Overdrive2k: Efficient Secure MPC over $\mathbb{Z}_{2^k}$ from Somewhat Homomorphic Encryption

Emmanuela Orsini[1[0000−0002−1917−1833]], Nigel P. Smart[1,2[0000−0003−3567−3304]], and Frederik Vercauteren[1[0000−0002−7208−9599]]

[1] imec-COSIC, KU Leuven, Leuven, Belgium.
[2] University of Bristol, Bristol, UK.
emmanuela.orsini@kuleuven.be, nigel.smart@kuleuven.be, frederik.vercauteren@kuleuven.be

Abstract. Recently, Cramer et al. (CRYPTO 2018) presented a protocol, SPDZ2k, for actively secure multiparty computation for dishonest majority in the pre-processing model over the ring $\mathbb{Z}_{2^k}$, instead of over a prime field $\mathbb{F}_p$. Their technique used oblivious transfer for the pre-processing phase, more specifically the MASCOT protocol (Keller et al. CCS 2016). In this paper we describe a more efficient technique for secure multiparty computation over $\mathbb{Z}_{2^k}$ based on somewhat homomorphic encryption. In particular we adapt the Overdrive approach (Keller et al. EUROCRYPT 2018) to obtain a protocol which is more like the original SPDZ protocol (Damgård et al. CRYPTO 2012). To accomplish this we introduce a special packing technique for the BGV encryption scheme operating on the plaintext space defined by the SPDZ2k protocol, extending the ciphertext packing method used in SPDZ to the case of $\mathbb{Z}_{2^k}$. We also present a more complete pre-processing phase for secure computation modulo 2^k by adding a new technique to produce shared random bits.

1 Introduction

The last ten years have seen a remarkable advance in practical protocols and systems to perform secure Multi-Party Computation (MPC). A major pillar of this advance has been in the case of a dishonest majority, in which one can obtain so-called active-security-with-abort. In this situation one is interested in MPC protocols for n parties, where $n \geq 2$, which are practical even for values of n in the tens (or potentially hundreds). Following the initial work of Bendlin et al. [BDOZ11], the main breakthrough came with the SPDZ protocol by Damgård et al. [DPSZ12] and its improvements, e.g. [DKL+13]. This protocol works in an offline/online manner over finite fields. In the offline phase, function-independent pre-processing is performed, typically to generate Beaver triples [Bea92]. In the online phase, this pre-processing is consumed as the desired function is securely evaluated. Active security is obtained by parties not only sharing data, but also sharing a linear MAC on this data together with a share of the MAC key. Validation of correct behavior is done via a MAC check protocol which verifies that all opened data shares and all privately held MAC and key shares are consistent.

Over the previous decade there has been a multitude of methods to produce the offline data needed for the SPDZ protocol. The initial protocol, [BDOZ11], in this family used a linearly homomorphic encryption scheme, and pairwise zero-knowledge proofs to correctly generate the offline data. This approach works well for a small number of parties, but does not scale for larger values of n. The linearly homomorphic encryption method was replaced in the SPDZ paper [DPSZ12] by a level-one Somewhat Homomorphic Encryption (SHE) scheme. The main efficiency improvement came from using the BGV [BGV12] SHE scheme, and making extensive use of the packing technique of Smart and Vercauteren [SV14]. On the other hand, the main inefficiency was that, to

obtain active security, one needed to prove knowledge of plaintexts and correctness of ciphertexts. These zero-knowledge proofs can (currently) only be done in a non-tight manner, and with a relatively large soundness error. This inefficiency in soundness error is usually overcome using standard amortization techniques. In [DKL+13], a different zero-knowledge proof was utilized which, whilst asymptotically better than that of [DPSZ12], turned out to be impractical.

Attention then switched to Oblivious Transfer (OT) based pre-processing, such as the Tiny-OT [NNOB12] and MASCOT [KOS16] protocols. Finally, in the last two years attention switched back to homomorphic encryption based protocols with the Overdrive paper by Keller et al. [KPR18]. Overdrive gives two variants of the SPDZ protocol: Low-Gear and High-Gear. The Low-Gear variant uses the original linearly homomorphic encryption based methodology of [BDOZ11], but implements it using a level-zero LWE-based SHE scheme (in this instance, BGV). The resulting method is very efficient for a small number of parties due to the inherent packing one can use. For two parties the authors of [KPR18] suggest it is six to fourteen times faster than MASCOT [KOS16][Table 2 and 4] (with the precise figure depending on the network latency).

In the High-Gear variant of Overdrive the authors return to the original zero-knowledge proofs of [DPSZ12], and make improvements by both reducing the lack of tightness (although not totally eliminating it), and enabling batching of the zero-knowledge proofs across all n parties on top of the usual amortization techniques. This last optimization results in an immediate improvement by a factor of n. Thus, for larger values of n, High-Gear is currently the best method for SPDZ-family style pre-processing over finite fields. In [KPR18][Table 2 and 4] the High-Gear protocol for two parties is shown to be up to six times faster than MASCOT (again depending on the network latency); whilst for 100 parties, [KPR18][Table 7] implies a 13 fold improvement over MASCOT.

Very recently a new protocol was introduced to the SPDZ family in the work of Cramer et al. [CDE+18], referred to there, and here, by the shorthand SPDZ2k. Instead of defining MPC protocols over a finite field, SPDZ2k defines MPC protocols over a ring $\mathbb{Z}_{2^k}$. Designing MPC protocols over rings $\mathbb{Z}_{2^k}$ is potentially useful in many applications, and could significantly simplify implementations, such as in the case of evaluations of functions containing comparisons and bit-wise operations. To enable computation over such rings, SPDZ2k makes changes to the way MACs are held, and verified, and more generally to how the pre-processing works. The paper [CDE+18] bases its pre-processing on a MASCOT-style methodology, hence the two protocols are inherently very similar. Indeed, recent work by Dåmgard et al. [DEF+19] implemented the SPDZ2k protocol showing that its performance is comparable to the MASCOT one.

Establishing whether an efficient pre-processing for MPC over $\mathbb{Z}_{2^k}$ can be provided via homomorphic encryption was left as an open problem by the authors of SPDZ2k. A quick naive investigation seems to imply that this is a non-starter. The main reason the SHE-based approach (either Low-Gear or High-Gear) is efficient is in the possibility of packing data into ciphertexts and performing many operations in parallel. For SPDZ over finite prime fields one selects the underlying ring in BGV (of degree N) to completely split over the finite field, thus one obtains N-fold parallelism. When extending the SHE schemes to work with a plaintext modulus of 2^k, instead of a prime p, the packing capacity decreases dramatically and one cannot approach anything like N-fold parallelism.

Our Contribution. In this paper we revisit the idea of using a SHE-based pre-processing, i.e. Overdrive-based, for the SPDZ2k family. We show that the above naive analysis, which would discount its applicability, is actually wrong.

Our *first contribution* is a new packing methodology which is particularly tailored to the pre-processing phase of SPDZ2k. In particular, we obtain (roughly speaking) a $N/5$ fold parallelism for High-Gear when mapped to working modulo 2^k. Since the High-Gear protocol is the state-of-the-art for the SPDZ family protocols in terms of efficiency for large numbers of parties, we focus our work on the High-Gear of Overdrive[3].

Using our new packing technique comes with difficulties. The main issue is that the packing for level-zero ciphertexts of a plaintext message is different from the packing used at level one. Thus there is a need to modify the distributed decryption procedure in one important case, namely when one needs to obtain a fresh encryption of the underlying plaintext rather than an additive secret sharing of it. This in turn raises another problem: the distributed decryption protocol requires pairs of ciphertexts with special properties associated to the packing. A party needs to generate two ciphertexts, one at level zero and one at level one, which encrypt the same value, but with different packings. Since parties could be adversarial, this means that we also need to adapt the zero-knowledge proofs associated with the High-Gear protocol to enable such pairs of ciphertexts to be produced correctly. Some of our amortized zero-knowledge proofs need to prove a more complex statement associated to our packing tecniques, with an overall estimated factor $2/3$ loss in performance compared to HighGear.

Given that Overdrive is up to fourteen times faster that MASCOT, depending on the number of parties, and that MASCOT and SPDZ2k perform very similarly, we expect that our protocol is up to two times more efficient than the OT-based protocols in the two party setting. As the number of parties grows this gap will increase. Whilst these only indicate rough expected performance figures, we give a more concrete estimation of the communication complexity of our protocol in Section 7.

Our *second contribution* is in the construction of a more complete preprocessing phase for SPDZ-like protocols modulo 2^k, with active security in the dishonest majority setting. Other than a protocol for producing multiplication triples, we show how to efficiently produce random shared bits in the SPDZ2k framework using a trick similar to the one used in the SPDZ protocol over $\mathbb{F}_p$. Protocols over fields make use of the squaring operation over finite fields of odd characteristic which is a 2-to-1 map, whereas, modulo 2^k, this operation is a 4-to-1 map. We show a simple trick that permits to use essentially the same technique used mod p in the modulo 2^k setting.[4]

Related works. Recently, we have seen a renewed interest in secure computation protocols over rings. Besides the well-known protocols given in [CFIK03] and [BLW08] that are restricted to the honest-majority case, and the SPDZ2k protocol mentioned above, there are several new protocols that have appeared in the last couple of years. Araki et al. [AFL+16] recently improved the efficiency of Sharemind [BLW08] with active security but in the honest-majority setting. In [DOS18], Damgård et al. describe a compiler that transforms a semi-honest protocol with t corruptions into a maliciously secure protocol with a smaller number of corruptions, i.e. $\sqrt{t} < n/2$, where n is the total number of parties. The work of Catalano et al. [CRFG19] uses the Joye-Libert homomorphic cryptosystem to design a maliciously secure two-party protocol for the pre-processing phase of SPDZ2k. Rathee et al. [RSS19] described a protocol, again in the 2-party case, based on RLWE-based additively homomorphic encryption and with passive security. This protocol is very efficient,

[3] Whilst writing this paper the TopGear [BCS19] variant of High-Gear was published on e-print. This essentially allows the High-Gear protocol to be run at higher security levels for roughly the same performance. The TopGear improvements *cannot* be applied directly to our work, since the zero-knowledge proofs here require challenge spaces to be in $\mathbb{F}_q$ to ensure correctness.

[4] A similar trick for random shared bit generation is described in a concurrent and independent work [DEF+19].

but extending it to the malicious setting would require expensive zero-knowledge proof of correct multiplication. In [DEF+19], SPDZ2k has been implemented showing that the efficiency of the OT-based protocol over the ring $\mathbb{Z}_{2^k}$ is comparable with MASCOT, i.e. the most efficient OT-based protocol over finite fields. However, like in MASCOT, the communication complexity , that is most of the time the main bottleneck of secure MPC protocols, is much higher than in Overdrive. More precisely, this work shows that SPDZ2k is only slightly slower than Overdrive in a LAN setting with a small number of parties, but several times slower in a WAN setting and when the number of parties increases. It is there left as future research the construction of a SHE-based pre-processing for secure computation over rings to close the performance gap between SPDZ2k and Overdrive.

2 Preliminaries

In this section we introduce some important notation, describe the security model, recap on the SPDZ2k paper's requirements for the offline phase [CDE+18], plus the necessary background on the BGV Somewhat Homomorphic Encryption (SHE) scheme [BGV12]. By way of notation we let $a \leftarrow A$ denote randomly assigning a value a from a set A, where we assume a uniform distribution on A. If A is an algorithm, we let $a \leftarrow A$ denote assignment of the output, where the probability distribution is over the random coins of A; we also let $a \leftarrow b$ be a shorthand for $a \leftarrow \{b\}$, i.e. to denote normal variable assignment. We denote by $[d]$ the set of integers $\{1, \ldots, d\}$.

Security Model. We prove security of our protocols in the universal composition (UC) framework of Canetti [Can01], and assume familiarity with this. Our protocols work with n parties, $P_1, \ldots, P_n$, and we consider security against malicious, static adversaries, i.e. corruption may only take place before the protocols start, corrupting up to $n-1$ parties. Informally, when we say that a protocol Π securely implements a functionality $\mathcal{F}$ with computational (resp. statistical) security parameter κ (resp. s), our theorems guarantee that the advantage of any environment $\mathcal{Z}$ in distinguishing the ideal and real executions is in $O(2^{-\kappa})$ (resp. $O(2^{-s})$).

In some of our protocols we will need a coin-tossing functionality $\mathcal{F}_{\mathsf{Rand}}$, which given a set $\mathcal{D}$, outputs a uniformly random element r from $\mathcal{D}$. This functionality can be efficiently implemented in the random oracle model as described in [CDE+18].

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

On input $(\mathsf{Rand}, \mathcal{D})$ from all parties, sample $r \leftarrow \mathcal{D}$, and output r to all parties.

Figure 1. Coin-tossing functionality

2.1 The SPDZ2k Protocol

The SPDZ2k protocol [CDE+18] is parametrized by two integers k and s, where k defines the modulus 2^k over which the MPC protocol will run, and s is a statistical security parameter, for simplicity of exposition we will set $t = k + s$. For the reader who is new to the SPDZ2k protocol think of $k = s = 64$. As we are mainly focusing on the offline phase our complexity does not depend on whether $k < s$ or $k \geq s$, it only depends on the value of $t = k + s$.

The protocol performs MPC over the underlying ring $\mathbb{Z}_{2^k}$, however each value $x \in \mathbb{Z}_{2^k}$ is secret shared amongst the n parties via values $[x]_i \in \mathbb{Z}_{2^t}$, such that $x = \sum_{i=1}^{n} [x]_i \pmod{2^k}$. By abusing notation we also think of x as the sum $\sum_{i=1}^{n} [x]_i \pmod{2^t}$, since in the main SPDZ2k online protocol the upper s bits of x will be ignored.

Sometimes we will use $[x]_i$ to denote additive sharings of values $x \in \mathbb{Z}_{2^t}$, and sometimes with domains different from $\mathbb{Z}_{2^t}$. We will explicitly point this out when we do such alterations to the basic sharing.

Each of the n parties also holds a share $[\alpha]_i \in \mathbb{Z}_{2^s}$ of a global MAC key $\alpha = \sum_{i=1}^{n} [\alpha]_i \pmod{2^t}$. The global MAC key is used to authenticate the shares held by a party, in particular each party holds a value $[\gamma_x]_i = [\alpha \cdot x]_i \in \mathbb{Z}_{2^t}$ such that

$$\gamma_x = \sum_{i=1}^{n} [\alpha \cdot x]_i = \alpha \cdot x \pmod{2^t}.$$

A secret value $x \in \mathbb{Z}_{2^t}$ shared in this way is represented by $\langle x \rangle = \{[x]_i, [\gamma_x]_i\}_{i \in [n]}$, and we let $\langle x \rangle_i$ denote the pair of values $([x]_i, [\alpha \cdot x]_i)$ held by party P_i in this sharing.

Using this secret sharing scheme any *linear* function can be computed locally by the parties, i.e. without any interaction. This is done using the method in Figure 2. We denote the process of executing this operation for a specific linear function as

$$\langle y \rangle \leftarrow c_0 + \sum_{i=1}^{k} c_i \cdot \langle x_i \rangle.$$

Procedure LinearFuncShares

This procedure allows the computation of an arbitrary linear function $y = c_0 + \sum_{j=1}^{k} c_j \cdot x_j \mod 2^t$ given public inputs $c_0, c_1, \ldots, c_k$ and the parties shares $\langle x_j \rangle_i = \{[x_j]_i, [\gamma_{x_j}]_i\}_{j=1}^{k}, i \in [n]$. The output are the shares of $\langle y \rangle$.

1. Each $P_i, i \neq 1$, sets $[y]_i = \sum_{j=1}^{k} c_j \cdot [x_j]_i \pmod{2^t}$
2. Party P_1 sets $[y]_1 = c_0 + \sum_{j=1}^{k} c_j \cdot [x_j]_1 \pmod{2^t}$
3. Each party P_i sets $[\alpha \cdot y]_i = [\alpha]_i \cdot c_0 + \sum_{j=1}^{k} c_j \cdot [\alpha \cdot x_j]_i \pmod{2^t}$
4. Each party P_i sets $\langle y \rangle_i = \{[y]_i, [\alpha \cdot y]_i\}$

Figure 2. Procedure to locally compute linear functions on shares

To perform non-linear operations the SPDZ2k protocol makes use of the offline-online paradigm. In the offline phase various generic pre-processed data items are produced which allow the online phase to proceed as a sequence of linear functions and opening operations. Each opening operation in the online phase needs to be checked for consistency, which can be done via the method introduced in [CDE+18] (which we recap on in the Appendix B). The overall protocol achieves actively secure MPC with abort, with a statistical error probability of roughly $2^{-s+\log_2 s}$ (see [CDE+18][Lemma 1] for more details).

2.2 The BGV SHE Scheme and Associated Number Theory

In this section we outline the details of what we require of the BGV encryption scheme. Most of the details can be found in [BGV12, GHS12b, GHS12c, GHS12a], although we will only require a variant, which supports circuits of multiplicative depth one.

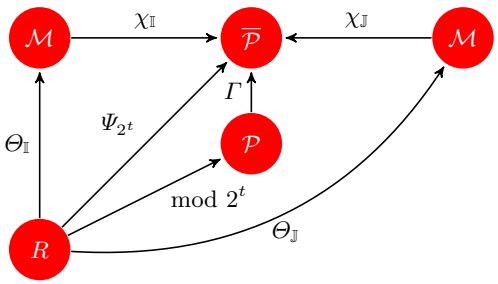

Fig. 3. Summary of the maps we use between different rings and representations

The Rings: The BGV encryption scheme, as we will use it, is built around the arithmetic of the cyclotomic ring $R = \mathbb{Z}[X]/(\Phi_m(X))$, where $\Phi_m(X)$ is the m-th cyclotomic polynomial. For an integer $q > 0$, we denote by R_q the ring obtained as reduction of R modulo q. In this work we will be taking m to be a prime p, and not the usual power of two as in most other papers. This is because we require that R factors modulo 2^t into a number r of distinct irreducible polynomials of degree d. To ensure better underlying geometry of the ring, i.e. the ring constant c_m is small (see [DPSZ12]), we then select m to be prime.

Our main optimization to enable an efficient offline phase for SPDZ2k will rely on us looking at the plaintext space in different ways. The main plaintext space $\mathcal{P}$ we will use is equivalent to the 2-adic local ring, approximated to the t-th coefficient, namely

$$\mathcal{P} = \mathbb{Z}_{2^t}[X]/(\Phi_p(X)).$$

As can be found in [Cas86], and used extensively in [GHS12a], the ring $\mathcal{P}$ decomposes into r irreducible factors each of degree d, as

$$\mathcal{P} \cong (\mathbb{Z}_{2^t}[X]/F_1(X)) \times \ldots \times (\mathbb{Z}_{2^t}[X]/F_r(X)) = \overline{\mathcal{P}},$$

where $\deg(F_i(X)) = d$ is the order of the element 2 in $\mathbb{F}_p^*$, and each $F_i(X)$ is the Hensel lift of the associated factor $f_i(X)$ of the factorization $\Phi_p(X) \equiv f_1(X) \cdots f_r(X) \pmod 2$. We write $N = \deg(\Phi_p(X)) = \phi(p) = p - 1$ and so $N = r \cdot d$. We will denote by $\Gamma : \mathcal{P} \longrightarrow \overline{\mathcal{P}}$ the map which takes elements in $\mathcal{P}$ and maps them to the slot representation $\overline{\mathcal{P}}$, and by Ψ_{2^t} the map from the global polynomial ring R representation to the slot $\overline{\mathcal{P}}$ representation, i.e.

$$\Psi_{2^t} : R \longrightarrow \overline{\mathcal{P}}.$$

Note that this map takes a polynomial f in R, maps it to $\mathcal{P}$, via reduction modulo 2^t, and then turns the resulting polynomial into its slot representation, thus $\Psi_{2^t}(f) = \Gamma(f \pmod{2^t})$. We also let Γ^{-1} denote the inverse map of Γ, which maps an element in $\overline{\mathcal{P}}$ to its equivalent element in $\mathcal{P}$. See Figure 3 for a summary of these, and other maps, we will be using [5].

It is well known that the number of monic irreducible polynomials of degree d over a finite field $\mathbb{F}_q$ is equal to

$$\frac{1}{d}\sum_{i|d} \mu(d/i) \cdot q^i,$$

[5] We will define the maps $\Theta_{\mathbb{I}}, \Theta_{\mathbb{J}}$ and $\chi_{\mathbb{I}}, \chi_{\mathbb{J}}$ in Figure 3 in the next section.

where $\mu(\cdot)$ is the Möbius function. This means that the number of SIMD "slots" r, using the packing technique of Smart and Vercauteren [SV14], is bounded by this value. In particular $r < 2^d$, and hence as N gets bigger we get progressively less efficient if we perform packing in a naive manner.

The problem occurs because we are interested in the plaintext space $\mathbb{Z}_{2^t}$, but the packing technique of [SV14] will only use the degree zero coefficient of each slot. Thus as d becomes larger for large N, the density of useful packing becomes smaller, and the ratio of data to plaintext space from this naive packing is $r/N = 1/d$.

The Distributions: Following [GHS12c] [Full version, Appendix A.5] and [ACK+19] [Documentation] we need different distributions in our protocol.

- $\mathsf{HWT}(h, N)$: This generates a vector of length N with elements chosen at random from $\{-1, 0, 1\}$ subject to the condition that the number of non-zero elements is equal to h.
- $\mathsf{ZO}(0.5, N)$: This generates a vector of length N with elements chosen from $\{-1, 0, 1\}$ such that the probability of each coefficient is $p_{-1} = 1/4$, $p_0 = 1/2$ and $p_1 = 1/4$.
- $\mathsf{dN}(\sigma^2, N)$: This generates a vector of length N with elements chosen according to an approximation to the discrete Gaussian distribution with variance σ^2.
- $\mathsf{RC}(0.5, \sigma^2, N)$: This generates a triple of elements (v, e_0, e_1) where v is sampled from $\mathsf{ZO}_s(0.5, N)$ and e_0 and e_1 are sampled from $\mathsf{dN}_s(\sigma^2, N)$.
- $\mathsf{U}(q, N)$: This generates a vector of length N with elements generated uniformly modulo q.

In the Appendix A we present the traditional noise analysis for the BGV scheme adapted to our specific application; this is adapted from [GHS12c], using the above distributions.

The Two Level BGV Scheme: We consider a two-leveled homomorphic scheme, given by three algorithms/protocols $\mathcal{E}_{\mathsf{BGV}} = \{\mathsf{BGV.KeyGen}, \mathsf{BGV.Enc}, \mathsf{BGV.Dec}\}$, which is parametrized by a security parameter κ, and defined as follows. First we fix two moduli q_0 and q_1 such that $q_1 = p_0 \cdot p_1$ and $q_0 = p_0$, where p_0, p_1 are prime numbers. Encryption generates level one ciphertexts, i.e. with respect to the largest modulo q_1, and level one ciphertexts can be moved to level zero ciphertexts via the modulus switching operation. We require

$$p_1 \equiv 1 \pmod{2^t} \quad \text{and} \quad p_0 - 1 \equiv p_1 - 1 \equiv 0 \pmod{p}.$$

The first condition is to enable modulus switching to be performed efficiently, whereas the second is to enable fast arithmetic using Number Theoretic Fourier Transforms.

- $\mathsf{BGV.KeyGen}(1^\kappa)$: It outputs a secret key $\mathfrak{sk}$ which is randomly selected from a distribution with Hamming weight h, i.e. $\mathsf{HWT}(h, N)$, much as in other systems, e.g. HELib [HS14] and SCALE [ACK+19] etc. The public key, $\mathfrak{pk}$, is of the form (a, b), such that

$$a \leftarrow \mathsf{U}(q_1, N) \quad \text{and} \quad b = a \cdot \mathfrak{sk} + 2^t \cdot \epsilon \pmod{q_1},$$

  where $\epsilon \leftarrow \mathsf{dN}(\sigma^2, N)$. This algorithm also outputs the relinearisation data $(a_{\mathfrak{sk},\mathfrak{sk}^2}, b_{\mathfrak{sk},\mathfrak{sk}^2})$ [BV11], where

$$a_{\mathfrak{sk},\mathfrak{sk}^2} \leftarrow \mathsf{U}(q_1, N) \quad \text{and} \quad b_{\mathfrak{sk},\mathfrak{sk}^2} = a_{\mathfrak{sk},\mathfrak{sk}^2} \cdot \mathfrak{sk} + 2^t \cdot e_{\mathfrak{sk},\mathfrak{sk}^2} - p_1 \cdot \mathfrak{sk}^2 \pmod{q_1},$$

  with $e_{\mathfrak{sk},\mathfrak{sk}^2} \leftarrow \mathsf{dN}(\sigma^2, N)$. We fix $\sigma = 3.16$ in what follows.

- $\mathsf{BGV.Enc}(m, \mathbf{r}; \mathfrak{pk})$: Given a plaintext $m \in \mathcal{P}$, the encryption algorithm samples $\mathbf{r} = (v, e_0, e_1) \leftarrow \mathsf{RC}(0.5, \sigma^2, n)$, i.e.

$$v \leftarrow \mathsf{ZO}(0.5, N) \quad \text{and} \quad e_0, e_1 \leftarrow \mathsf{dN}(\sigma^2, N),$$

and then sets

$$c_0 = b \cdot v + 2^t \cdot e_0 + m \pmod{q_1}, \quad c_1 = a \cdot v + 2^t \cdot e_1 \pmod{q_1}.$$

Hence the initial ciphertext is $\mathfrak{ct} = (1, c_0, c_1)$, where the first index denotes the level (initially set to be equal to one). We define a modulus switching operation which allows us to move from a level one to a level zero ciphertext, *without altering* the plaintext polynomial, that is

$$(0, c_0', c_1') \leftarrow \mathsf{SwitchMod}((1, c_0, c_1)), \quad c_0', c_1' \in R_{q_0}.$$

- $\mathsf{BGV.Dec}((c_0, c_1); \mathfrak{sk})$: Decryption is obtained by switching the ciphertext to level zero (if it is not already at level zero) and then decrypting $(0, c_0, c_1)$ via the equation

$$(c_0 - \mathfrak{sk} \cdot c_1 \pmod{q_0}) \pmod{2^t},$$

which results in an element of $\mathcal{P}$. The notation cmod refers to centered modular reduction, i.e. the resulting coefficients are taken in the interval $(-q/2, q/2]$. In the next sections, we will extend the decryption algorithm to enable distributed decryption.
- *Homomorphic Operations:* Ciphertexts at the same level ℓ can be added,

$$(\ell, c_0, c_1) \boxplus (\ell, c_0', c_1') = (\ell, (c_0 + c_0' \pmod{q_\ell}), (c_1 + c_1' \pmod{q_\ell}),$$

with the result being a ciphertext, which encodes a plaintext that is the sum of the two plaintexts of the initial ciphertexts.
Ciphertexts at level one can be multiplied together to obtain a ciphertext at level zero, where the output ciphertext encodes a plaintext which is the product of the plaintexts encoded by the input plaintexts. We do not present the method here, although it is pretty standard consisting of a modulus-switch, tensor-operation, then relinearization. We write the operation as

$$(1, c_0, c_1) \odot (1, c_0', c_1') = (0, c_0'', c_1''), \quad \text{with} \quad c_0'', c_1'' \in R_{q_0}.$$

3 Modified SHE Scheme

In this section we present a modified form of the previously presented "standard" BGV scheme. The main difference is that we introduce a new form of packing, where at each ciphertext level we interpret the naive BGV plaintext space $\mathcal{P}$ in a different manner. This modification enables us to obtain a final pre-processing phase for our MPC protocol which is less inefficient than one would naively expect.

3.1 Our New Packing Technique

The standard packing method of using only the degree zero coefficient in each slot will result in a very inefficient use of resources, as we have already mentioned. Thus we introduce a new packing technique which uses more coefficients in each slot. To do so, we first define two sets $\mathbb{I} = \{i_1, \ldots, i_{|\mathbb{I}|}\}$

and $\mathbb{J} = \{j_1, \ldots, j_{|\mathbb{I}|}\}$, such that $|\mathbb{I}| = |\mathbb{J}|$, and $j_\ell = 2 \cdot i_\ell$, for all $\ell = 1, \ldots, |\mathbb{I}|$. The idea is to encode (in each slot) $|\mathbb{I}|$ messages as coefficients of the powers X^i, with $i \in \mathbb{I}$, as follows. We define a map $\omega_\mathbb{I}$ for the set $\mathbb{I}$, as

$$\omega_\mathbb{I} : \begin{cases} (\mathbb{Z}_{2^t})^{|\mathbb{I}|} & \longrightarrow & \mathbb{Z}_{2^t}[X] \\ (m_1, \ldots, m_{|\mathbb{I}|}) & \longmapsto & m_1 \cdot X^{i_1} + \ldots + m_{|\mathbb{I}|} \cdot X^{i_{|\mathbb{I}|}}, \end{cases}$$

and a similar one $\omega_\mathbb{J}$ for the set $\mathbb{J}$. The reason why we require $j_\ell = 2 \cdot i_\ell$, for all $\ell = 1, \ldots, |\mathbb{I}|$, is that the $\mathbb{J}$-encoding will typically be used to hold the result of a product of two $\mathbb{I}$-encodings. As such we are only interested in the product of two terms of the same degree (giving rise to the $2 \cdot i_\ell$) and will ignore all other cross-products that appear in the product of two $\mathbb{I}$-encodings (all terms of degree $i_j + i_k$ for $j \neq k \in [|\mathbb{I}|]$). For level one ciphertexts (namely fresh ciphertexts), we will pack a message value from $\mathcal{M} = (\mathbb{Z}_{2^t})^{r \times |\mathbb{I}|}$ into the plaintext space $\overline{\mathcal{P}}$ as follows

$$\chi_\mathbb{I} : \begin{cases} \mathcal{M} & \longrightarrow & \overline{\mathcal{P}} \\ (\mathbf{m}_1, \ldots, \mathbf{m}_r) & \longmapsto & (\omega_\mathbb{I}(\mathbf{m}_1), \ldots, \omega_\mathbb{I}(\mathbf{m}_r)), \end{cases}$$

with a similar map being defined for the set $\mathbb{J}$. It is straightforward to see that this is a valid packing, and will be consistent for all ciphertexts at level one, since linear operations on elements in $\mathsf{Im}(\chi_\mathbb{I})$ also lie in $\mathsf{Im}(\chi_\mathbb{I})$.

For ease of convenience, we also define an "inverse" map, $\chi_\mathbb{I}^{-1}$, of the map above, which is defined on $\overline{\mathcal{P}}$ and simply selects the correct coefficients, producing a final output in $\mathcal{M}$. We also define $\mathsf{Supp}(\mathbb{I})$, to be the set of (potentially) non-zero coefficients in each slot in the image of $\omega_\mathbb{I}$, in particular elements in $\mathsf{Supp}(\mathbb{I})$ are the only values which affect the value of $\chi_\mathbb{I}^{-1}$. Thus we have

$$\mathsf{Supp}(\mathbb{I}) = \{(1, i_1), \ldots, (1, i_{|\mathbb{I}|}), (2, i_1), \ldots, (r, i_{|\mathbb{I}|})\},$$

where the first element of each pair refers to which slot we are considering and the second element to the power of X in that particular slot. Given an element u in the global polynomial ring R we can define an element in $\mathcal{M}$ by reducing the polynomial u modulo 2^t then taking its image under one of the inverse maps above. Thus we have the map

$$\Theta_\mathbb{I} : \begin{cases} R & \longrightarrow & \mathcal{M} \\ u & \longmapsto & \chi_\mathbb{I}^{-1}(\Psi_{2^t}(u)) \end{cases}$$

Given an element $m \in \mathcal{M}$, there are infinitely many preimages under the map $\Theta_\mathbb{I}$. At various points we will need to select one subject to a given bound B on the coefficients of the polynomial in R. We therefore define, in Figure 4, a procedure which outputs an element in R, uniformly at random, subject to the constraint that its image under $\Theta_\mathbb{I}$ is equal to a given element $\mathbf{m} \in \mathcal{M}$ and its coefficients are bounded by B. Clearly, all of the above considerations apply also to the set $\mathbb{J}$.

3.2 The BGV Encryption Scheme with Double Packing Set

We are now ready to define our modified BGV scheme, $\mathcal{E}_{\mathsf{mBGV}} = \{\mathsf{mBGV.KeyGen}, \mathsf{mBGV.Enc}, \mathsf{mBGV.Dec}\}$, which uses plaintext space $\mathcal{M} = (\mathbb{Z}_{2^t})^{r \times |\mathbb{I}|}$. The key generation algorithm $\mathsf{mBGV.KeyGen}$ is the same as in the original BGV scheme presented earlier, i.e. given a security parameter κ, it outputs a public/private key pair $(\mathfrak{pk}, \mathfrak{sk})$ and the relinearisation data.

The encryption algorithm differs as it now encrypts using one of the two sets $\mathbb{I}$ or $\mathbb{J}$. To make the dependence clear on which set we are encrypting a message under, we write either

$$\mathfrak{ct}^\mathbb{I} = (1, c_0, c_1)^\mathbb{I} = \mathsf{mBGV.Enc}(\mathbf{m}, \mathbf{r}; \mathbb{I}, \mathfrak{pk}) = \mathsf{BGV.Enc}(\Gamma^{-1}(\chi_\mathbb{I}(\mathbf{m})), \mathbf{r}; \mathfrak{pk})$$

The Function $\Theta_{\mathbb{I}}^{-1}(m, B)$

1. Compute $m_{\overline{\mathcal{P}}} \in \overline{\mathcal{P}}$, the image of $\mathbf{m}$ under the map $\chi_{\mathbb{I}}$.
2. For all entries *not in* $\mathsf{Supp}(\mathbb{I})$, replace the zero coefficient in each slot by a uniformly random element selected from $[0, \ldots, 2^t]$, resulting in a uniformly random element $m'_{\overline{\mathcal{P}}} \in \overline{\mathcal{P}}$ whose image under $\chi_{\mathbb{I}}^{-1}$ is also $\mathbf{m}$.
3. Pull back $m'_{\overline{\mathcal{P}}}$ to R by computing the element $m'_R \leftarrow \Psi_{2^t}^{-1}(m'_{\overline{\mathcal{P}}})$ subject to all coefficients lying in $[0, \ldots, 2^t]$.
4. Select a uniformly random polynomial $u \in R$ whose coefficient infinity norm is bounded by $B/2^t$.
5. Output $m_R \leftarrow m'_R + 2^t \cdot u$.

Figure 4. The procedure $\Theta_{\mathbb{I}}^{-1}(\mathbf{m}, B)$ from R to $\mathcal{M}$

or

$$\mathfrak{ct}^{\mathbb{J}} = (1, c_0, c_1)^{\mathbb{J}} = \mathsf{mBGV.Enc}(\mathbf{m}, \mathbf{r}; \mathbb{J}, \mathfrak{pk}) = \mathsf{BGV.Enc}(\Gamma^{-1}(\chi_{\mathbb{J}}(\mathbf{m})), \mathbf{r}; \mathfrak{pk}),$$

where $\mathbf{m} \in \mathcal{M}$. Similarly, the decryption algorithm is defined as

$$\mathbf{m} = \mathsf{mBGV.Dec}(\mathfrak{ct}^{\mathbb{I}}; \mathfrak{sk}) = \chi_{\mathbb{I}}^{-1}(\Gamma(\mathsf{BGV.Dec}(\mathfrak{ct}^{\mathbb{I}}; \mathfrak{sk})))$$

and

$$\mathbf{m} = \mathsf{mBGV.Dec}(\mathfrak{ct}^{\mathbb{J}}; \mathfrak{sk}) = \chi_{\mathbb{J}}^{-1}(\Gamma(\mathsf{BGV.Dec}(\mathfrak{ct}^{\mathbb{J}}; \mathfrak{sk}))).$$

Addition and multiplication of ciphertexts are accomplished as in the "standard" BGV scheme, but with some notable differences. Notice we can now only add ciphertexts at the same level when they are with respect to the same encoding. Thus we have (say)

$$(1, c_0, c_1)^{\mathbb{I}} \boxplus (1, c'_0, c'_1)^{\mathbb{I}} = (1, c''_0, c''_1)^{\mathbb{I}}.$$

The idea is that the $\mathbb{I}$ encoding is used for messages at level one, and the $\mathbb{J}$ encoding is used for messages at level zero, typically obtained as the result of multiplying two level one ciphertexts.
In the following sections we will use the bracked exponent $\mathfrak{ct}^{(\ell)}$ on a ciphertext to denote the "level" which the ciphertext is at, with fresh ciphertext always being at level one. Hence, following the discussion above we will usually have:

$$\mathfrak{ct}^{(1)} = (1, c_0, c_1)^{\mathbb{I}} = \mathfrak{ct}^{\mathbb{I}} \quad \text{and} \quad \mathfrak{ct}^{(0)} = (0, c_0, c_1)^{\mathbb{J}} = \mathfrak{ct}^{\mathbb{J}}.$$

However we might need to encrypt some messages using index set $\mathbb{J}$, for example if we wish to encrypt a fresh message and then move it directly to level zero using a $\mathsf{SwitchMod}$ operation, as in $(0, c'_0, c'_1)^{\mathbb{J}} \leftarrow \mathsf{SwitchMod}((1, c_0, c_1)^{\mathbb{J}})$, where $(1, c_0, c_1)^{\mathbb{J}} = \mathsf{Enc.mBGV}(\mathbf{m}, \mathbf{r}; \mathbb{J}, \mathfrak{pk})$. The reason we switch encodings as we transfer between level one and level zero is that when two ciphertexts are multiplied at level one to produce a level zero ciphertext, the $\mathbb{I}$ packing will no longer be valid. So we switch to index set $\mathbb{J}$ at this point. Our multiplication is now an operation of the form

$$(1, c_0, c_1)^{\mathbb{I}} \odot (1, c'_0, c'_1)^{\mathbb{I}} = (0, c''_0, c''_1)^{\mathbb{J}}.$$

We will clarify the dependence on $\mathbb{I}$ or $\mathbb{J}$ and the encryption level ℓ when it is not clear from the context. More formally, in our MPC protocol, we will denote addition and multiplication of ciphertexts as follows:

$$\begin{aligned}
\mathfrak{ct}_{\mathbf{m}_1+\mathbf{m}_2}^{(\ell,\cdot)} &\leftarrow \mathfrak{ct}_{\mathbf{m}_1}^{(\ell,\cdot)} \boxplus \mathfrak{ct}_{\mathbf{m}_2}^{(\ell,\cdot)}, \\
\mathfrak{ct}_{\mathbf{a}\cdot\mathbf{m}}^{(\ell,\cdot)} &\leftarrow \mathbf{a} \odot \mathfrak{ct}_{\mathbf{m}}^{(\ell,\cdot)}, \quad \text{for } \mathbf{a} \in \mathcal{M}, \\
\mathfrak{ct}_{\mathbf{m}_1\cdot\mathbf{m}_2}^{(0,\mathbb{J})} &\leftarrow \mathfrak{ct}_{\mathbf{m}_1}^{(1,\mathbb{I})} \odot \mathfrak{ct}_{\mathbf{m}_2}^{(1,\mathbb{I})}.
\end{aligned}$$

Correctness. To have correctness we need to ensure that multiplication of two elements in $\mathsf{Im}(\chi_{\mathbb{I}})$ results in something correct when we restrict $\overline{\mathcal{P}}$ to the image of the $\chi_{\mathbb{J}}$ map, i.e. by ignoring coefficients which are not in the image of $\chi_{\mathbb{J}}$. This is because a product of two elements in $\mathsf{Im}(\chi_{\mathbb{I}})$ is *not* an element of $\mathsf{Im}(\chi_{\mathbb{J}})$. Looking ahead, when we use this packing technique in our MPC protocol we need to ensure that ignoring coefficients that are not in $\mathsf{Im}(\chi_{\mathbb{I}})$ does not leak information. We shall deal with this security issue in the next sections, so for now we consider only the correctness concern.

To select $\mathbb{I}$ we have two conditions: The first obvious correctness guarantee is that the product term does not wrap around modulo each factor $F_i(X)$, so that we require

$$\forall i \in \mathbb{I}, \quad 2 \cdot i < d.$$

Secondly, we need that any cross-product terms do not interfere with any of the desired slot terms. This is implied by the equation

$$\forall i_1, i_2, j \in \mathbb{I}, \quad i_1 + i_2 \neq 2 \cdot j, \text{ with } i_1 \neq j, i_2 \neq j.$$

In Figure 5 we plot the growth of the maximum size of $|\mathbb{I}|$ versus the size of d. As one can see, it grows in a step wise manner, looking like about $d^{0.6}$ in the range under consideration here.

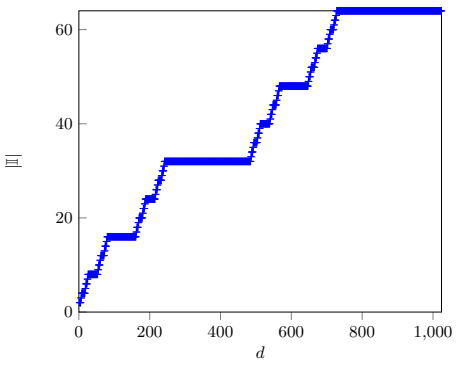

Fig. 5. Growth of $|\mathbb{I}|$ with d

This analysis gives the amount of packing we can produce in a given standard slot. To see what is the total packing ratio we can achieve, we need to look at the number theoretic properties of the polynomials $\Phi_p(X)$ for p prime. As remarked earlier these factor modulo 2 into r factors of degree d, where d is equal to the order of the element 2 in $\mathbb{F}_p^*$. We can then take the maximum value of $|\mathbb{I}|$ from the above calculations and compute the ratio of "useful" slots, in our application, as

$$\pi_p = \frac{r \cdot |\mathbb{I}|}{p-1}.$$

For security reasons in our MPC applications we will be taking p in the range $8192 < p < 65536$, so in Table 1 we present the prime values in this range which give us a ratio greater than 0.15. We see that it is possible to select p so that the packing ratio π_p approaches 0.2. Thus we can obtain an efficiency of packing of around $\phi(p)/5$, as mentioned in the introduction. All that remains is to adapt the MPC protocols to deal with this new packing methodology.

p	r	d	$\lvert\mathbb{I}\rvert$	$r \cdot \lvert\mathbb{I}\rvert$	π_p
9719	226	43	8	1808	.186
11119	218	51	8	1744	.156
11447	118	97	16	1888	.164
13367	326	41	8	2608	.195
14449	172	84	16	2752	.190
20857	316	66	12	3792	.181
23311	518	45	8	4144	.177
26317	387	68	12	4644	.176
29191	278	105	16	4448	.152
30269	329	92	16	5264	.173
32377	568	57	10	5680	.175
38737	538	72	13	6994	.180
43691	1285	34	8	10280	.235
61681	1542	40	8	12336	.200

Table 1. Primes with a packing density ratio greater than 0.15 in the range $8192 < p < 65536$

4 OverDrive Global ZKPoKs

Given a SHE scheme (in our case either $\mathcal{E}_{\mathsf{mBGV}}$ or $\mathcal{E}_{\mathsf{BGV}}$), we denote by $\mathcal{C}$ the set of admissible circuits for the SHE scheme, the exact choice of $\mathcal{C}$ will depend on the underlying construction. In our protocol the decryption function will be always correct assuming the input ciphertext is the evaluation of an admissible circuit from $\mathcal{C}$ applied to ciphertexts which are marked "correct enough". We shall call a ciphertext valid if it is either "correct enough", or is the output of a circuit in $\mathcal{C}$ applied to "correct enough" ciphertexts.

Looking ahead, in Section 5 we will extend the scheme $\mathcal{E}_{\mathsf{mBGV}}$, introduced in the previous section, to allow distributed decryption. The reason for using the term "correct enough" is that our distributed decryption protocol will be proved correct even if some ciphertexts are not completely valid, namely they are not generated using the standard encryption algorithm.

In describing our protocol, we assume a key generation functionality $\mathcal{F}_{\mathsf{KeyGen}}$as described in Figure 6. It runs $\mathsf{BGV.KeyGen}$ and outputs for each party P_i the public key $\mathfrak{pk}$ and an additive share $[\mathfrak{sk}]_i$ of $\mathfrak{sk}$ for performing distributed decryption. This means that given a public ciphertext, parties can use their shares of the $\mathfrak{sk}$ and collaborate to decrypt it. Just as in Overdrive, SPDZ and SCALE [KPR18, DPSZ12, ACK+19], we will assume a trusted dealer that implements the distributed key generation, possibly in practice via HSMs. Our goal here is to focus on the main part of the protocol and not on set-up assumptions, thus we do not discuss how to securely realise the ideal functionality$\mathcal{F}_{\mathsf{KeyGen}}$, as was done in the aforementioned works.

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

Let A be the set of corrupt parties.

1. On receiving (Init) from all honest parties run $(\mathfrak{pk}, \mathfrak{sk}) \leftarrow \mathsf{mBGV.KeyGen}(1^\kappa)$. Send $\mathfrak{pk}$ to the adversary.
2. Receive shares $[\mathfrak{sk}]_j$, $j \in A$, from the adversary.
3. Construct a complete set of shares $\{[\mathfrak{sk}]_1, \ldots, [\mathfrak{sk}]_n\}$ consistent with the adversary's choices and such that $\mathfrak{sk} = \sum_{i=1}^{n} [\mathfrak{sk}]_i$
4. Send $\mathfrak{pk}$ to all parties, and $[\mathfrak{sk}]_i$ to each honest party P_i.

Figure 6. The functionality $\mathcal{F}_{\mathsf{KeyGen}}$ for distributed key generation

4.1 Bounded Linearly Homomorphic Predicates

Here we show how to ensure that all the ciphertexts used in our protocol are valid. Compared to similar protocols in previous works, other than prove that our ciphertexts decrypt correctly, we also need to show that the underlying plaintexts satisfy a given predicate P which we call *bounded linearly homomorphic*.

Definition 4.1. *We say that a given predicate* P *is* bounded linearly homomorphic *if, given a bound* B *and values* $\mathbf{x}_1, \ldots, \mathbf{x}_\nu$*, where*

$$\mathbf{x}_1 = (x_{1,1}, \ldots, x_{u,1}) \in R^u, \ldots, \mathbf{x}_\nu = (x_{1,\nu}, \ldots, x_{u,\nu}) \in R^u,$$

such that

1. $\forall j \in [u]$, $\mathsf{P}(x_{j,1}, \ldots, x_{j,\nu}) = \mathsf{true}$*, and*
2. *the coefficient norm of each* $x_{j,k}$ *is bounded by* B*,*

then, for all $\mathbf{a} \in \{0,1\}^u$, $\mathsf{P}(\mathbf{a} \cdot \mathbf{x}_1, \ldots, \mathbf{a} \cdot \mathbf{x}_v) = \mathsf{true}$.

We will give two different instantiations of this definition. The first one is with the diagonal predicate $\mathsf{P} = \mathsf{Diag}$ also used in [DPSZ12]. This takes as input a single element $\mathbf{x}_1 \in R^u$, i.e. $\nu = 1$, and checks whether each of the slot entries in $\mathbf{x}_1$ (when mapped to $\overline{\mathcal{P}}$ via the map Ψ_{2^b} for $b = \lceil \log_2(u \cdot B) \rceil$), are identical to each other. Clearly if the predicate holds for input ciphertexts with plaintext coefficient norms bounded by B, then it also holds for a sum of u ciphertexts with plaintext coefficient norms bounded by $u \cdot B$.

The second instantiation works with $\nu = 2$. We recall from Section 3 that the maps $\Theta_{\mathbb{I}}$ and $\Theta_{\mathbb{J}}$ map an element $x \in R$ to an element in $\mathcal{M}$ according to $\chi_{\mathbb{I}}$ and $\chi_{\mathbb{J}}$, respectively. The predicate $\mathsf{P} = \mathsf{Pack}$ is then defined as follows:

- Let $\mathbf{m}_{\mathbb{I}} = \Theta_{\mathbb{I}}(x_1, B)$ and $\mathbf{m}_{\mathbb{J}} = \Theta_{\mathbb{J}}(x_2, B)$. The elements in $\mathsf{Supp}_{2^b}(\mathbf{m}_{\mathbb{I}})$, for $b = \lceil \log_2(u \cdot B) \rceil$, are indexed by $\mathsf{Supp}(\mathbb{I})$.
- If $\mathsf{Supp}_{2^b}(\mathbf{m}_{\mathbb{I}}) = \{c_{i,i_j}\}$, for $i \in [r]$ and $i_j \in \mathbb{I}$, then the coefficients in $\Psi_{2^b}(m_{\mathbb{J}})$ indexed by $(i, 2 \cdot i_j)$ are equal to c_{i,i_j}, and are uniformly random elsewhere. Being uniformly random in locations not indexed by $\mathbb{J}$ will be important for security of our distributed decryption protocol later.

Again it is straightforward to prove that this predicate is bounded linearly homomorphic.

4.2 Amortized Zero Knowledge Proof

Given the definition of a *bounded linearly homomorphic* predicate on the plaintexts, we are now ready to define what we mean by a *valid ciphertext* which encrypts such a plaintext. We recall that a ciphertext $\mathfrak{ct} = \mathsf{BGV.Enc}(x, \mathbf{r}; \mathfrak{pk})$ encrypts a plaintext value $x \in \mathcal{P}$ under randomness $\mathbf{r} = (v, e_0, e_1) \in R^3$. In our protocol we assume that $x = \Theta_{\mathbb{I}}^{-1}(\mathbf{m})$, for some $\mathbf{m} \in \mathcal{M}$. In a legitimate ciphertext, the plaintext x lies in $\mathcal{P}$ and the randomness values come from specific distributions (see Section 3). An adversarially chosen ciphertext may not be generated in this way, however, as long as the adversarial plaintexts and random coins are selected from some restricted set, the ciphertexts will correctly decrypt. A ciphertext which comes from this restricted set (no matter how it is generated) is said to be valid.

Protocol $\Pi^{\nu,\mathsf{flag}}_{\mathsf{gZKPoK}}$ - Part I

PARAMETERS: an integer ν, $u = \mathsf{ZK\_sec}$, $V = 2 \cdot \mathsf{ZK\_sec} - 1$, a $\mathsf{flag} \in \{\mathsf{Diag}, \mathsf{Pack}, \perp\}$ such that if $\mathsf{flag} = \mathsf{Diag}$ then $\mathsf{P} = \mathsf{Diag}$; if $\mathsf{flag} = \mathsf{Pack}$ then $\mathsf{P} = \mathsf{Pack}$ and if $\mathsf{flag} = \perp$ then $\mathsf{P} = \emptyset$.
INPUT: Each P_i inputs $u \cdot \nu$ BGV ciphertexts $\mathfrak{ct}^i_{j,k}, j \in [u], k \in [\nu]$, such that

$$\|v^i_{j,k}\|_\infty \leq \rho_1, \quad \|e^i_{0,j,k}\|_\infty, \|e^i_{1,j,k}\|_\infty \leq \rho_2, \quad \|x^i_{j,k}\|_\infty \leq \tau,$$

where $x^i_{j,k} \in R$ is the plaintext corresponding to $\mathfrak{ct}^i_{j,k}$, satisfying $\mathsf{P}(x^i_{j,1}, \ldots, x^i_{j,\nu}) = \mathsf{true}$, and for each $k \in [\nu]$, set:

$$\begin{aligned}
\mathbf{r}^i_k &= (v^i_{1,k}, \ldots, v^i_{u,k}, e^i_{0,1,k}, \ldots, e^i_{0,u,k}, e^i_{1,1,k}, \ldots, e^i_{1,u,k}) \in R^{u\times 3},\\
\mathbf{x}^i_k &= (x^i_{1,k}, \ldots, x^i_{u,k}) \in R^u\\
\mathbf{c}^i_k &= \mathfrak{ct}^i_k = (\mathfrak{ct}^i_{1,k}, \ldots, \mathfrak{ct}^i_{u,k}) \in R^{u\times 2}.
\end{aligned}$$

gZKPoK: If $\mathsf{flag} \in \{\mathsf{Diag}, \perp\}$ parties execute the following steps.
- For each $k \in [\nu]$ execute:

Commit:
- Each P_i broadcasts $\mathbf{c}^i_k = \mathsf{BGV.Enc}(\mathbf{x}^i_k, \mathbf{r}^i_k; \mathfrak{pk})$
- Each party P_i samples a new set of "plaintexts" $\mathbf{y}^i_k \in R^V$ and "randomness vectors" $\bar{\mathbf{r}}^i_k \in R^{V\times 3}$, such that, for $j \in [u]$ and $\mathsf{P}(y_{j,1}, \ldots, y_{j,\nu}) = \mathsf{true}$,

$$\begin{aligned}
\|y^i_{j,k}\|_\infty \leq 2^{\mathsf{ZK_sec}} \cdot \tau, \quad \|\bar{v}^i_{j,k}\|_\infty \leq 2^{\mathsf{ZK_sec}} \cdot \rho_1,\\
\|\bar{e}^i_{0,j,k}\|_\infty, \qquad \|\bar{e}_{1,j,k}\|_\infty \leq 2^{\mathsf{ZK_sec}} \cdot \rho_2.
\end{aligned}$$

- Each P_i computes and broadcasts $\mathbf{a}^i_k \leftarrow \mathsf{BGV.Enc}(\mathbf{y}^i_k, \bar{\mathbf{r}}^i_k; \mathfrak{pk})$, for $k \in [\nu]$.

Challenge: Parties call $\mathcal{F}_{\mathsf{Rand}}$ to get a random $\hat{\mathbf{e}}_k = (\hat{e}_{k,1}, \ldots, \hat{e}_{k,u}) \in \{0,1\}^u$.

Prove:
- Parties define $M_{\hat{\mathbf{e}}_k} \in \{0,1\}^{V\times u}$ to be the matrix such that $(M_{\hat{\mathbf{e}}_k})_{r,c} = \hat{e}_{k,r-c+1}$, for $1 \leq r-c+1 \leq u$, and 0 in all other entries.
- Each P_i computes and broadcasts the values $(\mathbf{z}^i_k, T^i_k)$, where $\mathbf{z}^{i\intercal}_k = \mathbf{y}^{i\intercal}_k + M_{\hat{\mathbf{e}}_k} \cdot \mathbf{x}^{i\intercal}_k$ and $T^i_k = \bar{\mathbf{r}}_k + M_{\hat{\mathbf{e}}_k} \cdot \mathbf{r}^i_k$.

Verify:
- Each party P_i computes $\mathbf{d}^i_k = \mathsf{BGV.Enc}(\mathbf{z}^i_k, T^i_k; \mathfrak{pk})$ and then stores the sum $\mathbf{d}_k = \sum_{i=1}^n \mathbf{d}^i_k$.
- The parties compute the values

$$\mathbf{c}_k = \sum_{i\in[n]} \mathbf{c}^i_k, \quad \mathbf{a}_k = \sum_{i\in[n]} \mathbf{a}^i_k, \quad \mathbf{z}_k = \sum_i \mathbf{z}^i_k, \quad T_k = \sum_{i\in[n]} T^i_k,$$

and conduct the following checks, where $t_{i,j,k}$ is the (i,j)-th element of T_k,

$$\mathbf{d}^\intercal_k = \mathbf{a}^\intercal_k + (M_{\hat{\mathbf{e}}_k} \cdot \mathbf{c}_k), \quad \|\mathbf{z}_k\|_\infty \leq 2 \cdot n \cdot 2^{\mathsf{ZK\_sec}} \cdot \tau \tag{1}$$

$$\|t_{i,1,k}\|_\infty \leq 2 \cdot n \cdot 2^{\mathsf{ZK\_sec}} \cdot \rho_1, \quad \|t_{i,2,k}\|_\infty, \quad \|t_{i,3,k}\|_\infty \leq 2 \cdot n \cdot 2^{\mathsf{ZK\_sec}} \cdot \rho_2.$$

- If $\mathsf{P} = \mathsf{Diag}$ the proof is rejected if $\mathsf{P}(z^i_{j,1}) \neq \mathsf{true}$ for any $j \in [u]$.

If the check passes, the parties output $\sum_{i\in[n]} \mathbf{c}^i_1, \ldots, \sum_{i\in[n]} \mathbf{c}^i_\nu$.

Figure 7. Protocol for global proof of knowledge of a ciphertext - Part I

Suppose we have $u \cdot \nu$ BGV ciphertexts $\mathfrak{ct}_j \leftarrow \mathsf{BGV.Enc}(x_j, \mathbf{r}_j, \mathfrak{pk}), j \in [u \cdot \nu]$, such that

$$\mathfrak{ct}_j = \sum_{i \in [n]} \mathfrak{ct}_j^i, \quad x_j = \sum_{i \in [n]} x_j^i, \quad \mathbf{r}_j = \sum_{i \in [n]} \mathbf{r}_j^i, \quad \forall j \in [u \cdot \nu],$$

i.e. $\mathfrak{ct}_j^i \leftarrow \mathsf{BGV.Enc}(x_j^i, \mathbf{r}_j^i, \mathfrak{pk}), x_j^i$ and $\mathbf{r}_j^i$ are respectively the ciphertext, the plaintext and the randomness held by party P_i. The protocol $\Pi^{\nu,\mathsf{flag}}_{\mathsf{gZKPoK}}$ (Figure 7, Figure 8) guarantees that each ciphertext $\mathfrak{ct}_j$ is both valid and satisfies the bounded linearly homomorphic predicate P. Our zero-knowledge proof is very similar to the one given in [KPR18], with some modifications due to our new packing technique, and it is a generalization to the multiparty setting of the amortized proof described in [DPSZ12] and [CD09]. Note that as done in Overdrive, our protocol does not check the correctness of every single share $\mathfrak{ct}_j^i$, but just of their sum since it is sufficient for our purpose.

To understand the proof $\Pi^{\nu,\mathsf{flag}}_{\mathsf{gZKPoK}}$, first, let us assume $\nu = 1$ and $\mathsf{flag} = \mathsf{Diag}$. Following Cramer et al. [CD09]'s blueprint, the protocol $\Pi^{1,\mathsf{Diag}}_{\mathsf{gZKPoK}}$ simultaneously *tries to* prove that u ciphertexts $\mathfrak{ct}_j$ are generated such that:

$$\|v_j\|_\infty \leq n \cdot \rho_1, \quad \|e_{0,j}\|_\infty, \|e_{1,j}\|_\infty \leq n \cdot \rho_2, \quad \|x_j\|_\infty \leq n \cdot \tau, \quad \forall j \in [u], \tag{2}$$

for $\tau = 2^{t-1}, \rho_1 = 1$ and $\rho_2 = 20$. This is done using an amortized Σ protocol that samples commitments $\bar{\mathfrak{ct}}_j \leftarrow \mathsf{BGV.Enc}(y_j, \bar{\mathbf{r}}_j, \mathfrak{pk}), j \in [u], \bar{\mathbf{r}}_j = (\bar{v}_j, \bar{e}_{0,j}, \bar{e}_{1,j})$, such that

$$\begin{gathered}\|\bar{v}_j\|_\infty \leq n \cdot 2^{\mathsf{ZK\_sec}} \cdot \rho_1, \\ \|\bar{e}_{0,j}\|_\infty, \|\bar{e}_{1,j}\|_\infty \leq n \cdot 2^{\mathsf{ZK\_sec}} \cdot \rho_2, \\ \|y_j\|_\infty \leq n \cdot 2^{\mathsf{ZK\_sec}} \cdot \tau, \quad \forall j \in [u],\end{gathered}$$

for some large enough $2^{\mathsf{ZK\_sec}}$. In this way we can form the responses $\mathbf{z}$ and T such that the terms $\mathbf{y}$ and $\bar{\mathbf{r}}$ statistically hide $M_e \cdot \mathbf{x}$ and $M_e \cdot \mathbf{r}$ respectively, for some challenge matrix M_e. The bounds on $\mathbf{z}$ and T imply bounds on $\mathbf{x}$ and $\mathbf{r}$. This implies that, instead of obtaining a proof that the input ciphertexts satisfy Equation 2, we get a proof that those values satisfy the following relationships:

$$\|v_j\|_\infty \leq n \cdot S \cdot \rho_1, \quad \|e_{j,0}\|_\infty, \|e_{j,1}\|_\infty \leq n \cdot S \cdot \rho_2, \quad \|x_j\|_\infty \leq n \cdot S \cdot \tau, \quad \forall j \in [u], \tag{3}$$

where $S = 2 \cdot 2^{3 \cdot \mathsf{ZK\_sec}/2+1}$. These bounds are clearly not tight and the value S is called the *soundness slack*.

When $\nu = 2$ and $\mathsf{P} = \mathsf{Pack}$, we need to repeat the above proof twice, or equivalently sample the challenge in $\{0,1\}^{2 \cdot \mathsf{ZK\_sec}}$, and add the proof for the predicate P. Line 2 of Figure 8 is checked by a verifier only that required equality between coefficients in the predicate holds. That the other coefficients are unformly distributed is not checked, indeed this is impossible to do. However, if the other coefficients are not uniformly distributed then the prover will loose the desired zero-knowledge property, thus it is not in the provers interest to produce values which are not uniformly distributed. In the case of our application an honest verifier is actually one of the n provers, and this is enough to ensure the desired uniform property holds on the required subset of coefficients.

Thus in both cases the protocol $\Pi^{\nu,\mathsf{flag}}_{\mathsf{gZKPoK}}$ is an honest-verifier zero-knowledge proof of knowledge for the relation

$$\mathcal{R}_{\mathsf{gZKPoK}} = \{(x, w) \,|\, x = (\mathbf{c}, \mathfrak{pk}), w = ((x_1, \mathbf{r}_1) \ldots, (x_{\nu \cdot u}, \mathbf{r}_{\nu \cdot u}))$$

$$
\begin{aligned}
&: \{u = \mathsf{ZK\_sec}, \|x_j\|_\infty \leq n \cdot S \cdot \tau, \mathbf{m}_j = \Theta_{\mathbb{I}}(x_j) \in \mathcal{M}, \\
&\mathbf{c} = (\mathfrak{ct}_1, \ldots, \mathfrak{ct}_u), \|v_j\|_\infty \leq n \cdot S \cdot \rho_1, \|e_{0,j}\|_\infty, \|e_{1,j}\|_\infty \leq n \cdot S \cdot \rho_2\} \\
&\wedge \{\mathsf{P}(x_{j,1}, \ldots, x_{j,\nu}) = \mathsf{true}, \forall j \in [u]\}\}
\end{aligned}
$$

Protocol $\Pi^{\nu,\mathsf{flag}}_{\mathsf{gZKPoK}}$ - Part II

If $\mathsf{flag} = \mathsf{Pack}$ then apply the proof for $\mathsf{flag} = \perp$ above, making sure the sampling in Step 4.1 follows the predicate P for Pack. Then, perform the following steps (using the values obtained whilst executing the above proof).

1. Each P_i computes and broadcasts the values
$$\mathbf{z}_2^{i\intercal} = \mathbf{y}_2^{i\intercal} + M_{\hat{\mathbf{e}}_2} \cdot \mathbf{x}_2^{i\intercal} \in R^V.$$
2. The proof is rejected if $\mathsf{P}(z^i_{j,1}, z^i_{j,2}) \neq \mathsf{true}$ for any $j \in [u]$. If the check passes, the parties output $\sum_{i\in[n]} \mathbf{c}^i_1, \ldots, \sum_{i\in[n]} \mathbf{c}^i_\nu$.

Figure 8. Protocol for global proof of knowledge of a ciphertext - Part II

Theorem 4.1. *The protocol $\Pi^{\nu,\mathsf{flag}}_{\mathsf{gZKPoK}}$ is an honest-verifier zero-knowledge proof of knowledge for the relation $\mathcal{R}_{\mathsf{gZKPoK}}$ with error probability $2^{-\mathsf{ZK\_sec}}$ and soundness slack $S = 2 \cdot 2^{3\cdot\mathsf{ZK\_sec}/2+1}$.*

We do not follow the Overdrive proof approach in our MPC protocol, i.e. we do not give an ideal functionality for $\Pi^{\nu,\mathsf{flag}}_{\mathsf{gZKPoK}}$. The reason is that a security proof for $\Pi^{\nu,\mathsf{flag}}_{\mathsf{gZKPoK}}$ would require rewinding the adversary to extract corrupt parties' inputs in the simulation, breaking the UC security of the protocol. Instead, we will use $\Pi^{\nu,\mathsf{flag}}_{\mathsf{gZKPoK}}$ inside our MPC protocol, as done in [DPSZ12], and prove UC security for this latter protocol. The complete proof of the theorem above is however still similar to the one in [KPR18].

Proof. We suppose that $\nu = 1$ and $\mathsf{P} = \mathsf{Diag}$, as the proof is similar in the other cases.

Correctness. Assume that all parties are honest. The equalities (1) and $\mathsf{Prep} = \mathsf{true}$ follow trivially from the homomorphic property of the encryption and the predicate. It remains to check the probability that honest parties will fail the bounds check on $\|\mathbf{z}\|_\infty$ and $\|t_{i,j}\|_\infty$. Remember that the honestly generated $\mathfrak{ct}^i$ are ciphertexts generated according to the true distribution (i.e. without the slack).

The bound check for the plaintext component will succeed if the infinity norm of $\sum_{i=1}^{n}(\mathbf{y}^i + \sum_{k=1}^{\mathsf{ZK\_sec}}(M_{\hat{e}_{jk}} \cdot \mathbf{x}^i))$ is at most $2 \cdot n \cdot \tau \cdot 2^{\mathsf{ZK\_sec}}$. This is always true because $\mathbf{y}^i$ is sampled such that $\|\mathbf{y}^i\|_\infty \leq 2^{\mathsf{ZK\_sec}} \cdot \tau$ and $\|M_{\hat{\mathbf{e}}} \cdot \mathbf{x}^{(i)}\|_\infty \leq \mathsf{sec} \cdot \tau \leq 2^{\mathsf{ZK\_sec}} \cdot \tau$. A similar argument holds regarding the three randomness components.

Special soundness. Given two accepting transcripts $(\mathbf{x}, \mathbf{a}, \hat{\mathbf{e}}, (\mathbf{z}, T))$ and $(\mathbf{x}, \mathbf{a}, \hat{\mathbf{e}}', (\mathbf{z}', T'))$, $\hat{\mathbf{e}} \neq \hat{\mathbf{e}}'$, we have to extract a valid witness $(\mathbf{x}, \mathbf{r})$ for $\mathbf{c}$. Recall that each party has a different secret $\mathbf{x}^i \in R^u$. Because both challenges have passed the equality check during the protocol, we obtain

$$(M_{\hat{\mathbf{e}}} - M_{\hat{\mathbf{e}}'}) \cdot \mathbf{c}^\intercal = (\mathbf{d} - \mathbf{d}')^\intercal \tag{4}$$

To find $(\mathbf{x}, \mathbf{r})$ such that $\mathbf{c} \leftarrow \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{x}, \mathbf{r})$, we first solve equation 4 for $\mathbf{c}$. Since $\hat{\mathbf{e}} \neq \hat{\mathbf{e}}'$, let j be the highest index such that $\hat{e_j} \neq \hat{e_j}'$ and consider the $\mathsf{ZK\_sec} \times \mathsf{ZK\_sec}$ sub-matrix matrix of $M_{\hat{\mathbf{e}}} - M_{\hat{\mathbf{e}}'}$

consisting of rows between j and $j+\mathsf{ZK\_sec}-1$ (both included). This matrix is invertible, it is then possible to find a solution for $\mathbf{c}$. Since the cryptosystem is linearly homomorphic and the values $\mathbf{z}, \mathbf{z}'$ and T, T' are publicly known, it is possible to solve the system for $\mathbf{x}$ and $\mathbf{r}$ from the bottom equation to the one in the middle with index $\mathsf{ZK\_sec}/2$. To establish the bounds, recall that the plaintexts $\mathbf{z}, \mathbf{z}^{'}$ have norms less than $2 \cdot n \cdot 2^{\mathsf{ZK\_sec}} \cdot \tau$ and the randomness used for encrypting them, $\mathbf{t}_k, \mathbf{t}_k^{'}$, have norms less than $2 \cdot n \cdot \rho_1 \cdot 2^{\mathsf{ZK\_sec}}$ in the first coordinate and $2 \cdot n \cdot \rho_2 \cdot 2^{\mathsf{ZK\_sec}}$ in the last two coordinates where k ranges through $1, \ldots, \mathsf{sec}$.

Solving the linear system from the bottom row to the middle row via substitution we obtain in the worst case: $\|\mathbf{x}_k\|_\infty \leq 2^k \cdot 2^{\mathsf{ZK\_sec}} \cdot n \cdot \tau$ and the infinity norm of $\mathbf{y}_k$ is less than $2^k \cdot n \cdot 2^{\mathsf{ZK\_sec}} \cdot \rho_1$ in the first coordinate and less than $2^k \cdot 2^{\mathsf{ZK\_sec}} \cdot n \cdot \rho_2$ in the last two coordinates, where k ranges through $1, \ldots, \mathsf{sec}/2$.

To solve for $\mathfrak{ct}_{\mathsf{ZK\_sec}/2}, \ldots \mathfrak{ct}_1$ consider the lowest index j such that $\hat{e_j} \neq \hat{e_j}'$ and construct a lower triangular matrix, and solve as we did above for the case of the upper triangular sub-matrix. The bound on the resulting values is similarly obtained.

Thus we obtain overall bounds of $(2 \cdot 2^{\mathsf{ZK\_sec}/2} \cdot n \cdot 2^{\mathsf{ZK\_sec}} \cdot \tau, 2 \cdot 2^{\mathsf{ZK\_sec}/2} \cdot n \cdot 2^{\mathsf{ZK\_sec}} \cdot \rho_1, 2^{\mathsf{ZK\_sec}/2} \cdot 2 \cdot n \cdot 2^{\mathsf{ZK\_sec}} \cdot \rho_2)$, i.e. $(S \cdot n \cdot \tau, S \cdot n \cdot \rho_1, S \cdot n \cdot \rho_2)$ with $S = 2 \cdot 2^{3\mathsf{ZK\_sec}/2+1}$.

Finally, if $\mathsf{P} = \mathsf{Diag}$, then parties accept if all the $\mathbf{z}^i$ decode to diagonal values.

Honest verifier zero-knowledge: Here we give a simulator $\mathcal{S}$ for an honest verifier (each party P_i acts as one at one point during the protocol). The simulator's purpose is to create a transcript with the verifier which is indistinguishable from the real interaction between the prover and the verifier. To achieve this, $\mathcal{S}$ samples uniformly $\hat{\mathbf{e}} \leftarrow \{0,1\}^{\mathsf{ZK\_sec}}$ and then creates the transcript accordingly: sample $\mathbf{z}^i$ and T^i with respect to the bounds in the final check. The simulator then fixes $\mathbf{a}^i = \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{z}^i, T^i) - (M_{\hat{\mathbf{e}}} \cdot \mathbf{c}_i)$, where the encryption is applied component-wise. Clearly the produced transcript $(\mathbf{a}^i, \hat{\mathbf{e}}^i, \mathbf{z}^i, T^i)$ passes the final checks and the statistical distance to the real one is $2^{-\mathsf{ZK\_sec}}$, which is negligible with respect to $\mathsf{ZK\_sec}$. □

5 Distributed Somewhat Homomorphic Encryption

We are now ready to describe and implement the functionality $\mathcal{F}_{\mathsf{DistrDec}}$ (Figure 9) that extends the scheme $\mathcal{E}_{\mathsf{mBGV}}$ introduced in the previous sections to allow distributed decryption. It will be the main building block of our MPC protocol in the next section.

As mentioned before, our protocol ensures that all the ciphertexts that are input of $\mathcal{F}_{\mathsf{DistrDec}}$ correctly decrypt. For this purpose we use the ideal functionality $\mathcal{F}^{\nu,\mathsf{flag}}_{\mathsf{GenValidCiph}}$ (see Figure 15). Given the procedures $\overline{\Gamma}_{\mathbb{I}}$ and $\overline{\Gamma}_{\mathbb{J}}$ described in Figure 10, and on inputs $[\mathbf{m}]_i \in \mathcal{M}$ from each P_i, where $\mathcal{M} = \mathbb{Z}_{2^t}^{r \times |\mathbb{I}|}$ is the plaintext space of our encryption scheme, and $r \times |\mathbb{I}|$ is the number of supported slots, the functionality $\mathcal{F}^{\nu,\mathsf{flag}}_{\mathsf{GenValidCiph}}$ returns:

- If $\nu = 1$ and $\mathsf{flag} = \perp$, a valid ciphertext $\mathfrak{ct}^{\mathbb{I}}_{\mathbf{m}} \leftarrow \mathsf{BGV.Enc}(\overline{\Gamma}_{\mathbb{I}}^{-1}(\chi_{\mathbb{I}}(\mathbf{m})), \mathbf{r}; \mathfrak{pk})$, such that $\mathbf{m} = \sum_{i \in [n]} [\mathbf{m}]_i$; If $\nu = 1$ and $\mathsf{flag} = \mathsf{Diag}$ a valid ciphertext computed as before and satisfying the predicate $\mathsf{P} = \mathsf{Diag}$;
- If $\nu = 2$ and $\mathsf{flag} = \mathsf{Pack}$, two ciphertexts $\mathfrak{ct}^{\mathbb{I}}_{\mathbf{m}} \leftarrow \mathsf{BGV.Enc}(\overline{\Gamma}_{\mathbb{I}}^{-1}(\chi_{\mathbb{I}}(\mathbf{m})), \mathbf{r}; \mathfrak{pk})$ and $\mathfrak{ct}^{\mathbb{J}}_{\mathbf{m}} \leftarrow \mathsf{BGV.Enc}(\overline{\Gamma}_{\mathbb{J}}^{-1}(\chi_{\mathbb{J}}(\mathbf{m})), \mathbf{r}; \mathfrak{pk})$ satisfying the predicate $\mathsf{P} = \mathsf{Pack}$.

The ideal functionality $\mathcal{F}^{\nu,\mathsf{flag}}_{\mathsf{GenValidCiph}}$ is implemented by $\Pi^{\nu,\mathsf{flag}}_{\mathsf{GenValidCiph}}$ (see Figure 16 later in this section).

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

Let A be the set of corrupt parties.
PARAMETERS: B_{DDec}, a bound on the coefficients of the mask values, and B_{noise} a bound on the noise of ciphertexts before decryption.
COMMON INPUT: A single valid level-zero ciphertext $\mathfrak{ct}_{\mathbf{m}}^{(0,\mathbb{J})} = (0, c_0, c_1)^{\mathbb{J}}$ from all the parties.

Initialize: On receiving (Init) from all parties the functionality, run $(\mathfrak{pk}, \mathfrak{sk}) \leftarrow \mathsf{mBGV.KeyGen}(1^{\kappa})$, sending the value $\mathfrak{pk}$ to the adversary and all the parties.

D1: On receiving the public input $(\mathsf{D1}, \mathfrak{ct}_{\mathbf{m}}^{(0,\mathbb{J})})$ from all the parties, where $\mathfrak{ct}_{\mathbf{m}}^{(0,\mathbb{J})}$ is valid level-zero ciphertext, the functionality performs the following steps.
- Execute $\mathbf{m} \leftarrow \mathsf{Dec}(\mathfrak{ct}_{\mathbf{m}}^{(0,\mathbb{J})}; \mathfrak{sk})$ and handle this value to the adversary.
- If P_1 *is honest*: Wait for the adversary to input either abort or $\boldsymbol{\delta}$. If abort, then forward abort to the honest parties and halt. Otherwise sample the honest shares $[\mathbf{m}]_i \leftarrow \mathcal{M}, i \notin A, i \neq 1$, at random and set $[\mathbf{m}]_1 = -\sum_{i \notin A, i \neq 1} [\mathbf{m}]_i + \mathbf{m} + \boldsymbol{\delta}$. Send $[\mathbf{m}]_i$ to $P_i, \forall i \notin A$.
- If P_1 *is corrupt*: Send $\mathbf{m}$ to the adversary. Wait for an input from the adversary. If this input is abort, then forward abort to the honest parties and halt. Otherwise receive $\mathbf{b}$. Sample the honest shares $[\mathbf{m}]_i \leftarrow \mathcal{M}, i \notin A$, at random but subject to the condition $\sum_{i \notin A} [\mathbf{m}]_i = \mathbf{b}$. Send these values $\mathbf{m}_i, i \notin A$ to the honest parties.

D2: On receiving $(\mathsf{D2}, \mathfrak{ct}_{\mathbf{m}}^{(0,\mathbb{J})})$ from all parties, the functionality performs the following steps.
- Execute $\mathbf{m} \leftarrow \mathsf{Dec}(\mathfrak{ct}_{\mathbf{m}}^{(0,\mathbb{J})}; \mathfrak{sk})$ and send $\mathbf{m}$ to the adversary.
- Wait for an input from the adversary: if abort is received, then abort.
- Otherwise receive $\mathbf{m}'$ and $\{[\mathbf{m}']_i\}_{i \in A}$. Sample random shares $\{[\mathbf{m}']_i\}_{i \notin A}$ such that $\sum_{i \in [n]} \{[\mathbf{m}']_i\} = \mathbf{m}'$.
- Output $\{[\mathbf{m}']_i\}_{i \notin A}$ to honest parties and $\hat{\mathfrak{ct}}_{\mathbf{m}'}^{(1,\mathbb{I})}$ to all parties.

Figure 9. The functionality for distributed decryption

The Procedures $\overline{\Gamma}_{\mathbb{I}}^{-1}(\mathbf{m})$ (resp. $\overline{\Gamma}_{\mathbb{J}}^{-1}(\mathbf{m})$)

1. If computing $\overline{\Gamma}_{\mathbb{I}}^{-1}(\mathbf{m})$ set all entries in $\mathbf{m}$ *not in* $\mathsf{Supp}(\mathbb{I})$ to zero.
2. If computing $\overline{\Gamma}_{\mathbb{J}}^{-1}(\mathbf{m})$ set all entries in $\mathbf{m}$ *not in* $\mathsf{Supp}(\mathbb{J})$ to a uniformly random element selected from $[0, \ldots, 2^t]$.
3. Output $\Gamma^{-1}(\mathbf{m})$.

Figure 10. The procedure $\overline{\Gamma}_{\mathbb{I}}^{-1}(\mathbf{m})$ (resp. $\overline{\Gamma}_{\mathbb{J}}^{-1}(\mathbf{m})$) from $\overline{\mathcal{P}}$ to $\mathcal{P}$

5.1 Distributed Decryption Protocols

Here we give two distributed decryption protocols, $\Pi_{\mathsf{DistrDec1}}$ and $\Pi_{\mathsf{DistrDec2}}$, in Figure 11 and Figure 13, respectively. The protocols $\Pi_{\mathsf{DistrDec1}}$ and $\Pi_{\mathsf{DistrDec2}}$ implement the functionality $\mathcal{F}_{\mathsf{DistrDec}}$ (Figure 9) on commands **D1** and **D2**, respectively. Notice that we do not perform a proper full distributed decryption, because the way we pack entries into a ciphertext would result in information leakage if we allowed all the parties to recover the plaintext corresponding to the public input ciphertext $\mathfrak{ct}_{\mathbf{m}}^{(0,\mathbb{J})}$, but both our protocols output to each party P_i an additive share $[\mathbf{m}]_i$ of $\mathbf{m}$. Both protocols depend on a constant B_{noise} which represents a bound on the ciphertext noise before a decryption occurs. For example, in case of fresh ciphertexts we have that $B_{\mathsf{noise}} = B_{\mathsf{clean}}^{\mathsf{dishonest}}$ (see Section A in the Appendix).

There are two main differences between the two protocols. The first one is in the way the shares $[\mathbf{m}]_i$ are computed. The protocol $\Pi_{\mathsf{DistrDec2}}$ is essentially the same as the $\mathsf{Reshare}$ protocol of [DPSZ12, DKL+13], where a masking ciphertext is used before the distributed decryption is performed. More precisely, parties call the functionality $\mathcal{F}_{\mathsf{GenValidCiph}}^{2,\mathsf{Pack}}$ which produces two ciphertexts

$(\mathfrak{ct}_{\mathbf{f}}^{(1,\mathbb{I})}, \mathfrak{ct}_{\mathbf{f}}^{(1,\mathbb{J})})$, with $\mathbf{f} = \sum_{i\in[n]}[\mathbf{f}]_i$; then they decrypt $\mathfrak{ct}_{\mathbf{m}+\mathbf{f}}^{(0,\mathbb{J})} = \mathfrak{ct}_{\mathbf{m}}^{(0,\mathbb{J})} \oplus \mathfrak{ct}_{\mathbf{f}}^{(0,\mathbb{J})}$, where $\mathfrak{ct}_{\mathbf{f}}^{(0,\mathbb{J})} = \mathsf{SwitchMod}(\mathfrak{ct}_{\mathbf{f}}^{(1,\mathbb{J})})$, so that each P_i can compute a share $[\mathbf{m}+\mathbf{f}]_i - [\mathbf{f}]_i$ of $\mathbf{m}$.

On the other hand, the protocol $\Pi_{\mathsf{DistrDec1}}$ uses random masks $f_i, i \in [n]$, inside the actual decryption to mask the decryption shares, so it does not require to perform any expensive zero-knowledge proof. Note that this approach cannot be used if the parties need to generate a new fresh ciphertext of $\mathbf{m}$ after the decryption, as happens in $\Pi_{\mathsf{DistrDec2}}$, where this fresh encryption is computed using the first ciphertext $\mathfrak{ct}_{\mathbf{f}}^{(1,\mathbb{I})}$ given by $\mathcal{F}_{\mathsf{GenValidCiph}}^{2,\mathsf{Pack}}$.

Protocol $\Pi_{\mathsf{DistrDec1}}$

PARAMETERS: The protocol is parametrized by two bounds: B_{DDec}, a bound on the coefficients of the mask values, and B_{noise} a bound on the noise of ciphertexts before decryption.
COMMON INPUT: A single valid level-zero ciphertext $\mathfrak{ct}_{\mathbf{m}}^{(0,\mathbb{J})} = (0, c_0, c_1)^{\mathbb{J}}$.

Initialize: Each party P_i calls $\mathcal{F}_{\mathsf{KeyGen}}$ receiving $(\mathfrak{pk}, [\mathfrak{sk}]_i)$.
D1: On input $(\mathsf{D1}, \mathfrak{ct}_{\mathbf{m}}^{(0,\mathbb{J})})$, where $\mathfrak{ct}_{\mathbf{m}}^{(0,\mathbb{J})} = (0, c_0, c_1)^{\mathbb{J}}$ is a (single) ciphertext, parties do as follows.

1. Each P_i samples $f_i \leftarrow [0, B_{\mathsf{DDec}}]^N$ (i.e. a polynomial in R with bounded coefficients).
2. P_1 computes $v_1 \leftarrow \big((c_0 - [\mathfrak{sk}]_1 \cdot c_1) + f_1 \pmod{q_0}\big) = w_1 + f_1 \pmod{q_0}$.
   Each $P_i, i \neq 1$ computes $v_i \leftarrow \big(-[\mathfrak{sk}]_i \cdot c_1 + f_i \pmod{q_0}\big) = w_i + f_i \pmod{q_0}$. All parties broadcast these values.
3. Parties check that $(\sum_i v_i \pmod{q_0})$ is bounded by $B_{\mathsf{noise}} + n \cdot B_{\mathsf{DDec}}$, if not abort.
4. P_1 computes $u_1 \leftarrow (\sum_{i=1}^{n} v_i \pmod{q_0}) - f_1 \pmod{2^t}$.
   Each $P_i, i \neq 1$ computes $u_i \leftarrow -f_i \mod 2^t$.
5. Each $P_i, i \in [n]$, sets $[\mathbf{m}]_i \leftarrow \chi_{\mathbb{J}}^{-1}(\Gamma(u_i))$.

Figure 11. Protocol implementing the command **D1** on $\mathcal{F}_{\mathsf{DistrDec}}$

Protocol $\Pi_{\mathsf{DistrDec1}}$. Given a public input ciphertext $\mathfrak{ct}_{\mathbf{m}}^{(0,\mathbb{J})}$, each party P_i samples a random polynomial f_i in R, with coefficients bounded by some fixed, large enough value B_{DDec} to avoid any leakage of information in the secret key, which is used to mask the decryption share.

Note that the correctness holds only if the values f_i introduced by the parties during the protocol are sampled from the right set, i.e. $\|f_i\|_\infty < B_{\mathsf{DDec}}$, and $\|\sum_{i\in[n]} v_i \pmod{q_0}\|_\infty < B_{\mathsf{noise}} + n \cdot B_{\mathsf{DDec}} < q_0/2$. We will derive the precise value B_{DDec} in the security proof.

In terms of protocol security, the intuition is that the polynomial f_i masks not only the values in $\mathsf{Supp}(\mathbb{J})$ which contain information, but also values not in $\mathsf{Supp}(\mathbb{J})$ which could contain residual information from prior homomorphic operations. So, the fact that the honest party effectively "forgets" the values corresponding to slot terms not in $\mathsf{Im}(\omega_{\mathbb{J}})$ results in the protocol not leaking information on these terms. A complete proof of this intuition can be found below.

Theorem 5.1. *The protocol $\Pi_{\mathsf{DistrDec1}}$ (Figure 11) implements the functionality $\mathcal{F}_{\mathsf{DistrDec}}.\mathsf{D1}$ (Figure 9) against any static, active adversary corrupting up to $n-1$ parties in the $\mathcal{F}_{\mathsf{KeyGen}}$-hybrid model with statistical security $2^{-\mathsf{DDec}}$ if $\big(B_{\mathsf{noise}} + 2^{\mathsf{DDec}} \cdot n \cdot (B_{\mathsf{noise}} + 2^t)\big) < q_0/2$.*

Proof. First we show correctness. We have to prove that the value $\mathbf{m}$ shared by the protocol equals the value $\chi_{\mathbb{J}}^{-1}(\Gamma(\mathsf{BGV.Dec}(\mathfrak{ct}_{\mathbf{m}}^{(0,\mathbb{J})}, \mathfrak{sk})))$. But this is immediate, from the equations

$$\mathbf{m} = \sum_i [\mathbf{m}]_i = \sum_i \chi_{\mathbb{J}}^{-1}(\Gamma(u_i)) = \chi_{\mathbb{J}}^{-1}(\Gamma(\sum_i u_i))$$

$$
\begin{aligned}
&= \chi_{\mathbb{J}}^{-1}\Big(\Gamma\big(\big(\sum_i v_i - f_1 \pmod{q_0}\big) - \sum_{i\neq 1} f_i \mod 2^t\big)\Big)\\
&= \chi_{\mathbb{J}}^{-1}\Big(\Gamma\big(\big(\sum_i w_i \pmod{q_0}\big) \mod 2^t\big)\Big)\\
&\quad + \chi_{\mathbb{J}}^{-1}\Big(\Gamma\big(\big(\sum_{i\neq 1} f_i - f_1 \pmod{q_0}\big) - \sum_{i\neq 1} f_i \mod 2^t\big)\Big)\\
&= \chi_{\mathbb{J}}^{-1}\Big(\Gamma\big((c_0 - \sum_i [\mathfrak{sk}]_i \cdot c_1 \pmod{q_0}) \mod 2^t\big)\Big)\\
&\quad + \chi_{\mathbb{J}}^{-1}\Big(\Gamma\big(\big(\tilde{f} - \tilde{f}_1 - \sum_{i\neq 1} f_i \mod 2^t\big)\Big) = \chi_{\mathbb{J}}^{-1}(\Gamma(\mathsf{BGV.Dec}(\mathfrak{ct}^{(0)}, \mathfrak{sk}))),
\end{aligned}
$$

where we denote $\tilde{f} = \sum_i f_i \pmod{q_0}$ and $\tilde{f}_1 = f_1 \pmod{q_0}$. Note that the correctness holds only if the values f_i introduced by the parties during the protocol are sampled from the right set, i.e. $\|f_i\|_\infty < B_{\mathsf{DDec}}$, and $\|\sum_{i\in[n]} v_i \mod q_0\|_\infty < B_{\mathsf{noise}} + n \cdot B_{\mathsf{DDec}} < q_0/2$.

Let $\mathcal{A}$ be a static real world adversary corrupting up to $n-1$ parties, we construct an ideal world adversary $\mathcal{S}$ (Figure 12) interacting with $\mathcal{F}_{\mathsf{DistrDec}}.\mathsf{D1}$, and show that no environment $\mathcal{Z}$ can distinguish between an interaction with $\mathcal{A}$ in the protocol $\Pi_{\mathsf{DistrDec1}}$ and an interaction with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{DistrDec}}.\mathsf{D1}$ in the ideal world.

To argue indistinguishability between the ideal and real execution to an environment $\mathcal{Z}$, recall that $\mathcal{Z}$ can choose the common input $\mathfrak{ct}_{\mathbf{m}}^{(0,\mathbb{J})}$, and that its view consists of this input, all the messages received by the adversary, namely the public key $\mathfrak{pk}$, $\{v_i\}_{i\notin A}$, other than all the adversary random tapes, and all the outputs $[\mathbf{m}]_i, i \in [n]$.

The simulator starts by emulating the $\mathcal{F}_{\mathsf{KeyGen}}$ functionality, obtaining the actual $[\mathfrak{sk}]_i, i \in A$, from the adversary and creates honest shares $[\mathfrak{sk}]_i, i \notin A$, such that the sum of all the secret key shares is a valid secret key. The distribution of the public key $\mathfrak{pk}$ that $\mathcal{S}$ sends to the adversary is exactly the same as in a real execution, as it is obtained by running KeyGen as in the real protocol. Using these $\mathfrak{sk}$'s shares the simulator can compute $w_i = -[\mathfrak{sk}]_i \cdot c_1, \forall i \in [n], i \neq 1$, and $w_1 = c_0 - [\mathfrak{sk}]_1 \cdot c_1$. After that we need to distinguish between the cases P_1 honest and P_1 corrupt.

First we recall that given a value $\mathbf{m} \in \mathcal{M}$, $\Theta_{\mathbb{J}}^{-1}(\mathbf{m}, B_{\mathsf{DDec}})$ is computed as follows: 1) First compute $m_{\overline{\mathcal{P}}} \in \overline{\mathcal{P}}$ using the map $\chi_{\mathbb{J}}^{-1}$ 2) For each entry not in $\mathsf{Supp}(\mathbb{J})$, sample a uniformly random element in $[0, \ldots, 2^t]$, so to obtain a uniform random element $m'_{\overline{\mathcal{P}}} \in \overline{\mathcal{P}}$ such that $\chi_{\mathbb{J}}(m'_{\overline{\mathcal{P}}}) = \mathbf{m}$ 3) Compute $m'_R \leftarrow \Psi_{2^t}^{-1}(m'_{\overline{\mathcal{P}}})$ with coefficients in $[0, \ldots, 2^t]$. 4) Sample $u \leftarrow R$ uniformly at random whose coefficient infinity norm is bounded by $B_{\mathsf{DDec}}/2^t$ 5) Output $m_R \leftarrow m'_R + 2^t \cdot u$. So if B_{DDec} is large enough, the output value m_R is within statistical distance from the uniform distribution in $[0, B_{\mathsf{DDec}}]$.

If P_1 *is honest*: $\mathcal{S}$ generates all the v_i's, $i \neq 1$, honestly, so from the discussion above we have that these values are perfectly simulated because they are obtained using shares of a possible secret key and random masks $f_i \leftarrow [0, B_{\mathsf{DDec}}]^N$, for large enough B_{DDec}. The value $v_1 = (-\sum_{i\neq 1} w_i + \Theta_{\mathbb{J}}^{-1}(\mathbf{m}, B_{\mathsf{DDec}}) \pmod{q_0})$ generated by the simulator is also statistical indistinguishable from the real word value $v_1 = w_1 + f_1 \pmod{q_0}$ except with negligible probability $2^{-\mathsf{DDec}}$, since f_1 has coefficients bounded by B_{DDec} in both executions, and hence, using the smudging lemma, the

two distributions are both within statistical distance from the uniform in $[0, B_{\mathsf{DDec}}]$, as long as $B_{\mathsf{DDec}} \geq 2^{\mathsf{DDec}} \cdot B_{\mathsf{noise}}$.

It remains to prove indistinguishability of the outputs. The environment sees the honest shares. These values are random but consistent with the actual plaintext $\mathbf{m}$, some adversarial chosen value $\boldsymbol{\delta}$ and the simulated $v_i, i \notin A$. More in particular, indistinguishability for shares $[\mathbf{m}]_i, i \notin A, i \neq 1$, is straightforward. The simulated value $[\mathbf{m}]_1$ is such that: $[\mathbf{m}]_1 = -\sum_{i \notin A, i \neq 1} [\mathbf{m}]_i + \mathbf{m} + \boldsymbol{\delta}$, with $\boldsymbol{\delta} \leftarrow \chi_{\mathbb{J}}^{-1}(\Gamma((E + \sum_{i \in A} f_i \pmod{q_0}) \mod 2^t))$ and in the real execution:

$$
\begin{aligned}
[\mathbf{m}]_1 &= \chi_{\mathbb{J}}^{-1}\Big(\Gamma\big(u_1\big)\Big) = \chi_{\mathbb{J}}^{-1}\Big(\Gamma\big(\big(\sum_{i\in[n]} v_i \pmod{q_0}\big) \mod 2^t\big)\Big) \\
&= \chi_{\mathbb{J}}^{-1}\Big(\Gamma\big(\big(\sum_{i\in[n]} w_i \pmod{q_0}\big) \mod 2^t\big)\Big) \\
&\quad + \chi_{\mathbb{J}}^{-1}\big(\Gamma\big((E \pmod{q_0}) \mod 2^t\big)\big) \\
&\quad + \chi_{\mathbb{J}}^{-1}\big(\Gamma\big(\big(\sum_{i\in n} f_i \pmod{q_0}\big) - f_1 \mod 2^t\big)\big) \\
&= \mathbf{m} + \chi_{\mathbb{J}}^{-1}\big(\Gamma\big(\big(\sum_{i\notin A} f_i \pmod{q_0} \mod 2^t\big)\big) \\
&+ \chi_{\mathbb{J}}^{-1}\big(\Gamma\big((E + \sum_{i\in A} f_i \pmod{q_0}) \mod 2^t\big)\big),
\end{aligned}
$$

so the two values are indistinguishable.

If P_1 *is corrupt*: The indistinguishability argument is similar to the previous case. Compared to that, we need to show that the value v_j computed in the ideal world is indistinguishable from the random value sampled in the real protocol. It is easy to see that this is the case as long as $B_{\mathsf{DistrDec}_1} > 2^{\mathsf{sec}} \cdot (B_{\mathsf{noise}} + 2^t)$. It is also easy to verify, similarly to the previous case that the honest output values are random and consistent with the $v_i, i \notin A$, generated by the simulator and sent to the adversary. □

Protocol $\Pi_{\mathsf{DistrDec2}}$. Given a public ciphertext $\mathfrak{ct}_{\mathbf{m}}^{(0,\mathbb{J})}$, the protocol $\Pi_{\mathsf{DistrDec2}}$ outputs a share $[\mathbf{m}]_i$ of the plaintext $\mathbf{m}$ and a fresh ciphertext $\mathfrak{ct}_{\mathbf{m}}^{(1,\mathbb{I})}$ to each party P_i. The protocol makes use of the command **Gen-2** of the functionality $\mathcal{F}_{\mathsf{GenValidCiph}}^{2,\mathsf{Pack}}$ (Figure 15), which is implemented in Figure 16. This command outputs two level-1 ciphertexts $\mathfrak{ct}_{\mathbf{f}}^{(1,\mathbb{I})}$ and $\mathfrak{ct}_{\mathbf{f}}^{(1,\mathbb{J})}$ of the same plaintext $\mathbf{f}$ corresponding to the set $\mathbb{I}$ and $\mathbb{J}$, respectively.

The ciphertext $\mathfrak{ct}_{\mathbf{f}}^{(1,\mathbb{J})}$, corresponding to the set $\mathbb{J}$, is used as a mask in the distributed decryption, and $\mathfrak{ct}_{\mathbf{f}}^{(1,\mathbb{I})}$, corresponding to the set $\mathbb{I}$, is used to create a fresh encryption $\hat{\mathfrak{ct}}_{\mathbf{m}}^{(1,\mathbb{I})}$ of $\mathbf{m}$.

The proof of security for this protocol is similar to the corresponding protocol in SPDZ [DPSZ12]. The major changes from SPDZ are that we need to produce two auxiliary ciphertexts per party $(\mathfrak{ct}_{\mathbf{f}_i}^{(1,\mathbb{I})}, \mathfrak{ct}_{\mathbf{f}_i}^{(1,\mathbb{J})})$, since we have different encodings at level zero and level one of the underlying message space. Intuitively, the protocol reveals no more information about the BGV plaintext inside $\mathfrak{ct}_{\mathbf{m}}^{(0,\mathbb{J})}$ because the honest parties are masking the coefficients not in $\mathsf{Supp}(\mathbb{J})$ using the coefficients from the plaintext inside $\mathfrak{ct}_{\mathbf{f}_i}^{(1,\mathbb{J})}$, which have been chosen to be uniformly random for coefficients not in $\mathsf{Supp}(\mathbb{J})$, using the procedure $\overline{\Gamma}_{\mathbb{J}}^{-1}$. A proof for this result is given below.

Theorem 5.2. *The protocol $\Pi_{\mathsf{DistrDec2}}$ implements the functionality $\mathcal{F}_{\mathsf{DistrDec}}.\mathsf{D2}$ (Figure 9) against any static, active adversary corrupting up to $n-1$ parties in the $(\mathcal{F}_{\mathsf{KeyGen}}, \mathcal{F}^{2,\mathsf{Pack}}_{\mathsf{GenValidCiph}})$-hybrid model with statistical security $2^{-\mathsf{DDec}}$ if $(B_{\mathsf{noise}} + 2^{\mathsf{DDec}} \cdot n \cdot (B_{\mathsf{noise}} + 2^t)) < q_0/2$*

Proof. The proof is essentially the same given in [DPSZ12]. Let $\mathcal{A}$ be a static real world adversary corrupting up to $n-1$ parties, we give an ideal world adversary $\mathcal{S}$ (Figure 14) interacting with $\mathcal{F}_{\mathsf{DistrDec}}.\mathsf{D2}$, and show that no environment $\mathcal{Z}$ can distinguish between an interaction with $\mathcal{A}$ in the protocol $\Pi_{\mathsf{DistrDec2}}$ and an interaction with $\mathcal{S}$ and the functionality in the ideal world. To prove the property of the simulation, essentially we need to prove that the simulated value v_j is indistinguishable from the real value. Again we can use the Smudging lemma, so the two values are indistinguishable from a uniform value in $[0, B_{\mathsf{DDec}}]$ as long as $2^{\mathsf{DDec}-t} \cdot (B_{\mathsf{noise}} + 2^t) \geq B_{\mathsf{DDec}}$ with negligible probability $2^{-\mathsf{DDec}}$. The shares $[\mathbf{m}]_i$ are also indistinguishable because in the real protocol the values $\mathfrak{ct}^{(1,\mathbb{J})}_{\mathbf{f}_i}$ and $\mathfrak{ct}^{(1,\mathbb{J})}_{\mathbf{f}}$ are obtained by using the $\overline{\Gamma}^{-1}_{\mathbb{J}}$ procedure that chooses the coefficients not in $\mathsf{Supp}(\mathbb{J})$ uniformly at random. □

5.2 Generating Valid Ciphertexts

Here we implement the ideal functionality $\mathcal{F}^{\nu,\mathsf{flag}}_{\mathsf{GenValidCiph}}$ to create valid ciphertexts, see Figure 16. To prove the security of $\Pi^{\nu,\mathsf{flag}}_{\mathsf{GenValidCiph}}$ we proceed like in [DPSZ12], that is we assume that the encryption scheme $\mathcal{E}_{\mathsf{mBGV}}$ has an additional key generation algorithm $\widetilde{\mathsf{KeyGen}}()$ that outputs a meaningless public key $\widetilde{\mathfrak{pk}}$ such that

- $\mathsf{Enc}(m, \widetilde{\mathfrak{pk}}) \overset{\mathrm{s}}{\approx} \mathsf{Enc}(0, \widetilde{\mathfrak{pk}})$, i.e. an encryption of any message m is statistically indistinguishable from an encryption of 0;
- If $\widetilde{\mathfrak{pk}} \leftarrow \widetilde{\mathsf{KeyGen}}()$ and $(\mathfrak{pk}, \mathfrak{sk}) \leftarrow \mathsf{KeyGen}()$, then $\mathfrak{pk} \overset{\mathrm{c}}{\approx} \widetilde{\mathfrak{pk}}$, namely the two public keys are computationally indistinguishable.

In $\mathcal{E}_{\mathsf{BGV}}$ the algorithm $\widetilde{\mathsf{KeyGen}}()$ just samples $\widetilde{\mathfrak{pk}} = (\tilde{a}, \tilde{b})$ uniformly at random $\mod q_1$.

The high level idea of the proof is then the following. We describe a simulator $\mathcal{S}$ and show that if an environment $\mathcal{Z}$ can distinguish the simulation from the real protocol execution, then we can construct a distinguisher that by rewinding the environment togheter with the adversary can distinguish between a public key $\mathfrak{pk}$ generated by KeyGen and a meaningless $\widetilde{\mathfrak{pk}}$ with non negligible probability. To this purpose we need to generalise the proof in [DPSZ12] to our multiparty global zero knowledge of plaintext knowledge.

Theorem 5.3. *The protocol $\Pi^{\nu,\mathsf{flag}}_{\mathsf{GenValidCiph}}$ securely implements the functionality $\mathcal{F}^{\nu,\mathsf{flag}}_{\mathsf{GenValidCiph}}$ (Figure 9) against any static, active adversary corrupting up to $n-1$ parties in the $(\mathcal{F}_{\mathsf{KeyGen}}, \mathcal{F}_{\mathsf{Rand}})$-hybrid model.*

Proof. We describe a simulator $\mathcal{S}$ (Figure 17) and show that any environment $\mathcal{Z}$ who can distinguish between the real and ideal execution of the protocol can be used to construct a distinguisher $\mathcal{D}$ which breaks the computational indistinguishability assumption between any normal $\mathfrak{pk} \leftarrow \mathsf{KeyGen}()$ and meaningless $\widetilde{\mathfrak{pk}} \leftarrow \widetilde{\mathsf{KeyGen}}()$.

First, the simulator emulates $\mathcal{F}_{\mathsf{KeyGen}}$ by running $\mathsf{KeyGen}()$ to generate $(\mathfrak{pk}, \mathfrak{sk})$, sends $\mathfrak{pk}$ to all parties and stores the secret key $\mathfrak{sk}$. So the simulator can decrypt all the received ciphertexts since

they know $\mathfrak{sk}$. Then they run the protocol honestly computing all the corrupt parties' plaintexts by decrypting.

We now show that no environment $\mathcal{Z}$ can distinguish between the real and ideal process. Using the standard UC notation, we let $\mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}$ denote the random variable describing the output of $\mathcal{Z}$ in a real execution of Π with adversary $\mathcal{A}$. Similarly, we let $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ denote the random variable describing the output of $\mathcal{Z}$ after interacting with the ideal execution with adversary $\mathcal{S}$ and functionality $\mathcal{F}$. We assume the output of $\mathcal{Z}$ to be a single bit, considered as a guess at one of the two executions REAL or IDEAL. The advantage of $\mathcal{Z}$ is then given by:

$$\mathrm{Adv}(\mathcal{Z}) = |\Pr[\mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}} = 1] - \Pr[\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} = 1]|.$$

Now, suppose there exists a set of inputs for which $\mathcal{Z}$ distinguishes between the two worlds with noticeable advantage, δ. We prove that there exists a distinguisher $\mathcal{D}$, which breaks the indistinguishability assumption of the output of KeyGen and $\widetilde{\mathsf{KeyGen}}$. $\mathcal{D}$ sets up a copy of $\mathcal{Z}$ and, on input $\mathfrak{pk}^*$, it goes through the protocol with $\mathcal{Z}$. Note $\mathfrak{pk}^*$ is either the output of $\mathsf{KeyGen}()$ or $\widetilde{\mathsf{KeyGen}}()$. Then it uses the output of this simulation to guess if $\mathfrak{pk}^*$ was generated by $\mathsf{KeyGen}()$ or $\widetilde{\mathsf{KeyGen}}()$. $\mathcal{D}$ runs the protocol in the same way the simulator would do, with some exceptions, as explained in the following.

- Use $\mathfrak{pk}^*$ as described above, instead of generating the public key using $\mathsf{KeyGen}()$
- To extract corrupt parties' input, sample $\mathbf{e} \in \{0,1\}^u$ and use the zero-knowledge proof. Receive $\{\mathbf{z}_k^i, T_k^i\}_{i\in A}$, and rewind the adversary. Sample $\bar{\mathbf{e}} \neq \mathbf{e}$ and receive back $\{\bar{\mathbf{z}}_z, T_k^i\}_{i\in A}$. Use the knowledge extractor to compute $\{\mathbf{x}_k^i\}_{i\in A}$
- To simulate honest parties' transcripts use the honest-verifier zero knowledge and generate $\{\mathbf{a}_k^i, \mathbf{e}, \mathbf{z}_k^i, T_k^i\}_{i\notin A}$.

$\mathcal{D}$ uniformly chooses to output a simulated (as described above) or a real view. We denote these views by $\mathcal{D}_{\mathsf{IDEAL}}$ and $\mathcal{D}_{\mathsf{REAL}}$, respectively. Now we distinguish two different cases:

- If $\mathfrak{pk}^* \leftarrow \mathsf{KeyGen}$: In this case the view generated by $\mathcal{D}$ is statistically indistinguishable to either the view generated by the simulator $\mathcal{S}$ or the one produced in the real protocol. So in this case $\mathcal{Z}$ is able to distinguish between $\mathcal{D}_{\mathsf{IDEAL}}$ and $\mathcal{D}_{\mathsf{REAL}}$ with advantage δ.
- If $\mathfrak{pk}^* \leftarrow \widetilde{\mathsf{KeyGen}}$: Since the key is meaningless and the encryptions contain statistically no information about the corresponding plaintexts, $\mathcal{D}_{\mathsf{IDEAL}} \overset{\mathrm{s}}{\approx} \mathcal{D}_{\mathsf{REAL}}$ and $\mathcal{Z}$ can only guess between the two with probability $1/2$.

Summing up:

$$\mathrm{Adv}(\mathcal{D}) \geq \mathrm{Adv}(\mathcal{Z})/2 - \epsilon = \delta/2 - \epsilon,$$

for some negligible value ϵ. □

Simulator $\mathcal{S}_{\mathsf{DistrDec}_1}$

Let A be the set of corrupt parties.

- Emulate $\mathcal{F}_{\mathsf{KeyGen}}$ receiving $(\mathfrak{pk}, \mathfrak{sk})$ and $\{[\mathfrak{sk}]_i\}_{i\in A}$ from $\mathcal{A}$.
  Sample $\{[\mathfrak{sk}]_i\}_{i\notin A}$ consistently. Send $\mathfrak{pk}$ to the adversary.
- Receive $\mathbf{m} \leftarrow \chi_{\mathbb{J}}^{-1}\big(\Gamma(\mathsf{BGV.Dec}(\mathfrak{ct}_{\mathbf{m}}^{(0,\mathbb{J})}, \mathfrak{sk}))\big)$ from the functionality.

P_1 **is honest:**

- Compute $w_i = -[\mathfrak{sk}]_i \cdot c_1, \forall i \neq 1$
- Sample random $\{f_i\}_{i\notin A}$
- Compute v_i honestly for each $P_i, i \notin A$, except for honest P_1
- Set
$$v_1 = -\sum_{i\neq 1} w_i + \Theta_{\mathbb{J}}^{-1}(\mathbf{m}, B_{\mathsf{DDec}}) \pmod{q_0}.$$
- Send $\{v_i\}_{i\notin A}$ to $\mathcal{A}$ and receive $\{v_i^*\}_{i\in A}$ from $\mathcal{A}$
- If $(\sum_{i\in[n]} v_i \pmod{q_0})$ is not bounded by the value $B_{\mathsf{noise}} + n \cdot B_{\mathsf{DDec}}$ send abort to the functionality, otherwise compute
$$\sum_{i\in A}(v_i^* - w_i^{)}) = \sum_{i\in A} \tilde{f}_i = E + \sum_{i\in A} f_i,$$
where $\{w_i\}_{i\in A}$ are honestly computed (i.e. computed used the actual secret keys obtained by $\mathcal{A}$), and $\{f_i\}_{i\in A}$ and E is an adversarial chosen value. Send
$$\boldsymbol{\delta} \leftarrow \chi_{\mathbb{J}}^{-1}(\Gamma((\sum_{i\in A} \tilde{f}_i \pmod{q_0}) \mod 2^t))$$
to the functionality.

P_1 **is corrupt:**

- Compute $w_i = -[\mathfrak{sk}]_i \cdot c_1, \forall i \in [n]$
- Sample random $\{f_i\}_{i\notin A}$ for honest parties
- Compute v_i honestly for each $P_i, i \notin A$, except for a honest P_j
- Set
$$v_j = -\sum_{i\neq 1} w_i + f_j + \Gamma^{-1}\Big(\chi_{\mathbb{J}}\big(\mathbf{m}\big)\Big) \pmod{q_0}.$$
- Send $\{v_i\}_{i\notin A}$ to $\mathcal{A}$ and receive $\{v_i^*\}_{i\in A}$ from $\mathcal{A}$
- If $(\sum_{i\in[n]} v_i \pmod{q_0})$ is not bounded by $B_{\mathsf{noise}} + n \cdot B_{\mathsf{DDec}}$ send abort to the functionality, otherwise compute
$$\bar{b}_i \leftarrow -f_i \mod 2^t \text{ and } [\mathbf{b}]_i \leftarrow \chi_{\mathbb{J}}^{-1}(\Gamma(\bar{b}_i)), \forall i \notin A.$$
Send $\mathbf{b} = \sum_{i\notin A}[\mathbf{b}]_i$ to the functionality.

Figure 12. Simulator for **D1**

Protocol $\Pi_{\mathsf{DistrDec2}}$

PARAMETERS: The protocol is parametrized by B_{DDec}.
COMMON INPUT: A single valid level-zero ciphertext $\mathfrak{ct}_{\mathbf{m}}^{(0,\mathbb{J})} = (0, c_0, c_1)^{\mathbb{J}}$.

Initialize: Each party P_i calls $\mathcal{F}_{\mathsf{KeyGen}}$ receiving $(\mathfrak{pk}, [\mathfrak{sk}]_i)$

D2: On input $(\mathsf{D2}, \mathfrak{ct}_{\mathbf{m}}^{(0,\mathbb{J})})$ from all parties, where $\mathfrak{ct}_{\mathbf{m}}^{(0)} = (0, c_0, c_1)^{\mathbb{J}}$ is a (single) ciphertext.

1. Parties call the functionality $\mathcal{F}_{\mathsf{GenValidCiph}}^{2,\mathsf{Pack}}$ on input $[\mathbf{f}]_i, \forall i \in [n]$, which returns the ciphertexts $(\mathfrak{ct}_{\mathbf{f}}^{(1,\mathbb{I})}, \mathfrak{ct}_{\mathbf{f}}^{(1,\mathbb{J})})$ to *all* parties.
2. All the parties locally compute $\mathfrak{ct}_{\mathbf{f}}^{(0,\mathbb{J})} = \mathsf{SwitchMod}(\mathfrak{ct}_{\mathbf{f}}^{(1,\mathbb{J})})$.
3. The parties homomorphically compute $\mathfrak{ct}_{\mathbf{m}+\mathbf{f}}^{(0,\mathbb{J})} = \mathfrak{ct}_{\mathbf{m}}^{(0)} \oplus \mathfrak{ct}_{\mathbf{f}}^{(0,\mathbb{J})}$, and let $\mathfrak{ct}_{\mathbf{m}+\mathbf{f}}^{(0,\mathbb{J})}$ be $(0, c_0, c_1)$.
4. P_1 computes $v_1 \leftarrow (c_0 - \mathfrak{sk}_1 \cdot c_1) \pmod{q_0} \in R_{q_0}$.
5. $P_i, i \neq 1$ computes $v_i \leftarrow -\mathfrak{sk}_i \cdot c_1 \pmod{q_0} \in R_{q_0}$.
6. All parties compute and broadcast $t_i = v_i + 2^t \cdot r_i$ for some random element $r_i \in R_{q_0}$ with infinity norm bound B_{DDec}.
7. The parties compute $(\mathbf{m} + \mathbf{f}) = \chi_{\mathbb{J}}^{-1}(\Psi_{2^t}(\sum t_i \pmod{q_0})) \in \mathcal{M}$.
8. Party P_1 sets $[\mathbf{m}]_1 \leftarrow (\mathbf{m} + \mathbf{f}) - [\mathbf{f}]_1$, party $P_i, i \neq 1$ sets $[\mathbf{m}]_i \leftarrow -[\mathbf{f}]_i$.
9. All parties compute, using some default value $\mathbf{0}$ for the randomness,

$$\hat{\mathfrak{ct}}_{\mathbf{m}}^{(1,\mathbb{I})} \leftarrow \mathsf{BGV.Enc}(\Psi_{2^t}^{-1}(\chi_{\mathbb{I}}(\mathbf{m} + \mathbf{f})), \mathbf{0}, \mathfrak{pk}) \ominus \mathfrak{ct}_{\mathbf{f}}^{(1,\mathbb{I})}.$$

Figure 13. Protocol implementing the command **D2** on $\mathcal{F}_{\mathsf{DistrDec}}$

Simulator $\mathcal{S}_{\mathsf{DistrDec}_2}$

- Emulate $\mathcal{F}_{\mathsf{KeyGen}}$ receiving $\{[\mathfrak{sk}]_i\}_{i \in A}$ from $\mathcal{A}$ and $\mathfrak{pk}$.
  Sample $\{[\mathfrak{sk}]_i\}_{i \notin A}$ at random, but such that $\sum_{i \in [n]} [\mathfrak{sk}]_i = \mathfrak{sk}$, and send $\mathfrak{pk}$ to $\mathcal{A}$.
- Emulate $\mathcal{F}_{\mathsf{GenValidCiph}}^{2,\mathsf{Pack}}$ to get $\mathfrak{ct}_{\mathbf{f}}^{(1,\mathbb{I})}$ and $\mathfrak{ct}_{\mathbf{f}}^{(1,\mathbb{J})}$ with inputs $\{\mathbf{f}_i\}_{i \in A}$ from the adversary. If the output is abort, send abort to the functionality and halt.
- Compute $\mathfrak{ct}_{\mathbf{f}}^{(0,\mathbb{J})} = \mathsf{SwitchMod}(\mathfrak{ct}_{\mathbf{f}}^{(1,\mathbb{J})})$ and $\mathfrak{ct}_{\mathbf{f}+\mathbf{m}}^{(0,\mathbb{J})} = (0, c_0, c_1)$
- Compute all the $v_i's$ honestly and $\mathbf{m} + \mathbf{f}$, where $\mathbf{f} = \sum_i \mathbf{f}_i$
- Compute t_i honestly for each $P_i, i \notin A$, except for honest P_j
- Sample random r_j with infinity norm bounded by B_{DDec} and set

$$\tilde{t_j} = -\sum_{i \in [n]} v_i + 2^t \cdot r_j + (m + f) \pmod{q_0},$$

  where $m + f = \chi_{\mathbb{J}}(\psi_{2^t}^{-1}(\mathbf{m} + \mathbf{f}))$.
- Send $\{t_i\}_{i \notin A}$ to $\mathcal{A}$ and receive $\{t_i^*\}_{i \in A}$ from $\mathcal{A}$
- Let $T = \sum_{i \notin A} t_i + \tilde{t_j} + \sum_{i \in A} t_i^*$ and compute $(\mathbf{m} + \mathbf{f})' = \chi_{\mathbb{J}}^{-1}(\psi_{2^t}(T)) \in \mathcal{M}$.
- Compute $\delta_{\mathbf{m}} \leftarrow (\mathbf{m} + \mathbf{f})' - (\mathbf{m} + \mathbf{f})$
- Compute and store $[\mathbf{m}]_1 \leftarrow (\mathbf{m} + \mathbf{f})' - \mathbf{f}_1$ and $[\mathbf{m}]_i \leftarrow -\mathbf{f}_i, i \neq 1$
- Perform the last step honestly. Send $\{[\mathbf{m}]_i\}_{i \in A}$ and $\mathbf{m}' = \sum_{i \in [n]} [\mathbf{m}]_i$ to the functionality.

Figure 14. Simulator for **D2**

Functionality $\mathcal{F}^{\nu,\mathsf{flag}}_{\mathsf{GenValidCiph}}$

Let A be the set of corrupt parties.
PARAMETERS: an integer ν, a security parameter $\mathsf{ZK\_sec}$, a $\mathsf{flag} \in \{\mathsf{Diag}, \mathsf{Pack}, \perp\}$ such that: If $\mathsf{flag} = \mathsf{Diag}$, then $\mathsf{P} = \mathsf{Diag}$; If $\mathsf{flag} = \mathsf{Pack}$, then $\mathsf{P} = \mathsf{Pack}$ and if $\mathsf{flag} = \perp$, then $\mathsf{P} = \emptyset$.

Initialize: On receiving (Init) from all parties run $(\mathfrak{pk}, \mathfrak{sk}) \leftarrow \mathsf{BGV.KeyGen}(1^\kappa)$, sending the value $\mathfrak{pk}$ to the adversary and all the parties.
Gen-1: On input $(\mathsf{Gen\text{-}1}, \mathsf{flag}, [\mathbf{m}]_i)$ from all parties $P_i, i \in [n]$, do the following:
- If the adversary sends abort, return abort
- Otherwise receive $\mathfrak{ct}^{(1,\mathbb{I})}_{\mathbf{m}}$ and send this value to the parties

Gen-2: On input $(\mathsf{Gen\text{-}2}, \mathsf{flag}, [\mathbf{m}]_i)$ from all parties, proceed as follows:
- If the adversary sends abort, return abort
- Otherwise receive $\mathfrak{ct}^{(1,\mathbb{I})}_{\mathbf{m}}$ and $\mathfrak{ct}^{(1,\mathbb{J})}_{\mathbf{m}'}$ and send these values to all parties

Figure 15. The functionality $\mathcal{F}^{\nu,\mathsf{flag}}_{\mathsf{GenValidCiph}}$ to generate valid ciphertexts

Protocol $\Pi^{\nu,\mathsf{flag}}_{\mathsf{GenValidCiph}}$

PARAMETERS: an integer ν, a security parameter $\mathsf{ZK\_sec}$, a $\mathsf{flag} \in \{\mathsf{Diag}, \mathsf{Pack}, \perp\}$ such that: If $\mathsf{flag} = \mathsf{Diag}$, then $\mathsf{P} = \mathsf{Diag}$; If $\mathsf{flag} = \mathsf{Pack}$, then $\mathsf{P} = \mathsf{Pack}$ and if $\mathsf{flag} = \perp$, then $\mathsf{P} = \emptyset$.

Initialize: Each party P_i calls $\mathcal{F}_{\mathsf{KeyGen}}$ receiving $(\mathfrak{pk}, [\mathfrak{sk}]_i)$.
Gen-1: Each P_i inputs $(\mathsf{Gen\text{-}1}, \mathsf{flag}, [\mathbf{m}]_i)$, where $\mathsf{flag} \in \{\mathsf{Diag}, \perp\}$ and $[\mathbf{m}]_i$ are private inputs and if $\mathsf{flag} = \mathsf{Diag}$ then all slots of $[\mathbf{m}]_i$ are equal.
1. Each P_i sets $[m_\mathbb{I}]_i \leftarrow \chi_\mathbb{I}([\mathbf{m}]_i) \in \overline{\mathcal{P}}$ and computes $\mathfrak{ct}^{\mathbb{I}}_{m_i} \leftarrow \mathsf{BGV.Enc}(\overline{\Gamma}^{-1}_{\mathbb{I}}([m_\mathbb{I}]_i), \mathbf{r}_i; \mathfrak{pk})$.
2. Parties run the protocol $\Pi^{1,\mathsf{flag}}_{\mathsf{gZKPoK}}$ receiving either $\mathfrak{ct}^{\mathbb{I}}_{\mathbf{m}}$ or abort.

Gen-2: Each P_i inputs $(\mathsf{Gen\text{-}2}, \mathsf{flag}, [\mathbf{m}]_i)$, where $\mathsf{flag} = \mathsf{Pack}$ and $[\mathbf{m}]_i$ are private inputs :
1. Each P_i sets $[m_\mathbb{I}]_i \leftarrow \chi_\mathbb{I}([\mathbf{m}]_i) \in \overline{\mathcal{P}}$ and $[m_\mathbb{J}]_i \leftarrow \chi_\mathbb{J}([\mathbf{m}]_i) \in \overline{\mathcal{P}}$, then they compute $\mathfrak{ct}^{\mathbb{I}}_{m_i} \leftarrow \mathsf{Enc.BGV}(\overline{\Gamma}^{-1}_{\mathbb{I}}([m_\mathbb{I}]_i), \mathbf{r}_i; \mathfrak{pk})$ and $\mathfrak{ct}'^{\mathbb{J}}_{m_i} \leftarrow \mathsf{Enc.BGV}(\overline{\Gamma}^{-1}_{\mathbb{J}}([m_\mathbb{J}]_i), \mathbf{r}'_i; \mathfrak{pk})$.
2. Parties run the protocol $\Pi^{1,\mathsf{flag}}_{\mathsf{gZKPoK}}$ receiving either $(\mathfrak{ct}^{\mathbb{I}}_{\mathbf{m}}, \mathfrak{ct}'^{\mathbb{J}}_{\mathbf{m}})$ to all the parties or abort.

Figure 16. Protocol for generating valid encryption on random shared values

Simulator $\mathcal{S}_{\mathsf{GenValidCrt}}$

Initialize: Emulate $\mathcal{F}_{\mathsf{KeyGen}}$ to obtain $(\mathfrak{pk}, \mathfrak{sk})$ and send $\mathfrak{pk}$ to $\mathcal{A}$
Gen-1: Perform the first step according to the protocol, with values $[\mathbf{m}]_{i\in A}$ received from $\mathcal{A}$. Run $\Pi^{1,\mathsf{flag}}_{\mathsf{gZKPoK}}$ honestly receiving $\{\mathfrak{ct}_{\mathbf{m}_i}\}_{i\in A}$ from the adversary. Decrypt every broadcast ciphertext. Send abort or $\mathfrak{ct}^{\mathbb{I}}_{\mathbf{m}}$ accordingly.
Gen-2: Perform the first step according to the protocol, with values $[\mathbf{m}]_{i\in A}$ received from $\mathcal{A}$. Run $\Pi^{2,\mathsf{flag}}_{\mathsf{gZKPoK}}$ honestly with the inputs $\{\mathfrak{ct}_{\mathbf{m}_i}\}_{i\in A}$ and $\{\mathfrak{ct}'_{\mathbf{m}_i}\}_{i\in A}$ provided by $\mathcal{A}$ and decrypting all the broadcast ciphertexts. Send abort or $(\mathfrak{ct}_{\mathbf{m}}, \mathfrak{ct}'_{\mathbf{m}})$ accordingly.

Figure 17. Simulator for $\mathsf{GenValidCrt}$

6 SPD$\mathbb{Z}_{2^k}$ from Somewhat Homomorphic Encryption - Pre-processing Phase

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

Let A be the set of corrupt parties. The functionality generates shares of the global MAC key, it then provides an interface to enable the generation of authenticated (potentially correlated) random shared values.

Initialize: On command (Init), wait for an input from the adversary, if it sends abort, return abort. Otherwise receive $\{[\alpha]_i\}_{i\in A} \in \mathbb{Z}_{2^t}$ and sample $\{[\alpha]_i\}_{i\notin A} \leftarrow \mathbb{Z}_{2^t}$ uniformly at random. Store $\alpha = \sum_{i\in[n]}[\alpha]_i$.

Input: On receiving (Input, P_i) from all parties, this command produces ξ input masks. If the adversary sends abort, return abort. Otherwise

- If $i \in A$: Receive $\mathbf{m} \in (\mathbb{Z}_{2^t})^\xi$ and all the shares $[\mathbf{m}]_i \in (\mathbb{Z}_{2^t})^\xi$, $i \in [n]$, and $\{[\boldsymbol{\gamma}_\mathbf{m}]_i\}_{i\in A}$, from the adversary. Sample honest shares $[\mathbf{m}]_i$ and $[\boldsymbol{\gamma}_\mathbf{m}]_i$ such that $\mathbf{m} = \sum_{i\in[n]}[\mathbf{m}]_i$ and $\boldsymbol{\gamma}_\mathbf{m} = \mathbf{m}\cdot\boldsymbol{\alpha}$
- If $i \notin A$: Receive from the adversary only the values $[\mathbf{m}]_i \in (\mathbb{Z}_{2^t})^\nu$ and $\{[\boldsymbol{\gamma}_\mathbf{m}]_i\}$ for $i \in A$. Sample honest shares at random but such that $\boldsymbol{\gamma}_\mathbf{m} = \mathbf{m}\cdot\boldsymbol{\alpha}$.
- Output $\langle\mathbf{m}\rangle$

Triple: On input (Triple) from all parties, this command produces ξ random authenticated triples.

- Sample honest shares $([\mathbf{a}]_i, [\mathbf{b}]_i) \in (\mathbb{Z}_{2^t})^{2\cdot\xi}, i \notin A$, at random
- Wait for an input from the adversary, if it sends abort return abort.
  Otherwise receive $([\mathbf{a}]_i, [\mathbf{b}]_i, [\mathbf{c}]_i) \in (\mathbb{Z}_{2^t})^{3\cdot\xi}$ and $([\boldsymbol{\gamma}_\mathbf{a}]_i, [\boldsymbol{\gamma}_\mathbf{b}]_i, [\boldsymbol{\gamma}_\mathbf{c}]_i) \in (\mathbb{Z}_{2^t})^{3\cdot\xi}$, $i \in A$
- Sample $\{[\mathbf{c}]_i\}_{i\notin A} \in (\mathbb{Z}_{2^t})^\xi$ at random such that $\mathbf{c} = \mathbf{a}\cdot\mathbf{b}$, and $\{[\boldsymbol{\gamma}_\mathbf{a}]_i, [\boldsymbol{\gamma}_\mathbf{b}]_i, [\boldsymbol{\gamma}_\mathbf{c}]_i\}_{i\notin A}$ such that, for each $\mathbf{x} \in \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$, $\boldsymbol{\gamma}_\mathbf{x} = \mathbf{x}\cdot\boldsymbol{\alpha}$.
- Output $\langle\mathbf{a}\rangle, \langle\mathbf{b}\rangle, \langle\mathbf{c}\rangle$

Square: On input (Square) from all parties, this command produces ξ random authenticated squares Proceed like in the Triple command, but with $\mathbf{a}\cdot\mathbf{a} = \mathbf{b}$. Output $\langle\mathbf{a}\rangle, \langle\mathbf{b}\rangle$

Bit: On input (Bit) from all parties, this command produces ξ random authenticated bits

- If the adversary sends abort, return abort Otherwise receive $[\mathbf{b}]_i \in \{0,1\}^\xi$ and $[\boldsymbol{\gamma}_\mathbf{b}]_i \in (\mathbb{Z}_{2^t})^\xi$, $i \in A$
- Sample $\{[\mathbf{b}]_i\}_{i\notin A}$ at random in $\{0,1\}^\xi$ and $\{[\boldsymbol{\gamma}_\mathbf{b}]_i\}_{i\notin A}$ at random but such that $\boldsymbol{\gamma}_\mathbf{b} = \mathbf{b}\cdot\boldsymbol{\alpha}$
- Output $\langle\mathbf{b}\rangle$

Figure 18. Offline Functionality $\mathcal{F}_{\mathsf{Prep}}$

We now present our offline protocol based on the homomorphic scheme $\mathcal{E}_{\mathsf{mBGV}}$ described in Section 3. Even if the online computation is assumed to be performed over $\mathbb{Z}_{2^k}$, we produce random authenticated data over $\mathbb{Z}_{2^{k+s}}$. We use the same MAC scheme (and MACCheck procedure) used in SPDZ2k, with the difference that in our protocol also the shares $[\alpha]_i, i \in [n]$, of the secret global key α are in $\mathbb{Z}_{2^{k+s}}$. We set $k+s=t$ and $\mathcal{M} = (\mathbb{Z}_{2^t})^\rho$, where ρ is the number of packing slots.

The main task of the pre-processing protocol, which implements the ideal functionality $\mathcal{F}_{\mathsf{Prep}}$, (given in Figure 18) is to produce the following type of random authenticated values:

Input masks: $(\langle r\rangle, P_i)$, with the authenticated shared valued r known by P_i.
Triples: $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$, where $a, b, c \in \mathbb{Z}_{2^t}$ are random shared values and $c = a\cdot b$.
Squares: $(\langle a\rangle, \langle b\rangle)$, where $a \in \mathbb{Z}_{2^t}$ is a random secret shared value and $b = a^2$.
Bits: $\langle b\rangle$, where b is a random secret shared bit.

We first implement a weaker form of pre-processing functionality $\mathcal{F}_{\mathsf{wPrep}}$ (Figure 20 and Figure 21), that might output incorrect values. After that, in protocol Π_{Prep} (Figure 19), we will bootstrap outputs from $\mathcal{F}_{\mathsf{wPrep}}$ to implement the desired functionality preprocessing functionality $\mathcal{F}_{\mathsf{Prep}}$ which returns different types of correct random authenticated values to be used in the online evaluation.

Protocol Π_{Prep}

Initialize: Same as in Π_{wPrep}

Input: Call the command (wInput, P_i) of $\mathcal{F}_{\mathsf{wPrep}}$ to generate ξ input mask $\langle \mathbf{r} \rangle$. Parties check a random linear combination of these input masks as follows:

1. Parties run $\mathcal{F}_{\mathsf{Rand}}$ to obtain random values $\chi_1, \ldots \chi_\xi$ in $\mathbb{Z}_{2^s}$
2. P_i computes $y = \sum_{k \in \xi} \chi_k \cdot r_k$ and broadcasts this value
3. Each party P_j computes a random linear combination of their MAC shares $[\gamma_y]_j = \sum_k \chi \cdot [\gamma_{r_k}]_i$
4. Run $\mathsf{MACCheck}$ on y and γ_y, if the check fails, output abort

Triple:

1. Call the command $(\mathsf{wTriple})$ of $\mathcal{F}_{\mathsf{wPrep}}$ to obtain a (weak) shared triple $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$.
2. Parties call the same command again, but using the same value a as in the previous step, so to get a (weak) shared triple $(\langle a \rangle, \langle \hat{b} \rangle, \langle \hat{c} \rangle)$.
3. Parties call $\mathcal{F}_{\mathsf{Rand}}$ to obtain $r \in \mathbb{Z}_{2^s}$.
4. Parties execute the **Sacrifice** step as follows:
   (a) Parties compute and run **Open** on $\langle \rho \rangle \leftarrow r \cdot \langle b \rangle - \langle \hat{b} \rangle$
   (b) Parties compute and open $\langle \tau \rangle \leftarrow r \cdot \langle c \rangle - \langle \hat{c} \rangle - \rho \cdot \langle a \rangle$.
   (c) If $\tau \neq 0 \pmod{2^t}$ then abort.
5. The parties run $\mathsf{MACCheck}$, if the check fails, then abort, else output $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$.

Square:

1. Call $(\mathsf{wSquare})$ on $\mathcal{F}_{\mathsf{wPrep}}$ 2 times so as to obtain a (weak) pair of shared squares $(\langle a \rangle, \langle b \rangle)$, $(\langle \hat{a} \rangle, \langle \hat{b} \rangle)$.
2. Parties execute the **Sacrifice** step as follows:
   (a) The parties call $\mathcal{F}_{\mathsf{Rand}}$ to obtain a random $r \in \mathbb{Z}_{2^s}$.
   (b) Parties compute $\langle \rho \rangle \leftarrow r \cdot \langle a \rangle - \langle \hat{a} \rangle$ and run **Open** on this value
   (c) Parties compute and open $\langle \tau \rangle \leftarrow r^2 \cdot \langle b \rangle - \langle \hat{b} \rangle - \rho_i \cdot (r \cdot \langle a \rangle + \langle \hat{a} \rangle)$.
   (d) If $\tau \neq 0 \pmod{2^t}$ then abort.
3. The parties call $\mathsf{MACCheck}$ if the check fails, then abort, else output $(\langle a \rangle, \langle b \rangle)$.

Bit:

1. Parties call (wBit) and $(\mathsf{wSquare})$ on $\mathcal{F}_{\mathsf{wPrep}}$ 2 times so to obtain a (weak) shared square $(\langle a \rangle, \langle b \rangle)$ and a (weak) shared bit $\langle c \rangle$.
2. The parties call $\mathcal{F}_{\mathsf{Rand}}$ to obtain a random $r \in \mathbb{Z}_{2^s}$.
3. Parties execute the **Sacrifice** step as follows:
   (a) Parties compute and open $\langle \rho \rangle \leftarrow r \cdot \langle c \rangle - \langle a \rangle$
   (b) Parties compute and open $\langle \tau \rangle \leftarrow r^2 \cdot \langle c \rangle - \langle b \rangle - \rho \cdot (r \cdot \langle c \rangle + \langle a \rangle)$.
   (c) If $\tau \neq 0 \pmod{2^t}$ then abort.
4. Parties call the $\mathsf{MACCheck}$ if the check fails, then abort, else output $\langle a \rangle$.

Figure 19. Offline protocol Π_{Prep}

6.1 Weak Offline Protocol

We only describe our new protocol for producing random authenticated bits, the remaining commands are implemented similarly to the SPDZ2k paper. In all steps we produce $\rho = r \cdot |\mathbb{I}|$ random pre-processed values at a time, since values are produced in the set $\mathcal{M}$. As before, given $\mathbf{m} \in \mathcal{M}$ we write $[\mathbf{m}]_i$ to denote an additive share of $\mathbf{m}$ and $[\alpha \cdot \mathbf{m}]_i$ to denote an additive share of the scalar multiplication of $\mathbf{m}$ by the scalar α, and reserve the notation $\langle x \rangle$ for authenticated sharings of values $x \in \mathbb{Z}_{2^t}$.

On input of a random private value $[\alpha]_i$ from each party P_i, the command **Initialize** outputs a public ciphertext $\mathfrak{ct}_\alpha$, where $\alpha = \sum_{i=n} [\alpha]_i$, that is used to authenticate secret values calling the subprotocol Π_{Auth} (Figure 23). The commands **wInput, wTriple** and **wSquare**, output shares of input masks, triples and squares, respectively. These are very similar to the ones in [DPSZ12] and [DKL+13], and are described in Figure 22 In what follows we are going to describe the command **wBit**, which implements our new technique to produce random authenticated bits.

Functionality $\mathcal{F}_{\mathsf{wPrep}}$- Part 1

Let A be the set of corrupt parties. The functionality generates shares of the global MAC key, it then provides an interface to enable the generation of authenticated (potentially correlated) random shared values. Let $\rho = r \times |\mathbb{I}|$.

Initialize: On command (Init) do as follows.

1. Wait for an input from the adversary, if it sends abort, return abort. Otherwise receive $\{[\alpha]_i\}_{i\in A} \in \mathbb{Z}_{2^t}$ and sample $\{[\alpha]_i\}_{i\notin A} \leftarrow \mathbb{Z}_{2^t}$ uniformly at random. Store $\alpha = \sum_{i\in[n]}[\alpha]_i$.

wInput: On input (wInput, P_i) do as follows.

1. If $i \in A$, receive $\mathbf{m} \in (\mathbb{Z}_{2^t})^\rho$ and all the shares $[\mathbf{m}]_i \in (\mathbb{Z}_{2^t})^\rho, i \in [n]$, from the adversary. Otherwise, receive from the adversary the values $[\mathbf{m}]_i \in (\mathbb{Z}_{2^t})^\rho$, for $i \in A$.
2. Receive an error value $\boldsymbol{\delta}_{\mathbf{m}} \in (\mathbb{Z}_{2^t})^\rho$ and a MAC error value $\boldsymbol{\delta}_{\boldsymbol{\gamma}}$ from the adversary and sample $[\mathbf{m}]_i \in (\mathbb{Z}_{2^t})^\rho$, for $i \notin A$, random but subject to $\mathbf{m} + \boldsymbol{\delta}_{\mathbf{m}} = \sum_{i\in[n]}[\mathbf{m}]_i \pmod{2^t}$.
3. Run the macro $\langle\mathbf{m}\rangle \leftarrow \mathsf{Auth}([\mathbf{m}]_1, \ldots, [\mathbf{m}]_n, \boldsymbol{\delta}_{\boldsymbol{\gamma}})$.
4. Output $\langle\mathbf{m}\rangle_i$ to party P_i

wTriple: On input (wTriple) do as follows.

1. Sample random shares $([a_j]_i, [b_j]_i) \in (\mathbb{Z}_{2^t})^2$ for $i \notin A, j \in [\rho]$.
2. Wait to receive values $([a_j]_i, [b_j]_i, [c_j]_i) \in (\mathbb{Z}_{2^t})^3, j \in [\rho]$ for $i \in A$. Set $\mathbf{a} \leftarrow \sum_{i=1}^n [\mathbf{a}]_i \pmod{2^t}$ and $\mathbf{b} \leftarrow \sum_{i=1}^n [\mathbf{b}]_i \pmod{2^t}$, where $\mathbf{a} = (a_1, \ldots, a_\rho)$ and $\mathbf{b} = (b_1, \ldots, b_\rho)$.
3. Receive a set of MAC offsets $(\boldsymbol{\delta}_{\gamma_a}, \boldsymbol{\delta}_{\gamma_b}, \boldsymbol{\delta}_{\gamma_c})$ and $\boldsymbol{\delta}_c \in (\mathbb{Z}_{2^t})^\rho$ from the adversary.
4. Set $\mathbf{c} \leftarrow \mathbf{a} \cdot \mathbf{b} + \boldsymbol{\delta}_c \pmod{2^t}$, and sample $[\mathbf{c}]_i \in (\mathbb{Z}_{2^t})^\rho$ for $i \notin A$ such that $\sum_{i=1}^n [\mathbf{c}]_i = \mathbf{c} \pmod{2^t}$.
5. Run $\langle\mathbf{a}\rangle \leftarrow \mathsf{Auth}([\mathbf{a}]_1, \ldots, [\mathbf{a}]_n, \delta_{\gamma_\mathbf{a}})$, $\langle\mathbf{b}\rangle \leftarrow \mathsf{Auth}([\mathbf{b}]_1, \ldots, [\mathbf{b}]_n, \delta_{\gamma_\mathbf{b}})$, and $\langle\mathbf{c}\rangle \leftarrow \mathsf{Auth}([\mathbf{c}]_1, \ldots, [\mathbf{c}]_n, \delta_{\gamma_\mathbf{c}})$.
6. Output $(\langle\mathbf{a}\rangle_i, \langle\mathbf{b}\rangle_i, \langle\mathbf{c}\rangle_i)$ to party P_i.

Figure 20. Weak offline functionality $\mathcal{F}_{\mathsf{wPrep}}$ - Part 1

Functionality $\mathcal{F}_{\mathsf{wPrep}}$- Part 2

Let A be the set of corrupt parties. The functionality generates shares of the global MAC key, it then provides an interface to enable the generation of authenticated (potentially correlated) random shared values. Let $\rho = r \times |\mathbb{I}|$.

wSquare: On input (wSquare) do as follows.

1. Randomly sample shares $[a_j]_i \in (\mathbb{Z}_{2^t})^2$ for $i \notin A$, $j \in [\rho]$.
2. Wait to receive values $([a_j]_i, [b_j]_i) \in (\mathbb{Z}_{2^t})^2$ for $i \in A$, $j \in [\rho]$. Set $\mathbf{a} \leftarrow \sum_{i=1}^n [\mathbf{a}]_i \pmod{2^t}$
3. Receive a set of MAC offsets $(\boldsymbol{\delta}_{\gamma_\mathbf{a}}, \boldsymbol{\delta}_{\gamma_\mathbf{b}})$ and an offset value $\boldsymbol{\delta}_\mathbf{b} \in (\mathbb{Z}_{2^t})^\rho$ from the adversary.
4. Set $\mathbf{b} \leftarrow \mathbf{a} \cdot \mathbf{a} + \boldsymbol{\delta}_\mathbf{b} \pmod{2^t}$.
5. Sample $[\mathbf{b}]_i \in (\mathbb{Z}_{2^t})^\rho$ for $i \notin A$ such that $\sum_{i=1}^n [\mathbf{b}]_i = \mathbf{b} \pmod{2^t}$.
6. Run $\langle\mathbf{a}\rangle \leftarrow \mathsf{Auth}([\mathbf{a}]_1, \ldots, [\mathbf{a}]_n, \boldsymbol{\delta}_{\boldsymbol{\gamma}_\mathbf{a}})$ and $[\mathbf{b}] \leftarrow \mathsf{Auth}([\mathbf{b}]_1, \ldots, [\mathbf{b}]_n, \delta_{\boldsymbol{\gamma}_\mathbf{b}})$.
7. Output $(\langle\mathbf{a}\rangle_i, \langle\mathbf{b}\rangle_i)$ to party P_i.

wBit: On input (wBit) do as follows.

1. Sample a random bit $d_j \in \{0, 1\}$, $j \in [\rho]$. Set $\mathbf{d} = (d_1, \ldots, d_\rho)$.
2. Wait to receive values $[d_j]_i \in \mathbb{Z}_{2^t}$ for $i \in A$.
3. Receive a MAC offset $\boldsymbol{\delta}_{\boldsymbol{\gamma}_\mathbf{d}}$ and an offset value $\boldsymbol{\delta}_\mathbf{d} \in (\mathbb{Z}_{2^t})^\rho$ from the adversary.
4. Run $\langle\mathbf{d}\rangle \leftarrow \mathsf{Auth}([\mathbf{d}]_1, \ldots, [\mathbf{d}]_n, \boldsymbol{\delta}_{\boldsymbol{\gamma}_\mathbf{d}})$.
5. Output $\langle\mathbf{d}\rangle_i$ to party P_i.

Macro $\mathsf{Auth}([\mathbf{x}]_1, \ldots, [\mathbf{x}]_n, \boldsymbol{\delta}_{\boldsymbol{\gamma}})$**:** This is an internal subroutine. Let $\mathbf{x} = (x_1, \ldots, x_\nu) \in (\mathbb{Z}_2)^\nu$.

1. Set $\mathbf{x} \leftarrow \sum_{i=1}^n [\mathbf{x}]_i \pmod{2^t}$.
2. Set $\boldsymbol{\gamma}_\mathbf{x} \leftarrow \boldsymbol{\alpha} \cdot \mathbf{x} + \boldsymbol{\delta}_{\boldsymbol{\gamma}} \pmod{2^t}$.
3. Wait to receive $[\boldsymbol{\gamma}_\mathbf{x}]_i \in (\mathbb{Z}_{2^t})^\rho$ for $i \in A$ from the adversary.
4. Select $[\boldsymbol{\gamma}_\mathbf{x}]_i \in (\mathbb{Z}_{2^t})^\rho$ for $i \notin A$ subject to $\sum_{i=1}^n [\boldsymbol{\gamma}_\mathbf{x}]_i = \boldsymbol{\gamma}_\mathbf{x} \pmod{2^t}$.
5. Return $\langle\mathbf{x}\rangle = \big(([\mathbf{x}]_1, [\boldsymbol{\gamma}_\mathbf{x}]_1), \ldots, ([\mathbf{x}]_n, [\boldsymbol{\gamma}_\mathbf{x}]_n)\big)$.

Figure 21. Weak offline functionality $\mathcal{F}_{\mathsf{wPrep}}$ - Part 2

Protocol Π_{wPrep} - Part 1

PARAMETERS: Let $\rho = r \times |\mathbb{I}|$ be the number of random authenticated data we produce for each call of the following commands.

Initialize: On command (Init) the parties do as follows.

1. Call $\mathcal{F}_{\mathsf{DistrDec}}.\mathsf{Init}$ to obtain $\mathfrak{pk}$
2. Parties sample random $[\alpha]_i \leftarrow \mathbb{Z}_{2^t}, i \in [n]$. Let $[\boldsymbol{\alpha}]_i \leftarrow \mathcal{M}$ denote a plaintext with all the slots set to $[\alpha]_i$. Set $\alpha = \sum_{i\in[n]}[\alpha]_i$.
3. The parties call the functionality $\mathcal{F}^{1,\mathsf{Diag}}_{\mathsf{GenValidCiph}}$ on private inputs $[\alpha]_i$ so that each party P_i receives $\mathfrak{ct}_{\boldsymbol{\alpha}}$.

Input: On input (Input, P_i) from all other parties, this commands produces ρ random masks for P_i.

1. P_i samples a random $\mathbf{r} \in \mathcal{M}$, creates random additive shares $[\mathbf{r}]_j$ of $\mathbf{r}$ and sends them to the designated party P_j
2. Parties call the functionality $\mathcal{F}^{1,\perp}_{\mathsf{GenValidCiph}}$ on input $(\mathsf{Gen}\text{-}1, \perp, [\mathbf{r}]_i)$, $\forall i \in [n]$, receiving $\mathfrak{ct}^{(1,\mathbb{I})}_{\mathbf{r}}$
3. Parties call the subprotocol Π_{Auth} on input $\mathfrak{ct}^{(1,\mathbb{I})}_{\mathbf{r}}$, so to obtain $\langle\gamma_{\mathbf{r}}\rangle$.

wTriple: On input (wTriple), this command produces ρ triples in one execution.

1. The parties call $\mathcal{F}^{1,\perp}_{\mathsf{GenValidCiph}}$ on random inputs $[\mathbf{a}]_i, [\mathbf{b}]_i$, so that each party receives $\mathfrak{ct}_{\mathbf{a}}$ and $\mathfrak{ct}_{\mathbf{b}}$.
2. Parties locally compute $\mathfrak{ct}_{\mathbf{c}} \leftarrow \mathfrak{ct}_{\mathbf{a}} \odot \mathfrak{ct}_{\mathbf{b}}$
3. The parties call $\mathcal{F}_{\mathsf{DistrDec}}.\mathsf{D2}$ on input $\mathfrak{ct}_{\mathbf{c}}$, so that each P_i receives $[\mathbf{c}]_i$ and a fresh ciphertext $\mathfrak{ct}'_{\mathbf{c}}$
4. Parties run Π_{Auth} on inputs $\mathfrak{ct}_{\mathbf{a}}, \mathfrak{ct}_{\mathbf{b}}, \mathfrak{ct}'_{\mathbf{c}}$ to obtain $\langle\gamma_{\mathbf{a}}\rangle$, $\langle\gamma_{\mathbf{b}}\rangle$, $\langle\gamma_{\mathbf{c}}\rangle$.

wSquare: On input (wSquare), this command produces ρ random authenticated squares.

1. This is exactly the same as wTriple above, except that we only sample the messages/ciphertexts for $\mathbf{a}$ and then set $\mathbf{b} = \mathbf{a}^2$.

Figure 22. Weak offline protocol Π_{wPrep} - Part 1

Subprotocol Π_{Auth}

On input $(\mathsf{Auth}, \mathfrak{ct}_{\mathbf{x}})$, where $\mathfrak{ct}_{\mathbf{x}}$ is a public valid ciphertext

1. Parties locally compute $\mathfrak{ct}_{\boldsymbol{\alpha}\cdot\mathbf{x}} = \mathfrak{ct}_{\mathbf{x}} \odot \mathfrak{ct}_{\boldsymbol{\alpha}}$
2. Parties call $\mathcal{F}_{\mathsf{DistrDec}}.\mathsf{D1}$ on input $\mathfrak{ct}_{\boldsymbol{\alpha}\cdot\mathbf{x}}$, so that each P_i receives $[\boldsymbol{\gamma}_x]_i = [\boldsymbol{\alpha} \cdot \mathbf{x}]_i$.

Figure 23. Subprotocol Π_{Auth}

Note that, as said before, the outputs of Π_{wPrep} might be incorrect. This is because the distributed decryption, needed to produce and authenticate pre-processing data, allows the adversary to introduce errors in both the shares and MACs. In Π_{Prep} we will check the correctness of these values using the standard techniques of sacrificing ([DPSZ12] and [CDE+18] in the $\mathbb{Z}_{2^k}$ setting).

Authenticated Bits. The standard trick in the modulo p setting, see [DKL+13], is to use the 2-to-1 map induced by squaring modulo p, inverting it, and taking an element in the kernel by dividing the initial value by the obtained square root, i.e. $x/\sqrt{x^2} \in \{-1, 1\}$. When working modulo 2^t this is no longer possible, as the squaring map is 4-to-1. However, by temporarily working modulo 2^{t+1} and then reducing the roots modulo 2^t we can again obtain a 2-to-1 map. Furthermore, since we need to be able to divide by the $\sqrt{x^2}$, we will limit ourselves to invertible x's, i.e. such that $x = 1 \pmod 2$. The protocol to generate a random element in $\{-1, 1\}$ is therefore as follows:

1. Given $a \leftarrow \mathbb{Z}_{2^t}$, compute $b \leftarrow 1 + 2a \pmod{2^{t+1}}$ (b is determined mod 2^{t+1})
2. $v \leftarrow b^2 \pmod{2^{t+2}}$ (note that v is determined modulo 2^{t+2} since $b + 2^{t+1}$ has the same square as b).
3. $\hat{b} \leftarrow \sqrt{v} \pmod{2^{t+1}}$ (A fixed square root is taken. Notice since v is a square, square roots exist, and there are four such square roots modulo 2^{t+2}, namely: b, $-b$, $b + 2^{t+1}$ and $-b + 2^{t+1}$. However, when reduced modulo 2^{t+1} there are only two possibilities, namely b and $-b$.

4. $d \leftarrow b/\hat{b} \pmod{2^{t+1}} \in \{-1, 1\}$.

Since we are interested in sharing bits in $\{0, 1\}$, not in $\{-1, 1\}$, we have to convert d. To perform the conversion in the large prime case of "standard" SPDZ, one can simply add one and then divide by two, but in our case division by two is impossible. However, we have a well defined division-by-2 map from $\mathbb{Z}_{2^{t+1}}$ to $\mathbb{Z}_{2^t}$ that maps $x \in \mathbb{Z}_{2^{t+1}}$ with $x = 0 \pmod 2$ to $x/2 \in \mathbb{Z}_{2^t}$, losing one bit of precision in the process. As such we can replace step 5 by:

5. $d \leftarrow (b/\hat{b} + 1)/2 \pmod{2^t} = (a/\hat{b} + (1 + \hat{b})/2\hat{b}) \pmod{2^t} \in \{0, 1\}$.

Note that since $\hat{b}$ is odd, the expression $(1 + \hat{b})/2$ is well defined modulo 2^t. We are now ready to give the wBit procedure of Π_{wPrep}, where we map these operations to the ciphertext space and the shares of a so as to produce shared bits in $\{0, 1\}$. In particular, given a sharing $[a]_i$ of a, it is easy to compute a sharing of d by defining $[d]_1 = [a]_1/\hat{b} + (1 + \hat{b})/(2\hat{b}) \pmod{2^t}$ and $[d]_i = [a]_i/\hat{b} \pmod{2^t}$ for $i > 1$.

Protocol Π_{wPrep} - Part 2

PARAMETERS: Let $\rho = r \times |\mathbb{I}|$ be the number of random authenticated data we produce for each call of the following commands.

wBit: This command produces ρ random authenticated bits in one execution.

1. Parties call $\mathcal{F}^{1,\perp}_{\mathsf{GenValidCiph}}$ on command (Gen-1, $\perp$) with random inputs $[\mathbf{a}]_i \in \mathcal{M}, i \in [n]$, so that each P_i receives $\mathfrak{ct}_{\mathbf{a}}$. Parties locally compute $\mathfrak{ct}_{\mathbf{b}} = \mathfrak{ct}_{\mathbf{a}} \boxplus \mathfrak{ct}_{\mathbf{a}} \boxplus \mathfrak{ct}_{\mathbf{1}}$, where $\mathfrak{ct}_{\mathbf{1}}$ a trivial encryption of the all one vector.
2. Parties set $\mathfrak{ct}_{\mathbf{v}} \leftarrow \mathfrak{ct}_{\mathbf{b}} \odot \mathfrak{ct}_{\mathbf{b}}$.
3. The parties call $\mathcal{F}_{\mathsf{DistrDec}}$.D1 on input $\mathfrak{ct}_{\mathbf{v}}$ and so each party P_i obtains $[\mathbf{v}]_i \in \mathcal{M}'$. Note $\mathcal{M}'$ is mod 2^{t+2}.
4. The parties broadcast $[\mathbf{v}]_i$ and set $\mathbf{v} \leftarrow [\mathbf{v}]_1 + \ldots + [\mathbf{v}]_n \pmod{2^{t+2}}$.
5. Parties set $\hat{\mathbf{b}} \leftarrow \sqrt{\mathbf{v}} \pmod{2^{t+1}}$, where a fixed square root value is taken in each slot position modulo 2^{t+1}. If a square root does not exists, abort.
6. Parties locally set

$$\mathfrak{ct}_{\mathbf{d}} \leftarrow \mathfrak{ct}_{\mathbf{a}}/\hat{\mathbf{b}} \boxplus \mathfrak{ct}_{(\hat{\mathbf{b}}+\mathbf{1})/2\hat{\mathbf{b}}},$$
$$[\mathbf{d}]_1 \leftarrow [\mathbf{a}]_1/\hat{\mathbf{b}} + (\hat{\mathbf{b}} + 1)/2\hat{\mathbf{b}} \pmod{2^t},$$
$$[\mathbf{d}]_i \leftarrow [\mathbf{a}]_i/\hat{\mathbf{b}} \pmod{2^t}, \text{ for } i > 1,$$

   where $\mathfrak{ct}_{(\hat{\mathbf{b}}+\mathbf{1})/2\hat{\mathbf{b}}}$ is a deterministic encryption of the public value $(\hat{\mathbf{b}} + \mathbf{1})/2\hat{\mathbf{b}}$.
7. Parties run Π_{Auth} on input $\mathfrak{ct}^{(1,\mathbb{I})}_{\mathbf{d}}$, so to obtain $[\boldsymbol{\gamma}_{\mathbf{d}}]_i, \forall i \in [n]$, i.e. each party P_i obtains $[\boldsymbol{\alpha} \cdot \mathbf{d}]_i$.
8. For each slot in the plaintext space $\mathcal{M}$ each party P_i can obtain a value of $\langle d_j \rangle_i$, $j \in [\rho]$, (a sharing modulo 2^t) from the plaintexts $([\mathbf{d}]_i, [\boldsymbol{\alpha} \cdot \mathbf{d}]_i)$.
9. Each party P_i's output is $\langle d_j \rangle_i, j \in [\rho]$.

Figure 24. Weak offline protocol Π_{wPrep} - wBit

Note that since we do not expose a direct distributed decryption operation on the $\mathcal{F}_{\mathsf{DistrDec}}$ functionality we need to obtain the clear value of $\mathbf{v}$ via sharing and opening, unlike in [DKL+13]. Also note again unlike [DKL+13], we produce exactly the given number of slots in each call to Bit, as we do not need to cope with the case of square roots of zero in this method.

Theorem 6.1. *The Protocol Π_{wPrep} (Figure 22 and Figure 24) securely realises the ideal functionality $\mathcal{F}_{\mathsf{wPrep}}$ in the ($\mathcal{F}_{\mathsf{GenValidCiph}}$,$\mathcal{F}_{\mathsf{DistrDec}}$)-hybrid model.*

Proof. We construct a simulator $\mathcal{S}_{\mathsf{wPrep}}$ (Figure 25 and Figure 26), such that no environment $\mathcal{Z}$ can distinguish between a real execution with the adversary $\mathcal{A}$ and Π_{wPrep} and an ideal execution with the simulator $\mathcal{S}_{\mathsf{wPrep}}$ and $\mathcal{F}_{\mathsf{wPrep}}$.

Simulator $\mathcal{S}_{\mathsf{wPrep}}$ - Part I

Let A be the set of corrupt parties.

Initialize: 1. Emulate $\mathcal{F}_{\mathsf{DistrDec}}.\mathsf{Init}$ and send $\mathfrak{pk}$ to the adversary
2. Emulate $\mathcal{F}^{1,\mathsf{Diag}}_{\mathsf{GenValidCiph}}$ on inputs $\{[\boldsymbol{\alpha}]_i\}_{i\in A}$ specified by $\mathcal{A}$ and random $\{[\boldsymbol{\alpha}]_i\}_{i\notin A}$ for honest parties. Send the output of $\mathcal{F}_{\mathsf{GenValidCiph}}$ to $\mathcal{A}$. If $\mathcal{F}_{\mathsf{GenValidCiph}}$ returns abort, send abort to the functionality.

Input: 1. If P_i *honest*: Sample a random $\mathbf{r} \in \mathcal{M}$ and shares $\{[\mathbf{r}]_i\}_{i\in[n]}$. Send $\{[\mathbf{r}]_i\}_{i\in A}$ to $\mathcal{A}$. If P_i *corrupt*: Receive $\mathbf{r} \in \mathcal{M}$ and shares $\{[\mathbf{r}]_i\}_{i\in[n]}$ from the adversary.
2. Emulate the functionality $\mathcal{F}^{1,\perp}_{\mathsf{GenValidCiph}}$ with inputs $[\mathbf{r}^*]_i, i \in A$, specified by the adversary. If the output is abort, send abort to the functionality. Otherwise, compute $\boldsymbol{\delta}_{\mathbf{r}} = \sum_{i\in A}([\mathbf{r}]_i - [\mathbf{r}^*]_i)$ and send $\mathfrak{ct}_{\mathbf{r}^*}$ to $\mathcal{A}$, where $\mathbf{r}^* = \mathbf{r} + \boldsymbol{\delta}_{\mathbf{r}}$.
3. Compute $\mathfrak{ct}_{\boldsymbol{\alpha}\cdot\mathbf{r}^*}$ and emulate $\mathcal{F}_{\mathsf{DistrDec}}.\mathsf{D1}$.
4. Send $\{[\mathbf{r}^*]_i, \boldsymbol{\gamma}_i = \boldsymbol{\gamma}_{[\mathbf{r}^*]_i}\}_{i\in A}, \boldsymbol{\delta}_{\mathbf{r}}, \boldsymbol{\delta}_{\boldsymbol{\gamma}}$ to the functionality. If P_i is corrupt send also $\mathbf{r}$ and $\{[\mathbf{r}]_i\}_{i\notin A}$.

wSquare: Simulate this command similarly to the previous one

Figure 25. Simulator for the weak-preprocessing protocol - Part I

Simulator $\mathcal{S}_{\mathsf{wPrep}}$ - Part II

Let A be the set of corrupt parties.

wTriple: 1. Emulate $\mathcal{F}^{1,\perp}_{\mathsf{GenValidCiph}}$ on input $\{[\mathbf{a}]_i, [\mathbf{b}]_i\}_{i\in A}$ specified by $\mathcal{A}$. If abort, return abort to the functionality, otherwise send $\mathfrak{ct}_{\mathbf{a}}, \mathfrak{ct}_{\mathbf{b}}$ to the adversary.
2. Compute $\mathfrak{ct}_{\mathbf{c}}$ and emulate $\mathcal{F}_{\mathsf{DistrDec}}.\mathsf{D2}$. Send $\mathfrak{ct}_{\mathbf{c}^*}$ to the advesary, where $\mathbf{c}^* = \mathbf{c} + \boldsymbol{\delta}_{\mathbf{c}}$.
3. Compute $\mathfrak{ct}_{\boldsymbol{\alpha}\cdot\mathbf{c}^*}$, $\mathfrak{ct}_{\boldsymbol{\alpha}\cdot\mathbf{a}}$, $\mathfrak{ct}_{\boldsymbol{\alpha}\cdot\mathbf{b}}$, and emulate $\mathcal{F}_{\mathsf{DistrDec}}.\mathsf{D1}$ on these inputs.
4. Send $\{[\mathbf{a}]_i$, $[\mathbf{b}]_i$, $[\mathbf{c}^*]_i\}_{i\in A}$, $\{[\boldsymbol{\gamma}_{\mathbf{a}}]_i$, $[\boldsymbol{\gamma}_{\mathbf{b}}]_i$, $[\boldsymbol{\gamma}_{\mathbf{c}^*}]_i\}_{i\in A}$, $\boldsymbol{\delta}_{\mathbf{c}}, \boldsymbol{\delta}_{\boldsymbol{\gamma}_{\mathbf{a}}}, \boldsymbol{\delta}_{\boldsymbol{\gamma}_{\mathbf{b}}}, \boldsymbol{\delta}_{\boldsymbol{\gamma}_{\mathbf{c}^*}}$ to the functionality.

wSquare: Simulate this command similarly to the previous one. Send $\boldsymbol{\delta}_{\mathbf{a}}, \boldsymbol{\delta}_{\mathbf{b}}, \boldsymbol{\delta}_{\boldsymbol{\gamma}_{\mathbf{a}}}, \boldsymbol{\delta}_{\boldsymbol{\gamma}_{\mathbf{b}}}$ to the functionality.

wBit: 1. Run the protocol honestly with corrupt shares specified by $\mathcal{A}$ when emulating $\mathcal{F}^{1,\perp}_{\mathsf{GenValidCiph}}$ and $\mathcal{F}_{\mathsf{DistrDec}}.\mathsf{D1}$.
2. Receive $\{[\mathbf{v}^*]_i\}_{i\in A}$ from $\mathcal{A}$ and send honest shares to $\mathcal{A}$. Compute $\boldsymbol{\delta}_{\mathbf{v}} = \sum_{i\in A}([\mathbf{v}]_i - [\mathbf{v}^*]_i)$. Compute $\mathbf{v}^* = \sum_{i\in A}[\mathbf{v}^*]_i + \sum_{i\notin A}[\mathbf{v}]_i$. If $\hat{\mathbf{b}}$ does not exist, send abort to the functionality. Otherwise go to the next step.
3. Simulate the rest of the protocol honestly and send $\boldsymbol{\delta}_d$, $\boldsymbol{\delta}_{\gamma_d}$ to the functionality

Figure 26. Simulator for the weak-preprocessing protocol - Part II

First, we note that the simulator generates $(\mathfrak{pk}, \mathfrak{sk})$ emulating $\mathcal{F}_{\mathsf{DistrDec}}.\mathsf{Init}$, and that indistinguishability of ciphertexts, for example in the **Initialize** step, follows from the CPA-security of the underlying homomorphic scheme. In the **wInput** step, if P_i is honest, the values $\{[\mathbf{r}]_i\}_{i\in A}$ are uniformly random in both executions; if P_i is corrupt, the shares are entirely specified by $\mathcal{A}$. Then the simulator honestly emulates the functionality $\mathcal{F}^{1,\perp}_{\mathsf{GenValidCiph}}$ with corrupt inputs specified by the adversary and honest random inputs, so the output of this command has exactly the same distribution in both worlds, and if abort occurs in the real execution, so it does in the ideal one except with negligible probability. The same arguments hold for the commands **wTriple** and **wSquare**. In the **wBit** command the $\{[\mathbf{v}]_i\}_{i\in A}$ values sent to the adversary are obtained by emulating $\mathcal{F}_{\mathsf{DistrDec}}.\mathsf{D1}$, so have the same distribution of real ones. And, if the sum of these shares with those provided

by the adversary results in a value for which the square root does not exist, then both executions abort. Note that this happens independently of the honest shares. The rest of the protocol does not require further communication, other than in the emulation of $\mathcal{F}_{\mathsf{DistrDec}}$.D1. □

6.2 From $\mathcal{F}_{\mathsf{wPrep}}$ to $\mathcal{F}_{\mathsf{Prep}}$ - Sacrificing

We can now show how to turn the Π_{wPrep} protocol into a protocol which realises the $\mathcal{F}_{\mathsf{Prep}}$ functionality. As said before, the authenticated shared data generated by $\mathcal{F}_{\mathsf{wPrep}}$ are incorrect if corrupt parties cheated in the distributed decryption, i.e. the output of $\mathcal{F}_{\mathsf{wPrep}}$ is a set of sharings $\{\langle a\rangle, \langle b\rangle, \langle c\rangle\}$ (resp. $\{\langle a\rangle, \langle b\rangle\}$ or $\{\langle a\rangle\}$) where we have $c = a \cdot b + \delta_c$ (resp. $b = a^2 + \delta_b$ or $a \in \{a, a + \delta_a\}$) for some adversarially chosen error value $\delta \in \mathbb{Z}_{2^t}$ and shared values $a, b, c \in \mathbb{Z}_{2^t}$. In a nutshell, the protocol of Π_{Prep} takes the output of Π_{wPrep} and ensures that the adversarially chosen values δ's are all equal to zero using the standard technique of sacrificing.

However, also the MACs on these values might be incorrect, i.e. we might have $\gamma_a = \sum_i [\alpha \cdot a]_i + \delta_{\gamma_a}$ for each authenticated value a. We can check the MAC on all the opened values at the end of the offline phase, and also check that the input masks are correctly MAC'd, by performing a MACCheck on a random linear combination of them. We add these checks in our preprocessing protocol, but in practice we do no worry about the errors δ_γ's on the MAC equations, since they can be dealt with later during the online phase, when all the values opened during the circuit evaluation are checked.

Theorem 6.2. *The protocol Π_{Prep} securely implements the ideal functionality $\mathcal{F}_{\mathsf{Prep}}$ against any static, active adversary corrupting up to $n-1$ parties in the $(\mathcal{F}_{\mathsf{wPrep}}, \mathcal{F}_{\mathsf{Rand}})$-hybrid model.*

Proof. The proof is essentially the same as in [DPSZ12] and [DKL+13]. The high level idea is that the simulator emulates the weak pre-processing functionality $\mathcal{F}_{\mathsf{wPrep}}$ with corrupt shares provided by the adversary. Note that when the protocol calls the command **Open**, it refers to the command described in the MACCheck procedure given in Figure 27.

Therefore, to simulate the **Input** command, first the simulator emulates the $\mathcal{F}_{\mathsf{wPrep}}$ with corrupt inputs provided by $\mathcal{A}$ and then the $\mathcal{F}_{\mathsf{Rand}}$ functionality, giving the random output values to $\mathcal{A}$. If P_i is corrupt, $\mathcal{S}$ receives the value y from the adversary and, if this value is inconsistent with previously computed values, it sends abort to the functionality. If P_i is honest, $\mathcal{S}$ sends y to $\mathcal{A}$ and waits for a reply. If $\mathcal{A}$ sends abort, $\mathcal{S}$ forwards abort to the functionality.

The simulation for the commands **Triple, Square** and **Bit** is similar: $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{wPrep}}$ and $\mathcal{F}_{\mathsf{Rand}}$, then during the sacrifice step it opens a random value in $\mathbb{Z}_{2^t}$ and, if the output of $\mathcal{F}_{\mathsf{wPrep}}$ was incorrect, it sends abort to the functionality. Otherwise it checks the consistency of opened values, sending abort to the functionality in the case the check fails.

To argue indistinguishability between the real and ideal executions, we recall that in the real-world execution the probability of passing the sacrificing step with incorrect values is negligible, in particular this happens with probability 2^{-s} [CDE+18][Claim 4]. While the probability of passing the MAC Check on inconsistent values is bounded by $2^{-2+\log(s+1)}$ [CDE+18][Theorem 1], which is again negligible for large enough s. □

7 Communication Efficiency Analysis

Here we analyse the communication efficiency of our preprocessing protocol, when compared to the method of [CDE+18]. To simplify matters we focus just on the cost of our triple generation

Protocol	N	$\log_2 q$	k	s	sec	Triple Cost
This paper	14449	270	32	32	26	**72.8**
SPDZ2k	-	-	32	32	26	79.87
This paper	32377	520	64	64	57	**153.3**
SPDZ2k	-	-	64	64	57	319.488
This paper	32377	720	128	64	57	**212.2**
SPDZ2k	-	-	128	64	57	557.06

Table 2. Amortized communication cost (in kbit) of producing triples of our protocol and SPDZ2k

procedure as it is the most expensive step of the preprocessing phase. The entire protocol we want to maintain the same level of statistical security, which is equal to $\mathsf{sec} = s - \log_2 s$.

The most expensive step in our protocol is the zero-knowledge proof that proves that ZK_sec ciphertexts are valid with ZK_sec bits of statistical security. Once this parameter is fixed, to sec, the protocol $\Pi^{1,\mathsf{flag}}_{\mathsf{gZKPoK}}$ requires $2 \times \mathsf{ZK\_sec}$ broadcasts of ciphertexts in R^2 and the broadcast of $\mathbf{z}^i$ and T_i, which gives a total cost of $4 \cdot \mathsf{ZK\_sec} \cdot N \cdot \log(q) + 8 \cdot \mathsf{ZK\_sec} \cdot N - 4 \cdot N \cdot \mathsf{ZK\_sec}$ bits.

To generate $\mathsf{ZK\_sec} \cdot \rho$ triples $\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle, \langle \mathbf{c} \rangle$ we need two calls to $\mathcal{F}^{1,\perp}_{\mathsf{GenValidCiph}}$, to create ZK_sec ciphertexts $\mathfrak{ct}_{\mathbf{a}}, \mathfrak{ct}_{\mathbf{b}}$, after that one call to $\mathcal{F}_{\mathsf{DistrDec}}.\mathsf{D2}$, to produce shares of $\mathfrak{ct}_{\mathbf{c}}$ and ZK_sec fresh ciphertexts $\mathfrak{ct}'_{\mathbf{c}}$. These are used later to produce the MAC shares $[\gamma_{\mathbf{a}}]_i, [\gamma_{\mathbf{b}}]_i, [\gamma_{\mathbf{c}}]_i$, obtained by running $3 \times \mathsf{ZK\_sec}$ times the subprotocol Π_{Auth}. Notice that, as done in [CDE+18], we are ignoring here the cost of the MACCheck, as it can be done in the online phase and, in any case, it is independent of the number of generated triples, and the cost of $\mathcal{F}_{\mathsf{Rand}}$ and sacrificing, as it is negligible compared to the cost of the rest of the protocol. This gives a total cost (amortized) of roughly $4 \cdot (12 \cdot \log(q) \cdot N/\rho + N/\rho \cdot q)$ bits per triple,where ρ is the amount of packing in a single ciphertext.

We then estimate for various values of (k, s) the values of N and q which satisfy the bounds in Appendix A, and which give the best values for packing from Table 1. We select parameters which give us roughly 128 bits of computational security according to the tool obtained from https://bitbucket.org/malb/lwe-estimator. This allows us to give an estimation of the communication complexity of our protocol and SPDZ2k in the case of two parties creating one triple, see Table 2. In the important cases of statistical security of 64 bits in SPDZ2k over 64 and 128-bit data types we have a reduction in communication of over a half. In addition our protocol will get progressively more efficient than the OT-based pre-processing of SPDZ2k as the number of parties increases.

Acknowledgments

We thank Cyprien Delpech de Saint Guilhem for many helpful discussions. This work has been supported in part by ERC Advanced Grant ERC-2015-AdG-IMPaCT, by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under contract No. N66001-15-C-4070, and by the FWO under an Odysseus project GOH9718N.

References

ACK+19. Abdelrahaman Aly, Daniele Cozzo, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Nigel P. Smart, and Tim Wood. SCALE-MAMBA v1.6: Documentation, 2019.

ADPS16. Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A New Hope. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 327–343. USENIX Association, 2016.

AFL+16. Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 805–817. ACM Press, October 2016.

BCS19. Carsten Baum, Daniele Cozzo, and Nigel P. Smart. Using TopGear in Overdrive: A more efficient ZKPoK for SPDZ. Cryptology ePrint Archive, Report 2019/035, 2019. http://eprint.iacr.org/2019/035.

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

Bea92. Donald Beaver. Foundations of secure interactive computing. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 377–391. Springer, Heidelberg, August 1992.

BGV12. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012*, pages 309–325. ACM, January 2012.

BLW08. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS 2008*, volume 5283 of *LNCS*, pages 192–206. Springer, Heidelberg, October 2008.

BV11. Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In Rafail Ostrovsky, editor, *52nd FOCS*, pages 97–106. IEEE Computer Society Press, October 2011.

Can01. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

Cas86. John W.S. Cassels. *Local Fields*. Cambridge University Press, 1986.

CD09. Ronald Cramer and Ivan Damgård. On the amortized complexity of zero-knowledge protocols. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 177–191. Springer, Heidelberg, August 2009.

CDE+18. Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 769–798. Springer, Heidelberg, August 2018.

CFIK03. Ronald Cramer, Serge Fehr, Yuval Ishai, and Eyal Kushilevitz. Efficient multi-party computation over rings. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 596–613. Springer, Heidelberg, May 2003.

CRFG19. Dario Catalano, Mario Di Raimondo, Dario Fiore, and Irene Giacomelli. MonZa: Fast maliciously secure two party computation on $\mathbb{Z}_{2^k}$. *IACR Cryptology ePrint Archive*, 2019:211, 2019.

DEF+19. Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *2019 IEEE Symposium on Security and Privacy*, pages 1102–1120. IEEE Computer Society Press, May 2019.

DKL+13. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, Heidelberg, September 2013.

DOS18. Ivan Damgård, Claudio Orlandi, and Mark Simkin. Yet another compiler for active security or: Efficient MPC over arbitrary rings. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 799–829. Springer, Heidelberg, August 2018.

DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

GHS12a. Craig Gentry, Shai Halevi, and Nigel P. Smart. Better bootstrapping in fully homomorphic encryption. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, *PKC 2012*, volume 7293 of *LNCS*, pages 1–16. Springer, Heidelberg, May 2012.

GHS12b. Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 465–482. Springer, Heidelberg, April 2012.

GHS12c. Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 850–867. Springer, Heidelberg, August 2012.

HS14. Shai Halevi and Victor Shoup. Algorithms in HElib. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 554–571. Springer, Heidelberg, August 2014.

KOS16. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 830–842. ACM Press, October 2016.

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

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

RSS19. Deevashwer Rathee, Thomas Schneider, and K. K. Shukla. Improved multiplication triple generation over rings via RLWE-based AHE. In *18. International Conference on Cryptology And Network Security (CANS'19)*, volume 11829 of *LNCS*. Springer, 2019.

SV14. Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. *Des. Codes Cryptography*, 71(1):57–81, 2014.

A BGV Noise Analysis

Here we present the noise analysis for the scheme. There is nothing here which has not appeared in other works, in fact we simply replace the bound p on the message coefficients for the BGV scheme with 2^t. However, we present the noise bounds here for completeness, and because they are important when analysing the zero-knowledge proofs in the paper. Our analysis follows closely that described in [GHS12c], and [ACK+19]: the reader is referred to those papers for a complete explanation of the notation. The analysis here is to convince the reader that no additional surprises occur due to moving from working with a plaintext mod p to one mod 2^t, and in our modified protocol for shared bit creation.

We first define a number of parameters:

- $\mathsf{ZK\_sec}$: This defines the soundness error of the zero-knowledge proofs.
- $\mathsf{DD\_sec}$: This defines the closeness of the distribution produced in the distributed decryption protocol below to the uniform distribution.
- ϵ: This defines the noise bounds for the FHE scheme below in the following sense. A FHE ciphertext in the protocol is guaranteed to decrypt correctly, even under adversarial inputs assuming the ZKPoKs are applied correctly, with probability $1 - 2^{-\epsilon}$. In fact this is an underestimate of the probability. From ϵ we define e_i such that $\mathsf{erfc}(e_i)^i \approx 2^{-\epsilon}$ and then we set $c_i = e_i^i$. In [GHS12c] this parameter is implicitly set to be 55.
- σ: The standard deviation for our approximate discrete Gaussians, we implicitly assume $\sigma = 3.17 = \sqrt{10}$ below (as proposed by [GHS12c]), and we utilize the NewHope [ADPS16] method of approximating the discrete Gaussian (and hence all samples are bounded in size by 20 in absolute value).
- h: This defines the number of non-zero coefficients in the FHE secret key as used in [GHS12c].

A.1 Noise for Fresh Ciphertexts

We calculate a bound (with high probability) on the output noise of an honestly generated ciphertext to be

$$\|c_0 - \mathfrak{sk} \cdot c_1\|_\infty = \|((a \cdot \mathfrak{sk} + 2^t \cdot \epsilon) \cdot v + 2^t \cdot e_0 + m - (a \cdot v + 2^t \cdot e_1) \cdot \mathfrak{sk}\|_\infty^{\mathsf{can}}$$

$$
\begin{aligned}
&= \|m + 2^t \cdot (\epsilon \cdot v + e_0 - e_1 \cdot \mathfrak{s}\mathfrak{k})\|_\infty^{\mathsf{can}} \\
&\leq \|m\|_\infty^{\mathsf{can}} + 2^t \cdot (\|\epsilon \cdot v\|_\infty^{\mathsf{can}} + \|e_0\|_\infty^{\mathsf{can}} + \|e_1 \cdot \mathfrak{s}\mathfrak{k}\|_\infty^{\mathsf{can}}) \\
&\leq \phi(p) \cdot 2^{t-1} \\
&\qquad + 2^t \cdot \sigma \cdot \left(c_2 \cdot \phi(p)/\sqrt{2} + c_1 \cdot \sqrt{\phi(p)} + c_2 \cdot \sqrt{h \cdot \phi(p)}\right) \\
&= B_{\mathsf{clean}}.
\end{aligned}
$$

Note this is a probabilistic bound and not an absolute bound.

However, below we will only be able to guarantee m, v, e_0 and e_1 are selected subject to

$$
\begin{aligned}
\|v\|_\infty &\leq 2^{3 \cdot \mathsf{ZK\_sec}/2+1} \cdot n \\
\|e_0\|_\infty, \|e_1\|_\infty &\leq 20 \cdot 2^{3 \cdot \mathsf{ZK\_sec}/2+1} \cdot n \\
\|m\|_\infty &\leq 2^{3 \cdot \mathsf{ZK\_sec}/2+1} \cdot n \cdot 2^{t-1},
\end{aligned}
$$

where sec is our statistical security parameter. In this situation we obtain the bound, using the inequality above between the infinity norm in the polynomial embedding and the infinity norm in the canonical embedding,

$$
\begin{aligned}
\|c_0 - \mathfrak{s}\mathfrak{k} \cdot c_1\|_\infty^{\mathsf{can}} &\leq \|m\|_\infty^{\mathsf{can}} + 2^t \cdot (\|\epsilon \cdot v\|_\infty^{\mathsf{can}} + \|e_0\|_\infty^{\mathsf{can}} + \|e_1 \cdot \mathfrak{s}\mathfrak{k}\|_\infty^{\mathsf{can}}) \\
&\leq \|m\|_\infty^{\mathsf{can}} + 2^t \cdot (\|\epsilon\|_\infty^{\mathsf{can}} \cdot \|v\|_\infty^{\mathsf{can}} + \|e_0\|_\infty^{\mathsf{can}} + \|e_1\|_\infty^{\mathsf{can}} \cdot \|\mathfrak{s}\mathfrak{k}\|_\infty^{\mathsf{can}}) \\
&\leq \phi(p) \cdot 2^{3 \cdot \mathsf{sec}/2+1} \cdot n \cdot 2^t \\
&\qquad \cdot \left(1/2 + 20 \cdot c_1 \cdot \sigma \cdot \sqrt{\phi(p)} + 20 + 20 \cdot c_1 \cdot \sqrt{h}\right) \\
&= B_{\mathsf{clean}}^{\mathsf{dishonest}}
\end{aligned}
$$

Again this is a probabilistic bound (assuming validly distributed key generation), but assumes the worst case for the ciphertext bounds.

A.2 Noise After SwitchMod

This takes as input a ciphertext modulo q_1 and outputs a ciphertext mod q_0. The initial ciphertext is at level $q_1 = p_0 \cdot p_1$, with $q_0 = p_0$. If the input ciphertext has noise bounded by ν in the canonical embedding then the output ciphertext will have noise bounded by ν' in the canonical embedding, where

$$
\nu' = \frac{\nu}{p_1} + B_{\mathsf{scale}}.
$$

The value B_{scale} is an upper bound on the quantity $\|\tau_0 + \tau_1 \cdot \mathfrak{s}\mathfrak{k}\|_\infty^{\mathsf{can}}$, where τ_i is drawn from a distribution which is close to a complex Gaussian with variance $\phi(p) \cdot 2^{2 \cdot t}/12$. Therefore, we can (with high probability) take the upper bound to be

$$
B_{\mathsf{scale}} = c_1 \cdot 2^t \cdot \sqrt{\phi(p)/12} + c_2 \cdot 2^t \cdot \sqrt{\phi(p) \cdot h/12}.
$$

This is again a probabilistic analysis, assuming validly generated public keys.

A.3 Noise After Addition

Noise from an addition operation is purely additive.

A.4 Noise After Multiplication

Multiplication is performed by first switching modulus/levels down to level zero (whose noise we analysed above), we then tensor the resulting ciphertext (which results in a degree two ciphertext whose noise is a product of the noise of the input level zero ciphertexts), we then need to relinearize (to produce a degree one ciphertext again).

In order to estimate the output noise term in the canonical embedding for the relinearization operation we need first to estimate the size of the term (again probabilistically, assuming validly generated public keys)

$$\begin{aligned}\|2^t \cdot d_2 \cdot e_{\mathfrak{s}\mathfrak{k},\mathfrak{s}\mathfrak{k}^2}\|_\infty^{\mathsf{can}} &\leq 2^t \cdot c_2 \cdot \sqrt{q_0^2/12 \cdot \sigma^2 \cdot \phi(m)^2} \\ &= 2^t \cdot c_2 \cdot q_0 \cdot \sigma \cdot \phi(m)/\sqrt{12} \\ &= B_{\mathsf{KS}} \cdot q_0.\end{aligned}$$

Then if the input to relinearization has noise bounded by ν then the output noise value in the canonical embedding will be bounded by

$$\nu + \frac{B_{\mathsf{KS}} \cdot q_0}{p_1} + B_{\mathsf{scale}}.$$

Combining all the above, if we take two ciphertexts of level one with input noise in the canonical embedding bounded by ν and ν', the output noise level from multiplication will be bounded by

$$\nu'' = \left(\frac{\nu}{p_1} + B_{\mathsf{scale}}\right) \cdot \left(\frac{\nu'}{p_1} + B_{\mathsf{scale}}\right) + \frac{B_{\mathsf{KS}} \cdot p_0}{p_1} + B_{\mathsf{scale}}.$$

This provides a suitable bound for the "circuit" used to produce multiplication triples. It is easily seen, by examining the calculations used to produce the shared random bits, that the same bound holds in this case as well. This is because, apart from a small amount of additions of ciphertexts, we replace the multiplication of ciphertexts above with a multiplication of a ciphertext and a scalar ring element. This last operation produces less noise than the former, and so the above bound will hold in this case as well.

A.5 Valid Decryption

A ciphertext, with noise ν'', will decrypt validly if we have $c_m \cdot \nu'' < q_0/2$. However, as we always take prime cyclotomics we have $c_m = 1.2732$, [DPSZ12].

B MACCheck Procedure

In this section we recall the MACCheck procedure over $\mathbb{Z}_{2^k}$ introduced by Cramer et al. in [CDE+18]. When a secret shared value $\langle x \rangle$ is opened, i.e. the parties broadcast their shares $x^i = [x]_i \mod 2^k$, the parties need to check the MAC on this value. Note that for security reasons the parties do not send the upper bits of their shares.

The procedure in Figure 27 consists of three different commands: the **Open** command that given a secret shared value $\langle x \rangle$ outputs a value $x' \in \mathbb{Z}_{2^t}$ such that $x \equiv x' \mod 2^k$; the **Single Check** command that checks the MAC on a single shared value $\langle x \rangle$; the **Batch Check** command that checks the MACs on m shared values $\langle x_1 \rangle, \ldots, \langle x_m \rangle$. In the last two commands we assume the parties have access to a random secret shared value $\langle r \rangle$, with $r \in \mathbb{Z}_{2^s}$.

Procedure MACCheck

Open: Given a secret shared values $\langle x \rangle$, do the following.

1. Each party P_i broadcast their share $x^i = [x]_i \mod 2^k$
2. The parties reconstruct the value $\tilde{x} = \sum_{i \in [n]} x^i \mod 2^t$.

Single Check: Given an opened value $\tilde{x} \in \mathbb{Z}_{2^t}$ and a random secret shared value $\langle r \rangle$, parties proceed as follows.

1. Compute $\langle y \rangle = \langle \tilde{x} + 2^k \cdot r \rangle$
2. Each party broadcasts their shares $[y]_i$ and reconstructs $y' = \sum_{i \in [n]} [y]_i \mod 2^t$
3. Each P_i commits to $[z]_i = [\gamma_y]_i - y' \cdot [\alpha]_i$
4. All parties open their commitments and check that $\sum_{i \in [n]} [z]_i \equiv 0 \mod 2^t$
5. If the check passes then outputs $y' \mod 2^k$, otherwise output abort.

Batch Check: Given opened values $\tilde{x}_1, \ldots, \tilde{x}_m \in \mathbb{Z}_{2^t}$ and a random secret shared value $\langle r \rangle, r \in \mathbb{Z}_{2^s}$, parties proceed as follows.

1. Parties call the functionality $\mathcal{F}_{\mathsf{Rand}}$ to obtain t random values $\chi_1, \ldots, \chi_m \in \mathbb{Z}_{2^s}$, and compute
$$\tilde{y} = \sum_{j \in [m]} \chi_j \cdot \tilde{x}_j \mod 2^t.$$
2. Each party P_i computes $p^i = \sum_{j \in [m]} \chi_j \cdot p^i_j$, where $p^i_j = \frac{[x_j]_i - x^i_j}{2^k}$, and broadcasts $\tilde{p}^i = p^i + [r]_i \mod 2^s$
3. Parties compute $\tilde{p} = \sum_{i \in [n]} \tilde{p}^i \mod 2^s$
4. Each party P_i computes $\gamma^i = \sum_{j \in [m]} \chi_j \cdot [\gamma_{x_j}]_i \mod 2^t$ and
$$z^i = \gamma^i - [\alpha]_i \cdot \tilde{y} - 2^k \cdot \tilde{p} \cdot [\alpha]_i + 2^k \cdot [\gamma_r]_i \mod 2^t.$$
Then it commits to z^i
5. Parties open their commitments and verify that $\sum_{i \in [n]} z^i \equiv 0 \mod 2^t$. If the check passes output $\tilde{x_j} \mod 2^k$, otherwise output abort.

Figure 27. Procedure for opening secred shared values and checking MACs

Theorem B.1 ([CDE+18]).

1. *If the* **Single Check** *passes, the opned value $\tilde{x}$ is correct, i.e. $\tilde{x} = x \mod 2^k$, except with probability 2^{-s}.*
2. *If the* **Batch Check** *passes, the opned values $\{\tilde{x}_i\}_{i \in [m]}$ are correct, i.e. $\tilde{x}_i = x_i \mod 2^k, \forall i \in [m]$, except with probability $2^{-s+\log(s+1)}$.*