Mon$\mathbb{Z}_{2^k}$a: Fast Maliciously Secure Two Party Computation on $\mathbb{Z}_{2^k}$

Dario Catalano[1], Mario Di Raimondo[1], Dario Fiore[2], and Irene Giacomelli[3]

[1] Dipartimento di Matematica e Informatica, Università di Catania, Italy
[2] IMDEA Software Institute, Madrid, Spain
[3] Protocol Labs, USA

Abstract. In this paper we present a new 2-party protocol for secure computation over rings of the form $\mathbb{Z}_{2^k}$. As many recent efficient MPC protocols supporting dishonest majority, our protocol consists of a heavier (input-independent) pre-processing phase and a very efficient online stage. Our offline phase is similar to BeDOZa (Bendlin *et al.* Eurocrypt 2011) but employs Joye-Libert (JL, Eurocrypt 2013) as underlying homomorphic cryptosystem and, notably, it can be proven secure without resorting to the expensive sacrifice step. JL turns out to be particularly well suited for the ring setting as it naturally supports $\mathbb{Z}_{2^k}$ as underlying message space. Moreover, it enjoys several additional properties (such as valid ciphertext-verifiability and efficiency) that make it a very good fit for MPC in general. As a main technical contribution we show how to take advantage of all these properties (and of more properties that we introduce in this work, such as a ZK proof of correct multiplication) in order to design a two-party protocol that is efficient, fast and easy to implement in practice.
Our solution is particularly well suited for relatively large choices of k (*e.g.* $k = 128$), but compares favorably with the state of the art solution of SPD$\mathbb{Z}_{2^k}$ (Cramer *et al.* Crypto 2018) already for the practically very relevant case of $\mathbb{Z}_{2^{64}}$.

1 Introduction

Secure Multi-Party Computation (MPC) allows a set of mutually mistrusting parties to jointly compute a function f of their inputs $x_1, \ldots x_n$ in such a way that correctness and security are guaranteed. Correctness means that at the end of the protocol the parties have computed $f(x_1, \ldots, x_n)$. Security means that, at the end of the interaction, party P_i, holding x_i, learns only (the i-th component of) the output $f(x_1, \ldots, x_n)$ and nothing else. The interesting feature of MPC is that security should be preserved even when there is an adversary $\mathcal{A}$ that controls some of the participants and, for the case of malicious security, takes full control of the corrupted parties, influencing their behaviors in arbitrary ways. The security model for MPC (*e.g.*, the Universal Composability framework [7]) formalizes this by stating that a protocol should be considered secure if its execution is essentially equivalent to an ideal protocol where the computation is performed by a fully trusted third party.

In terms of applications, a particularly relevant case is the two party setting or, more in general, the case where the adversary (maliciously) controls half or more users. This scenario is notoriously hard to handle efficiently. Indeed, it is well-known that fast information theoretic solutions are not possible and expensive public key cryptography needs to be employed to achieve security.

In recent years, several works (*e.g.* [4,13]) noticed that one can improve efficiency by dividing the computation in two stages: an expensive *offline* stage where public key cryptography is used in order to perform a pre-computation independent of the inputs, and an *online* stage in which, once the inputs become available, one performs the actual computation in a fast way, using only information theoretic techniques. More in detail, in these works pre-computation essentially consists in creating random triples of the form (a, b, ab). There are two main approaches to create these triples: using fast, but bandwidth inefficient, oblivious transfer extensions (*e.g.* [18]), or using more compact, but less computationally efficient, homomorphic encryption schemes.

When it comes to achieve security against malicious adversaries, the main technique used by these protocols are unconditionally secure MACs. For instance, in the celebrated SPDZ protocol [13,11] a MAC key is shared and used to authenticate the random triples generated in the offline phase; this prevents players from cheating when using this same material in the on-line phase. Since information theoretically secure MACs are typically constructed over finite fields, most existing solutions for dishonest majority MPC assume that the computation takes the form of an arithmetic circuit over a finite field (such as $\mathbb{Z}_p$ for prime p). An exception is the recent work of Cramer *et al.* [9] (SPD$\mathbb{Z}_{2^k}$) that proposes an efficient protocol that supports operations modulo 2^k. This choice comes particularly handy in practice: for instance, working modulo 2^k (and specifically 2^{64}) closely matches modern CPU computations and allows protocol designers to directly apply optimizations and tricks that are possible there and that are often expensive to emulate modulo p. In order to handle operations in $\mathbb{Z}_{2^k}$, the key technical contribution of the Cramer *et al.* solution is a new information theoretic MAC that allows to authenticate messages in this ring. In a nutshell, they achieve this by choosing a random secret key in a sufficiently large space $\mathbb{Z}_{2^s}$ and by performing all the computations in the larger ring $\mathbb{Z}_{2^{k+s}}$ so as to be able to bound with 2^{-s} the probability that an information theoretic adversary can forge a valid MAC. The new MAC is then used to construct an online protocol a-la SPDZ where computation is done in the ring $\mathbb{Z}_{2^{k+s}}$ (*i.e.* the values and the MACs are additively secret-shared in $\mathbb{Z}_{2^{k+s}}$). The preprocessing stage, on the other hand, is implemented via a MASCOT-like [18] protocol, whose communication costs are roughly twice those of the original MASCOT.

Our Contribution. In this paper we propose MonZa[4], a fast, two-party protocol for secure computation over the ring $\mathbb{Z}_{2^k}$. Our solution uses the authentication mechanism of [9], but we generate random triples using homomorphic encryption. Specifically, we use the Joye-Libert [17,5] additively homomorphic

[4] The name MonZa is inspired by the famous race track hosting the Formula One Italian Grand Prix.

cryptosystem (JL from now on), that turns out to be very well suited for our setting as it naturally supports $\mathbb{Z}_{2^n}$ (for flexible choices of n) as underlying message space. This scheme is efficient both in terms of encryption/decryption costs and in terms of bandwidth consumption (much more efficient than Paillier, for instance). More crucially, the JL cryptosystem has three additional properties that make it a perfect fit for multiparty computation. First, in JL all valid ciphertexts are publicly and efficiently recognizable. Second, JL has circuit privacy (for linear functionalities) in a very natural way. Third, one can generate different instances of JL that share the same plaintext space. The first two properties are particularly useful as they allow us to *avoid the use of expensive zero-knowledge proofs* for proving ciphertexts validity; this is in contrast to solutions based on lattice-based schemes where ciphertexts validity and circuit privacy require cumbersome techniques (related to preventing the injection of "bad noise" by a dishonest party). Moreover, since the scheme naturally works over $\mathbb{Z}_{2^n}$ we also do not need zero-knowledge proofs to show that a plaintext lies in a certain range (this would be needed if using Paillier, for instance).

In this paper we show how to take advantage of all the aforementioned properties of the JL cryptosystem (and even more properties that we add in this work – see slightly below) in order to design an efficient 2PC protocol for computations over the ring $\mathbb{Z}_{2^k}$.

We fully implemented MonZa's off-line phase[5] and performed a collection of experiments in order to evaluate, in terms of both bandwidth and computation, the efficiency of our solution. Details are given in Section 5. Notably, our bandwidth analysis shows that MonZa is particularly convenient for relatively large choices of k (*e.g.* $k = 64$ or 128) in which case it compares favorably with the state of the art solution of SPD$\mathbb{Z}_{2^k}$ [9]. The benchmarks confirm the practical efficiency of our protocol.

An Overview of Our Techniques. In order to design an efficient (preprocessing) 2PC protocol based on JL we cannot simply plug it as "yet another additively-homomorphic encryption" in existing approaches.

If we consider SPDZ [13], one could in principle enhance JL to support one homomorphic multiplication using the transformation of [8]; SPDZ however requires parties to threshold-decrypt ciphertexts at the end of preprocessing, and one drawback of JL is that it misses an efficient threshold decryption protocol[6].

Another option is to plug JL into a BeDOZa-style protocol [4]. In addition to the fact that BeDOZa works over a finite field while in our case we work in a ring with non-invertible elements, a major challenge is that in BeDOZa each party must execute a ZK protocol for correct multiplication, and such a protocol is *not* available for JL. Moreover, due to the fact that not all elements of the ring are invertible, one cannot use classical Sigma-protocol techniques to get it.

[5] We only focuses on the preprocessing stage since the online one is identical to [9].

[6] Also, coming up with an efficient, constant-round, protocol for a threshold JL decryption seems far from trivial due to the bit-by-bit extraction technique in the algorithm.

Finally, if one is concerned with avoiding proofs of correct multiplication, the recent Overdrive protocol [19] (still working over finite fields) showed how to avoid them if the linearly homomorphic encryption scheme satisfies a stronger security notion called *enhanced CPA*. Very informally, this property states that non-linear operations on ciphertexts are not possible. Somewhat surprisingly, this route turns out to not be viable in the setting of $\mathbb{Z}_{2^n}$. We formally prove that no encryption scheme that is linearly homomorphic over plaintext space $\mathbb{Z}_{2^n}$ can achieve enhanced CPA security. This essentially tells us that, in the $\mathbb{Z}_{2^n}$ setting, proofs of correct multiplication are sort of unavoidable.

Our (preprocessing) protocol shares some similarities with both BeDOZa [4] and Overdrive [19] in the sense that it employs an asymmetric Gilboa-like [15] multiplication protocol: P_1 has a key pair $(\mathsf{sk}, \mathsf{pk})$ and P_2 has the public key pk. To multiply their shares a_1 and b_2 the parties perform the following simple protocol. P_1 sends $\mathsf{Enc}_{\mathsf{pk}}(a_1)$ to P_2. P_2 chooses a random $r \in \mathbb{Z}_{2^n}$ and sends $C = \mathsf{Enc}_{\mathsf{pk}}(a_1)^{b_2}\mathsf{Enc}_{\mathsf{pk}}(-r) = \mathsf{Enc}_{\mathsf{pk}}(a_1b_2 - r)$ back to P_1. P_1 decrypts the received plaintext and sets it as its share of the product a_1b_2. P_2's share is just r. Notice that both BeDoZa and Overdrive use this protocol in a symmetric way: each player has a different key pair and to compute the shares of the product of secret-shared values in the two-party setting the protocol is executed two times (once for each mixed product). On the other hand, the design of the offline phase of our MonZa protocol is *asymmetric*: we require only one key pair and one party computes the intermediate ciphertexts of the form of C for both mixed products, while the other party decrypts. Since generating a ciphertext C is much less expensive than decrypting it (in JL), our MonZa protocol is well-suited for applications in the server-client model, where one party has less computational power than the other one.

Making the basic multiplication protocol described before secure against malicious adversaries requires more work though. Intuitively, P_2 has to show that he performed the above operation correctly. In principle this can be done with a ZK proof protocol where P_2 sends a commitment $\mathsf{Com}(a_2)$ and convinces P_1 that C satisfies the multiplicative relation $C = \mathsf{Enc}_{\mathsf{pk}}(a_1)^{a_2}\mathsf{Enc}_{\mathsf{pk}}(-r)$. A difficulty arises from the fact that doing this with JL is tricky. Solving these challenges is one of the main technical contributions of this paper.

To illustrate the problem let us consider the simpler case of proving knowledge of a JL plaintext. Informally, JL can be seen as a generalization of the well known Goldwasser-Micali cryptosystem [16]. The message space is $\mathcal{M} = \mathbb{Z}_{2^n}$, and the public key is N, g, where $N = pq$ is the product of two primes $p = 2^n p' + 1$ and $q = 2q' + 1$ such that p', q' are also primes[7], and g is an element of maximal order in $\mathbb{Z}_N^*$ and whose Jacobi symbol is 1. To encrypt $m \in \mathcal{M}$ one chooses a random $x \in \mathbb{Z}_N^*$ and sets $C = g^m x^{2^n} \bmod N$. To prove knowledge of m one would be tempted to use (an adapted version of) a standard, Schnorr-like, three move protocol. Very roughly this would go as follows. The prover starts by sending the encryption R of a random message r and, upon receiving a challenge $e \in \{0,1\}^n$,

[7] We remark that the original scheme from [17] allows more flexibility in the choice of p and q. For the sake of this discussion the choices above are good enough.

it sends z, y such that $g^z y^{2^n} = RC^e \bmod N$. Completeness and (honest) verifier ZK are easy to argue, but the problems are in proving (special) soundness. Indeed two accepting transcripts (for the same R) lead to an equation of the form $g^{z_1 - z_2} \hat{y}^{2^n} = C^{e_1 - e_2} \bmod N$ from which we *cannot* always extract the message since $e_1 - e_2$ might well be non invertible in $\mathbb{Z}_{2^n}$.

We overcome this issue by defining a slightly different protocol and by doing a careful analysis which shows that one can actually extract the least $n - s$ significant bits of the plaintext encrypted in C. More importantly, we extend this technique to work in the more involved case of proving a multiplication relation. Precisely, we propose an HVZK sigma-protocol for proving knowledge of $b, r \bmod 2^{n-s}$ such that $C = A^b \mathsf{Enc}_{\mathsf{pk}_1}(r)$ and $B = \mathsf{Enc}_{\mathsf{pk}_2}(b)$, where pk_1 pk_2 are public keys of two different JL instances with the same message space $\mathbb{Z}_{2^n}$. Our protocol for correct multiplication is quite efficient – the prover sends 7 elements of $\mathbb{Z}_N^*$ and 2 values of n bits each – and this is partly due to the fact that JL allows to naturally create two instantiations with the same message space (this is for example not possible with Paillier's encryption scheme). In order to cope with the limitation of extracting fewer bits in our applications, we show that we can instantiate JL with a larger message space $\mathbb{Z}_{2^{k+2s}}$ while keeping the shares of our triples over $\mathbb{Z}_{2^{k+s}}$.

As additional remark, we point out that our MonZa protocol departs from previous work [4,13] also in the fact that it does not resort to the expensive sacrifice step to guarantee security. Informally, many existing protocols check the validity of each produced triple by "sacrificing" another triple where the same multiplication relation is expected to hold. This techniques makes the resulting protocols less efficient than one would like them to be as one needs to generate twice as many triples than needed. By exploiting both the algebraic properties of JL and the fact that our protocol is specifically tailored to the two party setting, we manage to replace the sacrifice step with a simplified (and more efficient) version of the HVZK sigma-protocol discussed above.

Other Related Work. There are several works about MPC protocol based on secret-sharing, however only few of these focus on computation over the rings [10]. For the ring $\mathbb{Z}_{2^k}$ besides the SPD$\mathbb{Z}_{2^k}$ protocol mentioned above, Sharemind [6] is a well-known and efficient protocol based on replicated secret-sharing. Sharemind works in the 3-party setting with honest-majority and it is passively secure only. Recently, Araki *et al.* [2] improved the efficiency of Sharemind, while [1,14] extended it to the case of active corruption. However, all these works are restricted to the case of honest majority. Damgaard *et al.* [12] present a compiler for achieving active security starting from a passively-secure MPC protocol that can be used for ring-MPC protocols too. The compiler is perfectly secure, however the active security comes at the price of reducing the corruption threshold (from t corrupted players to approximately $\sqrt{t}$).

In a concurrent and independent work, Orsini *et al.* [20] proposed a protocol, Overdrive2k, to perform secure MPC over $\mathbb{Z}_{2^k}$ from somewhat homomorphic encryption. Similarly to ours, their solution improves SPD$\mathbb{Z}_{2^k}$ in terms of bandwidth consumption. In terms of techniques, Overdrive2k and MonZa are rather different. At the heart of Overdrive2k is a new packing technique for the BGV cryptosystem that works for $\mathbb{Z}_{2^k}$; also their protocol works in the general multiparty setting (*i.e.*, the number of participants is ≥ 2). Our solution, on the other hand, is tailored to the two-party setting and builds on new zero-knowledge techniques for the JL cryptosystem, and the overall protocol is arguably mathematically simpler.

Road Map. We start describing the notation, the cryptography primitives and the security model used in this paper in Section 2. In particular, Section 2.5 recalls the information theoretic MAC defined in SPD$\mathbb{Z}_{2^k}$ and also used by MonZa. Then, our MPC protocol is described in the following two sections: Section 3 describes the new offline phase that we design for MonZa (protocol Π_{Offline}), while the online phase, which follows the SPD$\mathbb{Z}_{2^k}$ blueprint, is described in the full version of this paper. Section 4 recalls the JL encryption scheme and presents the new proof of correct multiplication for this encryption scheme (protocol Π_{ZKPoCM}). Finally, we conclude with an analysis of the efficiency of Π_{Offline} and Π_{ZKPoCM} in Section 5.

2 Preliminaries

2.1 Notation

Given a finite set D, sampling a uniformly random element from D is denoted by $r \leftarrow D$. We denote by $\mathbb{Z}_M$ the ring of the integers modulo M (where $M \geq 2$). We say that a function ϵ is negligible in n if for every positive polynomial p there exists a constant c such that $\epsilon(n) < \frac{1}{p(n)}$ when $n > c$. Two families $X = \{X_n\}_{n\in\mathbb{N}}$ and $Y = \{Y_n\}_{n\in\mathbb{N}}$ of random variables are said to be *statistically indistinguishable*, denoted by $X \approx_s Y$, if it holds that $\sum_a \mid \Pr[X_n = a] - \Pr[Y_n = a] \mid$ is negligible in n. Two ensembles are said to be *computationally indistinguishable*, denoted by $X \approx_c Y$, if it holds that for any computationally bounded (non-uniform probabilistic polynomial-time (PPT)) distinguisher D $\mid \Pr[D(X_n) = 1] - \Pr[D(Y_n) = 1] \mid$ is negligible in n.

2.2 Linearly-Homomorphic Encryption for Messages in $\mathbb{Z}_{2^n}$

To design our protocols, we use a public-key encryption scheme whose message space is the ring $\mathbb{Z}_{2^n}$ and it has a linear homomomorphic property. More precisely, we assume that there exists a triple of algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ with the following property:

Algorithms: $\mathsf{Gen}(1^\lambda, n)$ is a randomized procedure that takes as input the security parameter λ and the message bit-length n, and outputs a matching pair of secret and public keys $(\mathsf{sk}, \mathsf{pk})$. The public key defines a ciphertext space $\mathcal{C}$.
Enc is a randomized algorithm keyed by pk that takes as input values in $\mathbb{Z}_{2^n}$. We write $\mathsf{Enc}_{\mathsf{pk}}(m, r)$ when we want to explicitly indicate that r is the random value used in the procedure, otherwise we write $\mathsf{Enc}_{\mathsf{pk}}(m)$.

Dec is a deterministic function keyed by sk. It holds that for any $m \in \mathbb{Z}_{2^n}$, $\Pr[\mathsf{Dec}_{\mathsf{sk}}(\mathsf{Enc}_{\mathsf{pk}}(m)) = m] = 1$ (the probability is taken over the random coins of Gen and Enc).

Additive property: Let $\mathcal{C}$ be the set of all possible ciphertexts, then there exists an operation $\odot$ on $\mathcal{C}$ such that for any a-tuple of ciphertexts $c_1 \leftarrow \mathsf{Enc}_{\mathsf{pk}}(m_1), \ldots, c_a \leftarrow \mathsf{Enc}_{\mathsf{pk}}(m_a)$ (a positive integer), it holds that $\Pr[\mathsf{Dec}_{\mathsf{sk}}(c_1 \odot \cdots \odot c_a) = m_1 + \cdots + m_a \mod 2^n] = 1$. We will use the notation $c^{\odot a} = c \odot \cdots \odot c$ (a times).

Lossy keys[8]: We also require the existence of a modified key generation algorithm, $\widetilde{\mathsf{Gen}}$, that on the same input λ, n generates a public key $\widetilde{\mathsf{pk}}$ with the following property. For any $m \in \mathbb{Z}_{2^n}$, $\{\mathsf{Enc}_{\widetilde{\mathsf{pk}}}(m)\}_\lambda \approx_s \{\mathsf{Enc}_{\widetilde{\mathsf{pk}}}(0)\}_\lambda$ (*i.e.*, $\mathsf{Enc}_{\widetilde{\mathsf{pk}}}(m)$ is statistically indistinguishable from an encryption of zero). Moreover, public keys produced by $\widetilde{\mathsf{Gen}}$ (called *lossy keys*) are computationally indistinguishable from those produced by the standard key generation algorithm.

Notice that *semantic security* follows from the indistinguishability of keys and the indistinguishability of encryption under the lossy keys.

Circuit privacy for linear functions: Informally, this property states that ciphertexts obtained through homomorphic evaluations are statistically indistinguishable from fresh encryptions of the resulting message. For simplicity, in our work we assume that homomorphic operations (i.e., $\odot$) are deterministic, and we state circuit privacy slightly differently: for any $a, b \in \mathbb{Z}_{2^n}$ and any ciphertext $A \in \mathsf{Enc}_{\mathsf{pk}}(a), B \in \mathsf{Enc}_{\mathsf{pk}}(b)$ we have that $A \odot B \odot \mathsf{Enc}_{\mathsf{pk}}(0) \approx_s \mathsf{Enc}_{\mathsf{pk}}(a + b)$. An implication of this property (that we use in our protocols) is that for any plaintexts $\alpha, \beta, \gamma \in \mathbb{Z}_{2^n}$ and any $C \in \mathsf{Enc}_{\mathsf{pk}}(\gamma)$, it holds $C^{\odot\alpha} \odot \mathsf{Enc}_{\mathsf{pk}}(\beta) \approx_s \mathsf{Enc}_{\mathsf{pk}}(\alpha\gamma + \beta)$.

Publicly Checkable Ciphertexts: we require that membership of a ciphertext in the ciphertext space, *i.e.*, $C \in \mathcal{C}$, can be efficiently and publicly tested given only the public key.

2.3 Commitments

Another building block we use in our constructions is an extractable commitment scheme for messages in $\mathbb{Z}_{2^n}$. That is, in the following we assume that there exists a tuple of algorithms $(\mathsf{cGen}, \mathsf{Com})$ with the following properties:

Algorithms: The procedure $\mathsf{cGen}(1^\lambda, n)$ takes as input the security parameter λ and the message bit-length n. The output is the commitment key ck and the extraction trapdoor information t_X.

Com is a randomized algorithm keyed by ck that takes as input values in $\mathbb{Z}_{2^n}$. We write $\mathsf{Com}_{\mathsf{ck}}(m, r)$ when we want to explicitly indicate that r is the random value used in the procedure, otherwise we write $\mathsf{Com}_{\mathsf{ck}}(m)$.

[8] For a CPA-secure additive encryption scheme this property always holds: include $C = \mathsf{Enc}_{\mathsf{pk}}(b)$ in the public key with $b = 0$ for Gen and $b = 1$ for $\widetilde{\mathsf{Gen}}$, and redefine encryption as $\mathsf{Enc}_{\mathsf{pk}}(m) = C^{\odot m} \odot \mathsf{Enc}_{\mathsf{pk}}(0)$.

Computationally hiding and unconditionally binding: We require that (1) for any $m, m' \in \mathbb{Z}_{2^n}$, $\mathsf{Com}_{\mathsf{ck}}(m) \approx_c \mathsf{Com}_{\mathsf{ck}}(m')$, and (2) for any C in the commitment space there exists at most one pair (m, r) such that it holds that $C = \mathsf{Com}_{\mathsf{ck}}(m, r)$.

Extractability: Finally, we require the existence of a PPT algorithm that allows to compute m from a commitment $C = \mathsf{Com}_{\mathsf{ck}}(m, r)$ and the trapdoor t_X.

Finally, we also require the existence of lossy keys for the commitment scheme too. That is, there exists a modified key-generation algorithm $\widetilde{\mathsf{cGen}}$ that generates lossy commitment keys (*i.e.*, any $\mathsf{Com}_{\widetilde{\mathsf{ck}}}(m)$, where $\widetilde{\mathsf{ck}} \leftarrow \widetilde{\mathsf{cGen}}$, is statistically indistinguishable from a commitment to zero) that are computationally indistinguishable from those produced by the standard key generation algorithm. From the above description it is rather clear that such a commitment scheme can be instantiated using a public key encryption scheme with the lossy key property. Indeed, in Section 4 we show that the Joye-Libert encryption scheme [17,5] satisfies the definition of additive encryption scheme given in Section 2.2 and can be used to instantiate the commitment scheme with the properties required here.

In the following we will use the notation $\mathsf{Enc}_{\mathsf{pk}}(m)$ (or $\mathsf{Com}_{\mathsf{ck}}(m)$) for a message $m \in \mathbb{Z}_{2^{n'}}$ also when the encryption (or commitment) scheme has message space $\mathbb{Z}_{2^n}$ (with $n \geq n'$). Indeed, we can think $\mathbb{Z}_{2^{n'}}$ as a subset of $\mathbb{Z}_{2^n}$.

2.4 Security Model

The protocols presented in this paper are for two parties, P_1 and P_2, and they are proven secure in the Universal Composability (UC) model [7]. In particular, our protocols will be proven secure against a malicious static adversary. In other words, the adversary may deviate from the protocol in any arbitrary way and can only corrupt parties before the protocol execution starts. Since it is not possible to construct an UC-secure MPC protocol with dishonest majority without a set-up assumption, in this paper we rely on the registered public-key model [3]. In particular, we assume that there is a functionality $\mathcal{F}_{\mathsf{KeyGen}}$ (described in Figure 1) that generates correct keys for both the additive encryption scheme and the mixed commitment scheme.

Finally, for the sake of simpler protocol description, we will use a standard coin tossing functionality $\mathcal{F}_{\mathsf{Rand}}$ to generate public randomness. When activated from all the parties with input (rand, u), the functionality $\mathcal{F}_{\mathsf{Rand}}$ samples $r \leftarrow \{0, 1\}^u$ and return it to all parties. $\mathcal{F}_{\mathsf{Rand}}$ can be implemented using commitments of random values in the random oracle model or additive encryption in the key-register model.

2.5 Value-Representation in SPD$\mathbb{Z}_{2^k}$

The SPD$\mathbb{Z}_{2^k}$ protocol [9] is an n-party MPC protocol in the preprocessing model for computation over a ring. The backbone of this protocol is the representation of values: each element is authenticated via an information-theoretic MAC and

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

Let $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be additive encryption scheme and $(\mathsf{cGen}, \mathsf{Com})$ an extractable commitment scheme. $\mathcal{F}_{\mathsf{KeyGen}}$ interacts with the parties P_1 and P_2 and the simulator $\mathcal{S}$, which can abort any time, and proceeds as follows.

- When activated on input $(\mathsf{EncKeys}, sid, \lambda, n)$ from both parties: if P_1 is honest, $\mathcal{F}_{\mathsf{KeyGen}}$ runs $\mathsf{Gen}(1^\lambda, n)$ and generates a key-pair $(\mathsf{sk}, \mathsf{pk})$; if P_1 is corrupted the functionality gets r^* from $\mathcal{S}$ and it runs $\mathsf{Gen}(1^\lambda, n)$ using r^* as random tape. The functionality sends $(\mathsf{sk}, \mathsf{pk})$ to P_1 and pk to P_2.
- When activated on input $(\mathsf{ComKey}, sid, \lambda, n)$ from both parties: if P_2 is honest, $\mathcal{F}_{\mathsf{KeyGen}}$ runs $\mathsf{cGen}(1^\lambda, n)$ and generates a commitment key ck; if P_2 is corrupted the functionality gets r' from $\mathcal{S}$ and it runs $\mathsf{cGen}(1^\lambda, n)$ using r' as random tape. The functionality sends ck to both parties.

Fig. 1. Functionality for the keys generation.

both the value and the MAC are secret-shared among the parties. In this section we recall the details of the SPD$\mathbb{Z}_{2^k}$ value-representation because our 2-party protocol will use it.

The MAC scheme has two parameter: k, where $\mathbb{Z}_{2^k}$ is the ring in which the inputs lie, and the security parameter s. The MAC key[9] α is sampled uniformly at random from $\mathbb{Z}_{2^{k+s}}$ and the MAC of a value $x \in \mathbb{Z}_{2^k}$ is defined as

$$m(x) = \alpha \cdot \tilde{x} \mod 2^{k+s}$$

where $\tilde{x} \in \mathbb{Z}_{2^{k+s}}$ such that $x = \tilde{x} \mod 2^k$. Then the values $\tilde{x}$ and $m(x)$ are additively secret-shared among the parties. The key α is fixed and also additively shared (*i.e.* $\alpha = \sum_{i=1}^n \alpha^{(i)} \mod 2^{k+s}$ and $\alpha^{(i)} \in \mathbb{Z}_{2^{k+s}}$ held by player P_i). In other words, the $[\cdot]$-representation of a value $x \in \mathbb{Z}_{2^k}$ is given by:

$$[x] = \{(x^{(i)}, m^{(i)}(x))\}_{i=1,\dots,n} \text{ and } \sum_{i=1}^n m^{(i)}(x) = (\sum_{i=1}^n x^{(i)}) \cdot (\sum_{i=1}^n \alpha^{(i)}) \mod 2^{k+s}$$

where $(x^{(i)}, m^{(i)}(x)) \in (\mathbb{Z}_{2^{k+s}})^2$ is known by player P_i.

Linear operations on shared and authenticated values are possible. In particular, we recall here the procedure `AffineComb` of [9]: the parties have u values $[x_1], \dots, [x_u]$, to compute the representation of $y = c + \sum_{i=1}^u c_i \cdot x_i \mod 2^k$, where $c, c_1, \dots, c_u$ are public values, the parties proceed as follow:

1. Party P_1 sets $y^{(1)} = c + \sum_{i=1}^u c_i \cdot x_i^{(1)} \mod 2^{k+s}$;
2. Each party P_j with $j \neq 1$ sets $y^{(j)} = \sum_{i=1}^u c_i \cdot x_i^{(j)} \mod 2^{k+s}$;
3. Each party P_j sets $m^{(j)}(y) = \alpha^{(j)} \cdot c + \sum_{i=1}^u c_i \cdot m^{(j)}(x_i) \mod 2^{k+s}$;

[9] The last (most significant) k bits of the MAC key are not actually required to be random, since the security of the MPC protocol follows from $\alpha \mod 2^s$ being random. However, sampling α from $\mathbb{Z}_{2^{k+s}}$ simplifies the description of the protocols.

In the following, we will say that parties compute $[y] = c + \sum_{i=1}^{u} c_i \cdot [x_i]$ to indicate that this procedure is executed.

3 Offline Phase

Our 2-party MPC protocol is divided in two phases: an offline phase, which is independent of both the input and the function, and an online phase, where the actual computation takes place. In the offline phase, the parties generate correlated randomness in the form of *singles* and *triples*. Then, in the on-line phase, as in the SPD$\mathbb{Z}_{2^k}$ protocol, these values are consumed to create representation of the inputs, and to multiply shared and authenticated values and to verify the MACs (more details in the full version of this paper).

The exact functionality $\mathcal{F}_{\mathsf{Offline}}$ that is implemented in the offline phase is described in Figure 2. The correlated randomness generated by $\mathcal{F}_{\mathsf{Offline}}$ for honest players has three forms: (1) authenticated single[10] $(j, [r])$, where r is sampled uniformly at random from $\mathbb{Z}_{2^{k+s}}$, and r is expressed in the $[\cdot]$-representation using a trivial sharing: $r^{(j)} = r$ and the other share is zero (*i.e.*, r is known by P_j only), (2) shared and authenticated single $[r]$, where again r is sampled uniformly at random from $\mathbb{Z}_{2^{k+s}}$ and expressed using the $[\cdot]$-representation, but no party knows the value, and (3) shared and authenticated triple $[a], [b], [c]$. Here, a, b, c are all shared and authenticated singles over $\mathbb{Z}_{2^{k+s}}$ such that it holds $c = a \cdot b \mod 2^k$.

The idea behind the specification of the corruption is that the environment is allowed to specify the share of a single for P_i corrupted ($i = 1$ or $i = 2$), and the share of c in a triple and the share of the MACs for P_2 corrupted. Then, the data for the honest party is chosen consistently with the values given by the environment to guarantee correctness of the MAC and the multiplication. Notice that the environment has no power to choose some of the shares of a corrupted P_1 (*i.e.*, the share of c and of the MACs); this is to reflect the different roles that the two parties have in our offline protocol and, in particular in the multiplication sub-protocol (more detail in the following).

The basic building block we use to generate both a single and a triple is a 2-party multiplication protocol (*i.e.*, a protocol to compute an additive sharing of the product of two secret values). Indeed in the 2-party case, and due to the nature of the MACs used in the $[\cdot]$-representation, such multiplication protocol is sufficient for computing both the product of secret-shared values and to authenticate a secret-shared value. Similarly to other MPC protocols like BeDOZa [4] and Overdrive [19], in order to implement the 2-party multiplication protocol we use an additive encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ as defined in Section 2.2. The high-level idea is simple: assume that party P_1 has a pair $(\mathsf{pk}, \mathsf{sk})$ and input $x^{(1)}$, while party P_2 knows only the public key pk and has input $x^{(2)}$. To compute an additive sharing of $x^{(1)} \cdot x^{(2)}$, P_1 sends $C_1 = \mathsf{Enc}_{\mathsf{pk}}(x^{(1)})$

[10] The $[\cdot]$-representation for a value x of $k + s$ bits means that we additively share in $\mathbb{Z}_{2^{k+s}}$ the value x and its MAC $x \cdot \alpha \bmod 2^{k+s}$. However, only the first k bits of x are authenticated.

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

$\mathcal{F}_{\mathsf{Offline}}$ interacts with the parties P_1 and P_2 and the simulator $\mathcal{S}$, which can abort any time, and proceeds as follows.
For the sake of brevity, the description of functionality uses the following macro (*i.e.*, internal subroutine) that is executed to compute an additive secret-sharing of the MAC of secret-shared values respect to a given global key α.

$\mathsf{Auth}(x^{(1)}, x^{(2)})$:

1. Let $x = x^{(1)} + x^{(2)} \bmod 2^{k+s}$ and $m(x) = \alpha \cdot x \bmod 2^{k+s}$;
   If P_2 is corrupted, wait for $m_2 \in \mathbb{Z}_{2^{k+s}}$ from $\mathcal{S}$, otherwise sample $m_2 \leftarrow \mathbb{Z}_{2^{k+s}}$ at random. Define $m_1 = m(x) - m_2 \mod 2^{k+s}$.
2. Send m_1 to P_1, and send m_2 to P_2 if P_2 honest.

Initialize: When activated on the first time on input $(\mathsf{Init}, sid, k, s)$ from all the parties, the functionality stores k and s. Then, for $j = 1, 2$, $\mathcal{F}_{\mathsf{Offline}}$ waits for $\mathcal{S}$ to send $\alpha^{(j)} \in \mathbb{Z}_{2^{k+s}}$ if P_j is corrupted, otherwise $\mathcal{F}_{\mathsf{Offline}}$ samples $\alpha^{(j)} \leftarrow \mathbb{Z}_{2^{k+s}}$ and forwards it to P_j. The functionality stores $\alpha = \alpha^{(1)} + \alpha^{(2)} \bmod 2^{k+s}$.

In each other activation,

Single: On input $(\mathsf{Single}, P_j, sid, ssid)$ from all parties, the functionality does the following.

1. $\mathcal{F}_{\mathsf{Offline}}$ waits for $\mathcal{S}$ to send $r \in \mathbb{Z}_{2^{k+s}}$ if P_j is corrupted, otherwise it samples $r \leftarrow \mathbb{Z}_{2^{k+s}}$ and forwards it to P_j.
2. $\mathcal{F}_{\mathsf{Offline}}$ executes $\mathsf{Auth}(r, 0)$: P_1 gets $m^{(1)}(r)$ and P_2 gets $m^{(2)}(r)$. The values $(ssid, r, m^{(j)}(r))$ and $(ssid, 0, m^{(i)}(r))$ are stored as local share of $(j, [r])$ by P_j and the other player P_i.

On input $(\mathsf{Single}, sid, ssid)$ from all parties, the functionality does the following.

1. For $j = 1, 2$, $\mathcal{F}_{\mathsf{Offline}}$ waits for $\mathcal{S}$ to send $r^{(j)} \in \mathbb{Z}_{2^{k+s}}$ if P_j is corrupted; otherwise it samples $r^{(j)} \leftarrow \mathbb{Z}_{2^{k+s}}$ and forwards it to P_j.
2. $\mathcal{F}_{\mathsf{Offline}}$ executes $\mathsf{Auth}(r^{(1)}, r^{(2)})$: for $j = 1, 2$, P_j gets $m^{(j)}(r)$ and stores $(ssid, r^{(j)}, m^{(j)}(r))$ as its local share of $[r]$.

Triple: On input $(\mathsf{Triple}, sid, ssid)$ from all parties, the functionality does the following.

1. For $j = 1, 2$, $\mathcal{F}_{\mathsf{Offline}}$ waits for $\mathcal{S}$ to send $a^{(j)}, b^{(j)} \in \mathbb{Z}_{2^{k+s}}$ if P_j is corrupted, otherwise $\mathcal{F}_{\mathsf{Offline}}$ samples $a^{(j)}, b^{(j)} \leftarrow \mathbb{Z}_{2^{k+s}}$ and forwards these to P_j. Let $a = a^{(1)} + a^{(2)} \bmod 2^{k+s}$, $b = b^{(1)} + b^{(2)} \bmod 2^{k+s}$ and $c \in \mathbb{Z}_{2^{k+s}}$ such that $c = a \cdot b \bmod 2^{k}$.
2. If P_2 is corrupted, wait for $c^{(2)} \in \mathbb{Z}_{2^{k+s}}$ from $\mathcal{S}$, otherwise sample $c^{(2)} \leftarrow \mathbb{Z}_{2^{k+s}}$ at random. Define $c^{(1)} = c - c^{(2)} \mod 2^{k+s}$. The functionality sends $c^{(1)}$ to P_1, and sends $c^{(2)}$ to P_2 if P_2 honest.
3. $\mathcal{F}_{\mathsf{Offline}}$ executes $\mathsf{Auth}(a^{(1)}, a^{(2)})$, $\mathsf{Auth}(b^{(1)}, b^{(2)})$ and $\mathsf{Auth}(c^{(1)}, c^{(2)})$: for $j = 1, 2$, P_j gets $m^{(j)}(a)$, $m^{(j)}(b)$ and $m^{(j)}(c)$; party P_j stores $(ssid_i, a^{(j)}, m^{(j)}(a))$, $(ssid_i, b^{(j)}, m^{(j)}(b))$ and $(ssid_i, c^{(j)}, m^{(j)}(c))$ as its share of $([a], [b], [c])$.

Fig. 2. Functionality for the offline phase (preprocessing). It generates the shares of the global MAC key, and it produces singles and triples.

to P_2, who samples $y^{(2)}$ uniformly at random from the message space and computes $C = C_1^{\odot x^{(2)}} \odot \mathsf{Enc}_{\mathsf{pk}}(y^{(2)})$. Now, P_2 sends C to P_1, who decrypts and get $y^{(1)} = x^{(1)} \cdot x^{(2)} + y^{(2)}$. Passive security follows easily from the properties of the underlying encryption scheme. To achieve active security, we need to assure that P_1 sends an actual encryption and that P_2 computes C following the instruction in the protocol. The first property is easy to guarantee because we assume that the underlying encryption scheme has a publicly checkable ciphertext space. For the other task, we use a Zero-Knowledge (ZK) proof.

More precisely in the description of protocol Π_{Offline}, we assume the existence of the sub-protocol Π_{ZKPoCM}. This is a 3-move standard Σ-protocol where the functionality $\mathcal{F}_{\mathsf{Rand}}$ (Section 2.4) generates the challenge sent in the second messages for both players. We assume that the keys for an additive encryption scheme and an extractable commitment scheme have been generated correctly by an invocation to $\mathcal{F}_{\mathsf{KeyGen}}$. Both schemes have the same message space $\mathbb{Z}_{2^{k+2s}}$. The prover wants to convince the verifier that a given ciphertext C satisfy a precise relation among a value it knows and another public ciphertext C_1. That is, the common input is two ciphertexts, C and C_1, and a commitment C_2, the private input of the prover is $m, r \in \mathbb{Z}_{2^{k+s}}$ such that $C_2 = \mathsf{Com}_{\mathsf{ck}}(\tilde{m})$ and $C = C_1^{\odot m} \odot \mathsf{Enc}_{\mathsf{pk}}(\tilde{r})$ where $\tilde{m}$ and $\tilde{r}$ are values in the (larger) message space such that $m = \tilde{m} \bmod 2^{k+s}$ and $r = \tilde{r} \bmod 2^{k+s}$. We give more details on this and an instantiation of this sub-protocol in Section 4.1.

Protocol Π_{Offline} is described in Figure 3 and Figure 4. For the sake of brevity, we use the sub-protocol Mult that captures the actively secure multiplication protocol described before (assuming that the ciphertext C_1 and the commitment C_2 were sent previously). Mult is used to compute both the MAC of a given value and the product of shared values. For example, to implement (Single, P_1) (*i.e.*, to authenticate a value r known by P_1), the parties need to compute the shares of the product $r \cdot \alpha^{(2)} \bmod 2^{k+s}$ (where $\alpha^{(2)}$ is P_2's share of the global MAC key). This is done by running the 2-party multiplication protocol Mult where C_1 is an encryption of r done by P_1 and C_2 is a commitment to $\alpha^{(2)}$ (Figure 3).

Analogously, to compute the mixed products for generating a triple (*e.g.*, $a^{(1)} \cdot b^{(2)} \bmod 2^{k+s}$ where $a^{(i)}$, $b^{(j)}$ are shares of singles) the parties execute Mult two times (in the example, C_1 is an encryption of $a^{(1)}$ and C_2 is a commitment to $b^{(2)}$). Finally, the Mult sub-protocol is used again to authenticate the product c (Figure 4). Notice that the sub-protocol Mult does not commit a party to its output, therefore for the triple generation we need to add an extra check. This guarantees that a party uses the correct value (*i.e.* its output from the multiplication step) in the authentication step. Without this check a corrupted party could authenticate a wrong share $\tilde{c}^{(i)}$ and this would create an insecure triple (*i.e.*, a triple where $c = a \cdot b + \Delta \bmod 2^{k+s}$ and $\Delta \neq 0 \bmod 2^k$ known by the corrupted party). We implement the check using again a ZK proof for encrypted/committed values[11]. In particular, we use a modified (simpler) version of Π_{ZKPoCM}. This version, which we call Π_{ZKPoMCV}, allows the prover to convince

[11] In order to use the same ZK-proof for both players we need to assume that the commitment scheme has the same homomorphic property as the encryption scheme.

Protocol Π_{Offline}

The protocol is run by parties P_1 and P_2. $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is an additive encryption scheme and $(\mathsf{cGen}, \mathsf{Com})$ is an extractable commitment scheme as defined in Section 2. For both these schemes, the message space is in $\mathbb{Z}_{2^{k+2s}}$.

In the steps of Π_{Offline} described in the following, we will use multiple times the sub-protocol Mult described below.

$\mathsf{Mult}(x^{(1)}, C_1, x^{(2)}, C_2)$:

Common input: the commitment $C_2 = \mathsf{Com}_{\mathsf{ck}}(x^{(2)})$ and the ciphertext $C_1 = \mathsf{Enc}_{\mathsf{pk}}(x^{(1)})$; Input for P_1: $x^{(1)} \in \mathbb{Z}_{2^{k+s}}$; Input for P_2: $x^{(2)} \in \mathbb{Z}_{2^{k+s}}$.

1. P_2 samples $\tilde{r} \leftarrow \mathbb{Z}_{2^{k+2s}}$, sends $D = C_1^{\odot x^{(2)}} \odot \mathsf{Enc}_{\mathsf{pk}}(\tilde{r})$ to P_1 and invokes Π_{ZKPoCM} playing the role of the prover with private input $(x^{(2)}, \tilde{r} \bmod 2^{k+s})$ and public input (C_1, D, C_2);
2. If Π_{ZKPoCM} doesn't abort, P_1 computes $\tilde{y}^{(1)} = \mathsf{Dec}_{\mathsf{sk}}(D)$

Output: for P_1 the value $y^{(1)} = \tilde{y}^{(1)} \bmod 2^{k+s}$, for P_2 the value $y^{(2)} = -\tilde{r} \bmod 2^{k+s}$. Notice that $y^{(1)} + y^{(2)} = x^{(1)} \cdot x^{(2)} \mod 2^{k+s}$.

Initialize:

1. For $i = 1, 2$, when activated on the first time on input $(\mathsf{Init}, sid, k, s)$, P_i sends $(\mathsf{EncKeys}, sid, \lambda, k + 2s)$ and $(\mathsf{ComKey}, sid, \lambda, k + 2s)$ to $\mathcal{F}_{\mathsf{KeyGen}}$; P_1 gets $\mathsf{sk}, \mathsf{pk}, \mathsf{ck}$ and P_2 gets pk, ck.
2. P_1 samples $\alpha^{(1)} \leftarrow \mathbb{Z}_{2^{k+s}}$ and sends $\Delta_1 = \mathsf{Enc}_{\mathsf{pk}}(\alpha^{(1)})$ to P_2;
3. P_2 samples $\alpha^{(2)} \leftarrow \mathbb{Z}_{2^{k+s}}$ and sends $\Delta_2 = \mathsf{Com}_{\mathsf{ck}}(\alpha^{(2)})$ to P_1.

In each other activation,

Single:

On input $(\mathsf{Single}, P_1, sid, ssid)$, the parties do the following.

1. P_1 samples $r \leftarrow \mathbb{Z}_{2^{k+s}}$, sends $R = \mathsf{Enc}_{\mathsf{pk}}(r)$ to P_2;
2. P_2 invokes $\mathsf{Mult}(r, R, \alpha^{(2)}, \Delta_2)$, P_1 gets $y^{(1)}$ and P_2 gets $y^{(2)}$;
3. P_1 sets $m^{(1)}(r) = \alpha^{(1)} \cdot r + y^{(1)} \mod 2^{k+s}$ and stores $(ssid, r, m^{(1)}(r))$ as its share of $(1, [r])$, P_2 stores $(ssid, 0, y^{(2)})$ as its share of $(1, [r])$.

On input $(\mathsf{Single}, P_2, sid, ssid)$, the parties do the following.

1. P_2 samples $r \leftarrow \mathbb{Z}_{2^{k+s}}$ and sends $R = \mathsf{Com}_{\mathsf{ck}}(r)$ to P_1;
2. P_1 invokes $\mathsf{Mult}(\alpha^{(1)}, \Delta_1, r, R)$, P_1 gets $y^{(1)}$ and P_2 gets $y^{(2)}$;
3. P_2 sets $m^{(2)}(r) = \alpha^{(2)} \cdot r + y^{(2)} \mod 2^{k+s}$ and stores $(ssid, r, m^{(2)}(r))$ as its share of $(2, [r])$, P_1 stores $(ssid, 0, y^{(1)})$ as its share of $(2, [r])$.

On input $(\mathsf{Single}, sid, ssid)$, the parties do the following.

1. Run $(\mathsf{Singles}, P_1)$ and $(\mathsf{Singles}, P_2)$ and generate $(1, [r^{(1)}])$ and $(2, [r^{(2)}])$, respectively;
2. Compute $[r] = [r^{(1)}] + [r^{(2)}]$ and store it with index $ssid$.

Fig. 3. Protocol for preprocessing.

the verifier that, given three ciphertexts (or commitments) $A, B, \tilde{C}$, the prover knows b such that $\tilde{C} = A^{\odot b}$. We give more details on this and an instantiation of this ZK proof in Section 4.2.

Protocol Π_{Offline} (continued)

Triple: On input $(\mathsf{Triple}, sid, ssid)$, the parties do the following.

1. The parties run two times the Single command and get their shares of $[a]$ and $[b]$ (let $A^{(1)} = \mathsf{Enc}_{\mathsf{pk}}(a^{(1)})$, $A^{(2)} = \mathsf{Com}_{\mathsf{ck}}(a^{(2)})$ and $B^{(1)} = \mathsf{Enc}_{\mathsf{pk}}(b^{(1)})$, $B^{(2)} = \mathsf{Com}_{\mathsf{ck}}(b^{(2)})$ be the intermediate values computed during the execution of the Single steps);
2. *Multiplication*:
   - P_2 invokes $\mathsf{Mult}(a^{(1)}, A^{(1)}, b^{(2)}, B^{(2)})$, P_i gets $y^{(i)}$ for $i = 1, 2$;
     (let $D = (A^{(1)})^{\odot b^{(2)}} \odot \mathsf{Enc}_{\mathsf{pk}}(-y^{(2)})$ be the ciphertext computed and sent by P_2 in this Mult and let $R = \mathsf{Com}_{\mathsf{ck}}(-y^{(2)})$ be the commitment computed and send by P_2 during the corresponding Π_{ZKPoCM}, see Section 4.1)
   - P_2 invokes $\mathsf{Mult}(b^{(1)}, B^{(1)}, a^{(2)}, A^{(2)})$, P_i gets $z^{(i)}$ for $i = 1, 2$;
     (let $D' = (B^{(1)})^{\odot a^{(2)}} \odot \mathsf{Enc}_{\mathsf{pk}}(-z^{(2)})$ be the ciphertext computed and sent by P_2 in this Mult and let $R' = \mathsf{Com}_{\mathsf{ck}}(-z^{(2)})$ be the commitment computed and send by P_2 during the corresponding Π_{ZKPoCM}, see Section 4.1)
   - For $j = 1, 2$, P_j sets $c^{(j)} = a^{(j)} \cdot b^{(j)} + y^{(j)} + z^{(j)} \mod 2^{k+s}$.
3. *Authentication*:
   - P_1 computes $\tilde{C}^{(1)} = \mathsf{Enc}_{\mathsf{pk}}(a^{(1)}b^{(1)})$ and P_2 computes $\tilde{C}^{(2)} = \mathsf{Com}_{\mathsf{ck}}(a^{(2)}b^{(2)})$; for $j = 1, 2$, P_j sends $\tilde{C}^{(j)}$ to the other party and invokes Π_{ZKPoMCV} playing the role of the prover with public input $(A^{(j)}, B^{(j)}, \tilde{C}^{(j)})$. If the ZK proofs do not abort, P_2 computes $C^{(1)} = \tilde{C}^{(1)} \odot D \odot D'$ and P_1 computes $C^{(2)} = \tilde{C}^{(2)} \odot^{-1} R \odot^{-1} R'$;
   - P_2 invokes $\mathsf{Mult}(c^{(1)}, C^{(1)}, \alpha^{(2)}, \Delta_2)$, P_i gets $y^{(i)}$ for $i = 1, 2$;
   - P_2 invokes $\mathsf{Mult}(\alpha^{(1)}, \Delta_1, c^{(2)}, C^{(2)})$, P_i gets $z^{(i)}$ for $i = 1, 2$;
   - For $j = 1, 2$, P_j sets $m^{(j)}(c) = c^{(j)} \cdot \alpha^{(j)} + y^{(j)} + z^{(j)} \mod 2^{k+s}$ and store $(c^{(j)}, m^{(j)}(c))$ as its share of $[c]$ $(c = a \cdot b \bmod 2^{k+s})$;

Fig. 4. Triple generation in the preprocessing.

Theorem 1. *Assume that the underlying encryption scheme and commitment scheme satisfy the definitions in Section 2. Then, protocol* Π_{Offline} *implements* $\mathcal{F}_{\mathsf{Offline}}$ *with computational security against any static active adversary in the* $(\mathcal{F}_{\mathsf{KeyGen}}, \mathcal{F}_{\mathsf{Rand}})$*-hybrid model.*

Proof. We use the variant of the UC model where the environment $\mathcal{Z}$ plays the role of both the distinguisher and the adversary. The environment always

If the commitment scheme is instantiated using the encryption as observed in Section 2.3, the homomorphic property clearly holds.

chooses the input for the honest player and gets its output when the execution is done. Moreover, in the protocol execution $\mathcal{Z}$ corrupts P_i ($i = 1$ or $i = 2$) and takes control of its actions (*i.e.* $\mathcal{Z}$ decides the messages sent by P_i and reads the message received by this party). We argue about UC security, defining a simulator $\mathcal{S}_i$ that interacts with $\mathcal{Z}$ and the functionality $\mathcal{F}_{\mathsf{Offline}}$ and simulates the view of $\mathcal{Z}$ when attacking the protocol execution. The simulator $\mathcal{S}_i$ has the power of choosing the input that P_i sends to $\mathcal{F}_{\mathsf{Offline}}$ and getting its output. In Figure 5 and Figure 6 we define $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively. The simulator $\mathcal{S}_i$ behaves as an honest party P_{3-i} running the protocol with the environment $\mathcal{Z}$ controlling the corrupted party. Here we show that a poly-time environment $\mathcal{Z}$ can not distinguish between the real view (*i.e.*, the view in the execution of the protocol) and an ideal view (*i.e.* the view in the interaction with the simulator).

Case $i = 1$ (P_1 is corrupted). We will argue now that the existence of a poly-time environment $\mathcal{Z}$ that distinguishes a real-view from an ideal one contradicts the key-indistinguishability property of the underlying commitment scheme. More in details, assume that there exists a $\mathcal{Z}$ that can distinguish between a real view and an ideal one with significant probability ϵ. We construct a distinguisher D that given a commitment key ck^* produces a *view* of the same form as what $\mathcal{Z}$ sees and with the following property: D uniformly chooses a bit b, if ck^* is a standard key, then *view* is an ideal-view when $b = 1$ and *view* is an real-view when $b = 0$; if ck^* is lossy, then *view* generated when $b = 0$ and *view* generated when $b = 1$ are statistically indistinguishable. The *view* produced by D is given to $\mathcal{Z}$ that outputs a bit b' (*i.e.*, $b' = 0$ means protocol execution and $b' = 1$ means simulated execution); if $b' = b$, D outputs "standard key", otherwise it outputs "lossy key". It is easy to see that D wins with probability close to $\epsilon/2$. We define D as follow.

On input ck^*, D generates $(\mathsf{sk}, \mathsf{pk})$ using Gen, initializes a local copy of $\mathcal{Z}$, sends ck^* and $(\mathsf{sk}, \mathsf{pk})$ to $\mathcal{Z}$ and starts executing the protocol Π_{Offline} where $\mathcal{Z}$ controls party P_1 and D plays P_2. The distinguisher D samples a bit $b \leftarrow \{0, 1\}$. If $b = 1$, D plays P_2 running the same instruction written for the simulator $\mathcal{S}_1$. D completes *view* choosing the outputs for P_2 as $\mathcal{F}_{\mathsf{Offline}}$ would do. If $b = 0$, D follows the instructions for an honest P_2 in the protocol Π_{Offline} and P_2's outputs in *view* are the values used in this execution. By construction, if ck^* is a standard key, then the view produced by D corresponds to a real-view if $b = 0$, and to an ideal-view if $b = 1$. On the other hand, if ck^* is a lossy key, then in any view each commitment is statistically indistinguishable from a commitment to zero and the messages produced as prover in Π_{ZKPoCM} are statistically indistinguishable because they can be simulated by the ZK simulator (unconditional zero-knowledge property, special case of Theorem 3). The same holds for Π_{ZKPoMCV} (the messages produced as prover in Π_{ZKPoMCV} are statistically indistinguishable because of the unconditional zero-knowledge property, refer to Section 4.2). Moreover, in any view each ciphertext of the form $C = C_1^{\odot b} \odot \mathsf{Enc}_{\mathsf{pk}}(r)$ is statistically indistinguishable to a fresh encryption of a random message (circuit privacy). Therefore the view produced by D when $b = 0$ is statistically close to the one produced when $b = 1$.

The simulator $\mathcal{S}_1$ is defined by the following instructions:

- Simulating the initialize command:
  1. Simulation of the call to $\mathcal{F}_{\mathsf{KeyGen}}$: $\mathcal{S}_1$ runs $\mathsf{Gen}(1^\lambda, k+2s)$, $\mathsf{cGen}(1^\lambda, k+2s)$ and gets $(\mathsf{sk}, \mathsf{pk})$ and ck. $\mathcal{S}_1$ sends $\mathsf{pk}, \mathsf{sk}, \mathsf{ck}$ to $\mathcal{Z}$.
  2. $\mathcal{S}_1$ receives $\varDelta'_1$ from $\mathcal{Z}$, computes $\alpha'^{(1)} = \mathsf{Dec}_{\mathsf{sk}}(\varDelta'_1)$ and sends $(\mathsf{Init}, \alpha'^{(1)})$ to $\mathcal{F}_{\mathsf{Offline}}$.
  3. The simulator behaves as an honest P_2 in the protocol $\varPi_{\mathsf{Offline}}$: $\mathcal{S}_1$ samples $\alpha'^{(2)} \leftarrow \mathbb{Z}_{2^{k+s}}$ and sends $\varDelta'_2 = \mathsf{Com}_{\mathsf{ck}}(\alpha'^{(2)})$ to $\mathcal{Z}$.
- Simulating the (Single, P_1) command:
  1. $\mathcal{S}_1$ receives $R' \in \mathcal{C}$ from $\mathcal{Z}$ and computes $r' = \mathsf{Dec}_{\mathsf{sk}}(R')$.
  2. The simulator behaves as an honest P_2 in the sub-protocol $\mathsf{Mult}(r', R', \alpha'^{(2)}, \varDelta'_2)$: $\mathcal{S}_1$ samples $\tilde{r}' \leftarrow \mathbb{Z}_{2^{k+2s}}$, computes $C' = R'^{\odot \alpha'^{(2)}} \odot \mathsf{Enc}_{\mathsf{pk}}(\tilde{r}')$ and sends C' to $\mathcal{Z}$. Then the simulator behaves as an honest prover in the protocol $\varPi_{\mathsf{ZKPoCM}}$ with private input $(\alpha'^{(2)}, \tilde{r}' \bmod 2^{k+s})$ and common input $C', R', \varDelta'_2$ (the simulator also simulates $\mathcal{F}_{\mathsf{Rand}}$). If there is no abort, the simulator sends r' to $\mathcal{F}_{\mathsf{Offline}}$.
- Simulating the (Single, P_2) command:
  1. The simulator behaves as an honest P_2 in the protocol $\varPi_{\mathsf{Offline}}$: $\mathcal{S}_1$ samples $r' \leftarrow \mathbb{Z}_{2^{k+s}}$ and sends $R' = \mathsf{Com}_{\mathsf{ck}}(r')$ to $\mathcal{Z}$.
  2. The simulator behaves as an honest P_2 in the sub-protocol $\mathsf{Mult}(\alpha'^{(1)}, \varDelta'_1, r', R')$: $\mathcal{S}_1$ samples $\tilde{r}' \leftarrow \mathbb{Z}_{2^{k+2s}}$, computes $C' = \varDelta_1'^{\odot r'} \odot \mathsf{Enc}_{\mathsf{pk}}(\tilde{r}')$ and sends C' to $\mathcal{Z}$. Then the simulator behaves as an honest prover in the protocol $\varPi_{\mathsf{ZKPoCM}}$ with private input $(r', \tilde{r}' \bmod 2^{k+s})$ and common input $C', R', \varDelta'_1$.
- Simulating the Single command:
  1. The same as before in (Single, P_1) to extract $(1, r'^{(1)})$ and emulating an honest P_2 in (Single, P_2) to generate $(2, r'^{(2)})$.
- Simulating the Triple command:
  1. The same as in Single to extract $a'^{(1)}$ and $b'^{(1)}$, and emulating an honest P_2 to generate $a'^{(2)}$ and $b'^{(2)}$.
  2. In any invocation of the sub-protocols Mult and $\varPi_{\mathsf{ZKPoMCV}}$, the simulator behaves as an honest P_2.
  3. If the ZK-proofs do not fail, $\mathcal{S}_1$ sends $a'^{(1)}$ and $b'^{(1)}$ to $\mathcal{F}_{\mathsf{Offline}}$.

Fig. 5. Simulator for a corrupted P_1 in the $\varPi_{\mathsf{Offline}}$ protocol.

The simulator $\mathcal{S}_2$ is defined by the following instructions:

- Simulating the initialize command:
  1. Simulation of the call to $\mathcal{F}_{\mathsf{KeyGen}}$: $\mathcal{S}_2$ runs $\mathsf{Gen}(1^\lambda, k+2s)$, $\mathsf{cGen}(1^\lambda, k+2s)$ and it gets $(\mathsf{sk}, \mathsf{pk})$ and (ck, t_X). $\mathcal{S}_2$ sends pk, ck to $\mathcal{Z}$ and stores the trapdoor t_X.
  2. The simulator behaves as an honest P_1 in the protocol Π_{Offline}: $\mathcal{S}_2$ samples $\alpha'^{(1)} \leftarrow \mathbb{Z}_{2^{k+s}}$, sends $\Delta'_1 = \mathsf{Enc}_{\mathsf{pk}}(\alpha'^{(1)})$ to $\mathcal{Z}$.
  3. $\mathcal{S}_2$ receives Δ'_2 from $\mathcal{Z}$, extracts $\alpha'^{(2)}$ from Δ'_2 using t_X and sends $(\mathsf{Init}, \alpha'^{(2)})$ to $\mathcal{F}_{\mathsf{Offline}}$.
- Simulating the (Single, P_1) command:
  1. The simulator behaves as an honest P_1 in the protocol Π_{Offline}: $\mathcal{S}_2$ samples $r' \leftarrow \mathbb{Z}_{2^{k+s}}$, sends $R' = \mathsf{Enc}_{\mathsf{pk}}(r')$ to $\mathcal{Z}$.
  2. Simulation of the sub-protocol $\mathsf{Mult}(r', R', \alpha'^{(2)}, \Delta'_2)$: $\mathcal{S}_2$ receives C' and behaves as an honest verifier in the protocol Π_{ZKPoCM} on public input (R', Δ'_2, C') ($\mathcal{S}_2$ simulates $\mathcal{F}_{\mathsf{Rand}}$ too). If the proof is accepted, the simulator computes $y'^{(2)} = r' \cdot \alpha'^{(2)} - \mathsf{Dec}_{\mathsf{sk}}(C') \bmod 2^{k+s}$ and sends $(\mathsf{Single}, P_2, y'^{(2)})$ to $\mathcal{F}_{\mathsf{Offline}}$.
- Simulating the (Single, P_2) command:
  1. $\mathcal{S}_2$ receives R' from $\mathcal{Z}$ and extracts r' from R' using t_X;
  2. Simulation of the sub-protocol $\mathsf{Mult}(\alpha'^{(1)}, \Delta'_1, r, R')$: $\mathcal{S}_2$ receives C' and behaves as an honest verifier in the protocol Π_{ZKPoCM} on common input (Δ'_1, R', C'). If the proof is accepted, $\mathcal{S}_2$ computes $y'^{(2)} = r' \cdot \alpha'^{(1)} - \mathsf{Dec}_{\mathsf{sk}}(C') \bmod 2^{k+s}$ and sends $(\mathsf{Single}, P_2, r', y'^{(2)})$ to $\mathcal{F}_{\mathsf{Offline}}$.
- Simulating the Single command:
  1. The same as before in (Single, P_1) and (Single, P_2) to extract $(2, r'^{(2)}, m^{(2)}(r'))$.
- Simulating the Triple command:
  1. The same as in Single to extract $(a'^{(2)}, m^{(2)}(a'))$ and $(b'^{(2)}, m^{(2)}(b'))$. In a similar way, the simulator extracts the environment's shares $(c'^{(2)}, m^{(2)}(c'))$ from the ciphertexts received in the multiplication and the authentication step.
  2. In any invocation of the sub-protocol Π_{ZKPoMCV}, the simulator behaves as an honest P_1.
  3. If the ZK-proofs do not fail, $\mathcal{S}_2$ sends the extracted values to $\mathcal{F}_{\mathsf{Offline}}$.

Fig. 6. Simulator for a corrupted P_2 in the Π_{Offline} protocol.

Case $i = 2$ (P_2 is corrupted). The rationale is the same as in the previous case: we show that a poly-time environment $\mathcal{Z}$ that distinguishes a real view from an ideal one can be used to construct a distinguisher D that contradicts the key-indistinguishability property of the underlying encryption scheme. We define D as follow.

On input pk^*, D generates (ck, t_X) using cGen, initializes a copy of $\mathcal{Z}$, sends pk^* and ck to $\mathcal{Z}$ and starts executing the protocol Π_{Offline} where $\mathcal{Z}$ controls party P_2 and D plays P_1. The distinguisher D samples a bit $b \leftarrow \{0,1\}$. If $b = 1$, D plays P_1 running the same instruction written for the simulator $\mathcal{S}_2$ and completes *view* choosing the outputs for P_1 as $\mathcal{F}_{\mathsf{Offline}}$ would do. If $b = 0$, D follows the instructions for an honest P_1 in the protocol. However, in the Mult sub-protocol, when D receives the ciphertext C, it can not decrypt because it does not have the secret key. On the other hand, D is allowed to rewind its copy of $\mathcal{Z}$ and therefore it can use the knowledge extractor of protocol Π_{ZKPoCM} (Theorem 4). For example, if the proof Π_{ZKPoCM} is run to check $C = \mathsf{Enc}_{\mathsf{pk}}(a)^{\odot \tilde{b}} \odot \mathsf{Enc}_{\mathsf{pk}}(-\tilde{r})$, D gets from the knowledge extractor $b = \tilde{b} \bmod 2^{k+s}$ and $r = \tilde{r} \bmod 2^{k+s}$ and it can compute its share as $y = a \cdot b - r \bmod 2^{k+s}$, and then continues the protocol as if it had decrypted. Again, by construction, if pk^* is a standard key, then the view produced by D corresponds (statistically) to a real-view if $b = 0$, and to an ideal-view if $b = 1$. On the other hand, if pk^* is a lossy key, the ciphertexts contained in the two views are statistically indistinguishable by definition of lossy key. And, as in case $i = 1$, the messages produced as prover in Π_{ZKPoMCV} (and contained in the two views) are statistically indistinguishable because of the unconditional zero-knowledge property.

3.1 On the Impossibility of Enhanced-CPA Security in $\mathbb{Z}_{2^n}$: Comparing with Overdrive offline phase.

Recently Keller et al. [19] constructed an n-party MPC protocol in the preprocessing model, where the online phase goes as the one in the SPDZ protocol, while the offline is base on a 2-party multiplication protocol similar to the one used in our paper. However, in [19] the ZK proof of correct multiplication is replaced by a postponed check to verify the correctness of the output (similar to the "SPDZ sacrifice"). The possibility of a selective failure attack that this approach introduces is avoided assuming that the underlying encryption scheme satisfies a stronger notion of security called *enhanced CPA*. This notion is recalled in the full version of this paper. Here we prove that, somewhat surprisingly, this notion cannot be achieved by encryption schemes that are linearly homomorphic over rings of the form $\mathbb{Z}_{2^n}$. More precisely, we show that any encryption scheme that is both linearly homomorphic and whose message space is $\mathbb{Z}_{2^n}$ cannot satisfy enhanced CPA security.

Theorem 2. *Let* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be an additive encryption scheme whose message space is* $\mathbb{Z}_{2^n}$ *(see Section 2.2), then the scheme cannot achieve enhanced CPA security (see the full version of this paper for further details).*

Proof. We prove the theorem by showing an efficient adversary $\mathcal{A}$ that successfully wins in the enhanced CPA-security game, with non negligible advantage. $\mathcal{A}$ works as follows. It receives from the challenger both the public key pk and the encryption $C = \mathsf{Enc}_{\mathsf{pk}}(m)$ of a random message $m \in \mathbb{Z}_{2^n}$. Using the homomorphic properties of the scheme, $\mathcal{A}$ computes a new ciphertext C' that encrypts the original message "shifted" by $n-1$ positions to the left. Notice that this only amounts at (homomorphically) multiplying the plaintext by the constant 2^{n-1} (*i.e.*, $C' = C^{\odot 2^{n-1}} = \mathsf{Enc}_{\mathsf{pk}}(2^{n-1} \cdot m)$). $\mathcal{A}$ proceeds by querying the oracle on input C': if the answer is yes $\mathcal{A}$ learns that the least significant bit (lsb) of m is 0; otherwise it learns that it is 1. Now, when the challenger sends out the test message m', $\mathcal{A}$ checks if lsb$(m') \neq$lsb(m) and outputs 1 if this is the case (and 0 otherwise). It is easy to check that such an adversary manages to guess the secret bit chosen by the challenger much better than at random (*i.e.* the winning probability for $\mathcal{A}$ is 1/4).

4 Joye-Libert Cryptosystem and Companion Protocols

In this section we recall the Joye-Libert (JL) cryptosystem [17,5], we refer to the original papers for details missing here.

$\mathsf{Gen}(1^\lambda, n)$: The algorithm starts by choosing two random λ-bit primes p, q, satisfying the following constraints $p \equiv 1 \bmod 2^n$ and $q \equiv 3 \bmod 4$. For simplicity, we let $p = 2^n p' + 1$ and $q = 2q' + 1$ where both p' and q' are primes.[12] Let g be a random generator of both $\mathbb{Z}_p^*$ and $\mathbb{Z}_q^*$, $N = pq$, and $\mu = p'$. The public key is $\mathsf{pk} = (g, n, N)$ and the secret key is $\mathsf{sk} = \mu$.
The message space is $\mathcal{M} = \{0,1\}^n$ while the ciphertext space $\mathcal{C}$ is the subset of $\mathbb{Z}_N^*$ with Jacobi symbol 1. We note that membership in $\mathcal{C}$ can be efficiently and publicly checked by computing the Jacobi symbol $\left(\frac{C}{N}\right)$ of a purposed ciphertext C.

$\mathsf{Enc}_{\mathsf{pk}}(m)$: Choose a random $x \in \mathbb{Z}_N^*$ and output $C = g^m x^{2^n} \bmod N$. With a slight abuse of notation we write $\mathsf{Enc}_{\mathsf{pk}}(m; x)$ to specify the randomness used.

$\mathsf{Dec}_{\mathsf{sk}}(C)$: First, compute $d = C^\mu \bmod p$ and then retrieve m bit by bit, as follows. Notice that $d = (g^\mu)^m \bmod p$ where g^μ is an element of order 2^n in $\mathbb{Z}_p^*$. One can compute the least significant bit m_0 of $m = m_{n-1}...m_0$ by computing $d^{2^{n-1}} \bmod p$. Indeed, this is 1 if and only if $m_0 = 0$. Knowing $m_{i-1}...m_0$ one computes m_i as follows: set $m_i = 0$ if and only if

$$\left(d/(g^{\mu(m_{i-1}...m_0)})\right)^{2^{n-i-1}} = 1 \bmod p$$

If one is interested in retrieving only the lowest $n' < n$ bits of the message, the above mechanism can simply stop at the n'-th step. We can use this optimization

[12] It is not strictly necessary that p' and q' are both primes: nevertheless for security each of them should contain a big enough prime factor.

in our application where $n = k + 2s$ and one is supposed to decrypt and then take the result mod 2^{k+s}. It is worthy to note that the decryption cost is linear in the message bit-size: for the considered settings it can be even faster than a Paillier cryptosystem as confirmed by experiments in Section 5. As shown in [17,5], the scheme is linearly homomorphic over $\mathbb{Z}_{2^n}$.

Security. As shown in [17,5], the JL scheme is semantically secure under the n-quadratic residuosity (n-QR) assumption (that is like the standard quadratic residuosity for a $p \equiv 1 \bmod 2^n$). Moreover, the security analysis shows that the scheme has the nice property of lossy public keys that we require in our applications (see Section 2.2). The "lossy" key generation algorithm Gen consists into sampling g as a 2^n-residue, i.e., $g \leftarrow h^{2^n}$ for a random $h \in \mathbb{Z}_N^*$. Indistinguishability of lossy keys from real ones is proven in [17,5]. Finally, observe that for JL circuit privacy holds whenever one adds a fresh encryption of 0 after an homomorphic computation (or equivalently, as used in our applications, the homomorphic computation involves an addition of a freshly generated ciphertext).

JL as a commitment scheme. It is straightforward to see that the JL cryptosystem is a perfectly binding and computationally hiding commitment scheme for messages in $\mathbb{Z}_{2^n}$: opening simply consists into revealing the randomness used to generate a ciphertext. Such commitments are extractable using an X-trapdoor that is the decryption key. Moreover, the lossy keys property immediately yields that JL is also a "mixed" commitment. Indeed, when generating the public key in lossy mode, commitments become computationally binding and perfectly hiding.

Here we show that in lossy mode, the commitment is also equivocable. This result is of independent interest since we do not use equivocation in our protocols.

First, recall that a key in equivocation mode is a $g = h^{2^n}$ for a random $h \in \mathbb{Z}_N^*$ that is stored as the equivocation trapdoor. Given h one can equivocate a commitment to m with randomness r to an arbitrary m' as follows. Let $C = g^m r^{2^n} \bmod N = (h^m r)^{2^n} \bmod N$ and let $m' = m + \alpha$ over integers; we can rewrite the previous equation as $(h^{m+\alpha-\alpha} r)^{2^n} \bmod N = g^{m'} (h^{-\alpha} r)^{2^n} \bmod N$ and thus setting $r' = h^{-\alpha} r \bmod N$ does the job.

Companion Protocols. In the next section we propose an HVZK protocol for proving correct multiplication relations. Then we show a protocol for proving (partial) knowledge of plaintexts of JL ciphertexts. This is not used in our 2PC protocol but is of independent interest and is given in the full version of this paper.

4.1 Zero-Knowledge Proof of Correct Multiplication

Here we propose an instantiation of the protocol Π_{ZKPoCM}. For $i = 1, 2$, let $\mathsf{pk}_i = (g_i, n, N_i)$ be a JL public key (both working with the same message space) and let $\mathcal{C}_i$ be the respective ciphertext spaces. In Figure 7 we describe a Σ-protocol for the NP relation $\mathcal{R}' \subseteq (\mathbb{Z}_{2^{n-s}})^2 \times \mathcal{C}_1^2 \times \mathcal{C}_2$:

$$\begin{aligned}\mathcal{R}' = \{ & ((b,r),(A,C,B)) \mid \exists\, (\tilde{b},\tilde{r}) \in (\mathbb{Z}_{2^n})^2, (x_r,x_b) \in \mathbb{Z}_{N_1}^* \times \mathbb{Z}_{N_2}^* \text{ s.t.} \\ & B = \mathsf{Enc}_{\mathsf{pk}_2}(\tilde{b},x_b),\ C = A^{\odot \tilde{b}} \odot \mathsf{Enc}_{\mathsf{pk}_1}(\tilde{r},x_r),\ b = \tilde{b} \bmod 2^{n-s},\ r = \tilde{r} \bmod 2^{n-s}\}.\end{aligned}$$

This proof system allows one to prove knowledge of the least $n-s$ significant bits of the messages $\tilde{b}, \tilde{r}$ used to define the ciphertext C.

Intuitively, the reason why we do not prove knowledge of the entire messages is that, for technical reasons related to the fact that not all messages are invertible, this is actually not possible. Interestingly enough, however, if we set challenges to be integers of s bits, then we can recover all but the s most significant bits. This means that if one carefully encrypts messages that are small enough (e.g., all the s most significant bits are zero), then one can actually recover the full message.

In what follows we prove that the protocol Π_{ZKPoCM} guarantees correctness, (honest verifier) zero knowledge and special soundness.

Completeness. This can be seen by inspection of the protocol.

Protocol Π_{ZKPoCM}

Common input for prover and verifier: two JL public keys $\mathsf{pk}_i = (g_i, n, N_i)$, for $i = 1,2$, and JL ciphertexts $A, C \in \mathcal{C}_1$ and $B \in \mathcal{C}_2$.
Private input for the prover: $\tilde{b}, \tilde{r} \in \mathbb{Z}_{2^n}$ and $(x_r, x_b) \in \mathbb{Z}_{N_1}^* \times \mathbb{Z}_{N_2}^*$ such that $B = \mathsf{Enc}_{\mathsf{pk}_2}(\tilde{b}, x_b)$ and $C = A^{\odot \tilde{b}} \odot \mathsf{Enc}_{\mathsf{pk}_1}(\tilde{r}, x_r)$.

1. P samples $\beta \in \mathbb{Z}_{N_2}^*$ and computes $R = \mathsf{Enc}_{\mathsf{pk}_2}(\tilde{r}, \beta) = g_2^{\tilde{r}} \beta^{2^n} \bmod N_2$. Also it samples $x, y \leftarrow \mathbb{Z}_{2^n}$ and $v \leftarrow \mathbb{Z}_{N_1}^*, \gamma_x, \gamma_y \leftarrow \mathbb{Z}_{N_2}^*$ and computes: $D = A^x g_1^y v^{2^n} \bmod N_1$, $X = g_2^x \gamma_x^{2^n} \bmod N_2$, $Y = g_2^y \gamma_y^{2^n} \bmod N_2$. It sends R, D, X, Y to the verifier.
2. The verifier sends back[a] $e \leftarrow \mathbb{Z}_{2^s}$.
3. The prover computes $z_b = x + e\tilde{b} \bmod 2^n$, $z_r = y + e\tilde{r} \bmod 2^n$ and q_b, q_r such that $q_b 2^n = x + e\tilde{b} - z_b$ and $q_r 2^n = y + e\tilde{r} - z_r$, computes $\delta_b = \gamma_x x_b^e g_2^{q_b} \bmod N_2, \delta_r = \gamma_y \beta^e g_2^{q_r} \bmod N_2, \omega = A^{q_b} x_r^e g_1^{q_r} v \bmod N_1$, and sends to the verifier $z_b, z_r, \delta_b, \delta_r, \omega$.
4. The verifier accepts if and only if all the following checks pass
   (a) $DC^e = A^{z_b} \mathsf{Enc}_{\mathsf{pk}_1}(z_r, \omega)$
   (b) $XB^e = g_2^{z_b} \delta_b^{2^n} \bmod N_2 = \mathsf{Enc}_{\mathsf{pk}_2}(z_b, \delta_b)$
   (c) $YR^e = g_2^{z_r} \delta_r^{2^n} \bmod N_2 = \mathsf{Enc}_{\mathsf{pk}_2}(z_r, \delta_r)$
   and if $A, C, D \in \mathcal{C}_1$ and $B, R, X, Y \in \mathcal{C}_2$ hold.

[a] For the sake of simplicity, $\mathcal{F}_{\mathsf{Rand}}$ is used to generate the challenge e when Π_{ZKPoCM} is used as sub-protocol of Π_{Offline}.

Fig. 7. Proof of correct multiplication for JL-encryptions

Theorem 3 (Honest-Verifier Zero-Knowledge). *If JL is a semantically secure public key encryption, then the protocol in Figure 7 is honest-verifier zero-knowledge. Furthermore, if in the protocol the public key pk_2 is generated in lossy mode, then honest-verifier zero-knowledge holds unconditionally.*

Proof. First, we describe a simulator that works as follows. Given a challenge e and JL ciphertexts A, B, C: sample $z_b, z_r \leftarrow \mathbb{Z}_{2^n}$, $R \leftarrow \mathcal{C}_2$, $\delta_b, \delta_r \leftarrow \mathbb{Z}^*_{N_2}$, $\omega \leftarrow \mathbb{Z}^*_{N_1}$, and set $D = A^{z_b} g_1^{z_r} \omega^{2^n} C^{-e} \bmod N_1$, $X = g_2^{z_b} \delta_b^{2^n} B^{-e} \bmod N_2$ and $Y = g_2^{z_r} \delta_r^{2^n} R^{-e} \bmod N_2$.

We claim that the simulated proof is computationally indistinguishable from the real one under the assumption that JL is semantically secure. The only (information-theoretic) difference between the real proof and the simulated one is that in the simulation R is the encryption of a random message, not the same $\tilde{r}$ known by the honest prover. This however is not noticeable to a computationally-bounded distinguisher. More formally, this can be argued by defining an hybrid simulator that takes as input $\tilde{r}$ and computes the proof as the simulator above with the only difference that R is a fresh encryption of $\tilde{r}$. The proofs created by this hybrid simulator are computationally indistinguishable from the ones created by the ZK simulator under the assumption that JL (over public key pk_2) is semantic secure. As a next step, one must argue that the proofs created by this hybrid simulator and the ones of the honest prover are distributed identically. This can be verified by inspection.

Finally, when pk_2 is lossy, then we can skip the computational step of the proof since, even if R is sampled randomly, by the lossy property is distributed identically to a lossy encryption of $\tilde{r}$.

Theorem 4 (Special Soundness). *The protocol in Figure 7 has special soundness.*

Proof. We prove that a prover cannot succeed in proving a wrong statement unless with negligible probability. We prove this as follows.

Assume that, for the same values used in steps 1 and 2 of the protocol, a prover manages to successfully answer for a non negligible fraction of challenges e. This means that there exist e_1, e_2, $e_1 \neq e_2$ (and wlog $e_1 > e_2$) such that

1. $C^{\Delta e = e_1 - e_2} = A^{\Delta z_b = z_{b1} - z_{b2}} \mathsf{Enc}(\Delta z_r = z_{r1} - z_{r2}, \omega/\omega')$
2. $B^{\Delta e} = g_2^{\Delta z_b} (\delta_b/\delta'_b)^{2^n} \bmod N_2$
3. $R^{\Delta e} = g_2^{\Delta z_r} (\delta_r/\delta'_r)^{2^n} \bmod N_2$

We distinguish between 2 cases, depending on whether $\gcd(\Delta e, 2^n) = 1$ or not.

Case $gcd(\Delta e, 2^n) = 1$. In this case one can easily extract a full $\tilde{b} \in \mathbb{Z}_{2^n}$ as $\tilde{b} = \Delta z_b / \Delta e \bmod 2^n$ and $\tilde{r} \in \mathbb{Z}_{2^n}$ as $\tilde{r} = \Delta z_r / \Delta e \bmod 2^n$.

Case $gcd(\Delta e, 2^n) \neq 1$. In this case let $\gcd(\Delta e, 2^n) = 2^t$ for some $t \leq s$ (the latter holds because $e_1, e_2 \in \mathbb{Z}_{2^s}$). We can rewrite the three equations above as follows

1. $C^{2^t e'} = A^{\Delta z_b} \mathsf{Enc}(\Delta z_r, \omega/\omega')$

2. $B^{2^t e'} = g_2^{\Delta z_b}(\delta_b/\delta'_b)^{2^n} \bmod N_2$
3. $R^{2^t e'} = g_2^{\Delta z_r}(\delta_r/\delta'_r)^{2^n} \bmod N_2$

From now on let us focus on the second equation above (the same argument will trivially hold for the third equation). First let d be the inverse of $e' \bmod 2^n$. Exponentiating both sides of the equation to d leads to the following $B^{2^t} = g_2^{d\Delta z_b}\left((\delta_b/\delta'_b)^d\right)^{2^n} \bmod N_2$. Notice that since g_2 is *not* a quadratic residue, the integer $d\Delta z_b$ must be even. Let t' be the largest integer such that $2^{t'}$ divides $d\Delta z_b$, i.e., $d\Delta z_b = 2^{t'}d'$ for some odd number d'. Clearly $t' \leq n$. We can rewrite the equation as

$$B^{2^t} = g_2^{2^{t'}d'}\left((\delta_b/\delta'_b)^d\right)^{2^n} \bmod N_2 \tag{1}$$

We distinguish two cases: (a) $t > t'$ and (b) $t \leq t'$.

Case (a) $t > t'$: If (1) holds and $B^{2^{t-t'}}/(g_2^{d'}\left((\delta_b/\delta'_b)^d\right)^{2^{n-t'}}) \bmod N_2 \notin \{-1, 1\}$, then we can immediately factor N_2 since we found a nontrivial root of unity. Given the factorization of N_2 extracting $\tilde{b}$ from B is possible using decryption. Otherwise, we have that

$$B^{2^{t-t'}} = u \cdot g_2^{d'}\left((\delta_b/\delta'_b)^d\right)^{2^{n-t'}} \bmod N_2 \tag{2}$$

for $u = 1$ or $u = -1$. We show that neither of the cases can occur. If $u = 1$, the equality (2) is not possible because d' is odd and g_2 is not a quadratic residue by construction. If $u = -1$, (2) is not possible because in this group setting ($p \equiv 1 \bmod 2^n$ and $q \equiv 3 \bmod 4$) -1 has Jabobi symbol -1 in $\mathbb{Z}^*_{N_2}$ (see [5, Theorem 1]) whereas all the other terms of the equation have Jacobi symbol 1. This concludes case (a).

Case (b) $t \leq t'$: Let $\tilde{b} \in \mathbb{Z}_{2^n}$ be the integer encrypted in B. By the homomorphic property of JL we have that B^{2^t} is a ciphertext that encrypts $2^t\tilde{b} \bmod 2^n = 2^t(\tilde{b} \bmod 2^{n-t}) = 2^t b_t$.

From equation (1), we can write B^{2^t} as an encryption of $2^{t'}d'$. Combined with the previous observation we have $2^t(\tilde{b} \bmod 2^{n-t}) = 2^{t'}d'$ and using the fact that $t \leq t'$ we obtain that $b_t = \tilde{b} \bmod 2^{n-t} = 2^{t'-t}d' = d\Delta z_b 2^{-t}$. This shows that $d\Delta z_b 2^{-t} \in \mathbb{Z}_{2^{n-t}}$ is the $(n-t)$-bit portion of the message encrypted in B. Finally, since $t \leq s$ we can set $b = (d\Delta z_b 2^{-t}) \bmod 2^{n-s}$. This concludes the proof about extractability of b.

By applying exactly the same argument above to R and the third verification equation, we can extract $r \in \mathbb{Z}_{2^{n-s}}$ as $r = (d\Delta z_r 2^{-t}) \bmod 2^{n-s}$.

Towards concluding the proof, let us recall that the relation requires

$$B = \mathsf{Enc}_{\mathsf{pk}_2}(\tilde{b}, x_b),\ C = A^{\odot \tilde{b}} \odot \mathsf{Enc}_{\mathsf{pk}_1}(\tilde{r}, x_r),\ b = \tilde{b} \bmod 2^{n-s},\ r = \tilde{r} \bmod 2^{n-s}$$

We have already extracted b and r; in what follows we need to argue that they satisfy the relation above. The check about B is already satisfied. So let us focus on the remaining checks.

Let $\tilde{a}$ be the integer encrypted in A, namely let us write $A = g_1^{\tilde{a}} x_a^{2^n}$. Similarly, let $\tilde{c} \in \mathbb{Z}_{2^n}$ be the integer encrypted in C. Then showing that the extracted values satisfy the relation means to show that $\tilde{c} = \tilde{a}\tilde{b} + \tilde{r} \bmod 2^n$ such that the least $n-s$ significant bits of $\tilde{b}, \tilde{r}$ are b and r respectively. More formally, this means to show that there is some q_s such that $\tilde{c}$ can be written as $a_s b + r + q_s 2^{n-s}$, for $a_s = \tilde{a} \bmod 2^{n-s}$. In other words, $c_s = \tilde{c} \bmod 2^{n-s} = a_s b + r \bmod 2^{n-s}$.

Now let us consider the first equation. By the homomorphic property we have that C^{2^t} is a ciphertext that encrypts $c' = 2^t \tilde{c} \bmod 2^n = 2^t(\tilde{c} \bmod 2^{n-t}) = 2^t c_t$.

From the first verification equation, exponentiating both sides of the equation by $d = e'^{-1} \bmod 2^n$, we get $C^{2^t} = A^{d\Delta z_b} g_1^{d\Delta z_r} \left((\omega/\omega')^d\right)^{2^n} \bmod N_1$ and using the expression of A, we can rewrite the equation as

$$\begin{aligned} C^{2^t} &= g_1^{d(\tilde{a}\Delta z_b + \Delta z_r)} \left(x_a^{d\Delta z_b} (\omega/\omega')^d\right)^{2^n} \bmod N_1 \\ &= g_1^{\tilde{a}(d\Delta z_b) + (d\Delta z_r)} \left(x_a^{d\Delta z_b} (\omega/\omega')^d\right)^{2^n} \bmod N_1 \\ &= g_1^{\tilde{\alpha}} \left(g_1^{q_\alpha} x_a^{d\Delta z_b} (\omega/\omega')^d\right)^{2^n} \bmod N_1 \end{aligned}$$

where in the last equation we used $\tilde{a} d\Delta z_b + d\Delta z_r = \tilde{\alpha} + q_\alpha 2^n$.

Thus we have that $c' = \alpha$. Notice that 2^t divides both $d\Delta z_b$ and $d\Delta z_r$ (this follows from the arguments used in the extractability of b and r), and thus by definition of $\tilde{\alpha}$, $2^t \mid \tilde{\alpha}$. In particular, $\tilde{\alpha} 2^{-t} = \tilde{a} d\Delta z_b 2^{-t} + d\Delta z_r 2^{-t} - q_\alpha 2^{n-t}$.

Therefore, $c_t = c' 2^{-t} = \tilde{\alpha} 2^{-t} = \tilde{a} d\Delta z_b 2^{-t} + d\Delta z_r 2^{-t} - q_\alpha 2^{n-t} = \tilde{a} b_t + r_t - q_\alpha 2^{n-t} = a_t b_t + r_t + q 2^{n-t}$.

If we take both sides mod 2^{n-s} (recall that $t \leq s$), we have that $c_s = \tilde{c} \bmod 2^{n-s} = a_s b + r \bmod 2^{n-s}$ as it was to be proven.

4.2 Zero-Knowledge Proof of Correct Multiplication of Two Committed (or Encrypted) Values

Here we propose an instantiation of the protocol Π_{ZKPoMCV} that allows a prover to show that she correctly performed multiplication of two committed (or encrypted) values. The protocol is given in Figure 8. Essentially, it considers a special case of the relation supported by the protocol in the previous section in which $r = 0$ and the ciphertexts are under the same public key. This specialization allows us to simplify and optimize the resulting protocol. Let $\mathsf{pk} = (g, n, N)$ be a JL public key and $\mathcal{C}$ its corresponding ciphertext space[13]. Specifically, we give a Σ-protocol for the NP relation $\mathcal{R} \subseteq (\mathbb{Z}_{2^{n-s}}) \times \mathcal{C}^3$:

$$\begin{aligned} \mathcal{R} = \{&(b, (A, C, B)) \mid \exists\, \tilde{b} \in \mathbb{Z}_{2^n}, (x_r, x_b) \in (\mathbb{Z}_N^*)^2 \text{ s.t.} \\ &B = \mathsf{Enc}_{\mathsf{pk}}(\tilde{b}, x_b),\ C = A^{\odot \tilde{b}} \odot \mathsf{Enc}_{\mathsf{pk}}(0, x_r),\ b = \tilde{b} \bmod 2^{n-s}\}. \end{aligned}$$

[13] When Π_{ZKPoMCV} is used in the offline phase of Mon$\mathbb{Z}_{2^k}$a, we have $\mathsf{pk} = \mathsf{pk}_1$ if the P_1 is the prover or $\mathsf{pk} = \mathsf{pk}_2$ if P_2 is the prover ($\mathsf{pk}_1, \mathsf{pk}_2$ are the keys used in Π_{ZKPoCM}).

As in previous protocols in this paper, this proof system allows one to prove knowledge of the least $n-s$ significant bits of the message $\tilde{b}$ used to define the ciphertext C.

Notice that correctness of the protocol $\mathit{\Pi}_{\mathsf{ZKPoCM}}$ can be easily inferred by inspection. Special soundness follows as a special case of Thm 4 (when ignoring the third equation and setting $r=0$). Honest verifier zero-knowledge also follows as a special case of Thm 3. Interestingly, however, in this protocol $\mathit{\Pi}_{\mathsf{ZKPoMCV}}$ the zero knowledge property holds unconditionally. Recall that in the proof of Thm. 3 the only reason we needed to resort to the semantic security of JL was because of the possible difference between the ciphertext R used by the prover and the one sampled by the simulator. Since in our case there is no such a difference, there is also no difference between the real proof and the simulated one.

Protocol $\mathit{\Pi}_{\mathsf{ZKPoMCV}}$

Common input for prover and verifier: A JL public key $\mathsf{pk}=(g,n,N)$, and JL commitments (ciphertexts) $A,B,C \in \mathcal{C}$.
Private input for the prover: $\tilde{b} \in \mathbb{Z}_{2^n}$ and $x_b, x_r \in \mathbb{Z}_N^*$ such that $B=\mathsf{Enc}_{\mathsf{pk}}(\tilde{b},x_b)$ and $C=A^{\odot\tilde{b}} \odot \mathsf{Enc}_{\mathsf{pk}}(0,x_r)$.

1. P samples $x \leftarrow \mathbb{Z}_{2^n}$ and $v,\gamma_x \leftarrow \mathbb{Z}_N^*$ and computes: $D=A^x v^{2^n} \bmod N$, $X=g^x\gamma_x^{2^n} \bmod N$. It sends D,X to the verifier.
2. The verifier sends back[a] $e \leftarrow \mathbb{Z}_{2^s}$.
3. The prover computes $z_b=x+e\tilde{b} \bmod 2^n$ and q_b such that $q_b 2^n = x+e\tilde{b}-z_b$, computes
   $\delta_b=\gamma_x x_b^e g^{q_b} \bmod N, \omega=A^{q_b} v x_r^e \bmod N$, and sends to the verifier z_b,δ_b,ω.
4. The verifier accepts if and only if all the following checks pass
   (a) $DC^e=A^{z_b}\mathsf{Enc}_{\mathsf{pk}}(0,\omega)$
   (b) $XB^e=g^{z_b}\delta_b^{2^n} \bmod N=\mathsf{Enc}_{\mathsf{pk}}(z_b,\delta_b)$
   and if $A,B,C,D,X \in \mathcal{C}$ holds.

[a] Again, for the sake of simplicity, $\mathcal{F}_{\mathsf{Rand}}$ is used to generate the challenge e when $\mathit{\Pi}_{\mathsf{ZKPoMCV}}$ is used as sub-protocol of $\mathit{\Pi}_{\mathsf{Offline}}$.

Fig. 8. Modified (simpler) version of $\mathit{\Pi}_{\mathsf{ZKPoCM}}$.

5 Efficiency Analysis

Here we turn to estimate the efficiency of our preprocessing protocol with respect to SPD$\mathbb{Z}_{2^k}$ in [9]; the online phase is essentially the same. Before entering into the details of the evaluation, in the next section we discuss a variant of our offline protocol that significantly reduces the overall bandwidth consumption at the cost of (1) explicitly requiring the random oracle heuristic, and (2) increasing the computational overhead of both players. Next, we analyze the efficiency of both the base and optimized versions.

Optimization using random oracles. First, assume that P_1 knows the secret key corresponding to the encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ (as it already holds), and that P_2 is given the extraction trapdoor for the (extractable) commitment $(\mathsf{cGen}, \mathsf{Com})$. Since valid JL ciphertexts – and commitments – are both easy to recognize and easy to sample, the holder of the secret decryption key (resp. extraction trapdoor) has an alternative way to generate a couple $(m, \mathsf{Enc}(m))$ (resp. $(m, \mathsf{Com}(m))$ with m random: it first samples a random ciphertext $\mathsf{Enc}(m)$ (resp. commitment $\mathsf{Com}(m)$), and then extracts m using the secret key[14]. It is straightforward to see that these two sampling procedures (*i.e.*, via encryption or decryption) generate the same distribution.

The related security proofs would require minor changes to the simulators $\mathcal{S}_1, \mathcal{S}_2$: for example, $\mathcal{S}_1$ in (Single, P_1) would get R' from $\mathcal{F}_{\mathsf{Rand}}$, instead of from $\mathcal{Z}$, and in (Single, P_2) it would compute r' from R' (again received from $\mathcal{F}_{\mathsf{Rand}}$) using the extraction trapdoor of the commitment, as the honest P_2 would do.

This simple idea can be used to gain in communication complexity as follows. In protocol $\mathit{\Pi}_{\mathsf{Offline}}$ on input $(\mathsf{Single}, P_1, sid, ssid)$, the parties can get a common $R = \mathsf{Enc}_{\mathsf{pk}}(r)$, without any communication, by simply setting $R \leftarrow H_1(\omega_1, sid, ssid)$ where ω_1 is some common auxiliary information and H_1 is a random oracle mapping into the ciphertext space of $\mathsf{Enc}_{\mathsf{pk}}$. Similarly, on input $(\mathsf{Single}, P_2, sid, ssid)$, the parties can get a common $R = \mathsf{Com}_{\mathsf{ck}}(r)$ by setting $R \leftarrow H_2(\omega_2, sid, ssid)$ (where, again, ω_2 is some common auxiliary information and H_2 is a random oracle mapping into the commitment space of $\mathsf{Com}_{\mathsf{ck}}$).

Similarly, the communication complexity of $\mathit{\Pi}_{\mathsf{ZKPoCM}}$ (see section 4.1) can be reduced by generating X and Y using the random oracle in exactly the same way. Moreover the resulting protocols remain secure with these modifications as, all the quantities retain the same original distribution and, as proved in section 4.1, the (special) soundness of $\mathit{\Pi}_{\mathsf{ZKPoCM}}$ holds unconditionally. The same holds for $\mathit{\Pi}_{\mathsf{ZKPoMCV}}$ (see section 4.2): the transmission of X (an encryption of random value x) can be avoided.

Bandwidth usage. The bandwidth of our sub-protocols depends on few parameters: the size of the generic modulus used in the JL encryption/commitment schemes denoted as $|N|$, the message bit-length k, the statistical security parameter s and the internal parameter $n = k + 2s$.

We analyze the elements exchanged between the parties. The sub-protocol $\mathit{\Pi}_{\mathsf{ZKPoCM}}$ in Figure 7 sends a total of 7 elements of size $|N|$ and two of n bits. The sub-protocol $\mathit{\Pi}_{\mathsf{ZKPoMCV}}$ in Figure 8 sends four elements of size $|N|$ and one of n bits. The multiplication sub-protocol Mult in Figure 3 sends an element of size $|N|$ before an invocation of $\mathit{\Pi}_{\mathsf{ZKPoCM}}$. The sub-protocols ($\mathsf{Single}$, P_i) in Figure 3 send an encryption/commitment (size $|N|$) followed by an instance of Mult; the variant Single, used to generate a shared random value unknown to all parties, runs (Single, P_1) and (Single, P_2). Finally, in Triple one invokes two times Single, four times Mult, two times $\mathit{\Pi}_{\mathsf{ZKPoCM}}$ and sends four encryptions/commitments

[14] More precisely, in order for the above idea to be any useful in our protocols, we also need to extract the randomness associated to the encryption/commitment. Luckily, this happens to be the case when using JL as underlying building block.

Table 1. Bandwidth analysis of our sub-protocols

	$\mathit{\Pi}_{\mathsf{ZKPoCM}}$	$\mathit{\Pi}_{\mathsf{ZKPoMCV}}$	Mult	$(\mathsf{Single},\ P_i)$	Single	Triple
Mon$\mathbb{Z}_{2^k}$a base	$7\lvert N\rvert + 2n$	$4\lvert N\rvert + n$	$8\lvert N\rvert + 2n$	$9\lvert N\rvert + 2n$	$18\lvert N\rvert + 4n$	$78\lvert N\rvert + 18n$
Mon$\mathbb{Z}_{2^k}$a optim.	$5\lvert N\rvert + 2n$	$3\lvert N\rvert + n$	$6\lvert N\rvert + 2n$	$6\lvert N\rvert + 2n$	$12\lvert N\rvert + 4n$	$56\lvert N\rvert + 18n$

with size $|N|$ bits[15]. We also consider the optimized version of our protocols discussed in Section 5.

For a concrete comparison we consider some significant settings, varying the available parameters, and comparing the results with data on SPD$\mathbb{Z}_{2^k}$ in [9]. For each considered computational security level $S \in \{80, 112, 128\}$, we select a proper statistical security parameter s according to the message bit-length $k \in \{32, 64, 128\}$. The size of the modulus N is selected according to recent NIST recommendations[16]. The extended comparison is reported in Table 2 with bold remarks on the best values per triple and single generation[17]. The global costs to generate a triple and a single (input sharing) in SPD$\mathbb{Z}_{2^k}$ are computed according to the formulas $2(k + 2s)(9s + 4k)$ and $(s + 1)(k + 2s)$ reported in Section 7 of [9]. For the input sharing step of our protocols we consider the cost of $(\mathsf{Single},\ P_i)$ as a random shared value known to P_i is later used to share a secret input belonging to him .

Implementation and computational benchmark. We implemented the off-line phase of the base version of Mon$\mathbb{Z}_{2^k}$a[18]: it produces triples and singles that could be used in the on-line phase of SPD$\mathbb{Z}_{2^k}$. Our implementation is written in language C and uses the GNU Multiprecision Library[19] (GMP) for the MPI operations. We used two servers equipped with an Intel Xeon 8124M CPU running at 3.0 GHz: each server hosts a single thread running one of the two parties. We simulated three typical deploying scenarios: two servers connected by a common 1 Gigabit Ethernet LAN with an average latency (intended as Round Trip Time — RTT) of 0.5 ms and two servers hosted by two different data-centers connected by a fast WAN with 17 ms of latency[20] or by a very-limited WAN with 100 ms of latency and a bandwidth of 50 Mb/s.

The underlying JL encryption scheme has been implemented following the specifications in [5] with few adjustments: adaptation of the decryption algorithm

[15] Similarly to the analysis in [9], we ignore the negligible costs of $\mathcal{F}_{\mathsf{Rand}}$ and of the check of the openings in Triple as it can be performed in a batch when producing many triples at once.

[16] `https://keylength.com`

[17] For sake of completeness, in the border case with $S = 80$, $s = 40$ and $k = 128$, we considered a slightly larger modulus $|N| = 1160$ in order to satisfy the security constraint $k + 2s < \frac{1}{4}\log_2(N) - S$ on JL scheme from [5].

[18] The source code of our project is publicly available at: `https://github.com/crypto-unict/monza-mpc`

[19] `https://gmplib.org`

[20] We considered the actual ping delay between Amazon and Google data-centers.

Table 2. Bandwidth comparison with SPD$\mathbb{Z}_{2^k}$ (costs in kbit)

				SPD$\mathbb{Z}_{2^k}$		**Mon$\mathbb{Z}_{2^k}$a base**		**Mon$\mathbb{Z}_{2^k}$a optim.**	
S	$\|N\|$	k	s	triple	input	triple	input	triple	input
80	1024	32	32	79.87	**3.17**	81.60	9.41	**59.07**	6.34
		64	40	177.41	**5.90**	82.46	9.50	**59.94**	6.43
		128	40	362.75	8.53	94.22	10.86	**68.70**	**7.38**
112	2048	32	32	**79.87**	**3.17**	161.47	18.62	116.42	12.48
		64	56	267.52	**10.03**	162.91	18.78	**117.86**	12.64
		128	56	487.68	13.68	164.06	18.91	**119.01**	**12.77**
128	3072	32	32	**79.87**	**3.17**	241.34	27.84	173.76	18.62
		64	64	319.49	**12.48**	243.07	28.03	**175.49**	18.82
		128	64	557.06	**16.64**	244.22	28.16	**176.64**	18.94

(S: comp. sec. level; N: JL-schemes modulus; k: message bit-length; s: stat. sec. level)

to support the partial extraction of the plaintext[21] (as described in Section 4), usage of some precomputed values derived by some components of the public and secret keys (as described in Section 5.2 in [5]) and a faster encryption exploiting some fixed base exponentiations.

For each protocol we measured the effective (wall clock) time required to get the final output but also the CPU usage (in percent): indeed in a real implementation the CPU can become idle waiting for incoming values delayed by network latency. Even a medium latency can degrade the final performance of an interactive protocol: in order to overcome this limit, we engineered the possibility to run on a single CPU thread a batch of interlaced runs in order to piggyback the passing network messages. As shown by our tests, this allows to get even on a very slow WAN connection almost the same throughput rate of a LAN.

The experiments used the following parameters: message bit-length $k = 64$, computational security level $S = 112$, statistical security level $s = 56$ and JL modulus size $|N| = 2048$ bit. The benchmarks reported in Table 3 are obtained as average on a batch of several runs with low standard deviation (1%). The value in the column "average time" is intended as the average cost of a single item of the batch.

Acknowledgements. The research of Dario Catalano and Mario Di Raimondo has been partially supported by the Università degli Studi di Catania, "Piano della Ricerca 2016/2018 — Linea di intervento 2".

The research of Dario Fiore has been partially supported by the Spanish Government under projects SCUM (ref. RTI2018-102043-B-I00), CRYPTOEPIC

[21] JL decryption can be surprisingly fast for small messages; as reference a Paillier decryption, with identical parameters/machine/setting used in Table 3, has a cost that range from 7864 μs to 4323 μs (if CRT is exploited). JL requires only 4054 μs.

Table 3. Benchmarks on Mon$\mathbb{Z}_{2^k}$a off-line protocol

		triple				**input**			
latency	batch	av. time	CPU P_1	CPU P_2	throug.	av. time	CPU P_1	CPU P_2	throug.
(ms)	(items)	(ms)	(%)	(%)	(item/s)	(ms)	(%)	(%)	(item/s)
	1	56.65	80%	32%	17.65	7.99	70%	24%	125.16
0.5	**100**	**52.24**	86%	35%	**19.14**	**7.41**	74%	26%	**134.95**
(LAN)	1000	52.36	85%	35%	19.10	7.43	74%	26%	134.59
	1	253.68	18%	7%	3.94	40.37	14%	5%	24.77
17.0	1000	53.05	84%	34%	18.85	7.52	74%	25%	132.99
(WAN)	**2000**	**52.34**	85%	34%	**19.11**	**7.42**	74%	26%	**134.77**
	1	1252.53	4%	2%	206.85	40.37	3%	1%	4.83
100.0	1000	58.34	77%	31%	17.14	8.25	67%	23%	121.21
(WAN)	**4000**	**55.44**	81%	33%	**18.03**	**7.95**	70%	24%	**125.79**

(ref. EUR2019-103816), and SECURITAS (ref. RED2018-102321-T), and by the Madrid Regional Government under project BLOQUES (ref. S2018/TCS-4339).

References

1. Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *2017 IEEE Symposium on Security and Privacy*, pages 843–862. IEEE Computer Society Press, May 2017.
2. 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 16*, pages 805–817. ACM Press, October 2016.
3. Boaz Barak, Ran Canetti, Jesper Buus Nielsen, and Rafael Pass. Universally composable protocols with relaxed set-up assumptions. In *45th FOCS*, pages 186–195. IEEE Computer Society Press, October 2004.
4. 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.
5. Fabrice Benhamouda, Javier Herranz, Marc Joye, and Benoît Libert. Efficient cryptosystems from 2^k-th power residue symbols. *Journal of Cryptology*, 30(2):519–549, April 2017.
6. 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.

7. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.
8. Dario Catalano and Dario Fiore. Using linearly-homomorphic encryption to evaluate degree-2 functions on encrypted data. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 1518–1529. ACM Press, October 2015.
9. 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.
10. Ronald Cramer, Serge Fehr, Yuval Ishai, and Eyal Kushilevitz. Efficient multiparty computation over rings. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 596–613. Springer, Heidelberg, May 2003.
11. 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.
12. 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.
13. 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.
14. Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 225–255. Springer, Heidelberg, April / May 2017.
15. Niv Gilboa. Two party RSA key generation. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 116–129. Springer, Heidelberg, August 1999.
16. Shafi Goldwasser and Silvio Micali. Probabilistic encryption and how to play mental poker keeping secret all partial information. In *14th ACM STOC*, pages 365–377. ACM Press, May 1982.
17. Marc Joye and Benoît Libert. Efficient cryptosystems from 2^k-th power residue symbols. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 76–92. Springer, Heidelberg, May 2013.
18. 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 16*, pages 830–842. ACM Press, October 2016.
19. 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.
20. Emmanuela Orsini, Nigel P. Smart, and Frederik Vercauteren. Overdrive2k: Efficient secure mpc over z_{2^k} from somewhat homomorphic encryption. Cryptology ePrint Archive, Report 2019/153, 2019.