Better Preprocessing for Secure Multiparty Computation

Carsten Baum[1(✉)], Ivan Damgård[1], Tomas Toft[2], and Rasmus Zakarias[1]

[1] Department of Computer Science, Aarhus University, Aarhus, Denmark
cbaum@cs.au.dk
[2] Danske Bank, Copenhagen, Denmark

Abstract. We present techniques and protocols for the preprocessing of secure multiparty computation (MPC), focusing on the so-called SPDZ MPC scheme [14] and its derivatives [1,11,13]. These MPC schemes consist of a so-called preprocessing or offline phase where correlated randomness is generated that is independent of the inputs and the evaluated function, and an online phase where such correlated randomness is consumed to securely and efficiently evaluate circuits. In the recent years, it has been shown that such protocols (such as [5,17,18]) turn out to be very efficient in practice.

While much research has been conducted towards optimizing the online phase of the MPC protocols, there seems to have been less focus on the offline phase of such protocols (except for [11]). With this work, we want to close this gap and give a toolbox of techniques that aim at optimizing the preprocessing. We support both instantiations over small fields and large rings using somewhat homomorphic encryption and the Paillier cryptosystem [19], respectively. In the case of small fields, we show how the preprocessing overhead can basically be made independent of the field characteristic. In the case of large rings, we present a protocol based on the Paillier cryptosystem which has a lower message complexity than previous protocols and employs more efficient zero-knowledge proofs that, to the best of our knowledge, were not presented in previous work.

Keywords: Efficient multiparty computation · Preprocessing · Paillier encryption

1 Introduction

During the recent years, secure two- and multiparty computation ([16,21]) has evolved from a merely academic research topic into a practical technique for secure function evaluation (see e.g. [6]). Multiparty computation (MPC) aims

The authors acknowledge support from the Danish National Research Foundation and The National Science Foundation of China (under the grant 61061130540) for the Sino-Danish Center for the Theory of Interactive Computation, within which part of this work was performed; and also from the CFEM research center (supported by the Danish Strategic Research Council) within which part of this work was performed.

M. Manulis et al. (Eds.): ACNS 2016, LNCS 9696, pp. 327–345, 2016.
DOI: 10.1007/978-3-319-39555-5_18

at solving the following problem: How can a set of parties $\mathcal{P}_1, ..., \mathcal{P}_n$, where each party $\mathcal{P}_i$ has a secret input value x_i, compute a function $y = f(x_1, ..., x_n)$ on their values while not revealing any other information than the output y? Such function could e.g. compute a statistic on the inputs (to securely compute a mean or median) or resemble an online auction or election. Ideally, all these parties would give their secret to a trusted third party (which is incorruptible), that evaluates the function f and reveals the result y to each participant. Such a solution in particular guarantees two properties:

Privacy: Even if malicious parties collude, as long as they cannot corrupt the trusted third party they cannot gain any information except y and what they can derive from it using their inputs.

Correctness: After each party sent their input, there is no way how malicious parties can interfere with the computation of the trusted third party in such a way as to force it to output a specific result y' to the parties that are honest.

A secure multiparty computation protocol replaces such a trusted third party by an interactive protocol among the n parties, while still guaranteeing the above properties. In recent years, it has been shown that even if $n-1$ of the n parties can be corrupted, the efficiency of secure computation can be dramatically improved by splitting the protocol into different phases: During a *preprocessing* or *offline* phase, *raw material* or so-called correlated randomness is generated. This computation is both independent of f and the inputs x_i and can therefore be carried out any time before the actual function evaluation takes place. This way, a lot of the *heavy* computation that relies e.g. on public-key primitives (which we need to handle dishonest majority) will be done beforehand and need not be performed in the later *online* phase, where one can rely on *cheap* information-theoretic primitives.

This approach led to very efficient MPC protocols such as [11,13,14,17,18] to just name a few. In this work, we will primarily focus on variants of the so-called SPDZ protocol [11,14] and their preprocessing phases. They are secure against up to $n-1$ static corruptions, which will also be our adversarial model. For the preprocessing, they rely on very efficient lattice-based homomorphic cryptosystems that allow to perform both additions and multiplications on the encrypted ciphertexts and can pack a large vector of plaintexts into one ciphertext. Unfortunately, the current implementations of the preprocessing has several (non-obvious) drawbacks in terms of efficiency which we try to address in this work:

- The complexity of the preprocessing phase depends upon the size of the field over which the function f will be evaluated: It is much less efficient for small fields. The main reason behind it is that SHE schemes have no efficient reliable distributed decryption algorithm, so since the output from the preprocessing depends in part on decryption results, it must be checked for correctness. This is done by sacrificing some part of the computed data to check the remainder, but this approach only yields security inversely proportional to the field size.

Hence, especially for small fields, one has to repeat that procedure multiple times which introduces noticeable overhead.
– If the goal in the end is to do secure computation over the integers, one needs to use large fields or rings to avoid overflow. Unfortunately, the parameter sizes of SHE schemes grow very quickly if one increases the size of the underlying field, rendering them very slow in practice. This makes it interesting to investigate a preprocessing scheme using Paillier encryption, which comes with a very large ring as plaintext space.

1.1 Contributions and Technical Overview

In this work, we address the aforementioned problems and show the following results:

(1) We present a novel way of checking the correctness of shared multiplication triples for SHE schemes. In particular, we need to sacrifice only a constant fraction of the data to do the checking, where existing methods need to sacrifice a fraction $\Theta(1 - 1/\kappa)$ for error probability $2^{-\kappa}$.
(2) We show how the linearly homomorphic encryption scheme of Paillier and Damgård-Jurik [10,19] can be used more efficiently to produce multiplication triples by representing the data as polynomials and thereby reducing the amount of complex zero-knowledge proofs. Moreover, we also present zero-knowledge proofs for, e.g., plaintext knowledge that only require players to work modulo N even if the ciphertexts are defined modulo N^2. Though the technique may already be known, this did not appear in previous published work.

In the full version of this work [2] we also show a technique that improves the efficiency of the zero-knowledge proofs as used in [11,14]. Moreover, we present an optimized distributed decryption routine as it is required for our Paillier-based preprocessing. We will explain our contributions and techniques in more detail now.

Verifying Multiplicative Relations. Our goal is (somewhat simplified) to produce encrypted vectors $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ such that $\boldsymbol{x} \odot \boldsymbol{y} = \boldsymbol{z}$, where $\odot$ denotes the coordinate-wise product, or Schur product. The SPDZ protocol for creating such data uses distributed decryption during which errors may be introduced. To counter this, we encode the plaintexts in such a way that we can check the result later: We will let $\boldsymbol{x}, \boldsymbol{y}$ be codewords of a linear code. Those vectors can be put into SIMD ciphertexts of the SPDZ preprocessing scheme. Note that multiplying $\boldsymbol{x}$ and $\boldsymbol{y}$ coordinate-wise yields a codeword in a related code (namely its so-called Schur transform). Now we do a protocol to obtain an encryption of $\boldsymbol{z}$, which, however, uses unreliable decryption[1] underway. The next step is then to check

[1] With unreliable decryption, we mean that the result is only correct if no party acts maliciously during the decryption procedure.

if $\boldsymbol{z}$ is indeed codeword as expected, This can be done almost only by linear operations - which are basically *free* in the SPDZ MPC scheme, because they can all be done as local operations and do not involve sending messages.

Checking whether the result is a codeword is not sufficient, but if $\boldsymbol{z}$ is in the code and not equal to the codeword $\boldsymbol{x} \odot \boldsymbol{y}$, then an adversary would have to have cheated in a large number of positions (the minimum distance of the code). Thus, given the resulting vector $\boldsymbol{z}$ is a codeword, one checks a small number of random positions of the vector to see if it contains the product of corresponding positions in $\boldsymbol{x}$ and $\boldsymbol{y}$. During each check we have a constant probability of catching the adversary, and this quickly amplifies to our desired security levels.

Note that the only assumption that we have to make on the underlying field is that appropriate codes with good distance can be defined.

Paillier-Based Preprocessing for SPDZ. Paillier's encryption scheme is linearly homomorphic, so does not allow to perform multiplications of the plaintexts of two or more ciphertexts directly. On the other hand, it has a reliable decryption routine which is what we will make use of. Computing products of encryptions using linearly homomorphic encryption schemes is a well-known technique and works as follows: Assume $\mathcal{P}_1$ published some encryption $[\![a_1]\!], [\![b_1]\!]$, $\mathcal{P}_2$ published $[\![a_2]\!], [\![b_2]\!]$ and they want to compute values c_1, c_2 where $\mathcal{P}_1$ holds c_1 and $\mathcal{P}_2$ c_2 such that $(a_1 + a_2) \cdot (b_1 + b_2) = c_1 + c_2$.

In a protocol, $\mathcal{P}_2$ would send an encryption $[\![c_2']\!] := b_2' \cdot [\![a_1 + a_2]\!] + [\![-x_2]\!]$ to $\mathcal{P}_1$ and prove (among other things) that this b_2' is the same as the plaintext inside $[\![b_2]\!]$ (where $[\![x_2]\!]$ is an auxiliary value). $\mathcal{P}_1$ similarly sends $[\![c_1']\!] := b_1' \cdot [\![a_1 + a_2]\!] + [\![-x_1]\!]$ to $\mathcal{P}_2$ and proves a related statement. Afterwards, both use the distributed decryption to safely decrypt the value $c_1' + c_2'$, which does not reveal any information about the product if x_1, x_2 were appropriately chosen. $\mathcal{P}_1$ now sets $c_1 = c_1' + c_2' + x_1$ as her share, while $\mathcal{P}_2$ chooses $c_2 = x_2$. These shares do individually not reveal any information about the product.

Our approach is, instead of sampling all a_i, b_i independently, to let the factors be evaluations of a polynomial (that is implicitly defined), and then multiply these factors *unreliably*: Instead of giving a zero-knowledge proof that $b_2' = b_2$, we only need to prove that $\mathcal{P}_2$ knows b_2', x_2 such that the above equation is satisfied, which reduces the complexity of the proof. This means that the result is only correct if all parties honestly follow the multiplication protocol.

The products computed using unreliable multiplication now all lie on a polynomial as well, and using Lagrange interpolation one can evaluate the polynomial in arbitrary points. This can be used to efficiently (and almost locally) check if all products are correct.

We want to remark that this approach is asymptotically as efficient as existing techniques, but relies on zero-knowledge proofs with lower message complexity. It is an interesting open question how these approaches compare in practice.

1.2 Related Work

In an independent work, Frederiksen et al. showed how to preprocess data for the SPDZ MPC scheme using oblivious transfer ([15]). Their approach can make use of efficient OT-extension, but does only allow fields of characteristic 2. While this has some practical applications, it does not generalize (efficiently) to arbitrary fields. On the contrary, our techniques are particularly efficient for other use-cases when binary fields cannot be used to compute the desired function efficiently. Therefore, both results complement each other.

Our technique for checking multiplicative relations is related to the work in [4] for secret shared values in honest majority protocols and in [9] for committed values in 2-party protocols. To the best of our knowledge, this type of technique has not been used before for dishonest majority MPC.

Paillier Encryption: The Paillier encryption scheme has been used in MPC preprocessing before such as in [5]. Moreover it was also employed in various MPC schemes such as [6,8,12] to just name a few. The particular instance of the scheme that we use is from [10].

2 Preliminaries

Throughout this work, we assume that a secure point-to-point channels between the parties exist and that a broadcast channel is available. We make commitments abstractly available using the functionality $\mathcal{F}_{\text{Commit}}$ and assume the existence of a random oracle, which will be used in the coin-flipping protocol $\mathcal{P}_{\text{ProvideRandom}}$[2]. Both $\mathcal{F}_{\text{Commit}}, \mathcal{P}_{\text{ProvideRandom}}$ can be found in the full version [2]. We use $\odot$ for point-wise multiplication of vector entries, $(g, h) = d$ to denote that d is the greatest common divisor of g, h and let $[r]$ be defined as the set $[r] := \{1, ..., r\}$. We will denote vectors in bold lower-case letters such as e.g. $\boldsymbol{b}$ whereas matrices are bold upper-case letters such as $\boldsymbol{M}$. $[\![m]\!]$ denotes an encryption of a message m where the randomness is left implicit.

2.1 The SPDZ Multiparty Computation Protocol

We start out with a short primer on the [14] MPC protocol which we will mostly refer to as SPDZ. This we use not just as motivation for our results, but also to make the reader familiar with the notation.

SPDZ evaluates an arithmetic circuit C over a field $\mathbb{Z}_p$ on a gate-level, where there are addition and multiplication gates. Each value $c \in \mathbb{Z}_p$ of the computation (which is assigned to a wire in the process of the evaluation) is MACed using a uniformly random MAC secret MAC key α as $\alpha \cdot c$ and both of these values are then sum-shared among all parties. This MAC key α is fixed for all such shared values, and α is additionally sum-shared among the parties, where party $\mathcal{P}_i$ holds share α_i such that $\alpha = \sum_{i=1}^{n} \alpha_i$.

[2] In practice, this can be implemented in several ways, e.g. using a pseudorandom function and the commitment scheme $\mathcal{F}_{\text{Commit}}$.

To make the above more formal, we define the $\langle\cdot\rangle$-representation of a shared value as follows:

Definition 1. *Let $r, s, e \in \mathbb{Z}_p$, then the $\langle r\rangle$-representation of r is defined as*

$$\langle r\rangle := \big((r_1, ..., r_n), (\gamma(r)_1, ..., \gamma(r)_n)\big)$$

where $r = \sum_{i=1}^{n} r_i$ and $\alpha \cdot r = \sum_{i=1}^{n} \gamma(r)_i$. Each player $\mathcal{P}_i$ will hold his shares $r_i, \gamma(r)_i$ of such a representation. We define

$$\begin{aligned}
\langle r\rangle + \langle s\rangle &:= \big((r_1 + s_1, ..., r_n + s_n), (\gamma(r)_1 + \gamma(s)_1, ..., \gamma(r)_n + \gamma(s)_n)\big)\\
e \cdot \langle r\rangle &:= \big((e \cdot r_1, ..., e \cdot r_n), (e \cdot \gamma(r)_1, ..., e \cdot \gamma(r)_n)\big)\\
e + \langle r\rangle &:= \big((r_1 + e, r_2, ..., r_n), (\gamma(r)_1 + e \cdot \alpha_1, ..., \gamma(r)_n + e \cdot \alpha_n) \big)
\end{aligned}$$

This representation is closed under linear operations:

Proposition 1. *Let $r, s, e \in \mathbb{Z}_p$. We say that $\langle r\rangle \mathrel{\hat{=}} \langle s\rangle$ if both $\langle r\rangle, \langle s\rangle$ reconstruct to the same value. Then it holds that*

$$\langle r\rangle + \langle s\rangle \mathrel{\hat{=}} \langle r + s\rangle \quad \text{and} \quad e \cdot \langle r\rangle \mathrel{\hat{=}} \langle e \cdot r\rangle \quad \text{and} \quad e + \langle r\rangle \mathrel{\hat{=}} \langle e + r\rangle$$

In order to multiply two representations, we rely on a technique due to Beaver [3]: Let $\langle r\rangle, \langle s\rangle$ be two values where we want to calculate a representation $\langle t\rangle$ such that $t = r \cdot s$. Assume the availability of a triple[3] $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ such that a, b are uniformly random and $c = a \cdot b$. To obtain $\langle t\rangle$, one can use the procedure as depicted in Fig. 1. Correctness and privacy of this procedure were established before, e.g. in [14]. This already allows to compute on shared values, and inputting information into such a computation can also easily be achieved using standard techniques[4]. Checking that a value was indeed reconstructed correctly will be done using $\mathcal{P}_{\text{CheckMac}}$ which allows to check the MAC of the opened value without revealing the key α.

Procedure $\mathcal{P}_{\text{Mult}}$

Multiply$(\langle r\rangle, \langle s\rangle, \langle a\rangle, \langle b\rangle, \langle c\rangle)$**:**

(1) The players calculate $\langle\gamma\rangle = \langle r\rangle - \langle a\rangle, \langle\delta\rangle = \langle s\rangle - \langle b\rangle$.
(2) The players publicly reconstruct γ, δ.
(3) Each player locally calculates $\langle t\rangle = \langle c\rangle + \delta\langle a\rangle + \gamma\langle b\rangle + \gamma\delta$.
(4) Return $\langle t\rangle$ as the representation of the product.

Fig. 1. Procedure $\mathcal{P}_{\text{Mult}}$ to generate the product of two $\langle\cdot\rangle$-shared values.

[3] We will also refer to those triples as *multiplication triples* throughout this paper.
[4] Open a random value $\langle r\rangle$ to a party that wants to input x. That party then broadcasts $x - r$ and the parties jointly compute $(x - r) + \langle r\rangle = \langle x\rangle$.

This checking procedure will fail to detect an incorrect reconstruction with probability at most $2/p$ over fields of characteristic p, and similarly with probability $2/q$ over rings $\mathbb{Z}_N$ where q is the smallest prime factor of N. This in essence is captured by the following Lemma which we will also need in other cases (Fig. 2):

Lemma 1. *Assume that $\mathcal{P}_{\text{CheckMac}}$ is executed over the field $\mathbb{Z}_p$. The protocol $\mathcal{P}_{\text{CheckMac}}$ is correct and sound: It returns 1 if all the values v_i and their corresponding MACs $\gamma(v_i)$ are correctly computed and rejects except with probability $2/p$ in the case where at least one value or MAC is not correctly computed.*

Procedure $\mathcal{P}_{\text{CheckMac}}$

CheckOutput$(v_1, ..., v_t, m)$**:**

(1) The parties compute $\boldsymbol{r} \leftarrow \mathcal{P}_{\text{ProvideRandom}}.\text{ProvideRandom}(m, t)$.
(2) Each $\mathcal{P}_i$ computes $v = \sum_{i=1}^{t} \boldsymbol{r}[i] \cdot v_i$ and $\gamma_i = \sum_{j=1}^{t} \boldsymbol{r}[j] \cdot \gamma(v_j)$.
(3) Each $\mathcal{P}_i$ computes $\sigma_i = \gamma_i - \alpha_i \cdot v$ and commits to σ_i using $\mathcal{F}_{\text{Commit}}$ as c_i'.
(4) Each c_i' is opened towards all players using $\mathcal{F}_{\text{Commit}}$.
(5) If $\sigma = \sum_{i=1}^{n} \sigma_i$ is 0 then return 1, otherwise return 0.

Fig. 2. Procedure $\mathcal{P}_{\text{CheckMac}}$ to check validity of MACs.

Proof. See e.g. [11].

For some of our settings we will choose p to be rather small (i.e. of constant size in the security parameter). In this case, one can extend the $\langle\cdot\rangle-$representation as in Definition 1 by having a larger number of MACs and then check all of these MACs in parallel.

2.2 (Reed-Solomon) Codes

Let $q, k, m \in \mathbb{N}^+, m > k$ and q be a prime power. Consider the two vector spaces $\mathbb{F}_q^k, \mathbb{F}_q^m$ and a monomorphism $C : \mathbb{F}_q^k \to \mathbb{F}_q^m$ together as a *code*, i.e. $\boldsymbol{c} = C(\boldsymbol{x})$ as an encoding of $\boldsymbol{x}$ in $\mathbb{F}_q^m$. We assume that it is efficiently decidable whether $\boldsymbol{c'} \in C$ (error checking), where $\boldsymbol{c'} \in C \Leftrightarrow \exists \boldsymbol{x'} \in \mathbb{F}_q^k : \ C(\boldsymbol{x'}) = \boldsymbol{c'}$ and the minimum distance d of two codewords $\boldsymbol{x}, \boldsymbol{y} \in C$ should be large (meaning that the difference of any two distinct codewords should be nonzero in as many positions as possible). Such a code is called an $[m, k, d]$ code.

If, for every message $\boldsymbol{x} \in \mathbb{F}_q^k$ the message $\boldsymbol{x}$ reappears directly in $C(\boldsymbol{x})$ then the code is called systematic. Without loss of generality, one can assume that the first m positions of a codeword are equal to the encoded message in that case. The mapping of C can be represented as multiplication with a matrix $\boldsymbol{G}$ (called the *generator matrix*), and one can write the encoding procedure as $C : \boldsymbol{x} \mapsto \boldsymbol{G}\boldsymbol{x}$ where $\boldsymbol{G} \in \mathbb{F}_q^{m \times k}$. Similarly, we assume the existence of a *check matrix* $\boldsymbol{H} \in \mathbb{F}_q^{(m-k) \times m}$ where $\boldsymbol{H}\boldsymbol{x} = \boldsymbol{0} \Leftrightarrow \boldsymbol{x} \in C$.

For a $[m, k, d]$ code C, define the *Schur transform* (as in [13]) as $C^* = span(\{\boldsymbol{x} \odot \boldsymbol{y} \mid \boldsymbol{x}, \boldsymbol{y} \in C\})$. C^* is itself a code where the message length k' cannot be smaller than k. On the contrary, C^* has a smaller minimum distance $d' \leq d$. The actual values k', d' depend on the properties of the code C.

A code with small loss $d - d'$ with respect to the Schur transform (as we shall see later) is the so-called *Reed-Solomon* code ([20]), where the encoding C works as follows: Fix pairwise distinct and nonzero $z_1, ..., z_m \in \mathbb{F}_q$ and define the matrices $\boldsymbol{A_1} = V(z_1, ..., z_k)^{-1}$ and $\boldsymbol{A_2} = V(z_1, ..., z_m)$ where $V(\cdot)$ is the Vandermonde matrix. We then define the encoding as

$$\begin{aligned} C : \mathbb{F}_q^k &\to \mathbb{F}_q^m \\ \boldsymbol{x} &\mapsto \boldsymbol{A_2 A_1 x} \end{aligned}$$

This encoding can be made efficient since the matrices are decomposable for certain values $z_1, ..., z_k$ using the *Fast Fourier Transform* (FFT). The decoding works essentially the same way, where one computes $\boldsymbol{y}^\top \boldsymbol{A_2}^{-1} \boldsymbol{A_1}^{-1}$.

The intuition behind the encoding procedure is as follows: The k values uniquely define a polynomial f of degree at most $k - 1$, whose coefficients can be computed using $\boldsymbol{A_1}$ (as an inverse FFT). One evaluates the polynomial in the remaining $m - k$ positions using $\boldsymbol{A_2}$. The minimum distance d is exactly $m - k + 1$, since two polynomials of degree at most $k - 1$ are equal if they agree in at least k positions. Now, by letting $\boldsymbol{A_2}$ be another FFT matrix, the point-wise multiplication of codewords from C yields a codeword in C^* which is a polynomial of degree at most $2(k - 1)$ and the code C^* therefore has minimum distance $d' = m - 2k + 1$.

2.3 The Paillier Cryptosystem

We use the Paillier encryption scheme as defined in [10,19] (with some practical restrictions). Let $N = p \cdot q$ be the product of two odd, τ-bit safe-primes with $(N, \phi(N)) = 1$ (we choose τ such that the scheme has λ bit security). Paillier encryption of a message $x \in \mathbb{Z}/N\mathbb{Z}$ with randomness $r \in \mathbb{Z}/N\mathbb{Z}^*$ is defined as:

$$\mathsf{Enc}_{pk}(x, r) := r^N \cdot (N + 1)^x \bmod N^2$$

Knowing the factorization of N allows decryption of ciphertext $c \in \mathbb{Z}/N^2\mathbb{Z}^*$, e.g., by determining the randomness used,

$$r = c^{N^{-1} \bmod \phi(N)} \bmod N.$$

The decryption then proceeds as

$$x = ((c \cdot r^{-N} \mod N^2) - 1)/N \mod N$$

The KG algorithm samples an RSA modulus $N = p \cdot q$, and we let the public key be $pk = (N)$ and the secret key be $sk = (p, q, f = N^{-1} \mod \varphi(N))$. The encryption scheme is additively homomorphic and IND-CPA secure given the Composite Residuosity problem $CR[N]$ is hard.

Functionality $\mathcal{F}_{\text{KGD}}$

Generate key:

(1) On input $(\mathsf{generate\_key}, \tau, \kappa)$ by all parties, randomly sample two different primes $p, q \in \mathbb{P}$ of bit length approximately τ. Let $N = p \cdot q$ and compute $f = N^{-1} \bmod \varphi(N)$.
(2) Sample key shares $f_1, ..., f_n \in \mathbb{Z}/2^{\kappa}N\mathbb{Z}$.
(3) Output (N, f_i) to party $\mathcal{P}_i$, and save $(N, f, f_1, ..., f_n)$ locally.

Distributed decryption:

(1) When receiving $(\mathsf{decrypt}, f_i, c)$ from each party $\mathcal{P}_i$, check whether some $(N, f, f_1, ..., f_n)$ was stored. If not, return $\perp$.
(2) Send $(x, r) \leftarrow \mathsf{Dec}_{sk}(c)$ to the adversary. Upon receiving $x' \in \{(x, r), \perp\}$ from the adversary, send (result, x') to all players.

Fig. 3. Functionality $\mathcal{F}_{\text{KGD}}$ that provides shared keys and decrypts ciphertexts.

During the decryption of a ciphertext as described above one does completely recover the randomness used during encryption. This gives rise to a reliable distributed decryption algorithm, which we describe in the full version of this work[5] Both key generation and distributed decryption are described in the functionality Fig. 3.

Observe that the distributed decryption does also output the randomness used in the ciphertext. This can be harmful in some applications, but is sufficient for our application.

3 More Efficient Preprocessing from SHE

In this section, we present an improved preprocessing protocol for SPDZ over large fields. Towards achieving this, we overhaul the triple generation in a way that allows more efficient checks of correctness. This check uses the original SPDZ preprocessing as a black box[6] (see Fig. 4). Our approach introduces some computational overhead, but we show how this overhead can be reduced. In the full version of this work ([2]), we additionally present a technique to improve the zero-knowledge proofs of plaintext knowledge used in [11].

Offline Phase Protocol. Let C be some $[m, k, d]$ Reed-Solomon code as described in the previous section. Moreover, let C^* be its $[m, k', d']$ Schur transform. We assume the existence of a functionality that samples *faulty correlated*

[5] One can also find such an algorithm in [10], but our solution allows for a much simpler decryption routine. In particular, no zero-knowledge proofs are involved in the decryption process.

[6] We therefore abstain from introducing the concept of SHE in this work and refer the reader to [2,14] for more details on the subject.

Functionality $\mathcal{F}_{\text{TripleGen}}$

This functionality generates a shared MAC key α and (potentially faulty) $\langle\cdot\rangle$-representations.

Initialize: On input (init, p, C) from all players, the functionality stores the prime p and a description of the code C. $\mathcal{A}$ chooses the set of parties $\mathcal{I} \subset \{1, \ldots, n\}$ he corrupts.

(1) For all $i \in \mathcal{I}$, $\mathcal{A}$ inputs $\alpha_i \in \mathbb{Z}_p$, while for all $i \notin \mathcal{I}$, the functionality chooses $\alpha_i \leftarrow \mathbb{Z}_p$ at random.
(2) Set they key $\alpha = \sum_{i=1}^{n} \alpha_i$ and send α_i to $\mathcal{P}_i, i \notin \mathcal{I}$.

Triples: On input $(\mathsf{triples})$ from all parties, the functionality does the following to generate triples:

(1) For $i \notin \mathcal{I}$, the functionality samples $\boldsymbol{a}_i, \boldsymbol{b}_i \in C$ at random.
(2) For $i \in \mathcal{I}$, $\mathcal{A}$ inputs $\boldsymbol{a}_i, \boldsymbol{b}_i, \boldsymbol{c}_i, \boldsymbol{\delta}, \boldsymbol{\Delta}_{\gamma,a}, \boldsymbol{\Delta}_{\gamma,b}, \boldsymbol{\Delta}_{\gamma,c} \in \mathbb{Z}_p^m$. If $\boldsymbol{a}_i, \boldsymbol{b}_i \notin C$ then stop.
(3) Define $\boldsymbol{a} = \sum_{j=1}^{n} \boldsymbol{a}_j, \boldsymbol{b} = \sum_{j=1}^{n} \boldsymbol{b}_j$.
(4) Let $j \notin \mathcal{I}$ be the smallest index of an honest player. For all $i \notin \mathcal{I}, i \neq j$ choose $\boldsymbol{c}_i \in \mathbb{Z}_p^m$ uniformly at random. For $\mathcal{P}_j$ let $\boldsymbol{c}_j = \boldsymbol{a} \odot \boldsymbol{b} + \boldsymbol{\delta} - \sum_{i \in [n], i \neq j} \boldsymbol{c}_i$. Send $\boldsymbol{a}_i, \boldsymbol{b}_i, \boldsymbol{c}_i$ to each honest $\mathcal{P}_i$.
(5) Run the macros
$\langle \boldsymbol{a} \rangle \leftarrow \text{Angle}(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n, \alpha, \boldsymbol{\Delta}_{\gamma,a}, m, p)$,
$\langle \boldsymbol{b} \rangle \leftarrow \text{Angle}(\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n, \alpha, \boldsymbol{\Delta}_{\gamma,b}, m, p)$,
$\langle \boldsymbol{c} \rangle \leftarrow \text{Angle}(\boldsymbol{c}_1, \ldots, \boldsymbol{c}_n, \alpha, \boldsymbol{\Delta}_{\gamma,c}, m, p)$.
(6) Return $(\langle \boldsymbol{a} \rangle, \langle \boldsymbol{b} \rangle, \langle \boldsymbol{c} \rangle)$.

$\text{Angle}(\boldsymbol{r}_1, \ldots, \boldsymbol{r}_n, \alpha, \boldsymbol{\Delta}_\gamma, m, p)$**:** This macro will be run to create $\langle\cdot\rangle$-representations.

(1) Define $\boldsymbol{r} = \sum_{i=1}^{n} \boldsymbol{r}_i$
(2) For $i \in \mathcal{I}$, $\mathcal{A}$ inputs $\boldsymbol{\gamma}_i \in \mathbb{Z}_p^m$, and for $i \notin \mathcal{I}$, the functionality chooses $\boldsymbol{\gamma}_i \leftarrow \mathbb{Z}_p^m$ at random except for $\boldsymbol{\gamma}_j$, with j being the smallest index not in $\mathcal{I}$.
(3) Set $\boldsymbol{\gamma} = \alpha \cdot \boldsymbol{r} + \boldsymbol{\Delta}_\gamma$ and $\boldsymbol{\gamma}_j = \boldsymbol{\gamma} - \sum_{j \neq i=1}^{n} \boldsymbol{\gamma}_i$. For every honest party $\mathcal{P}_i$, send γ_i.
(4) Define $\langle \boldsymbol{r} \rangle = (\boldsymbol{r}_1, ..., \boldsymbol{r}_n, \boldsymbol{\gamma}_1, ..., \boldsymbol{\gamma}_n)$. Return $\langle \boldsymbol{r} \rangle$.

Fig. 4. Functionality $\mathcal{F}_{\text{TripleGen}}$ that generates potentially faulty triples.

randomness and which is depicted in Fig. 4. It generates random codewords as the shares of factors $\boldsymbol{a}, \boldsymbol{b}$ of multiplication triples and also enforces that malicious parties choose such codewords as their shares. The functionality then computes a product and shares it among all parties, subject to the constraint that $\mathcal{A}$ can arbitrarily modify the sum and the shares of malicious parties. Figure 4 can be implemented using a SHE scheme as was shown in [14]. As a twist, the zero-knowledge proofs must be slightly extended to show that the vectors inside the ciphertexts contain codewords from C. Based on this available functionality, we show that one can implement $\mathcal{F}_{\text{FullTripleGen}}$ as depicted in Fig. 5 using our protocol $\Pi_{\text{TripleCheck}}$. $\mathcal{F}_{\text{FullTripleGen}}$ is similar to $\mathcal{F}_{\text{TripleGen}}$ but additionally ensures that all multiplication triples are correct.

Functionality $\mathcal{F}_{\text{FULLTRIPLEGEN}}$

Let $\mathcal{I}$ be the set of parties that are controlled by $\mathcal{A}$, $u \in \mathbb{N}^+$. This functionality generates a shared MAC key α and $\langle\cdot\rangle$-representations. It uses the macro Angle as depicted in $\mathcal{F}_{\text{TRIPLEGEN}}$.

Initialize: On input (init, p, u) from all players, the functionality stores the prime p and the vector dimension u. $\mathcal{A}$ chooses the set of parties $\mathcal{I} \subset [n]$ he corrupts.

(1) For all $i \in \mathcal{I}$, $\mathcal{A}$ inputs $\alpha_i \in \mathbb{Z}_p$, while for all $i \notin \mathcal{I}$, the functionality chooses $\alpha_i \leftarrow \mathbb{Z}_p$ at random.
(2) Set they key $\alpha = \sum_{i=1}^{n} \alpha_i$ and send α_i to $\mathcal{P}_i, i \notin \mathcal{I}$.

Triples: On input $(\mathsf{triples})$ the functionality does the following

(1) Let $\mathcal{A}$ input $\boldsymbol{a}_i, \boldsymbol{b}_i, \boldsymbol{c}_i, \boldsymbol{\Delta}_{\gamma,a}, \boldsymbol{\Delta}_{\gamma,b}, \boldsymbol{\Delta}_{\gamma,c} \in \mathbb{Z}_p^u$ for each $i \in \mathcal{I}$.
(2) Choose $\boldsymbol{a}_i, \boldsymbol{b}_i \in \mathbb{Z}_p^u$ for each honest $\mathcal{P}_i$ uniformly at random. Set $\boldsymbol{a} = \sum_i \boldsymbol{a}_i, \boldsymbol{b} = \sum_i \boldsymbol{b}_i$ and define $\boldsymbol{c} = \boldsymbol{a} \odot \boldsymbol{b}$.
(3) Let j be the smallest number in $[n]\backslash\mathcal{I}$. Choose uniformly random $\boldsymbol{c}_i \in R^u$ for each $\mathcal{P}_i$ with $i \in [n]\backslash\mathcal{I}, i \neq j$ and set $\boldsymbol{c}_j = \boldsymbol{c} - \sum_{i\in[n], i\neq j} \boldsymbol{c}_i$.
(4) Run the macros
$\langle \boldsymbol{a} \rangle \leftarrow \text{Angle}(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n, \alpha, \boldsymbol{\Delta}_{\gamma,a}, u, p)$,
$\langle \boldsymbol{b} \rangle \leftarrow \text{Angle}(\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n, \alpha, \boldsymbol{\Delta}_{\gamma,b}, u, p)$,
$\langle \boldsymbol{c} \rangle \leftarrow \text{Angle}(\boldsymbol{c}_1, \ldots, \boldsymbol{c}_n, \alpha, \boldsymbol{\Delta}_{\gamma,c}, u, p)$.
(5) Return $(\langle \boldsymbol{a} \rangle, \langle \boldsymbol{b} \rangle, \langle \boldsymbol{c} \rangle)$.

Fig. 5. Functionality $\mathcal{F}_{\text{FULLTRIPLEGEN}}$ that generates correct triples.

The main idea of this protocol follows the outline as presented in the introduction:

(1) Check that the output vector $\boldsymbol{c}$ is a codeword of C^*. If so, then the error vector $\boldsymbol{\delta}$ is also a codeword, meaning that either it is $\mathbf{0}$ or *it has weight at least*d'.
(2) Open a fraction of the triples to check whether they are indeed correct. If so, then $\boldsymbol{\delta}$ must be the all-zero vector with high probability.

Due to the lack of space, the proof of security of $\Pi_{\text{TRIPLECHECK}}$ is postponed to the full version of this work [2], where the security is proven in the UC framework [7].

Fast and Amortized Checks. In the protocol presented in Fig. 7, we check each potential code vector separately. Let $\boldsymbol{H} \in \mathbb{Z}_p^{l\times m}$ be the check matrix of the Schur transform of the code. Multiplication with a check matrix $\boldsymbol{H}$ can be done in $O(m^2)$ steps - but assuming that this must be carried out for a number of e.g. m vectors this leads to $O(m^3)$ operations, if done trivially. Let us put all the l input vectors $\boldsymbol{a_1}, ..., \boldsymbol{a_l}$ into a matrix $\boldsymbol{A} = [\boldsymbol{a_1}||\boldsymbol{a_2}||...||\boldsymbol{a_l}]$. If all vectors are drawn from the code, then $\boldsymbol{HA} = \mathbf{0}$.

Now consider another generator matrix $\boldsymbol{G} \in \mathbb{Z}_p^{m' \times l}$ for a Reed-Solomon code of message dimension l, where we denote the redundancy as $d \in O(m)$ again (we can easily assume that $m' \in O(m)$). Multiplication of each of the matrices $\boldsymbol{H}, \boldsymbol{A}$ with $\boldsymbol{G}$ can be done in time $m'^2 \cdot \log(m')$ using the FFT, and one can precompute $\boldsymbol{GH}$ before the actual computation takes place. $\boldsymbol{GHAG}^\top$ is a zero matrix if $\boldsymbol{A}$

Procedure $\mathcal{P}_{\text{MatrixMultCheck}}$

CheckMultiplication($\boldsymbol{H}, \boldsymbol{A}$)**:**

(1) Compute the matrices $\boldsymbol{GH}$ and $\boldsymbol{AG}^\top$.
(2) For $j \in [m']$ select a pair $(x_j, y_j) \in \{1, ..., m'\}^2$.
(3) For $j \in [m']$, compute z_j as the inner product of the x_jth row of $\boldsymbol{GH}$ and the y_jth column of $\boldsymbol{AG}^\top$.
(4) If all z_i are 0 return *accept*, otherwise *reject*.

Fig. 6. Procedure $\mathcal{P}_{\text{MatrixMultCheck}}$ to check whether a matrix product is zero.

Protocol $\Pi_{\text{TripleCheck}}$

Let $\boldsymbol{H}$ be the check matrix of C^* and $t \in \mathbb{N}^+, t < k-1$ be the upper bound on the number of opened triples. We assume that both C, C^* are in systematic form, and are over the field $\mathbb{Z}_p$.

Initialize:

(1) All parties send (init, p, C) to $\mathcal{F}_{\text{TripleGen}}$ to receive their shares α_i of α.

Triples:

(1) All parties send $(\mathsf{triples})$ to $\mathcal{F}_{\text{TripleGen}}$ and obtain $(\langle \boldsymbol{a} \rangle, \langle \boldsymbol{b} \rangle, \langle \boldsymbol{c} \rangle)$.
(2) Let $\boldsymbol{c}_i$ be $\mathcal{P}_i$s share of $\langle \boldsymbol{c} \rangle$. Each party locally computes $\boldsymbol{\sigma}_i = \boldsymbol{H}\boldsymbol{c}_i$ and commits to $\boldsymbol{\sigma}_i$ using $\mathcal{F}_{\text{Commit}}$.
(3) Each party $\mathcal{P}_i$ opens its commitments to $\boldsymbol{\sigma}_i$ towards all parties. Check if $\boldsymbol{0} = \sum_i \boldsymbol{\sigma}_i$. If not, abort.
(4) Let $A = [m]$. For $j \in [t]$ all parties do the following
  (4.1) Sample the uniformly random value $r \leftarrow \text{ProvideRandom}(m, 1)$. Set $A \leftarrow A \backslash \{r\}$.
  (4.2) Each party $\mathcal{P}_i$ commits to its shares $\boldsymbol{a}_i[r], \boldsymbol{b}_i[r], \boldsymbol{c}_i[r]$ using $\mathcal{F}_{\text{Commit}}$.
  (4.3) Each party opens its commitments towards all other parties.
  (4.4) Each party checks that $(\sum_i \boldsymbol{a}_i[r]) \cdot (\sum_i \boldsymbol{b}_i[r]) = \sum_i \boldsymbol{c}_i[r]$. If not, then they abort.
(5) Let $U = [m] \backslash A$, where $U = \{u_1, ..., u_l\}$. Compute $d \leftarrow \mathcal{P}_{\text{CheckMac}}.\text{CheckOutput}(\boldsymbol{\sigma}, \boldsymbol{a}[u_1], \boldsymbol{b}[u_1], \boldsymbol{c}[u_1], ..., \boldsymbol{a}[u_l], \boldsymbol{b}[u_l], \boldsymbol{c}[u_l])$. If $d \neq 0$ the parties return $\perp$.
(6) Let $O \subset A$ be the smallest $k - t - 1$ indices of A. The parties output $(\langle \boldsymbol{a}[O] \rangle, \langle \boldsymbol{b}[O] \rangle, \langle \boldsymbol{c}[O] \rangle)$.

Fig. 7. Protocol $\Pi_{\text{TripleCheck}}$ that checks the correctness of triples.

only consists of codewords. On the other hand, consider $\boldsymbol{GHA}$: If one row is not a codeword, then it will be encoded to a vector with weight at least d due to the distance of the code. Multiplying with $\boldsymbol{G}^\top$ will then yield a matrix where at least d^2 entries are nonzero. Since both $m', d \in O(m)$, the fraction $\frac{d^2}{m'^2}$ is constant. One can compute both $\boldsymbol{GH}$ and $\boldsymbol{AG}^\top$ in time $m'^2 \cdot \log(m^{\,prime})$ using the FFT, and then choose rows/columns from both product matrices for which one then computes the scale product. In case that at least one $\boldsymbol{a}_i$ is not a codeword, it will be nonzero with constant probability. Repeating this experiment $\Omega(m')$ times yields 0 in all cases only with probability negligible in m' (Fig. 6). We refer to [13] for more details on this technique.

4 Preprocessing from Paillier Encryption

In this section we present a novel approach to produce multiplication triples using Paillier's cryptosystem. In comparison to previous work which uses heavy zero-knowledge machinery to prove that multiplications are done correctly, we choose a somewhat different approach that is related to the preprocessing protocol from the previous section. Moreover, we present two zero-knwoledge proofs which are used in the protocol. In comparison to previous work, they will require to send less bits per proof instance.

Protocol Π_{ZKPoPK}

$\mathcal{P}$ proves the relation R_{ZKPoPK}.

(1) $\mathcal{P}$ chooses $s \leftarrow \mathbb{Z}/N\mathbb{Z}^*$ and sends $t = s^N \mod N$ to $\mathcal{V}$.
(2) $\mathcal{V}$ chooses $e \leftarrow \mathbb{Z}/N\mathbb{Z}$ and sends it to $\mathcal{P}$.
(3) $\mathcal{P}$ sends $k = s \cdot r^e \mod N$ to $\mathcal{V}$.
(4) $\mathcal{V}$ accepts if $k^N = c^e \cdot t \mod N$ and otherwise rejects.

Fig. 8. Protocol Π_{ZKPoPK} to prove knowledge of plaintexts of Paillier encryptions.

4.1 Proving Statements About Paillier Ciphertexts

First, consider a regular proof of plaintext knowledge. For Paillier encryption, one would prove the following relation:

$$R_{\text{ZKPoPK}} = \big\{(a, w) \;\big|\; a = (c, pk) \wedge w = (x, r) \wedge x \in \mathbb{Z}/N\mathbb{Z} \wedge r \in \mathbb{Z}/N\mathbb{Z}^* \wedge c = \mathsf{Enc}_{pk}(x, r)\big\}$$

Throughout the protocol, the parties must compute products with ciphertexts, where we want to establish that a party *knows* which value it multiplied in. This can be captured as follows:

$$R_{\text{PoM}} = \big\{(a,w) \;\big|\; a = (z,\hat{z},pk) \wedge w = (b,c,r) \wedge b,c \in \mathbb{Z}/N\mathbb{Z} \wedge r \in \mathbb{Z}/N\mathbb{Z}^* \wedge \hat{z} = z^b \cdot \mathsf{Enc}_{pk}(c,r) \bmod N^2\big\}$$

Protocol Π_{PoM}

$\mathcal{P}$ proves the relation R_{PoM}.

(1) $\mathcal{P}$ generates $t,u \in \mathbb{Z}/N\mathbb{Z}, v \in \mathbb{Z}/N\mathbb{Z}^*$. He then sends $f = z^t \cdot \mathsf{Enc}_{pk}(u,v) \mod N^2$ to $\mathcal{V}$.
(2) $\mathcal{V}$ chooses a uniformly random $e \in \mathbb{Z}/N\mathbb{Z}$ and sends it to $\mathcal{P}$.
(3) $\mathcal{P}$ computes $g = t + e \cdot b \mod N, h = u + e \cdot c \mod N, i = v \cdot r^e \mod N$ and and sends (g,h,i) to $\mathcal{V}$.
(4) $\mathcal{V}$ accepts if $z^g \cdot \mathsf{Enc}_{pk}(h,i) = \hat{z}^e \cdot f \mod N^2$, and rejects otherwise.

Fig. 9. Protocol Π_{PoM} to prove linear relation on ciphertexts.

In the following, we present honest-verifier perfect zero-knowledge proofs for both $R_{\text{ZKPoPK}}, R_{\text{PoM}}$ between a prover $\mathcal{P}$ and verifier $\mathcal{V}$. In order to use them in the preprocessing protocol, one can either make them non-interactive using the Fiat-Shamir transformation in the Random Oracle Model, or use the secure coin-flip protocol $\mathcal{P}_{\text{ProvideRandom}}$ to sample the challenge e. Since during a protocol instance, many proofs are executed in parallel, one can use the same challenge for all instances and so the complexity of doing the coin-flip is not a significant cost.

For practical implementations, one can choose the random value e from a smaller interval like e.g. $[0, 2^\kappa]$ where κ is the statistical security parameter. This also yields negligible cheating probability[7]. The proof that $\Pi_{\text{ZKPoPK}}, \Pi_{\text{PoM}}$ are in fact honest-verifier zero-knowledge proofs for the relations $R_{\text{ZKPoPK}}, R_{\text{PoM}}$ can be found in the full version of this work.

4.2 Computing and Checking Triples

Our protocol $\Pi_{\text{PaillierTripleGen}}$, on a high level, runs in the following phases:

(1) In a first step, every party encrypts uniformly random values.
(2) Take $k+2$ values which define a polynomial A of degree $k+1$ uniquely (when considered as evaluations in the points $1, ..., k+2$). Interpolate this polynomial A in the next $k+2$ points locally, encrypt these points and prove that the encrypted values are indeed points that lie on A. Then the same is done for a polynomial B.

[7] For the soundness of the proof, we rely on the fact that $(e - e', N) = 1$ which indeed is always true if $e, e' \ll \sqrt{N}$ and N is a safe RSA modulus.

Protocol $\Pi_{\text{PaillierTripleGen}}$ (Part 1)

A protocol to perform preprocessing for the SPDZ protocol using Paillier encryption.

Initialize: On input $(\mathsf{init}, \mathbb{Z}/N\mathbb{Z}, k)$ the parties do the following:

(1) Each party $\mathcal{P}_i$ picks $\alpha_i \in \mathbb{Z}/N\mathbb{Z}$ uniformly at random, broadcasts a fresh encryption $[\![\alpha_i]\!]$ and proves knowledge of plaintext of $[\![\alpha_i]\!]$ using Π_{ZKPoPK}.
(2) The parties compute $[\![\alpha]\!] \leftarrow \prod_{i=1}^{n} [\![\alpha_i]\!]$.
(3) Each $\mathcal{P}_i$ stores $[\![\alpha]\!]$ as the encrypted MAC key and its share α_i of the MAC key.

Triples: On input $(\mathsf{triples})$ the parties do the following:

(1) For $j \in [k+2]$ each $\mathcal{P}_i$ picks $A_i(j), B_i(j) \in \mathbb{Z}/N\mathbb{Z}$ uniformly at random, computes $[\![A_i(j)]\!], [\![B_i(j)]\!]$ and broadcasts $([\![A_i(j)]\!], [\![B_i(j)]\!])_{j\in[k+2]}$ together with proofs of Π_{ZKPoPK}.
(2) For $j \in [k+2]$ every party $\mathcal{P}_i$ defines the polynomials $A_i(\cdot), B_i(\cdot)$ using $A_i(j), B_i(j)$ as evaluations. Each party computes and broadcasts $([\![A_i(l)]\!], [\![B_i(l)]\!])_{l=k+3,\ldots,2k+2}$ together with proofs of plaintext knowledge using Π_{ZKPoPK}.
(3) The parties locally compute

$$[\![A(l)]\!] = \prod_{i=1}^{n} [\![A_i(l)]\!] \text{ and } [\![B(l)]\!] = \prod_{i=1}^{n} [\![B_i(l)]\!]$$

(4) The parties sample $\beta \leftarrow \mathcal{P}_{\text{ProvideRandom}}.\text{ProvideRandom}(N-2k-3, 1)+2k+3$ so that $\beta \in \mathbb{Z}/N\mathbb{Z} \setminus \{0, ..., 2k+2\}$.
(5) Define $A^{\top}(\beta)$ to be the value $A(\beta)$ computed using Lagrange interpolation and the values $A(1), ..., A(k+2)$ and similarly $A^{\perp}(\beta)$ to be $A(\beta)$ computed using $A(1), ..., A(2k+2)$. Every $\mathcal{P}_i$ locally computes

$$[\![A^{\dagger}(\beta)]\!] = [\![A^{\top}(\beta)]\!]/[\![A^{\perp}(\beta)]\!] \text{ and } [\![B^{\dagger}(\beta)]\!] = [\![B^{\top}(\beta)]\!]/[\![B^{\perp}(\beta)]\!]$$

(6) The parties decrypt $[\![A^{\dagger}(\beta)]\!], [\![B^{\dagger}(\beta)]\!]$ and check whether $A^{\dagger}(\beta) = B^{\dagger}(\beta) = 0 \bmod N$. Otherwise they abort.

Fig. 10. Protocol $\Pi_{\text{PaillierTripleGen}}$ to generate correct random triples out of random single values, Part 1.

(3) An unreliable point-wise multiplication of A, B is performed. The resulting polynomial C is interpolated in a random point β, and it is checked whether the multiplicative relation holds. This is enough to check correctness of all triples due to the size of N.
(4) Share the points of C among all parties as random shares.
(5) For all of the shares of A, B, C that were generated in the protocol, products with the MAC key α are computed. Correctness of the multiplication with α is checked and if the check is passed, the MACs are reshared among the parties in the same way as the points of C.

Protocol $\Pi_{\text{PaillierTripleGen}}$ (Part 2)

Triples:

(7) For $j \in [2k+2]$ each $\mathcal{P}_i$ chooses $r_{i,j} \leftarrow \mathbb{Z}/N\mathbb{Z}^*$, computes encryptions

$$[\![\hat{c}_{i,j}]\!] \leftarrow [\![A(j)]\!]^{B_i(j)} \mathsf{Enc}_{pk}(0, r_{i,j})$$

broadcasts the $[\![\hat{c}_{i,j}]\!]$ and proves the relation using Π_{PoM}.

(8) For $j \in [2k+2]$ each $\mathcal{P}_i$ picks $\tilde{c}_{i,j} \in \mathbb{Z}/N\mathbb{Z}$ uniformly at random, computes $[\![\tilde{c}_{i,j}]\!]$ and broadcasts $([\![\tilde{c}_{i,j}]\!])_{j\in\{0,\dots,2k+3\}}$ together with proofs of Π_{ZKPoPK}.

(9) For $j \in [2k+2]$ the parties locally compute

$$[\![\hat{c}_j]\!] = \prod_{i=1}^{n} [\![\hat{c}_{i,j}]\!] / \prod_{i=1}^{n} [\![\tilde{c}_{i,j}]\!]$$

and publicly decrypt $\hat{c}_j$.

(10) For $j \in [2k+2]$ each party $\mathcal{P}_i$ sets

$$[\![C_1(j)]\!] = [\![\tilde{c}_{1,j}]\!] \cdot [\![\hat{c}_j]\!] \qquad \text{and} \qquad [\![C_i(j)]\!] = [\![\tilde{c}_{t,j}]\!]$$

for $t \in [n], t \neq 1$ and $[\![C(j)]\!] = \prod_{i=1}^{n} [\![C_i(j)]\!]$ and its share of $C(j)$ as

$$C_i(j) = \begin{cases} \tilde{c}_{1,j} + \hat{c}_j & \text{if } i = 1 \\ \tilde{c}_{i,j} & \text{else} \end{cases}$$

(11) The parties sample $\beta \leftarrow \mathcal{P}_{\text{ProvideRandom}}.\text{ProvideRandom}(N-k-1, 1)+k+1$ so that $\beta \leftarrow \mathbb{Z}/N\mathbb{Z} \setminus \{0, ..., k\}$.

(12) The parties compute $[\![A(\beta)]\!], [\![B(\beta)]\!], [\![C(\beta)]\!]$ locally using Lagrange interpolation and then decrypt these values.

(13) If $A(\beta) \cdot B(\beta) \neq C(\beta) \bmod N$ then abort.

(14) Each $\mathcal{P}_i$ picks $s_i \in \mathbb{Z}/N\mathbb{Z}$ uniformly at random, computes $[\![s_i]\!]$ and broadcasts $[\![s_i]\!]$ together with a proof of Π_{ZKPoPK}. Let $s = \sum_i s_i$.

(15) We define the following abbreviation:

$$t_{i,j} \leftarrow \begin{cases} s_i \text{ for } j = 0 \\ A_i(j) \text{ for } j = 1, ..., k \\ B_i(j) \text{ for } j = k+1, ..., 2k \\ C_i(j) \text{ for } j = 2k+1, ..., 3k \end{cases} \quad \text{and } t_j \leftarrow \begin{cases} s \text{ for } j = 0 \\ A(j) \text{ for } j = 1, ..., k \\ B(j) \text{ for } j = k+1, ..., 2k \\ C(j) \text{ for } j = 2k+1, ..., 3k \end{cases}$$

Fig. 11. Protocol $\Pi_{\text{PaillierTripleGen}}$ to generate correct random triples out of random single values, Part 2.

The protocol $\Pi_{\text{PaillierTripleGen}}$ can be found in Figs. 10, 11 and 12. The proof of security in the UC framework [7] as well as a short introduction into the UC framework can be found in the full version of this work.

Protocol $\Pi_{\text{PaillierTripleGen}}$ (Part 3)

Triples:

(16) For $j = 0, \ldots, 3k$ each $\mathcal{P}_i$ picks $r_{i,j} \in \mathbb{Z}/N\mathbb{Z}^*$ uniformly at random and computes

$$[\![t_{i,j} \cdot \alpha]\!] \leftarrow [\![\alpha]\!]^{t_{i,j}} \cdot \mathsf{Enc}_{pk}(0, r_{i,j})$$

then broadcasts $([\![t_{i,j} \cdot \alpha]\!])$ and proves the relation using Π_{PoM}.

(17) For $j = 0, \ldots, 3k$, $\mathcal{P}_1, \ldots, \mathcal{P}_n$ compute

$$[\![t_j \cdot \alpha]\!] \leftarrow \prod_{i=1}^{n} [\![t_{i,j} \cdot \alpha]\!]$$

(18) The parties sample $\beta \leftarrow \mathcal{P}_{\text{ProvideRandom}}.\text{ProvideRandom}(N, 1)$.

(19) All parties compute

$$[\![v]\!] \leftarrow \prod_{j=0}^{3k} [\![t_j]\!]^{\beta^j} \text{ and } [\![v']\!] \leftarrow \prod_{j=0}^{3k} [\![t_j \cdot \alpha]\!]^{\beta^j}$$

(20) The parties jointly decrypt $[\![v]\!]$ to v and check that the decryption was correct.

(21) The parties jointly decrypt

$$[\![M]\!] \leftarrow [\![\alpha]\!]^v / [\![v']\!]$$

and verify that $M = 0$, otherwise they abort. All parties verify correctness of decryption.

(22) For $j \in [3k]$ each $\mathcal{P}_i$ picks $m_{i,j} \in \mathbb{Z}/N\mathbb{Z}$ uniformly at random, computes $[\![m_{i,j}]\!]$ and broadcasts $([\![m_{i,j}]\!])_{j \in [3k]}$ together with proofs of Π_{ZKPoPK}.

(23) For each $j \in [3k]$, the parties compute

$$[\![O_j]\!] \leftarrow [\![t_j \cdot \alpha]\!] / \prod_{i=1}^{n} [\![m_{i,j}]\!]$$

and publicly decrypt $[\![O_j]\!]$. All parties verify correctness of decryption.

(24) For each $j \in [3k]$, each $\mathcal{P}_i$ determines its share $\gamma(t_j)_i$, of the MAC $\gamma(t_j)$ of t_j as

$$\gamma(t_j)_i \leftarrow \begin{cases} O_j + m_{i,j} & \text{for } i = 1 \\ m_{i,j} & \text{for } 1 < i \leq n \end{cases}$$

(25) Each party $\mathcal{P}_i$ uses $t_{i,j}, \gamma(t_j)_i$ as its shares of $\langle t_j \rangle$.

Fig. 12. Protocol $\Pi_{\text{PaillierTripleGen}}$ to generate correct random triples out of random single values, Part 3.

References

1. Baum, C., Damgård, I., Orlandi, C.: Publicly auditable secure multi-party computation. In: Abdalla, M., De Prisco, R. (eds.) SCN 2014. LNCS, vol. 8642, pp. 175–196. Springer, Heidelberg (2014)
2. Baum, C., Damgård, I., Toft, T., Zakarias, R.: Better preprocessing for secure multiparty computation (2016). https://eprint.iacr.org/2016/048

3. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 420–432. Springer, Heidelberg (1992)
4. Ben-Sasson, E., Fehr, S., Ostrovsky, R.: Near-linear unconditionally-secure multiparty computation with a dishonest minority. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 663–680. Springer, Heidelberg (2012)
5. Bendlin, R., Damgård, I., Orlandi, C., Zakarias, S.: Semi-homomorphic encryption and multiparty computation. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 169–188. Springer, Heidelberg (2011)
6. Bogetoft, P., et al.: Secure multiparty computation goes live. In: Dingledine, R., Golle, P. (eds.) FC 2009. LNCS, vol. 5628, pp. 325–343. Springer, Heidelberg (2009)
7. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: Proceedings of 42nd IEEE Symposium on Foundations of Computer Science, 2001, pp. 136–145. IEEE (2001)
8. Cramer, R., Damgård, I.B., Nielsen, J.B.: Multiparty computation from threshold homomorphic encryption. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 280–300. Springer, Heidelberg (2001)
9. Cramer, R., Damgård, I., Pastro, V.: On the amortized complexity of zero knowledge protocols for multiplicative relations. In: Smith, A. (ed.) ICITS 2012. LNCS, vol. 7412, pp. 62–79. Springer, Heidelberg (2012)
10. Damgård, I., Jurik, M.: A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. PKC 2001. LNCS, vol. 1992, pp. 119–136. Springer, Heidelberg (2001)
11. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority – or: breaking the SPDZ limits. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013. LNCS, vol. 8134, pp. 1–18. Springer, Heidelberg (2013)
12. Damgård, I.B., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 247–264. Springer, Heidelberg (2003)
13. Damgård, I., Zakarias, S.: Constant-overhead secure computation of boolean circuits using preprocessing. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 621–641. Springer, Heidelberg (2013)
14. Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer, Heidelberg (2012)
15. Frederiksen, T.K., Keller, M., Orsini, E., Scholl, P.: A unified approach to MPC with preprocessing using OT. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015. LNCS, vol. 9452, pp. 711–735. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48797-6_29
16. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game. In: Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, pp. 218–229. ACM (1987)
17. Lindell, Y., Pinkas, B., Smart, N.P., Yanai, A.: Efficient constant round multiparty computation combining BMR and SPDZ. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9216, pp. 319–338. Springer, Heidelberg (2015)
18. Nielsen, J.B., Nordholt, P.S., Orlandi, C., Burra, S.S.: A new approach to practical active-secure two-party computation. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 681–700. Springer, Heidelberg (2012)

19. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 223–238. Springer, Heidelberg (1999)
20. Reed, I.S., Solomon, G.: Polynomial codes over certain finite fields. J. Soc. Ind. Appl. Math. **8**(2), 300–304 (1960)
21. Yao, A.C.-C.: How to generate and exchange secrets. In: 27th Annual Symposium on Foundations of Computer Science, 1986, pp. 162–167. IEEE (1986)