of the program on the corresponding input is undefined. Otherwise, the output is the sequence of* Output *values. Note that we consider addition of input values merely for the purpose of efficiency.*

Remark 4.3. It may seem restrictive that we do not support negative values; however, our construction can easily be modified to support negative values by viewing integers modulo N as ranging from $-\frac{N}{2}$ to $\frac{N}{2}$ instead of from 0 to $N-1$, as done by Boyle *et al.* [BKS19]. (We would then use multiplicative and additive sharings instead of divisive and subtractive ones.) We choose to work with non-negative values for the sake of notational simplicity.

4.2 HSS from Paillier

We follow the blueprint from Fazio *et al.* [FGJS17], based on Boyle *et al.* [BGI16a] to build an HSS scheme for RMS programs. Our scheme requires that an encryption of the secret decryption key be available. However, for correctness, our scheme also requires that all ciphertexts encrypt values much smaller than N; so, we are forced to decompose our secret key into digits before encrypting it. We use B_{sk} to refer to the base used for this decomposition, or, in other words, as an upper bound on the size of each digit. B_{sk} affects the efficiency of our scheme, since it will take $\ell = \log_{B_{\mathsf{sk}}}(N^2)$ ciphertexts to contain our secret key. B_{sk} is also related to the bound B_{msg} on our message space, since we require that all our input and memory values *times a digit of the secret key* be at least 2^κ times smaller than N: we get $B_{\mathsf{msg}} = \frac{N}{B_{\mathsf{sk}}2^\kappa}$.

If we want $B_{\mathsf{msg}} = 2^\kappa$, then we get $B_{\mathsf{sk}} = \frac{N}{2^{2\kappa}}$. As long as N is at least 3κ bits long, B_{sk} will be at least κ bits long; so, we will need around 6 ciphertexts to contain our secret key.

As in RMS programs, we consider *input values* and *memory values*. Input values, denoted I, are the inputs to the computation, consisting of Paillier encryptions. Memory values, denoted M, are subtractively secret-shared intermediate values. More concretely, let $d^{(0)}, \ldots, d^{(\ell-1)}$ denote the digits of d (modulo some base B_{sk}), where d is the Paillier decryption key.

- An *input value* I_x consists of X, which is a Paillier encryption of x, and $X^{(0)}, \ldots, X^{(\ell-1)}$, which are Paillier encryptions of $d^{(0)}x, \ldots, d^{(\ell-1)}x$.
- A *memory value* $\mathsf{M}_x = (\mathsf{M}_{x,0}, \mathsf{M}_{x,1})$ consists of subtractive sharings of x and $d^{(0)}x, \ldots, d^{(\ell-1)}x$ over the integers. That is, party σ's memory value for x is $\mathsf{M}_{x,\sigma} = (\langle x\rangle_\sigma^{(\mathbb{Z})}, \langle xd^{(0)}\rangle_\sigma^{(\mathbb{Z})}, \ldots, \langle xd^{(\ell-1)}\rangle_\sigma^{(\mathbb{Z})})$.

We describe our HSS scheme for RMS programs in Construction 4.5.

Theorem 4.4. *Construction 4.5 is a secure HSS scheme assuming the KDM security of the Paillier encryption scheme, and assuming that* $F^{(N)}$ *is a secure PRF.*

Proof. We prove the correctness and security of our construction separately below.

Correctness. For computations where all input, intermediate and output values are much smaller than N, correctness largely follows by inspection. Correctness could only be violated if, during multiplication, when taking $\langle zd^{(i)}\rangle_\sigma^{(\mathbb{Z})} = \langle zd^{(i)}\rangle_\sigma^{(N)}$, we get $\langle zd^{(i)}\rangle_1^{(N)} - \langle zd^{(i)}\rangle_0^{(N)} \neq zd^{(i)}$. If $zd^{(i)} \ll N$ and $\langle zd^{(i)}\rangle_1^{(N)}$ is uniformly distributed in $\mathbb{Z}_N$, this happens with negligible probability, as described in Section 3.2. Recall that $\langle zd^{(i)}\rangle_1^{(N)}$ is computed as $\langle zd^{(i)}\rangle_1^{(N)} = \mathsf{DDLog}_N((X^{(i)})^{\langle yd\rangle_1^{(\mathbb{Z})}}) + F^{(N)}_{\mathsf{k}_{\mathsf{prf}}}(\mathsf{id}, i) \pmod N$. Since $F^{(N)}$ outputs values in $[N]$, by the security of $F^{(N)}$, $\langle zd^{(i)}\rangle_1^{(N)}$ will be indistinguishable from random in $\mathbb{Z}_N$.

Security. We show security in a series of hybrids (described below) played with a PPT adversary $\mathcal{A}$. Hybrid 0 gives the adversary inputs produced in accordance with the HSS security definition (Definition 4.1). We then alter the way the adversary's inputs are produced in three steps; each change should not affect the adversary's probability of successfully guessing b by more than a negligible amount. In the last hybrid, the adversary's probability of guessing b is exactly $\frac{1}{2}$, since its view in that hybrid is entirely independent of b. It follows that the adversary's probability of guessing b in the original hybrid can be only negligibly greater than $\frac{1}{2}$.

Hybrid 0**:** We begin by producing inputs for $\mathcal{A}$ in accordance with the security definition.

Hybrid 1**:** Now, during Setup, choose each element of ek_σ from $[2^\kappa B_{\mathsf{sk}}]$ uniformly at random. These are statistically close in distribution to a correctly generated ek_σ. If $\mathcal{A}$ guesses b with probability significantly different than in hybrid 0, then $\mathcal{A}$ can be leveraged to distinguish between two statistically close distributions.

Hybrid 2**:** Now, during Setup, instead of producing encryptions $D^{(0)}, \ldots, D^{(\ell-1)}$ of digits of the secret key honestly, we set them all to be independent encryptions of 0. If $\mathcal{A}$ guesses b with probability significantly different than in hybrid 1, then $\mathcal{A}$ can be run by a KDM adversary $\mathcal{A}_{\mathsf{KDM}}$ to break KDM security. (Note that at this point, $\mathcal{A}_{\mathsf{KDM}}$ does not need to use the secret key to produce any of the inputs for $\mathcal{A}$.) $\mathcal{A}_{\mathsf{KDM}}$ then outputs the guess b' produced by $\mathcal{A}$.

Hybrid 3**:** Now, during Input, instead of encrypting x_b, we always encrypt x_0. If $\mathcal{A}$ guesses b with probability significantly different than in hybrid 2, then $\mathcal{A}$ can be run by an algorithm $\mathcal{A}_{\mathsf{SS}}$ to break the semantic security of Paillier encryption.

Note that here, $\mathcal{A}$'s probability of guessing b is exactly $\frac{1}{2}$, since its view is completely independent of b.

□

Construction 4.5: Construction $\mathsf{HSS}_{\mathsf{Paillier}}$

$\mathsf{Setup}(1^\lambda)$**:** Set up the scheme.

1. Sample $(\mathsf{pk}_{\mathsf{Paillier}} = N, \mathsf{sk} = d) \leftarrow \mathsf{Paillier.Gen}(1^\lambda)$. Let $d^{(0)}, \ldots, d^{(\ell-1)}$ denote the digits of d base B_{sk}.
2. Subtractively secret share the digits of d as $\langle d^{(i)} \rangle_0^{(\mathbb{Z})}, \langle d^{(i)} \rangle_1^{(\mathbb{Z})}$ such that $\langle d^{(i)} \rangle_1^{(\mathbb{Z})} - \langle d^{(i)} \rangle_0^{(\mathbb{Z})} = d^{(i)}$. Each $\langle \cdot \rangle_1^{(\mathbb{Z})}$ is drawn uniformly at random from $[2^\kappa B_{\mathsf{sk}}]$; $\langle \cdot \rangle_0^{(\mathbb{Z})}$ is selected to complete the subtractive sharing.
3. Sample a key $\mathsf{k}_{\mathsf{prf}}$ for the prf $F^{(2^\kappa)}$ which outputs values in $[2^\kappa]$.
4. For $\sigma \in \{0,1\}$, let $\mathsf{ek}_\sigma = (\mathsf{k}_{\mathsf{prf}}, \langle d^{(0)} \rangle_\sigma^{(\mathbb{Z})}, \ldots, \langle d^{(\ell-1)} \rangle_\sigma^{(\mathbb{Z})})$.
5. Encrypt the digits of d as $D^{(0)} \leftarrow \mathsf{Paillier.Enc}(\mathsf{pk}_{\mathsf{Paillier}}, d^{(0)}), \ldots, D^{(\ell-1)} \leftarrow \mathsf{Paillier.Enc}(\mathsf{pk}_{\mathsf{Paillier}}, d^{(\ell-1)})$.
6. Let $\mathsf{pk} = (\mathsf{pk}_{\mathsf{Paillier}}, D^{(0)}, \ldots, D^{(\ell-1)})$.
7. Output $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1))$.

$\mathsf{Input}(\mathsf{pk}, x)$**:** Generate an input value for x.

1. Generate a Paillier ciphertext $X \leftarrow \mathsf{Paillier.Enc}(\mathsf{pk}_{\mathsf{Paillier}}, x)$.
2. For $i \in [0, \ldots, \ell - 1]$, generate an encryption $X^{(i)}$ of $d^{(i)}x$ by homomorphically multiplying $D^{(i)}$ by x, and then re-randomizing. Concretely using Paillier, $X^{(i)} = r_i^N (D^{(i)})^x$ for a randomly sampled $r_i \leftarrow \mathbb{Z}_{N^2}^*$.
3. Let $\mathsf{I} = (X, X^{(0)}, \ldots, X^{(\ell-1)})$.
4. Output $(\mathsf{I}_0 = \mathsf{I}, \mathsf{I}_1 = \mathsf{I})$.

$\mathsf{ConvertInput}(\sigma, \mathsf{ek}_\sigma, \mathsf{I} = (X, X^{(0)}, \ldots, X^{(\ell-1)}))$**:** Convert an input to a memory value. First, we take a canonical secret sharing of 1 as $\langle 1 \rangle_1^{(\mathbb{Z})} = F_{\mathsf{k}_{\mathsf{prf}}}^{(2^\kappa)}(1) + 1 \bmod N$, and $\langle 1 \rangle_0^{(\mathbb{Z})} = F_{\mathsf{k}_{\mathsf{prf}}}^{(2^\kappa)}(1)$. Then we create a memory value for 1 as $\mathsf{M}_{1,\sigma} = (\langle 1 \rangle_\sigma^{(\mathbb{Z})}, \langle d^{(0)} \rangle_\sigma^{(\mathbb{Z})}, \ldots, \langle d^{(\ell-1)} \rangle_\sigma^{(\mathbb{Z})})$ for $\sigma \in \{0,1\}$, and evaluate $\mathsf{Mul}(\sigma, \mathsf{ek}_\sigma, \mathsf{I}_x, \mathsf{M}_{1,\sigma})$.[a]

$\mathsf{Add}(\sigma, \mathsf{ek}_\sigma, \mathsf{M}_{x,\sigma}, \mathsf{M}_{y,\sigma})$**:** Add two memory values.

1. Parse $\mathsf{M}_{x,\sigma} = (\langle x \rangle_\sigma^{(\mathbb{Z})}, \langle xd^{(0)} \rangle_\sigma^{(\mathbb{Z})}, \ldots, \langle xd^{(\ell-1)} \rangle_\sigma^{(\mathbb{Z})})$, and $\mathsf{M}_{y,\sigma} = (\langle y \rangle_\sigma^{(\mathbb{Z})}, \langle yd^{(0)} \rangle_\sigma^{(\mathbb{Z})}, \ldots, \langle yd^{(\ell-1)} \rangle_\sigma^{(\mathbb{Z})})$.
2. Let $\langle z \rangle_\sigma^{(\mathbb{Z})} = \langle x \rangle_\sigma^{(\mathbb{Z})} + \langle y \rangle_\sigma^{(\mathbb{Z})}$, and $\langle zd^{(i)} \rangle_\sigma^{(\mathbb{Z})} = \langle xd^{(i)} \rangle_\sigma^{(\mathbb{Z})} + \langle yd^{(i)} \rangle_\sigma^{(\mathbb{Z})}$ for $i \in [0, \ldots, \ell - 1]$.
3. Output $\mathsf{M}_{z,\sigma} = (\langle z \rangle_\sigma^{(\mathbb{Z})}, \langle zd^{(0)} \rangle_\sigma^{(\mathbb{Z})}, \ldots, \langle zd^{(\ell-1)} \rangle_\sigma^{(\mathbb{Z})})$.

$\mathsf{Add}(\mathsf{pk}, \mathsf{I}_x = (X, X^{(0)}, \ldots, X^{(\ell-1)}), \mathsf{I}_y = (Y, Y^{(0)}, \ldots, Y^{(\ell-1)}))$**:** Add two input values by homomorphically evaluating addition on the ciphertexts to get $\mathsf{I}_z = (Z, Z^{(0)}, \ldots, Z^{(\ell-1)})$. Concretely using Paillier, $Z = XY \bmod N^2$, and $Z^{(i)} = X^{(i)} Y^{(i)} \bmod N^2$. Output I_z.

$\mathsf{Mul}(\sigma, \mathsf{ek}_\sigma, \mathsf{I}_x, \mathsf{M}_{y,\sigma}))$**:** Multiply an input value and a memory value. We let id be the index of this multiplication; all such indices are assumed to be unique.

1. Parse $\mathsf{I}_x = (X, X^{(0)}, \ldots, X^{(\ell-1)})$.
2. Parse $\mathsf{M}_{y,\sigma} = (\langle y \rangle_\sigma^{(\mathbb{Z})}, \langle yd^{(0)} \rangle_\sigma^{(\mathbb{Z})}, \ldots, \langle yd^{(\ell-1)} \rangle_\sigma^{(\mathbb{Z})})$.
3. Let $\langle yd \rangle_\sigma^{(\mathbb{Z})} = \sum_{i=0}^{\ell-1} {B_{\mathsf{sk}}}^i \langle yd^{(i)} \rangle_\sigma^{(\mathbb{Z})}$ be a subtractive share of yd over the integers.
4. Let
$$\langle z \rangle_\sigma^{(N)} = \mathsf{DDLog}_N((X)^{\langle yd \rangle_\sigma^{(\mathbb{Z})}}) + F_{\mathsf{k}_{\mathsf{prf}}}^{(N)}(\mathsf{id}) \pmod N.$$
This yields a subtractive sharing of $z = xy \pmod N$. Since $z \ll N$, we can take this to be a share of z over the integers; that is,
$$\langle z \rangle_\sigma^{(\mathbb{Z})} = \langle z \rangle_\sigma^{(N)}.$$

5. Similarly, let

$$\langle zd^{(i)}\rangle_\sigma^{(N)} = \mathsf{DDLog}_N((X^{(i)})^{\langle yd\rangle_\sigma^{(\mathbb{Z})}}) + F_{\mathsf{k_{prf}}}^{(N)}(\mathsf{id}, i),$$

and

$$\langle zd^{(i)}\rangle_\sigma^{(\mathbb{Z})} = \langle zd^{(i)}\rangle_\sigma^{(N)}.$$

6. Output $\mathsf{M}_{z,\sigma} = (\langle z\rangle_\sigma^{(\mathbb{Z})}, \langle zd^{(0)}\rangle_\sigma^{(\mathbb{Z})}, \ldots, \langle zd^{(\ell-1)}\rangle_\sigma^{(\mathbb{Z})})$.

$\mathsf{Output}(\sigma, \mathsf{ek}_\sigma, \mathsf{M}_{x,\sigma} = (\langle x\rangle_\sigma^{(\mathbb{Z})}, \langle xd^{(0)}\rangle_\sigma^{(\mathbb{Z})}, \ldots, \langle xd^{(\ell-1)}\rangle_\sigma^{(\mathbb{Z})}), n_{\mathsf{out}})$:

Output $\langle x\rangle_\sigma^{(\mathbb{Z})}$ mod n_{out}.

[a] Note that in our HSS construction based on Paillier, we do not actually use $\langle 1\rangle_\sigma^{(\mathbb{Z})}$ in the multiplication; it is only necessary for Output. However, in HSS constructions based on PaillierEG and BG described in Appendix B, this will be needed.)

4.3 HSS Variants

The HSS construction in the previous section has two drawbacks: (1) it requires a local trusted setup for each pair of parties, and (2) its security relies on the assumption that Paillier is KDM secure. We address both these issues by giving two alternative HSS constructions. In the first one we replace Paillier encryption with the Paillier-ElGamal encryption scheme [CS02, DGS03, BCP03], which is essentially ElGamal over the group $\mathbb{Z}_{N^2}^*$. In this variant multiple users can share the same modulus N, and the decryption key is a random exponent d (as in ElGamal). This has the advantage of only requiring a public-key style setup, where each party publishes a public key, and each can then non-interactively derive their shared public key and their own evaluation key. Note that the trusted setup now only contains the modulus N, and can be used by any number of parties. In the last construction we replace Paillier encryption with the provably KDM secure encryption scheme of Brakerski and Goldwasser [BG10]. This has the unexpected advantage that generating encryptions of the digits of the secret key can trivially be done having access to the public key only. While both alternative constructions follow the same blueprint as the one from "regular" Paillier, several details need to be taken care of. The details of the constructions are deferred to Appendix B.

5 Pseudorandom Correlation Functions

In this section, we present our constructions of pseudorandom correlation functions (PCFs). We first recap the syntax and definitions of a PCF in Section 5.1. Then, in Section 5.2, we give our PCF for the vector oblivious linear evaluation (VOLE) correlation, based on the DCR assumption, and in Section 5.3, our PCF for the oblivious transfer (OT) correlation based on quadratic residuosity.

Our public-key variants of these PCFs are deferred until Section 6. Finally, in Section 5.4, we also construct the weaker notion of a pseudorandom correlation generator (PCG) for the oblivious linear evaluation (OLE) correlation, based on a combination of the DCR and learning parity with noise assumptions.

5.1 Definitions

To formalize our constructions for VOLE and OT, we use the concept of a *pseudorandom correlation function* (PCF) by Boyle *et al.* [BCG+20a]. Informally, a pseudorandom correlation function enables two parties to sample an arbitrary amount of correlated randomness, given a one-time setup that outputs a pair of short, correlated keys. This extends the previous notion of a pseudorandom correlation generator [BCG+19], analogously to how a PRF extends a PRG, where in the latter, the outputs are typically of bounded length and/or must be computed all at once.

One example of desirable correlated randomness is an instance of random oblivious transfer (OT), where one party obtains (s_0, s_1) uniform over $\{0,1\}^2$, and the other obtains (b, s_b) for b uniform over $\{0,1\}$. Another example is vector oblivious linear evaluation (VOLE) over a ring R, where the parties obtain respective outputs $(u,v) \in R^2$ and $(x,w) \in R^2$, where u,v are random, $w = ux+v$, and x is sampled at random, but fixed for all samples from the correlation.

We model a target correlation as a probabilistic algorithm $\mathcal{Y}$, which produces a pair of outputs (y_0, y_1) for the two parties. To define security, we additionally require the correlation to be *reverse-sampleable*, meaning that given an output y_σ, there is an efficient algorithm which produces a $y_{1-\sigma}$ from the distribution of $\mathcal{Y}$ conditioned on y_σ. Note that in the case of VOLE, due to the fixed x, we also use a master secret key msk which parametrizes the algorithm $\mathcal{Y}$. Such a correlation with a master secret key is called a *correlation with setup*, which we focus on below.

Definition 5.1 (Reverse-sampleable correlation with setup). *Let* $1 \leq \ell_0(\lambda), \ell_1(\lambda) \leq \mathsf{poly}(\lambda)$ *be output-length functions. Let* $(\mathsf{Setup}, \mathcal{Y})$ *be a tuple of probabilistic algorithms, such that*

- Setup*, on input* 1^λ*, returns a master key* msk*, and*
- $\mathcal{Y}$*, on input* 1^λ *and* msk*, returns a pair of outputs* $(y_0, y_1) \in \{0,1\}^{\ell_0(\lambda)} \times \{0,1\}^{\ell_1(\lambda)}$.

We say that the tuple $(\mathsf{Setup}, \mathcal{Y})$ *defines a* reverse sampleable correlation with setup *if there exists a probabilistic polynomial time algorithm* $\mathsf{RSample}$ *such that*

- $\mathsf{RSample}$*, on input* 1^λ*,* msk*,* $\sigma \in \{0,1\}$ *and* $y_\sigma \in \{0,1\}^{\ell_\sigma(\lambda)}$*, returns* $y_{1-\sigma} \in \{0,1\}^{\ell_{1-\sigma}(\lambda)}$ *such that for all* $\mathsf{msk}, \mathsf{msk}'$ *in the image of* Setup *and all* $\sigma \in \{0,1\}$*, the following distributions are statistically close:*

$$\{(y_0, y_1) \mid (y_0, y_1) \leftarrow \mathcal{Y}(1^\lambda, \mathsf{msk})\}$$

$$\{(y_0, y_1) \mid (y_0', y_1') \leftarrow \mathcal{Y}(1^\lambda, \mathsf{msk}'), y_\sigma \leftarrow y_\sigma', y_{1-\sigma} \leftarrow \mathsf{RSample}(1^\lambda, \mathsf{msk}, \sigma, y_\sigma)\}$$

```
$\underline{\mathsf{Exp}^{\mathsf{pr}}_{\mathcal{A},Q,0}(\lambda):}$
$\mathsf{msk} \leftarrow \mathsf{Setup}(1^\lambda)$
for $i = 1$ to $Q(\lambda)$:
    $x^{(i)} \leftarrow \{0,1\}^{n(\lambda)}$
    $(y_0^{(i)}, y_1^{(i)}) \leftarrow \mathcal{Y}(1^\lambda, \mathsf{msk})$
$b \leftarrow \mathcal{A}(1^\lambda, (x^{(i)}, y_0^{(i)}, y_1^{(i)})_{i \in [Q(\lambda)]})$
return $b$
```

```
$\underline{\mathsf{Exp}^{\mathsf{pr}}_{\mathcal{A},Q,1}(\lambda):}$
$(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{PCF.Gen}(1^\lambda)$
for $i = 1$ to $Q(\lambda)$:
    $x^{(i)} \leftarrow \{0,1\}^{n(\lambda)}$
    for $\sigma \in \{0,1\}$: $y_\sigma^{(i)} \leftarrow \mathsf{PCF.Eval}(\sigma, \mathsf{k}_\sigma, x^{(i)})$
$b \leftarrow \mathcal{A}(1^\lambda, (x^{(i)}, y_0^{(i)}, y_1^{(i)})_{i \in [Q(\lambda)]})$
return $b$
```

Fig. 2. Pseudorandom $\mathcal{Y}$-correlated outputs of a PCF.

A PCF for a correlation $\mathcal{Y}$ consists of a key generation algorithm, Gen, which outputs a pair of correlated keys, together with an evaluation algorithm, Eval, which is given one of the keys and a public input, and produces a correlated output. In a *weak* PCF, we only consider running Eval with randomly chosen inputs, whereas in a *strong* PCF, the inputs can be chosen arbitrarily. Boyle *et al.* [BCG+20a] show that any weak PCF can be used together with a programmable random oracle to obtain a strong PCF, so from here on, our default notion of PCF will be a weak PCF.

There are two security requirements for a PCF: firstly, a pseudorandomness requirement, meaning that the joint distribution of both parties' outputs of Eval are indistinguishable from outputs of $\mathcal{Y}$. Secondly, there is a security property, which intuitively requires that pseudorandomness still holds even when given one of the parties' keys.

Definition 5.2 (Pseudorandom correlation function (PCF)). *Let* $(\mathsf{Setup}, \mathcal{Y})$ *fix a reverse-sampleable correlation with setup which has output length functions* $\ell_0(\lambda), \ell_1(\lambda)$*, and let* $\lambda \leq n(\lambda) \leq \mathsf{poly}(\lambda)$ *be an input length function. Let* $(\mathsf{PCF.Gen}, \mathsf{PCF.Eval})$ *be a pair of algorithms with the following syntax:*

- $\mathsf{PCF.Gen}(1^\lambda)$ *is a probabilistic polynomial time algorithm that on input* 1^λ*, outputs a pair of keys* $(\mathsf{k}_0, \mathsf{k}_1)$*;*
- $\mathsf{PCF.Eval}(\sigma, \mathsf{k}_\sigma, x)$ *is a deterministic polynomial-time algorithm that on input* $\sigma \in \{0,1\}$*, key* k_σ *and input value* $x \in \{0,1\}^{n(\lambda)}$*, outputs a value* $y_\sigma \in \{0,1\}^{\ell_\sigma(\lambda)}$*.*

We say $(\mathsf{PCF.Gen}, \mathsf{PCF.Eval})$ *is a (weak) pseudorandom correlation function (PCF) for* $\mathcal{Y}$*, if the following conditions hold:*

- ***Pseudorandom*** $\mathcal{Y}$***-correlated outputs.*** *For every* $\sigma \in \{0,1\}$ *and non-uniform adversary* $\mathcal{A}$ *of size* $\mathsf{poly}(\lambda)$*, and every* $Q = \mathsf{poly}(\lambda)$*, it holds that*

$$\left|\Pr[\mathsf{Exp}^{\mathsf{pr}}_{\mathcal{A},Q,0}(\lambda) = 1] - \Pr[\mathsf{Exp}^{\mathsf{pr}}_{\mathcal{A},Q,1}(\lambda) = 1]\right| \leq \mathsf{negl}(\lambda)$$

for all sufficiently large λ*, where* $\mathsf{Exp}^{\mathsf{pr}}_{\mathcal{A},Q,b}(\lambda)$ *for* $b \in \{0,1\}$ *is as defined in Figure 2. (In particular, where the adversary is given access to* $Q(\lambda)$ *samples.)*

```
$\underline{\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},Q,\sigma,0}(\lambda):}$
$(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{PCF.Gen}(1^\lambda)$
for $i = 1$ to $Q(\lambda)$:
    $x^{(i)} \leftarrow \{0,1\}^{n(\lambda)}$
    $y^{(i)}_{1-\sigma} \leftarrow \mathsf{PCF.Eval}(1-\sigma, \mathsf{k}_{1-\sigma}, x^{(i)})$
$b \leftarrow \mathcal{A}(1^\lambda, \sigma, k_\sigma, (x^{(i)}, y^{(i)}_{1-\sigma})_{i \in [Q(\lambda)]})$
return $b$
```

```
$\underline{\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},Q,\sigma,1}(\lambda):}$
$(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{PCF.Gen}(1^\lambda)$
$\mathsf{msk} \leftarrow \mathsf{Setup}(1^\lambda)$
for $i = 1$ to $Q(\lambda)$:
    $x^{(i)} \leftarrow \{0,1\}^{n(\lambda)}$
    $y^{(i)}_{\sigma} \leftarrow \mathsf{PCF.Eval}(\sigma, \mathsf{k}_{\sigma}, x^{(i)})$
    $y^{(i)}_{1-\sigma} \leftarrow \mathsf{RSample}(1^\lambda, \mathsf{msk}, \sigma, y^{(i)}_{\sigma})$
$b \leftarrow \mathcal{A}(1^\lambda, \sigma, k_\sigma, (x^{(i)}, y^{(i)}_{1-\sigma})_{i \in [Q(\lambda)]})$
return $b$
```

Fig. 3. Security of a PCF. Here, RSample is the algorithm for reverse sampling $\mathcal{Y}$ as in Definition 5.1.

- ***Security.*** *For each $\sigma \in \{0,1\}$ and non-uniform adversary $\mathcal{A}$ of size $B(\lambda)$, and every $Q = \mathsf{poly}(\lambda)$, it holds that*

$$\left|\Pr[\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},Q,\sigma,0}(\lambda) = 1] - \Pr[\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},Q,\sigma,1}(\lambda) = 1]\right| \leq \mathsf{negl}(\lambda)$$

for all sufficiently large λ, where $\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},Q,\sigma,b}(\lambda)$ for $b \in \{0,1\}$ is as defined in Figure 3 (again, with $Q(\lambda)$ samples).

5.2 PCF for Vector-OLE From Paillier

Vector oblivious linear evaluation, or VOLE, over a ring $R = R(\lambda)$, is a correlation defined by an algorithm Setup, which outputs $\mathsf{msk} = x$ for a random $x \in R$, and an algorithm $\mathcal{Y}_{\mathsf{VOLE}}$, which on input msk, samples random elements $u, v \in R$, computes $w = ux + v$ and outputs the pair $((u,v),(w,x))$. Note that w, v can be viewed as a subtractive secret sharing of the product ux. Since x is fixed, this means that a batch of VOLE samples can be used to perform scalar-vector multiplications on secret-shared inputs, as part of, for instance, a secure two-party computation protocol.

The main idea behind our PCF for VOLE is the following. In the standard Paillier cryptosystem, every element of $\mathbb{Z}^*_{N^2}$ defines a valid ciphertext, which makes it possible to obliviously sample an encryption of a random message, without knowing the underlying message. We exploit this by having both parties locally generate the same random ciphertexts[3], which are then viewed as encryptions of random inputs a in the HSS construction. Then, given a subtractive secret sharing of xd, where d is the secret key and $x \in \mathbb{Z}_N$ is some fixed value, the parties can use the distributed multiplication protocol from the HSS scheme to obtain shares z_0, z_1 such that $z_1 = z_0 + ax$. If one party is additionally given the secret key d (and hence learns the a's) and the other party is given x, then this process can be repeated to produce an arbitrarily long VOLE correlation.

[3] E.g. with a random oracle, or some other public source of randomness.

In the PCF construction, shown in Construction 5.4, the values x, d and shares of xd are distributed by the PCF Gen algorithm, while the random ciphertexts are given as public inputs to the Eval algorithm, since we are only building a weak PCF and not a strong one. Additionally, the parties use a PRF to randomize their output shares and ensure that these are uniformly distributed.

Theorem 5.3. *Suppose the DCR assumption holds, and that F is a secure PRF. Then Construction 5.4 is a secure PCF for the VOLE correlation, $\mathcal{Y}_{\mathsf{VOLE}}$, over the ring $\mathbb{Z}_N$.*

Proof. We first argue the pseudorandomness property, namely, that the joint distribution of outputs from Eval are indistinguishable from outputs of $\mathcal{Y}_{\mathsf{VOLE}}$ (see game in Fig. 2). First, note that by the correctness of DDLog (Lemma 3.3), and the fact that each random input c lies in $\mathbb{Z}^*_{N^2}$, the outputs satisfy $z_1 - z_0 = ax$. Also, note that x is uniform in $\mathbb{Z}_N$ in both the PCF and the ideal $\mathcal{Y}_{\mathsf{VOLE}}$ outputs.

Therefore, it suffices to show that the first party's input/output samples given by (c, a, z_0), where $a = \mathsf{Paillier.Dec}(d, c)$ and $z_0 = \mathsf{DDLog}_N(c^{y_0}) + F_{\mathsf{k}_{\mathsf{prf}}}(c)$, are computationally indistinguishable from samples of the form (c, a', z_0'), where a', z_0' are uniform in $\mathbb{Z}_N$. We can first replace each z_0 with a random z_0' from $\mathbb{Z}_N$, which is indistinguishable due to the security of the PRF F (remember that in the pseudorandomness definition of Fig. 2 the distinguisher does not see the PRF key). Next, we also replace each a with random values a' in $\mathbb{Z}_N$, instead of decryptions of c; this is indistinguishable under the DCR assumption, by the semantic security of Paillier encryption.

We now argue the security property from Definition 5.2. First consider the case $\sigma = 0$. We want to show that, given the key $\mathsf{k}_0 = (N, \mathsf{k}_{\mathsf{prf}}, y_0, d)$ from Gen, no adversary can distinguish a set of random inputs and evaluations (c, z_1, x) computed from k_1 from random, reverse-sampled values conditioned on the (z_0, a) computed from k_0. Note that the reverse-sampling algorithm simply picks a random $x \in \mathbb{Z}_N$ as its setup, and then on input (z_0, a), outputs $z_1 = z_0 + ax$. It is clear that the two distributions are identical, due to the correctness property and the fact that the key k_0 is sampled independently of x.

For the case $\sigma = 1$, in the real distribution the adversary is given $\mathsf{k}_1 = (N, \mathsf{k}_{\mathsf{prf}}, y_1, x)$ and a set of input/output samples of the form (c, z_0, a), where z_0, a are computed using $\mathsf{Eval}(0, \mathsf{k}_0, c)$. We consider the following sequence of hybrids.

In the first hybrid, instead of computing z_0 as in Eval, we compute $z_0 = z_1 - ax$. This is identically distributed to the original experiment, due to the correctness of DDLog.

Next, we sample the share y_1 uniformly from $[N^3 2^\kappa]$, instead of computing $y_1 = y_0 + x \cdot d$. Since y_0 is uniform in $[N^3 2^\kappa]$ and $xd < N^3$, this is within statistical distance $2^{-\kappa}$ of the previous distribution.

Finally, we replace each a by a uniform value from $\mathbb{Z}_N$. This is identical to the reverse-sampled distribution, and computationally indistinguishable from the previous case, since all other values seen by the adversary are independent of the secret key, so we can rely on the semantic security of Paillier under DCR. □

Construction 5.4: PCF for Vector Oblivious Linear Evaluation

Let $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \mathbb{Z}_N$ be a pseudorandom function.

Gen: On input 1^λ:

1. Sample $(N, p, q) \leftarrow \mathsf{GenPQ}(1^\lambda)$.
2. Compute $d \in \mathbb{Z}$ such that $d \equiv 0 \pmod{\varphi(N)}$ and $d \equiv 1 \pmod{N}$.
3. Sample $x \leftarrow [N], y_0 \leftarrow [N^3 2^\kappa]$, and let $y_1 = y_0 + x \cdot d$ over the integers.
4. Sample $\mathsf{k}_{\mathsf{prf}} \leftarrow \{0,1\}^\lambda$.
5. Output the keys $\mathsf{k}_0 = (N, \mathsf{k}_{\mathsf{prf}}, y_0, d)$ and $\mathsf{k}_1 = (N, \mathsf{k}_{\mathsf{prf}}, y_1, x)$.

Eval: On input $(\sigma, \mathsf{k}_\sigma, c)$, for a random input $c \in \mathbb{Z}^*_{N^2}$:

- If $\sigma = 0$, parse $\mathsf{k}_0 = (N, \mathsf{k}_{\mathsf{prf}}, y_0, d)$:
  1. Compute $a = \mathsf{Paillier.Dec}(d, c)$.
  2. Compute $z_0 = \mathsf{DDLog}_N(c^{y_0}) + F_{\mathsf{k}_{\mathsf{prf}}}(c) \bmod N$.
  3. Output (z_0, a)
- If $\sigma = 1$, parse $\mathsf{k}_1 = (N, \mathsf{k}_{\mathsf{prf}}, y_1, x)$:
  1. Compute $z_1 = \mathsf{DDLog}_N(c^{y_1}) + F_{\mathsf{k}_{\mathsf{prf}}}(c) \bmod N$.
  2. Output (z_1, x)

5.3 PCF for Oblivious Transfer From Quadratic Residuosity

To build a PCF for OT, we will first build a PCF for *XOR-correlated OT*, where the sender's messages are all of the form $z_1, z_1 \oplus s$ for some fixed string s. This is formally defined by a correlation $\mathcal{Y}_{\oplus\text{-}\mathsf{OT}}$, where the setup algorithm Setup picks a random $\mathsf{msk} = s \leftarrow \{0,1\}^\lambda$, and then each call to $\mathcal{Y}_{\oplus\text{-}\mathsf{OT}}(\mathsf{msk})$ first samples $b \leftarrow \{0,1\}, z_0 \leftarrow \{0,1\}^\lambda$, lets $z_1 = z_0 \oplus b \cdot s$, and outputs the pair $(z_0, b), (z_1, s)$.

Our PCF construction proceeds analogously to the VOLE case, except we rely on the Goldwasser-Micali cryptosystem instead of Paillier.

GM Encryption. We use the Goldwasser–Micali (GM) cryptosystem [GM82], with the simplified decryption procedure by Katz and Yung [KY02], which allows threshold decryption when p, q are both $3 \pmod 4$.[4]

$\mathsf{GM.Gen}(1^\lambda)$:
1. Sample $(N, p, q) \leftarrow \mathsf{GenPQ}(1^\lambda)$.
2. Let $d = \phi(N)/4 = (N - p - q + 1)/4$.
3. Output $\mathsf{pk} = N$, $\mathsf{sk} = d$.

$\mathsf{GM.Enc}(\mathsf{pk}, x \in \{0,1\})$:
1. Sample a random $r \leftarrow \mathbb{Z}_N$.
2. Output $\mathsf{ct} = r^2(-1)^x \bmod N$.

[4] One can also obtain a similar threshold-compatible decryption under more general requirements for the modulus; see Desmedt and Kurosawa [DK07].

Observe that, if $x = 0$, ct will be a random quadratic residue modulo N; if $x = 1$, ct will be a random non-residue.

$\mathsf{GM.Dec}(\mathsf{sk}, \mathsf{ct})$:

1. Compute $y = \mathsf{ct}^d \bmod N$, which is in $\{1, -1\}$, and output $x = 0$ if $y = 1$, or $x = 1$ if $y = -1$.

Notice that in the GM cryptosystem, $\mathbb{J}_N$ (the elements of $\mathbb{Z}_N$ with Jacobi symbol 1) defines the set of valid ciphertexts. This allows us to sample a random ciphertext without knowing the corresponding message, by generating a random element of $\mathbb{Z}_N$ and testing that it has Jacobi symbol 1, which can be done efficiently.

We also use the distributed discrete log procedure $\mathsf{DDLog}^{\mathsf{GM}}$, shown in Algorithm 5.5. By inspection, it can be seen that for any two inputs $a_0, a_1 \in \mathbb{Z}_N^*$ satisfying $a_1/a_0 = (-1)^b$ for a bit b, we have $\mathsf{DDLog}^{\mathsf{GM}}(a_0) \oplus \mathsf{DDLog}^{\mathsf{GM}}(a_1) = b$. Note that this procedure was previously used to construct trapdoor hash functions [DGI+19].

Algorithm 5.5: $\mathsf{DDLog}^{\mathsf{GM}}(a \in \mathbb{Z}_N)$

1. Map a to an integer in $\{0, \ldots, N-1\}$.
2. If $a < N/2$ then output $z = 1$, otherwise, output $z = 0$.

PCF for Oblivious Transfer. The construction proceeds similarly to the VOLE case, except instead of one sharing, the Gen algorithm samples λ subtractive sharings of $s_j \cdot d$, where d is the GM secret key and s_j is one bit of the sender's fixed correlated OT offset. Then, given a random encryption of a bit b in Eval, the parties run $\mathsf{DDLog}^{\mathsf{GM}}$ λ times to obtain XOR shares of the string $b \cdot s \in \{0,1\}^\lambda$, giving a correlated OT as required.

Construction 5.6: PCF for Oblivious Transfer

Let $F : \{0,1\}^\lambda \times \mathbb{Z}_N \to \{0,1\}^\lambda$ be a pseudorandom function.

Gen: On input 1^λ:

1. Sample $(N, p, q) \leftarrow \mathsf{GenPQ}(1^\lambda)$, and let $d = \varphi(N)/4$.
2. Sample $\mathsf{k}_{\mathsf{prf}} \leftarrow \{0,1\}^\lambda$.
3. For $j = 1, \ldots, \lambda$, sample $s_j \leftarrow \{0,1\}$, $y_{0,j} \leftarrow [N2^\kappa]$, and let $y_{1,j} = y_{0,j} + s_j \cdot d$ over the integers. Write $s = (s_1, \ldots, s_\lambda)$.
4. Output the keys $\mathsf{k}_0 = (N, \mathsf{k}_{\mathsf{prf}}, \{y_{0,j}\}_{j \in [\lambda]}, d)$ and $\mathsf{k}_1 = (N, \mathsf{k}_{\mathsf{prf}}, \{y_{1,j}\}_{j \in [\lambda]}, s)$.

Eval: On input $(\sigma, \mathsf{k}_\sigma, c)$, for a random input $c \in \mathbb{J}_N$:

- If $\sigma = 0$, parse $\mathsf{k}_0 = (N, \mathsf{k}_{\mathsf{prf}}, \{y_{0,j}\}_{j \in [\lambda]}, d)$:
  1. Compute $b = \mathsf{GM.Dec}(d, c)$ in $\{0,1\}$.
  2. For $j = 1, \ldots, \lambda$, compute $z_{0,j} = \mathsf{DDLog}^{\mathsf{GM}}(c^{y_{0,j}})$.

```
    3. Let $z_0 = (z_{0,0}, \dots, z_{0,\lambda}) \oplus F_{\mathsf{k}_{\mathsf{prf}}}(c)$.
    4. Output $(z_0, b)$.
  – If $\sigma = 1$, parse $\mathsf{k}_1 = (N, \mathsf{k}_{\mathsf{prf}}, \{y_{1,j}\}_{j \in [\lambda]}, s)$:
    1. For $j = 1, \dots, \lambda$, compute $z_{1,j} = \mathsf{DDLog}^{\mathsf{GM}}(c^{y_{1,j}})$.
    2. Let $z_1 = (z_{1,1}, \dots, z_{1,\lambda}) \oplus F_{\mathsf{k}_{\mathsf{prf}}}(c)$.
    3. Output $(z_1, s)$.
```

Theorem 5.7. *Suppose the QR assumption holds, and that F is a secure PRF. Then Construction 5.6 is a secure PCF for the correlated OT correlation, $\mathcal{Y}_{\oplus\text{-}\mathsf{OT}}$.*

Proof. We first argue $\mathcal{Y}_{\oplus\text{-}\mathsf{OT}}$-pseudorandomness. For each query on input c, where $b = \mathsf{Dec}(d, c)$, by the correctness of $\mathsf{DDLog}^{\mathsf{GM}}$ we have $z_{1,j} \oplus z_{0,j} = s_j \cdot b$, for each $j = 1, \dots, \lambda$, which implies $z_1 \oplus z_0 = b \cdot s$ as required. Further, by the same argument as for the PCF for VOLE, the input/output pairs $c, (z_0, b)$ for party $\sigma = 0$ are computationally indistinguishable from random, due to the security of the PRF, and the pseudorandom ciphertexts property of Goldwasser-Micali. This completes the pseudorandomness property.

Next, we consider the security property for $\sigma = 0$. Just as with the proof of Theorem 5.3, the expanded outputs of party 1 are identically distributed to reverse-sampled outputs from $\mathcal{Y}_{\oplus\text{-}\mathsf{OT}}$ (conditioned on the outputs from k_0), so we have perfect security due to the fact that k_0 is entirely independent of the sender's secret s.

When $\sigma = 1$, we follow the same sequence of hybrid games as in Theorem 5.3. First, we replace each $y_{1,j}$ with a uniform element of $[N2^\kappa]$, which is statistically close to the real $y_{1,j}$ since $s_j d < N$. Then, we replace each bit b with a random bit, instead of using d to decrypt c, which is computationally indistinguishable under the QR assumption. □

Extension to random oblivious transfer. A correlated OT can be locally coverted into a *random OT*, where both of the sender's messages are independently random, using a hash function and the technique of Ishai *et al.* [IKNP03]. The sender simply applies the hash function to compute its outputs $\mathsf{H}(z_1), \mathsf{H}(z_1 \oplus s)$, while the receiver outputs $\mathsf{H}(z_0) = \mathsf{H}(z_1 \oplus b \cdot s)$. Assuming a suitable correlation robustness property of H, the resulting OT messages are pseudorandom. It was shown by Boyle et al. [BCG+19, BCG+20a] that this transformation can be used to convert any PCF or PCG for correlated OT into one for the random OT correlation. Hence, we obtain the following.

Corollary 5.8. *Suppose the QR assumption holds, and there is a secure correlation-robust hash function. Then, there exists a secure PCF for the random oblivious transfer correlation.*

5.4 PCG for OLE and Degree-2 Correlations From LPN and Paillier

In Section 5.2, we showed how to build a PCF for VOLE, where the parties obtain $(u, v) \in R^2$ and $(x, w) \in R^2$, respectively, such that u, v are random, $w = ux + v$, and x is fixed for all samples from the correlation. In this section we show how to upgrade this to more general degree-2 correlations, including OLE, where x is sampled freshly at random for each instance. Of course, if we could get many VOLE PCF setups, each one of those could yield one OLE instance (if we only use it once!). Here, we show how to get m setups for the VOLE construction *all at once* from a smaller amount of correlated randomness, in what amounts to a PCG for OLE. (We emphasize that this is a pseudorandom correlation *generator*, not *function*, since it produces a fixed number of correlations.) Later, we also observe that this PCG for OLE can be generalized in several ways, to obtain a PCG for *general degree-2 correlations* over $\mathbb{Z}_N$, or $\mathbb{Z}_2$ when replacing Paillier by QR, and finally even to a pseudorandom correlation *function*, when replacing LPN by a variable-density variant of LPN [BCG+20a].

In the main construction we fix N, as well as the associated Paillier decryption key d, which we give to party 0, across all m instances. Our goal is run m copies of Construction 5.4 so we would *like* to give party 1 m random values $x_1, \ldots, x_m$, and secret share each dx_i over the integers between the two parties. However, this doesn't give us a PCG, because the size of our setup would be the same as the number of correlations we are able to produce. In order to keep our setup size much smaller than m, we instead produce the *setup* with a variant of a PCG based on the LPN assumption [BCGI18]. We give party 1 a *sparse* n-element vector $\boldsymbol{e}$ of elements in $[N]$, for $m < n$, which only contains $t = \mathsf{poly}(\lambda)$ non-zero elements. (Since it is sparse, it can actually be represented in $t \log(n) \log(N) \ll n$ bits.) By the dual form of the LPN assumption, $H \cdot \boldsymbol{e}$ for such a sparse $\boldsymbol{e}$ and some public $H \in \mathbb{Z}^{m \times n}$ looks pseudorandom (if $\boldsymbol{e}$ is unknown), so we can expand $\boldsymbol{e}$ to give m psuedorandom elements. In order to similarly compress a sharing of $d \cdot \boldsymbol{e}$, we use a *function secret sharing of the multi-point function* defined by $d \cdot \boldsymbol{e}$. (Note that we need to use a large enough modulus in the function secret sharing so that the output shares the parties obtain are shares over the integers with overwhelming probability.) This allows both parties to obtain shares of $d \cdot \boldsymbol{e}$, and then to compute shares of $H \cdot (d \cdot \boldsymbol{e})$. This completes the setup of m instances of our VOLE PCF; the parties then use each of those instances once, to get m instances of the OLE correlation.

We present the complete PCG in Construction 5.10. Preliminaries on FSS are deferred to Appendix A.

Theorem 5.9. *Let* H *be a random oracle,* F *a secure PRF, and suppose that both the LPN and DCR assumptions hold. Then Construction 5.10 is a secure PCG for the OLE correlation.*

Proof. We first argue the pseudorandomness property, by showing that the outputs of $\mathsf{PCG}_{\mathsf{OLE}}$ are indistinguishable from a set of m random OLEs. The outputs are given by $(\boldsymbol{a}, \boldsymbol{z}_0), (\boldsymbol{b}, \boldsymbol{z}_1)$, where each $\boldsymbol{z}_\sigma$ is obtained by first taking an FSS output $\boldsymbol{y}'_\sigma$, then multiplying by H, and finally applying DDLog and adding a PRF value to each entry of the vector.

Hybrid 0**:** First, we consider a hybrid where we replace the FSS output $\boldsymbol{y}'_1$ with $\boldsymbol{y}'_0 + d \cdot \boldsymbol{e}$ in $\mathbb{Z}_{N^3 2^\lambda}$. Due to the correctness of FSS, this is statistically indistinguishable from the previous experiment.

Hybrid 1**:** Next, we replace the other party's FSS output, $\boldsymbol{y}'_0$, with a uniform vector in $\mathbb{Z}_{N^3 2^\lambda}$, and compute $\boldsymbol{y}'_1$ using this. By the pseudorandom outputs property of FSS, this is computationally indistinguishable.

Hybrid 2**:** In this hybrid, instead of computing $\boldsymbol{y}'_1 = \boldsymbol{y}'_0 + d \cdot \boldsymbol{e}$ in $\mathbb{Z}_{N^3 2^\lambda}$, we compute it over the integers. Since each entry of $d \cdot \boldsymbol{e}$ is less than N^3, and $\boldsymbol{y}'_0$ is uniform in $\{0, \dots, N^3 2^\lambda\}$, this is statistically close to hybrid 1.
Note that now, we have $\boldsymbol{y}_1 - \boldsymbol{y}_0 = d \cdot (H \cdot \boldsymbol{e})$ over $\mathbb{Z}$, so by the correctness of DDLog and the fact that $a_j = \mathsf{Dec}_d(c_j)$, it holds that $\boldsymbol{z}_{1,j} - \boldsymbol{z}_{0,i} = a_i \cdot b_i \pmod{N}$ (where b_j is the j-the entry of $H \cdot \boldsymbol{e} \bmod N$).

Hybrid 3**:** In this hybrid, we replace the outputs $\boldsymbol{z}_0$ with a uniform vector in $\mathbb{Z}_N^m$, and compute $\boldsymbol{z}_1 = \boldsymbol{z}_0 + \boldsymbol{a} * \boldsymbol{b} \bmod N$. Since the outputs in Hybrid 2 already satisfied this relation, the two experiments are computationally indistinguishable due to the PRF.

Hybrid 4**:** Next, instead of computing $\boldsymbol{b} = H \cdot \boldsymbol{e} \bmod N$, we sample $\boldsymbol{b}$ uniformly at random from $\mathbb{Z}_N^m$. This is indistinguishable from Hybrid 3 under the LPN assumption.

Hybrid 5**:** In the final hybrid, instead of computing a_j by decrypting c_j (which is an output of the random oracle), we sample a_j uniformly at random. This is indistinguishable from the previous hybrid under DCR, following the semantic security of Paillier. It is also identical to the ideal $\mathcal{Y}_{\mathsf{OLE}}$ distribution.

Security, $\sigma = 0$. Here, we rely on the LPN assumption, as well as the security of FSS. Recall that the distinguisher is given $\mathsf{k}_0 = (N, \mathsf{k}_{\mathsf{prf}}, \mathsf{k}_0^{\mathsf{fss}}, d)$, and party 1's expanded outputs $(\boldsymbol{b}, \boldsymbol{z}_1)$. By the same argument as the first two hybrids previously, this is indistinguishable from the case where $\boldsymbol{z}_1$ is computed as $\boldsymbol{z}_0 + \boldsymbol{a} * \boldsymbol{b}$ (due to correctness of FSS and DDLog). We next consider a hybrid where $\mathsf{k}_0^{\mathsf{fss}}$ is replaced with a key generated from the FSS simulator, which is computationally indistinguishable from the real key. Since everything in the distinguisher's view, except for $\boldsymbol{b} = H\boldsymbol{e}$, is now now generated independently of the error vector $\boldsymbol{e}$, we can now use the LPN assumption to replace $\boldsymbol{b}$ with a uniform vector.

Security, $\sigma = 1$. This argument proceeds symmetrically to the case where $\sigma = 0$, except here, instead of replacing $\boldsymbol{b}$ with a random vector based on LPN, the final step is to replace $\boldsymbol{a}$ with a random vector, relying on the DCR assumption. □

Construction 5.10: $\mathsf{PCG}_{\mathsf{OLE}}$

Let $\mathsf{H} : \{0,1\}^* \to \mathbb{Z}_{N^2}$ be a hash function, modelled as a random oracle, and $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \mathbb{Z}_N$ be a PRF.

Let m, n be length parameters with $m < n$, and $H \in \mathbb{Z}^{m \times n}$ be a matrix for which the dual-LPN problem is hard over $\mathbb{Z}_N$.

Gen: On input 1^λ:

1. Sample $(N, p, q) \leftarrow \mathsf{GenPQ}(1^\lambda)$.
2. Compute $d \in \mathbb{Z}$ such that $d = 0 \bmod \varphi(N)$ and $d = 1 \bmod N$.
3. Sample $\mathsf{k}_{\mathsf{prf}} \leftarrow \{0,1\}^\lambda$.
4. Sample a vector $\boldsymbol{e} \in \mathbb{Z}_N^n$ with t random, non-zero entries, and zero elsewhere.
5. Generate FSS keys $\mathsf{k}_0^{\mathsf{fss}}, \mathsf{k}_1^{\mathsf{fss}}$ for the multi-point function defined by $d \cdot \boldsymbol{e}$, with domain size n and range $\mathbb{Z}_{N'}$, for $N' = N^3 2^\kappa$.[a]
6. Output the seeds $\mathsf{k}_0 = (N, \mathsf{k}_{\mathsf{prf}}, \mathsf{k}_0^{\mathsf{fss}}, d)$ and $\mathsf{k}_1 = (N, \mathsf{k}_{\mathsf{prf}}, \mathsf{k}_1^{\mathsf{fss}}, \boldsymbol{e})$.

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

- If $\sigma = 0$, parse $\mathsf{k}_0 = (N, \mathsf{k}_{\mathsf{prf}}, \mathsf{k}_0^{\mathsf{fss}}, d)$:
    1. Let $\boldsymbol{y}_0' = \mathsf{FSS.FullEval}(0, \mathsf{k}_0^{\mathsf{fss}})$ in $\mathbb{Z}^n$.
    2. Compute $\boldsymbol{y}_0 = H \cdot \boldsymbol{y}_0'$ in $\mathbb{Z}^m$.
    3. For $j = 1, \ldots, m$:
        (a) Let $c_j = \mathsf{H}(sid, j)$ in $\mathbb{Z}_{N^2}$
        (b) Compute $a_j = (c^d - 1)/N$ in $\mathbb{Z}_N$.
        (c) Compute $z_{0,j} = \mathsf{DDLog}_N(c_j^{y_{0,j}}) + F_{\mathsf{k}_{\mathsf{prf}}}(j)$.
    4. Output $\boldsymbol{a} = (a_1, \ldots, a_m)$ and $\boldsymbol{z}_0 = (z_{0,1}, \ldots, z_{0,m})$.
- If $\sigma = 1$, parse $\mathsf{k}_1 = (N, \mathsf{k}_{\mathsf{prf}}, \mathsf{k}_1^{\mathsf{fss}}, \boldsymbol{e})$:
    1. Let $\boldsymbol{y}_1' = \mathsf{FSS.FullEval}(1, \mathsf{k}_1^{\mathsf{fss}})$ in $\mathbb{Z}^n$.
    2. Compute $\boldsymbol{y}_1 = H \cdot \boldsymbol{y}_1'$ in $\mathbb{Z}^m$ and $\boldsymbol{b} = H \cdot \boldsymbol{e}$ in $\mathbb{Z}_N^m$.
    3. For $j = 1, \ldots, m$:
        (a) Let $c_j = \mathsf{H}(sid, j)$ in $\mathbb{Z}_{N^2}$
        (b) Compute $z_{1,j} = \mathsf{DDLog}_N(c_j^{y_{1,j}}) + F_{\mathsf{k}_{\mathsf{prf}}}(j)$.
    4. Output $\boldsymbol{b} = H \cdot \boldsymbol{e}$ and $\boldsymbol{z}_1 = (z_{1,1}, \ldots, z_{1,m})$, both in $\mathbb{Z}_N^m$.

[a] This ensures that $d \cdot \boldsymbol{e}$ is always much less than N', so we get shares over the integers.

Extensions to Degree-2 Correlations, and PCF. We now describe three natural extensions to the above construction, to degree-2 correlations over $\mathbb{Z}_N$, degree-2 correlations over $\mathbb{Z}_2$, and PCFs. Firstly, note that in step 3c of $\mathsf{PCG}_{\mathsf{OLE}}$, the parties use DDLog to obtain shares of $a_j \cdot b_j$. However, they could just as easily multiply $a_i \cdot b_j$, for any $i \neq j$, by using instead the corresponding ciphertext c_i. This means the parties can actually use the same PCG seeds to get shares of $f(\boldsymbol{a}, \boldsymbol{b})$, where f is any degree-2 function of $\boldsymbol{a}, \boldsymbol{b}$. This PCG for degree-2 correlations can be used, for instance, to get shares of matrix products and correlations for circuit-dependent preprocessing [BCG+20b].

Secondly, notice that the same construction also applies when using Goldwasser-Micali instead of Paillier. We use FSS in the same way to obtain shares of $d \cdot \boldsymbol{e}$, where now d is the GM decryption key (as in Construction 5.6). We also replace the matrix H with a binary matrix, used for LPN over $\mathbb{Z}_2$, and perform the DDLog step in the same way, using the corresponding algorithm for GM. This leads to a PCG for general, degree-2 correlations over $\mathbb{Z}_2$, based on LPN and QR.

Finally, we remark that all of these constructions can be upgraded to give a *pseudorandom correlation function*, where the number of outputs is not polynomially bounded, and they can be computed on-the-fly. This can be done by replacing the matrix H with a *variable-density* matrix, relying on variable-density LPN [BCG+20a]. The idea is that both H and $\boldsymbol{e}$ are chosen to be exponentially large, but sparse enough that they can still be compressed so that a single entry of the product $H \cdot \boldsymbol{e}$ can be computed efficiently. This follows by directly applying the variable-density LPN assumption introduced in [BCG+20a].

6 Public-key Setup for PCFs

6.1 Non-Interactive VOLE

In this section, we present our protocol for non-interactive VOLE with semi-honest security, based on Paillier. Party $\mathsf{P_A}$ has input values $a_1, \ldots, a_n$, while party $\mathsf{P_B}$ has a single input value x, and the goal is to obtain additive shares of $a_i \cdot x$ modulo N, by exchanging just one simultaneous message. We assume that the modulus N has been generated as a trusted setup, and no-one knows its factorization.

Our protocol starts off in the spirit of Bellare-Micali OT [BM90], where $\mathsf{P_B}$ sends g^s for a random s, and $\mathsf{P_A}$ sends $g^{r_i} \cdot C^{a_i}$, for random r_i, where g and C are some fixed random group elements. Note that in Bellare-Micali, a_i is a bit, whereas here it is in $\mathbb{Z}_N$. At this point (where we depart slightly from [BM90]), $\mathsf{P_A}$ can compute the keys $g^{r_i s}$, while $\mathsf{P_B}$ can compute, $g^{r_i s} \cdot C^{a_i s}$ (without knowing a_i). We then have that the *ratio* of each of the two parties' keys is $C^{a_i s}$. Next, we additionally have $\mathsf{P_B}$ send a correction value $D = C^s \cdot (1+N)^x$, which allows $\mathsf{P_A}$ to adjust its key so that the ratios become $(1+N)^{a_i x}$. Finally, each party locally applies the distributed discrete log procedure to convert each key into an additive share of $a_i \cdot x$ modulo N. To allow for simulation, both parties randomize their output shares. Since we are dealing with passive security, it is enough for one of the parties ($\mathsf{P_A}$) to sample those values. The full protocol is specified in Construction 6.3.

Theorem 6.1. *The protocol in Construction 6.3 securely implements Functionality 6.2 in the presence of passive, static corruptions under the DCR and QR assumptions.*

Proof. It is easy to verify that each pair of keys K_i, K'_i satisfies the right relation, that is:

$$K'_i / K_i = B^{r_i} \cdot D^{b_i} / A_i^s = C^{a_i s} \cdot (1+N)^{x a_i} / C^{a_i s} = (1+N)^{x a_i}$$

Then by Lemma 3.3 we have that, for each i, $y_{0,n} - y_{1,n} = x a_i$.

Towards proving security, note that the view of a corrupt $\mathsf{P_B}$ only consists of $(A_1, \ldots, A_n, t_1, \ldots, t_n)$. The A_i's can be efficiently simulated by simply choosing uniform values from $\mathbb{Z}_{N^2}^*$. However we also need to ensure that the outputs of of the computation are the same in the real and ideal protocol. Towards this, we set the t_i to be the unique values consistent with the output of the functionality i.e., $t_i = y_{0,i} - \mathsf{DDLog}_N(A_i^s)$. We then prove security with the following series of hybrids.

Hybrid 0: This is the simulated protocol i.e., on input $(x, y_{0,1}, \ldots, y_{0,n})$ the simulator picks a random $s \leftarrow [N^2]$, samples uniformly random $A_i \leftarrow \mathbb{Z}_{N^2}^*$, and then programs $t_i = y_{0,i} - \mathsf{DDLog}_N(A_i^s)$.

Hybrid 1: We replace the random (g, C) in the CRS and the A_i's with random N-th residues, that is, $(g')^N, (C')^N, (A_i')^N$, for random $g', C', A_i' \in \mathbb{Z}_{N^2}^*$. By the DCR assumption, these are indistinguishable.

Hybrid 2: Instead of sampling g', C', A_i' at random, we choose them to be random squares in $\mathbb{Z}_{N^2}^*$; this ensures that (g, C, A_i) are $2N$-th residues, which have order $p'q'$ (since N is a product of safe primes). This step is indistinguishable from the previous hybrid under the QR assumption.

Hybrid 3: We now compute A_i as in the protocol i.e., by sampling a random r_i and setting $A_i = g^{r_i} C^{a_i}$. Since g, C, A_i are all of order $p'q'$, this hybrid is distributed statistically close to the previous one (due to the choice of r_i).

Hybrid 4: We replace (g, C) in the CRS with uniform random elements from $\mathbb{Z}_{N^2}^*$. As in Hybrids 1–2, this is indistinguishable due to DCR and QR. Note that Hybrid 4 corresponds to the real protocol, with the only exception that the t_i's are programmed in this hybrid while they are random in the real protocol. However, note that each t_i is in fact identically distributed in the hybrid and in the real protocol, since it is the *only* value that leads to the correct output $y_{0,i}$. Therefore this concludes the proof of security for a corrupt $\mathsf{P_B}$.

The view of a corrupt $\mathsf{P_A}$ consists of the tuple (B, D). This can be simulated essentially by running the protocol as an honest $\mathsf{P_B}$ with input $x = 0$. Indistinguishability of the view follows due to Lemma 2.1. As for the case of the corrupted $\mathsf{P_B}$, we program the random values t_i's to be the only value which would produce the right output, namely $t_i = y_{1,i} - \mathsf{DDLog}_N(B^{r_i} D^{a_i})$. We then prove security with the following series of hybrids.

Hybrid 0: This is the simulated protocol where on input $(a_1, \ldots, a_n, y_{1,1}, \ldots, y_{1,n})$ the simulator picks a random $s \leftarrow [N^2]$ and computes $(B, D) = (g^s, C^s)$, samples uniformly random $r_i \leftarrow [N^2]$, and then programs $t_i = y_{0,i} - \mathsf{DDLog}_N(B^{r_i} D^{b_i})$.

Hybrid 1: We replace (g, C) in the CRS to be random N-th residues. Indistinguishability follows from DCR.

Hybrid 2: We replace D with $C^s(1 + N)^x$. Indistinguishability from the previous hybrid follows from Lemma 2.1.

Hybrid 3: We replace g, C in the CRS with uniformly random elements from $\mathbb{Z}_{N^2}^*$. This is indistinguishable due to DCR.

Note that Hybrid 3 corresponds to the real protocol, with the only exception that the t_i's are programmed in this hybrid while they are random in the real protocol. However, note that each t_i is in fact identically distributed in the hybrid and in the real protocol, since it is the *only* value that leads to the correct output $y_{1,i}$. Therefore this concludes the proof of security for a corrupt $\mathsf{P_A}$. □

Functionality 6.2: $\mathcal{F}_{\mathbb{Z}_N\text{-VOLE}}$

The functionality interacts with parties $\mathsf{P_B}$, $\mathsf{P_A}$ and an adversary $\mathcal{A}$.

On input $a_1, \ldots, a_n \in \mathbb{Z}_N$ from $\mathsf{P_A}$ and $x \in \mathbb{Z}_N$ from $\mathsf{P_B}$, the functionality does the following:

- Sample $y_{0,i} \leftarrow \mathbb{Z}_N$, for $i = 1, \ldots, n$, and set $y_{1,i} = y_{0,i} + a_i \cdot x$.
- Output $(y_{0,1}, \ldots, y_{0,n})$ to $\mathsf{P_B}$ and $(y_{1,1}, \ldots, y_{1,n})$ to $\mathsf{P_A}$.

Construction 6.3: Non-interactive VOLE protocol

CRS: The algorithms below implicitly have access to $\mathsf{crs} = (N, g, C)$, where $(N, p, q) \leftarrow \mathsf{GenPQ}(1^\lambda)$, and $g, C \leftarrow \mathbb{Z}^*_{N^2}$.

Message from $\mathsf{P_A}$: On input $(a_1, \ldots, a_n) \in \mathbb{Z}_N^n$, sample $r_i \leftarrow [N^2], t_i \leftarrow \mathbb{Z}_N$, compute $A_i = g^{r_i} \cdot C^{a_i}$ and send (A_i, t_i) for $i = 1, \ldots, n$.

Message from $\mathsf{P_B}$: On input $x \in \mathbb{Z}_N$, sample $s \leftarrow [N^2]$ and send (B, D) where $B = g^s$, $D = C^s \cdot (1+N)^x$.

Output of $\mathsf{P_A}$: On receiving (B, D), compute $K'_i = B^{r_i} \cdot D^{a_i}$ and output $(y_{1,1}, \ldots, y_{1,n})$, where $y_{1,i} = \mathsf{DDLog}_N(K'_i) + t_i$.

Output of $\mathsf{P_B}$: On receiving $(A_1, \ldots, A_n, t_1, \ldots, t_n)$, compute $K_i = A_i^s$ and output $(y_{0,1}, \ldots, y_{0,n})$, where $y_{0,i} = \mathsf{DDLog}_N(K_i) + t_i$.

6.2 Public-Key Silent PCFs

We can plug our non-interactive VOLE protocol into the PCFs of Section 5 to obtain a public-key variant of those protocols where, after independently posting a public key, each party can locally derive its PCF key. Using their PCF keys, together with a random oracle to generate the public random inputs, the parties can then silently compute an arbitrary quantity of OT or VOLE correlations, without any interaction beyond the PKI.

Formally speaking, we can model this by defining a *public-key PCF* the same way as a standard PCF, except we replace the Gen algorithm with two separate algorithms GenA and GenB, which output key pairs $(\mathsf{sk}_\mathsf{A}, \mathsf{pk}_\mathsf{A})$ and $(\mathsf{sk}_\mathsf{B}, \mathsf{pk}_\mathsf{B})$. After running these algorithms, we define the two parties' PCF keys to be $(\mathsf{sk}_\mathsf{A}, \mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{B})$ and $(\mathsf{sk}_\mathsf{B}, \mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{B})$, respectively, and the rest of the definition follows the same way as before.

We sketch the constructions below. To distinguish between the different moduli involved, we refer to the modulus in the CRS (needed for the NIVOLE protocol) as $\tilde{N}$, and we refer to the modulus in the PCF as N.

Public-Key Silent VOLE. We replace the Gen algorithm of Construction 5.4 with the following: Both parties generate the first message of a non-interactive key exchange (NIKE) protocol (this will be used to derive the PRF key k_prf). Party 0 runs $(N, p, q) \leftarrow \mathsf{GenPQ}(1^\lambda)$ and computes d as in Gen, while party 1 picks a random x, and they run the protocol in Construction 6.3 with $n = 1$ (that is, they implement a single OLE). They both include the message from the NIOLE and NIKE protocol in their public key, while party 0 includes the modulus N too. Upon receiving the public key of the other party, they can compute their PCF key k_σ by completing the NIKE and NIOLE protocols. Note that we require y_0, y_1 to be a share of $x \cdot d$ over the integers, so this requires the modulus $\tilde{N}$ in the CRS for the NIOLE to be sufficiently large i.e., $\tilde{N} > N^3 2^\kappa$.

Public-Key Silent OT. We replace the Gen algorithm of Construction 5.6 with the following: As above both parties generates the first message of a NIKE. Party 0 runs $(N, p, q) \leftarrow \mathsf{GenPQ}(1^\lambda)$ and computes d as in Gen, while party 1 picks random bits $s_j \in \{0, 1\}$ for $j \in [\lambda]$, and they run the protocol in Construction 6.3 with $n = \lambda$ (that is, this is a "proper" instance of VOLE). They both include the message from the NIVOLE and NIKE protocol in their public key, while party 0 includes the modulus N too. Upon receiving the public key of the other party, they can compute their PCF key k_σ by completing the NIKE and NIOLE protocol. Note that we require y_0, y_1 to be a share of $s_j \cdot d$ over the integers, so this requires the modulus $\tilde{N}$ in the CRS for the NIVOLE to be sufficiently large i.e., $\tilde{N} > N 2^\kappa$.

Acknowledgements. We would like to thank Damiano Abram for pointing out a bug in Theorem 6.1, and the anonymous reviewers for their time and feedback.

This work was supported by: the Concordium Blockhain Research Center, Aarhus University, Denmark; the Carlsberg Foundation under the Semper Ardens Research Project CF18-112 (BCM); the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme under grant agreement No 669255 (MPCPRO) and grant agreement No 803096 (SPEC); a starting grant from Aarhus University Research Foundation.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001120C0085. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the

Defense Advanced Research Projects Agency (DARPA). Distribution Statement “A” (Approved for Public Release, Distribution Unlimited).

References

AIK09. Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Cryptography with constant input locality. *Journal of Cryptology*, (4), October 2009.

BCG⁺17. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, and Michele Orrù. Homomorphic secret sharing: Optimizations and applications. In *ACM CCS 2017*. ACM Press, October / November 2017.

BCG⁺19. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In *CRYPTO 2019, Part III*, LNCS. Springer, Heidelberg, August 2019.

BCG⁺20a. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Correlated pseudorandom functions from variable-density lpn. In *FOCS*, 2020.

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

BCGI18. Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In *ACM CCS 2018*. ACM Press, October 2018.

BCP03. Emmanuel Bresson, Dario Catalano, and David Pointcheval. A simple public-key cryptosystem with a double trapdoor decryption mechanism and its applications. In *ASIACRYPT 2003*, LNCS. Springer, Heidelberg, November / December 2003.

BG10. Zvika Brakerski and Shafi Goldwasser. Circular and leakage resilient public-key encryption under subgroup indistinguishability - (or: Quadratic residuosity strikes back). In *CRYPTO 2010*, LNCS. Springer, Heidelberg, August 2010.

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

BGI16a. Elette Boyle, Niv Gilboa, and Yuval Ishai. Breaking the circuit size barrier for secure computation under DDH. In *CRYPTO 2016, Part I*, LNCS. Springer, Heidelberg, August 2016.

BGI16b. Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *ACM CCS 2016*. ACM Press, October 2016.

BGI17. Elette Boyle, Niv Gilboa, and Yuval Ishai. Group-based secure computation: Optimizing rounds, communication, and computation. In *EUROCRYPT 2017, Part II*, LNCS. Springer, Heidelberg, April / May 2017.

BGI⁺18. Elette Boyle, Niv Gilboa, Yuval Ishai, Huijia Lin, and Stefano Tessaro. Foundations of homomorphic secret sharing. In *ITCS 2018*. LIPIcs, January 2018.

BGMM20. James Bartusek, Sanjam Garg, Daniel Masny, and Pratyay Mukherjee. Reusable two-round mpc from ddh. Cryptology ePrint Archive, Report 2020/170, 2020. https://eprint.iacr.org/2020/170.

BKS19. Elette Boyle, Lisa Kohl, and Peter Scholl. Homomorphic secret sharing from lattices without FHE. In *EUROCRYPT 2019, Part II*, LNCS. Springer, Heidelberg, May 2019.

BM90. Mihir Bellare and Silvio Micali. Non-interactive oblivious transfer and applications. In *CRYPTO'89*, LNCS. Springer, Heidelberg, August 1990.

BV11. Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *52nd FOCS*. IEEE Computer Society Press, October 2011.

Cle91. Richard Cleve. Towards optimal simulations of formulas by bounded-width programs. *Computational Complexity*, 1:91–105, 1991.

CS02. Ronald Cramer and Victor Shoup. Universal hash proofs and a paradigm for adaptive chosen ciphertext secure public-key encryption. In *EUROCRYPT 2002*, LNCS. Springer, Heidelberg, April / May 2002.

DGI$^+$19. Nico Döttling, Sanjam Garg, Yuval Ishai, Giulio Malavolta, Tamer Mour, and Rafail Ostrovsky. Trapdoor hash functions and their applications. In *CRYPTO 2019, Part III*, LNCS. Springer, Heidelberg, August 2019.

DGS03. Ivan Damgård, Jens Groth, and Gorm Salomonsen. The theory and implementation of an electronic voting system. In *Secure Electronic Voting*, pages 77–98, 2003.

DHRW16. Yevgeniy Dodis, Shai Halevi, Ron D. Rothblum, and Daniel Wichs. Spooky encryption and its applications. In *CRYPTO 2016, Part III*, LNCS. Springer, Heidelberg, August 2016.

DK07. Yvo Desmedt and Kaoru Kurosawa. A generalization and a variant of two threshold cryptosystems based on factoring. In *ISC 2007*, LNCS. Springer, Heidelberg, October 2007.

DKK18. Itai Dinur, Nathan Keller, and Ohad Klein. An optimal distributed discrete log protocol with applications to homomorphic secret sharing. In *CRYPTO 2018, Part III*, LNCS. Springer, Heidelberg, August 2018.

FGJS17. Nelly Fazio, Rosario Gennaro, Tahereh Jafarikhah, and William E. Skeith III. Homomorphic secret sharing from paillier encryption. In *ProvSec 2017*, LNCS. Springer, Heidelberg, October 2017.

Gen09. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *41st ACM STOC*. ACM Press, May / June 2009.

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

GIS18. Sanjam Garg, Yuval Ishai, and Akshayaram Srinivasan. Two-round MPC: Information-theoretic and black-box. In *TCC 2018, Part I*, LNCS. Springer, Heidelberg, November 2018.

GM82. Shafi Goldwasser and Silvio Micali. Probabilistic encryption and how to play mental poker keeping secret all partial information. In *14th ACM STOC*. ACM Press, May 1982.

IKNP03. Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *CRYPTO 2003*, LNCS. Springer, Heidelberg, August 2003.

KY02. Jonathan Katz and Moti Yung. Threshold cryptosystems based on factoring. In *ASIACRYPT 2002*, LNCS. Springer, Heidelberg, December 2002.

Pai99. Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT'99*, LNCS. Springer, Heidelberg, May 1999.

WYG$^+$17. Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical private queries on public data. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017*, pages 299–313, 2017.

A Function Secret Sharing

Function secret sharing [BGI15, BGI16b] is a way of splitting up a function f into several shares, or keys, f_i, such that individually each f_i hides the function, but can be locally evaluated on any public input x to give $f_i(x)$, such that $\sum_i f_i(x) = f(x)$.

In this work, we consider two-party FSS for classes of functions with domain $\{1, \dots, n\}$ and range $\mathbb{G}$, where $(\mathbb{G}, +)$ is an abelian group, and both $n, \mathbb{G}$ may depend on the security parameter λ. For consistency with the rest of the paper, we use subtractive reconstruction instead of the standard additive shares.

Definition A.1 (Function Secret Sharing). *A* function secret sharing *(FSS) scheme for an infinite family of functions $\mathcal{F} = \{f : [n] \to \mathbb{G}\}$ with input domain $[n]$ and output range an abelian group $(\mathbb{G}, +)$ (both possibly depending on f), 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 $f \in \mathcal{F}$, outputs a pair of keys (K_0, K_1);*
- $\mathsf{FSS.FullEval}(\sigma, K_\sigma)$*, given a key K_σ for party $\sigma \in \{0,1\}$, the full-domain evaluation algorithm outputs 2^n group elements $(y_\sigma^1, \dots, y_\sigma^n) \in \mathbb{G}^n$.*

The scheme should satisfy the following requirements:

- **Correctness:** *For any $f \in \mathcal{F}$ and $x \in [n]$, we have*

$$\Pr\left[\mathsf{FSS.Eval}(1, K_1, x) - \mathsf{FSS.Eval}(0, K_0, x) = f(x) \;\middle|\; (K_0, K_1) \leftarrow \mathsf{FSS.Gen}(1^\lambda, f)\right] = 1$$

- **Security:** *For any $\sigma \in \{0,1\}$, there exists a PPT simulator $\mathcal{S}$ such that for any sequence $f_\lambda \in \mathcal{F}$ of polynomial-size function descriptions, the distributions*

$$\{(\sigma, f_\lambda, K_\sigma) \mid (K_0, K_1) \leftarrow \mathsf{FSS.Gen}(1^\lambda, f_\lambda)\} \text{ and } \{(\sigma, f_\lambda, K_\sigma) \mid K_\sigma \leftarrow \mathcal{S}(1^\lambda, \mathsf{Leak}(f_\lambda))\}$$

are computationally indistinguishable.

In the constructions we use, the leakage function $\mathsf{Leak} : \{0,1\}^* \to \{0,1\}^*$ *is given by* $\mathsf{Leak}(f_\lambda) = (n, \mathbb{G})$*, namely it outputs a description of the input and output domains of f.*
We use $\mathsf{FSS.FullEval}(\sigma, K_\sigma)$ *to refer to an evaluation of the entire domain.*

We also require that outputs of the Eval algorithm are pseudorandom.

Definition A.2 (FSS with Pseudorandom Outputs). *An FSS scheme* $(\mathsf{FSS.Gen}, \mathsf{FSS.Eval})$ *for $\mathcal{F}$ has* pseudorandom outputs *if for any $\sigma \in \{0,1\}$, any sequence of polynomial-size function descriptions $f_\lambda \in \mathcal{F}$ and any $n = \mathsf{poly}(\lambda)$, the distribution*

$$\left\{(y^{(i)})_{i=1}^n \mid y^{(i)} \leftarrow \mathsf{FSS.Eval}(\sigma, K_\sigma, i), (K_0, K_1) \leftarrow \mathsf{FSS.Gen}(1^\lambda, f_\lambda)\right\}_{\lambda \in \mathbb{N}}$$

is computationally indistinguishable from the uniform distribution on $\mathbb{G}^n$.

Multi-point FSS. We consider FSS for the class of *multi-point functions* with output space $\mathbb{Z}_{N'}$. Each function in the class is specified by a domain size n, modulus N' and set of points $(x_i, y_i) \in [n] \times \mathbb{Z}_{N'}$, for $i = 1, \ldots, t$, and defined by

$$f(x) = \begin{cases} y_i & \text{if } x = x_i \\ 0 & \text{o.w.} \end{cases}$$

We can equivalently define a function in the class by a vector $\boldsymbol{e} \in \mathbb{Z}_{N'}^n$ with t non-zero coordinates, where we use the set of points (i, e_i), where e_i is a non-zero entry of $\boldsymbol{e}$.

FSS for multi-point functions can be constructed using distributed point functions [GI14, BGI16b], for which there are very efficient constructions based only on one-way functions. The size of each key for a t-point FSS with domain size n can be $O(t\lambda \log n)$ bits.

Dual-LPN Assumption. For a vector $\boldsymbol{e}$, we denote by $\mathsf{HW}(\boldsymbol{e})$ the number of non-zero entries of $\boldsymbol{e}$.

Assumption 3 (Dual-LPN Over $\mathbb{Z}_N$)**.** *Let $N, n, m, t \in \mathbb{Z}_N$ and $H \in \mathbb{Z}_N^{m \times n}$ (with $m < n$) be parameters that depend implicitly on the security parameter λ. The* dual learning parity with noise *assumption over $\mathbb{Z}_N$ states that the distribution*

$$(H, H\boldsymbol{e}), \qquad \boldsymbol{e} \leftarrow \mathbb{Z}_N^n \text{ such that } \mathsf{HW}(\boldsymbol{e}) = t$$

is computationally indistinguishable from $(H, \boldsymbol{u})$ where $\boldsymbol{u} \leftarrow \mathbb{Z}_N^m$.

When H is a uniform matrix, Dual-LPN is equivalent to the standard LPN assumption with a bounded number of samples, and also to the syndrome decoding problem for random linear codes via a search-to-decision reduction [AIK09]. For greater efficiency, one can use a more structured matrix using, say, quasi-cyclic codes, which allow computing $H\boldsymbol{e}$ with a quasi-linear number of arithmetic operations. In our construction, we assume that given a modulus N, the parties have some way of locally agreeing upon an H for which LPN is hard. This could be done, for instance, by taking some pre-agreed H' over the integers and reducing this modulo N, or by generating H as the output of a random oracle.

B HSS Variants

In this section we describe two variants of our HSS scheme: one from Paillier-ElGamal encryption [CS02, DGS03, BCP03], and one from the circular-secure encryption scheme due to Brakerski and Goldwasser [BG10].

B.1 HSS from Paillier-ElGamal

We show how to build HSS from a different encryption scheme — Paillier-ElGamal. This has the advantage of only requiring a public-key style setup, where each party publishes a public key, and each can then non-interactively derive their shared public key and their own evaluation key. We also require a common reference string containing N; however, this can be reused by many different pairs of parties running HSS.

Paillier-ElGamal Encryption. We use Paillier-ElGamal encryption [CS02, DGS03, BCP03], which is essentially ElGamal over the group $\mathbb{Z}^*_{N^2}$. In this setting, we consider N and g, sampled in Gen for completeness, to be public parameters available to all algorithms. In our HSS scheme, these will be part of a common reference string (CRS).

```
$\mathsf{PaillierEG.Gen}(1^\lambda)$ :
    1. Sample $(N, p, q) \leftarrow \mathsf{GenPQ}(1^\lambda)$.
    2. Sample a random $g' \leftarrow [N^2]$, and let $g = (g')^{2N} \bmod N^2$.
    3. Sample a random $d \leftarrow [N^2]$.
    4. Output $\mathsf{pk} = g^d \bmod N^2$, $\mathsf{sk} = d$.
$\mathsf{PaillierEG.Enc}(\mathsf{pk}, x)$ :
    1. Sample a random $r \leftarrow [N]$.
    2. Output $\mathsf{ct} = (g^r, \mathsf{pk}^r(1+N)^x)$.
$\mathsf{PaillierEG.Dec}(\mathsf{sk}, \mathsf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1))$ :
    1. Let $\mathsf{ct}' = \mathsf{ct}_1(\mathsf{ct}_0)^{-d} \pmod{N^2}$.
    2. Output $x = \frac{\mathsf{ct}'-1}{N}$.
```

The security of this scheme can be based only on the DCR assumption (Assumption 1); it can be shown as a simple corollary of Lemma 2.1 with $l = 2$, $\mathbb{G} = \mathbb{Z}^*_{N^2}$, $\mathbb{G}_R$ being the group of Nth residues modulo N^2, and $\mathbb{G}_M$ being the group generated by $(1+N)$.

We take an adversary $\mathcal{A}_{\mathsf{PEG}}$ who can break the semantic security of the Paillier-ElGamal encryption scheme, and build an adversary $\mathcal{A}_{\mathsf{IV}}$ who can win the interactive vector game. $\mathcal{A}_{\mathsf{IV}}$ obtains (g_1, g_2) from its challenger $\mathcal{C}_{\mathsf{IV}}$; it sets the generator $g = g_2^2$, and forwards $\mathsf{pk} = g_1^2$ to $\mathcal{A}_{\mathsf{PEG}}$. (Because N is a product of safe primes, g is a generator of the cyclic subgroup of $2N$th residues modulo N^2, so pk is distributed appropriately.) Upon receiving a challenge $x_0, x_1 \in \mathbb{G}_M^2$, it sends $(a_1 = 0, a_2 = x_1)$ to $\mathcal{C}_{\mathsf{IV}}$. It forwards the output to $\mathcal{A}_{\mathsf{PEG}}$, and forwards the output of $\mathcal{A}_{\mathsf{PEG}}$ to $\mathcal{C}_{\mathsf{IV}}$ as b'. If $\mathcal{C}_{\mathsf{IV}}$ used $b = 1$, then the ciphertext sent to $\mathcal{A}_{\mathsf{PEG}}$ would be a valid encryption of x_1. Otherwise, its totally independent of both messages. (By a standard argument, if $\mathcal{A}_{\mathsf{PEG}}$ is able to distinguish an encryption of x_0 from an encryption of x_1, it should be able to distinguish an encryption of one of those from something independent of both messages.)

Building HSS. Using Paillier-ElGamal, we can build an HSS scheme which has the advantage of not requiring any correlated randomness. We still require setup in the form of the modulus N^2 and generator g known to all parties, and a public key infrastructure (PKI); however, there is no need to share and encrypt digits of the decryption key d in a trusted way, since d is now not constrained to have any specific value mod N or $\phi(N)$, and so can be generated in a distributed, non-interactive way by the two participating parties.

Note that in the previous construction, during setup, our parties acquired subtractive shares of the digits of d, and encryptions of those same digits. However, it is not actually important that the shares and encryptions correspond to actual digits of d: these must simply be small values (relative to N) such that a known linear combination of those values gives d (over the integers). This property will be useful later.

We next describe the setup and multiplication steps of HSS from Paillier-ElGamal. All the other steps remain the same, with small caveats: all instances of $\mathsf{Paillier.Enc}$ are replaced with $\mathsf{PaillierEG.Enc}$, and during Input, $X^{(i)}$ is computed as $(g^{r_i}(D_0^{(i)})^x, \mathsf{pk}_{\mathsf{PEG}}^{r_i}(D_1^{(i)})^x)$, which corresponds to re-randomization and homomorphic multiplication by a known value in this encryption scheme.

Setup: Now, setup is split into several steps: the setup of a common reference string (CRS) containing N and g, the setup of a PKI, and the use of the PKI to obtain the keys ek_σ used in HSS.

CRS: N sampled via GenPQ, and g sampled as per $\mathsf{PaillierEG.Gen}$, are published as a CRS. Additionally, a PRF key $\mathsf{k}_{\mathsf{prf}}$ is sampled and published.

PKI: Each party σ:

- Chooses s_σ uniformly at random from $[N^2]$. Let $d_\sigma^{(0)}, \ldots, d_\sigma^{(\ell-1)}$ denote the digits of d_σ in base B_{sk}.
- Lets $\mathsf{pk}_{\mathsf{PEG},\sigma} = g^{d_\sigma} \pmod{N^2}$, and produces encryptions $P_\sigma^{(i)} = (P_{\sigma,0}^{(i)}, P_{\sigma,1}^{(i)}) = \mathsf{PaillierEG.Enc}(\mathsf{pk}_{\mathsf{PEG},\sigma}, d_\sigma^{(i)}; r_\sigma^{(i)})$ (using fresh randomness $r_\sigma^{(i)}$).
- Sets its public key $\mathsf{pk}_\sigma = (\mathsf{pk}_{\mathsf{PEG},\sigma}, P_\sigma^{(0)}, \ldots, P_\sigma^{(\ell-1)})$, and its secret key $\mathsf{sk}_\sigma = (d_\sigma, r_\sigma^{(0)}, \ldots, r_\sigma^{(\ell-1)})$.

Obtaining pk and ek_σ: We define d as $d = d_0 + d_1$, and $d^{(i)}$ as $d^{(i)} = d_0^{(i)} + d_1^{(i)}$. These are no longer digits base B_{sk}, since it could be that $d^{(i)} \geq B_{\mathsf{sk}}$. However, it is still true that

$$d = \sum_{i=0}^{\ell-1} B_{\mathsf{sk}}^i d^{(i)}.$$

- Both parties compute $\mathsf{pk}_{\mathsf{PEG}} = \mathsf{pk}_{\mathsf{PEG},0}\mathsf{pk}_{\mathsf{PEG},1} \pmod{N^2}$.
- To obtain the sharings of the pseudo-digits $d^{(i)}$ of d, party 1 lets its subtractive share $\langle d^{(i)} \rangle_1^{(\mathbb{Z})}$ be $F_{\mathsf{k}_{\mathsf{prf}}}^{(2^\kappa B_{\mathsf{sk}})}(i) + d_1^{(i)}$, where $F^{(2^\kappa B_{\mathsf{sk}})}$ is a PRF outputting elements of $[2^\kappa B_{\mathsf{sk}}]$. Party 0 lets its subtractive share $\langle d^{(i)} \rangle_0^{(\mathbb{Z})}$ be $F_{\mathsf{k}_{\mathsf{prf}}}^{(2^\kappa B_{\mathsf{sk}})}(i) - d_0^{(i)}$.
- Obtaining encryptions of digits of the joint key d under that same shared key is a bit more tricky. We follow the blueprint of Boyle *et al.* [BGI17]. Party σ can obtain encryptions of $d_\sigma^{(i)}$ under $d = d_0 + d_1$, by taking

$$D_\sigma^{(i)} = (D_{\sigma,0}^{(i)}, D_{\sigma,1}^{(i)}) = \mathsf{PaillierEG.Enc}(\mathsf{pk}_{\mathsf{PEG}}, d_\sigma^{(i)}; r_\sigma^{(i)}).$$

It can then obtain encryptions of $d_{1-\sigma}^{(i)}$ under the same key by taking

$$D_{1-\sigma}^{(i)} = (D_{1-\sigma,0}^{(i)}, D_{1-\sigma,1}^{(i)}) = (P_{1-\sigma,0}^{(i)}, P_{1-\sigma,1}^{(i)}(P_{1-\sigma,0}^{(i)})^{d_\sigma}).$$

Note that both parties will obtain the *same* ciphertexts. They can then both compute the same encryption of $d^{(i)}$ as

$$D^{(i)} = (D_{1,0}^{(i)} D_{0,0}^{(i)}, D_{1,1}^{(i)} D_{0,1}^{(i)}).$$

Multiplication. Multiplication is exactly as in the regular Paillier version of HSS, with the only difference being how the inputs to DDLog are obtained. Letting $X = (X_0, X_1)$ be the ciphertext, the subtractive shares of z are now obtained as

$$\langle z \rangle_\sigma^{(N)} = \mathsf{DDLog}_N((X_1)^{\langle y \rangle_\sigma^{(\mathbb{Z})}} (X_0)^{-\langle yd \rangle_\sigma^{(\mathbb{Z})}}) + F_{\mathsf{k}_{\mathsf{prf}}}^{(N)}(\mathsf{id}) \pmod{N}.$$

To see that these inputs to DDLog are divisive shares of $(1+N)^{xy} \pmod{N^2}$, recall that $X = (X_0, X_1) = (g^r, \mathsf{pk}_{\mathsf{PEG}}^r (1+N)^x)$.

$$\begin{aligned}
\frac{(X_1)^{\langle y \rangle_1^{(\mathbb{Z})}} (X_0)^{-\langle yd \rangle_1^{(\mathbb{Z})}}}{(X_1)^{\langle y \rangle_0^{(\mathbb{Z})}} (X_0)^{-\langle yd \rangle_0^{(\mathbb{Z})}}} &= \frac{(\mathsf{pk}_{\mathsf{PEG}}^r (1+N)^x)^{\langle y \rangle_1^{(\mathbb{Z})}} (g^r)^{-\langle yd \rangle_1^{(\mathbb{Z})}}}{(\mathsf{pk}_{\mathsf{PEG}}^r (1+N)^x)^{\langle y \rangle_0^{(\mathbb{Z})}} (g^r)^{-\langle yd \rangle_0^{(\mathbb{Z})}}} \\
&= (\mathsf{pk}_{\mathsf{PEG}}^r (1+N)^x)^{\langle y \rangle_1^{(\mathbb{Z})} - \langle y \rangle_0^{(\mathbb{Z})}} (g^r)^{-\langle yd \rangle_1^{(\mathbb{Z})} + \langle yd \rangle_0^{(\mathbb{Z})}} \\
&= (\mathsf{pk}_{\mathsf{PEG}}^r (1+N)^x)^y (g^r)^{-yd} \\
&= (1+N)^{xy}.
\end{aligned}$$

The subtractive shares of $zd^{(i)}$ are obtained analogously.

Security. Correctness is straightforward to see, as in the Paillier-based HSS construction. For security, we can use the same argument as before, with the caveat that in all games, we allow party σ to run its setup correctly, so game 1 is skipped. In hybrid 2, party $1-\sigma$ publishes encryptions of 0; in hybrid 3, as before, we always encrypt x_0.

B.2 HSS from Circular-Secure Paillier

Circular-Secure Paillier-Based Encryption. In order to avoid making assumptions about circular security, we would sometimes prefer to use a scheme which was *proven* circular secure. One such scheme, based on the DCR assumption (Assumption 1), was introduced by Brakerski and Goldwasser [BG10]. It uses an additional security parameter l which parameterizes the KDM-security of the scheme, and is polynomially related to λ. As before, we consider N and $g_1, \ldots, g_l$, sampled in Gen, to be public parameters available to all algorithms.

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

1. Sample $(N, p, q) \leftarrow \mathsf{GenPQ}(1^\lambda)$.

2. Sample $g_1, \dots, g_l$ as random Nth residues in $\mathbb{Z}^*_{N^2}$.
3. Sample a length-l bit vector $d = (d^{(1)}, \dots, d^{(l)}) \in \{0,1\}^l$.
4. Output $\mathsf{pk} = g_0 = \prod_{i=1}^{l} g_i^{d^{(i)}} \bmod N^2$, $\mathsf{sk} = d$.

$\mathsf{BG.Enc}(\mathsf{pk}, x)$:

1. Sample a random $r \leftarrow [N]$.
2. Output $\mathsf{ct} = (g_1^r, \dots, g_l^r, g_0^r(1+N)^x)$.

$\mathsf{BG.Dec}(\mathsf{sk}, \mathsf{ct} = (\mathsf{ct}_1, \dots, \mathsf{ct}_l, \mathsf{ct}_0))$:

1. Let $\mathsf{ct}' = (\prod_{i=1}^{l} \mathsf{ct}_i^{-d^{(i)}})\mathsf{ct}_0 \pmod{N^2}$.
2. Output $x = \frac{\mathsf{ct}'-1}{N}$.

We can use the circular-secure encryption scheme by Brakerski and Goldwasser [BG10] to avoid having to assume Paillier is circular secure. The downside is that ciphertexts are much larger; they contain a number l of $\mathbb{Z}^*_{N^2}$ elements, where l is polynomial in the security parameter λ.

On the plus side, the secret key in the BG scheme is already used in terms of its individual bits, and it is possible to obtain a ciphertext that decrypts to any one of those bits from the public key alone. (Note that this ciphertext will be distributed differently from an honestly generated one; however, decryption will work just the same.) We create a ciphertext that decrypts to $d^{(i)}$ by taking

$$\mathsf{ct}_i = (g_1^r, \dots, g_{i-1}^r, (1+N)^{-1}g_i^r, g_{i+1}^r, \dots, g_l^r, g_0^r).$$

We can confirm that decryption outputs $d^{(i)}$ as follows:

$$\begin{aligned}\mathsf{ct}'_i = (\prod_{j=1}^{l} \mathsf{ct}_j^{-d^{(j)}})\mathsf{ct}_0 \pmod{N^2} &\equiv (\prod_{j=1}^{l} g_j^{d^{(j)}})^{-r}(1+N)^{d^{(i)}} g_0^r \pmod{N^2} \\ &\equiv g_0^{-r}(1+N)^{d^{(i)}} g_0^r \pmod{N^2} \\ &\equiv (1+N)^{d^{(i)}} \pmod{N^2}.\end{aligned}$$

Finally, $\frac{\mathsf{ct}'-1}{N} = d^{(i)}$. Obtaining a ciphertext that decrypts to a multiple of $d^{(i)}$ — say, $xd^{(i)}$ — can be similarly done by multiplying g_i^r by $(1+N)^{-x}$.

We next go over the setup and multiplication phases of BG-based HSS. As with $\mathsf{PaillierEG}$-based HSS, the other phases are largely unchanged, except in trivial ways.

Setup:

CRS: N sampled via GenPQ, and $g_1, \dots, g_l$ sampled as per $\mathsf{BG.Gen}$, are published as a common reference string. Additionally, a PRF key $\mathsf{k_{prf}}$ is sampled and published. (For notational simplicity, we re-use $\mathsf{k_{prf}}$ for multiple PRFs; assume that this key contains multiple keys, and the appropriate one is always used.)

PKI: Each party σ:

- Chooses $d_\sigma = (d_\sigma^{(1)}, \dots, d_\sigma^{(l)})$ uniformly at random from $\{0,1\}^l$.

- Lets $\mathsf{pk}_\sigma = \prod_{i=1}^{l} g_i^{d_\sigma^{(i)}} \bmod N^2$. Note that in this scheme, there is no need to publish encryptions of the bits of d_σ, as those can be publicly computed from the public key, as described above.

Obtaining pk and ek_σ:

- Both parties compute $\mathsf{pk} := \mathsf{pk}_0\mathsf{pk}_1 \pmod{N^2}$. We define $d^{(i)}$ as $d^{(i)} := d_0^{(i)} + d_1^{(i)}$. Note that, by inspection, encryptions to pk will correctly decrypt using $d = (d^{(1)}, \ldots, d^{(l)})$.
- To obtain the sharings of the pseudo-digits $d^{(i)}$ of d, party 1 lets its subtractive share $\langle d^{(i)} \rangle_1^{(\mathbb{Z})}$ be $F_{\mathsf{k}_{\mathsf{prf}}}^{(2^\kappa)}(i) + d_1^{(i)}$, where $F^{(2^\kappa)}$ is a PRF outputting elements of $[2^\kappa]$. Party 0 lets its subtractive share $\langle d^{(i)} \rangle_0^{(\mathbb{Z})}$ be $F_{\mathsf{k}_{\mathsf{prf}}}^{(2^\kappa)}(i) - d_0^{(i)}$.
- Both parties can obtain encryptions of $d^{(i)}$ under pk by taking $r = F_{\mathsf{k}_{\mathsf{prf}}}^{(N)}(i)$ and computing

$$D^{(i)} = (g_1^r, \ldots, g_{i-1}^r, (1+N)^{-1} g_i^r, g_{i+1}^r, \ldots, g_l^r, \mathsf{pk}^r).$$

Multiplication: As in the $\mathsf{PaillierEG}$-based HSS, multiplication is done similarly to the way it is done in the regular Paillier-based HSS, with several differences. First, since d is only ever used bit by bit, the shares $\langle yd \rangle_\sigma^{(\mathbb{Z})}$ need never be assembed. Second, the inputs to DDLog are obtained differently. Letting $X = (X_1, \ldots, X_l, X_0)$ be the ciphertext, the subtractive shares of z are now obtained as

$$\langle z \rangle_\sigma^{(N)} = \mathsf{DDLog}_N(\mathsf{ct}'_\sigma) + F_{\mathsf{k}_{\mathsf{prf}}}^{(N)}(\mathsf{id}) \pmod{N},$$

where

$$\mathsf{ct}'_\sigma = (X_0)^{\langle y \rangle_\sigma^{(\mathbb{Z})}} \left(\prod_{i=1}^{l} X_i^{-\langle y d^{(i)} \rangle_\sigma^{(\mathbb{Z})}} \right) \pmod{N^2}.$$

To see that $\mathsf{ct}'_0, \mathsf{ct}'_1$ are divisive shares of $(1+N)^{xy} \pmod{N^2}$, recall that $(\prod_{i=1}^{l} X_i^{-d^{(i)}})X_0 \pmod{N^2} = (1+N)^x$.

$$\begin{aligned}
\frac{(X_0)^{\langle y \rangle_1^{(\mathbb{Z})}} \left(\prod_{i=1}^{l} X_i^{-\langle y d^{(i)} \rangle_1^{(\mathbb{Z})}} \right)}{(X_0)^{\langle y \rangle_0^{(\mathbb{Z})}} \left(\prod_{i=1}^{l} X_i^{-\langle y d^{(i)} \rangle_0^{(\mathbb{Z})}} \right)} &\equiv X_0^y \left(\prod_{i=1}^{l} X_i^{-y d^{(i)}} \right) && \pmod{N^2} \\
&\equiv \big(X_0 \prod_{i=1}^{l} X_i^{-d^{(i)}} \big)^y && \pmod{N^2} \\
&\equiv (1+N)^{xy}. && \pmod{N^2}
\end{aligned}$$

The subtractive shares of $zd^{(i)}$ are obtained analogously.

Security. Correctness is straightforward to see, as in the Paillier-ElGamal construction. The security of the construction follows in the same way as the semantic security of the BG cryptosystem [BG10].