Dishonest Majority Multiparty Computation over Matrix Rings

Hongqing Liu, Chaoping Xing, Chen Yuan, Taoxu Zou

Shanghai Jiao Tong University, China

Abstract. The privacy-preserving machine learning (PPML) has gained growing importance over the last few years. One of the biggest challenges is to improve the efficiency of PPML so that the communication and computation costs of PPML are affordable for large machine learning models such as deep learning. As we know, linear algebra such as matrix multiplication occupies a significant part of the computation in the deep learning such as deep convolutional neural networks (CNN). Thus, it is desirable to propose the MPC protocol specialized for the matrix operations. In this work, we propose a dishonest majority MPC protocol over matrix rings which supports matrix multiplication and addition. Our MPC protocol can be seen as a variant of SPDZ protocol, i.e., the MAC and global key of our protocol are vectors of length m and the secret of our protocol is an $m \times m$ matrix. Compared to the classic SPDZ protocol, our MPC protocol reduces the communication complexity by at least m times. We also show that our MPC protocol is as efficient as [11] which also presented a dishonest majority MPC protocol specialized for matrix operations. The MPC protocol [11] resorts to the homomorphic encryption scheme (BFV scheme) to produce the matrix triples in the preprocessing phase. This implies that their protocol only supports the matrix operations over integer rings or prime fields of large size. On the contrary, we resort to vector oblivious linear evaluations and random vector oblivious linear evaluations to generate correlated randomness in the preprocessing phase. Thus, the matrices of our MPC protocol can be defined over any finite field or integer ring. Due to the small size of our MAC, the communication complexity of our MPC protocol remains almost the same regardless of the size of the field or the ring.

1 Introduction

Secure multiparty computation (MPC) allows a set of mutually distrustful parties $P_1, \cdots, P_n$ to jointly compute a public function f with their private inputs, and reveals nothing except the final output. The adversary could corrupt at most t of n parties to gain the private information of honest parties by either inspecting the transcripts between parties (semi-honest adversary) or arbitrarily deviating the protocol (malicious adversary). According to the number of corrupted parties t, MPC protocols can be classified into two categories: honest majority ($t \leq \frac{n}{2}$) and dishonest majority ($t < n$). The honest majority MPC

protocol can achieve information-theoretic security while the dishonest majority MPC protocol can only achieve computational security.

In MPC protocols, the public function f is generally modeled as an arithmetic circuit over a finite field or a ring, which consists of addition and multiplication gates. The computation over a ring is usually more complicated than the computation over a field. Before the advent of privacy preserving machine learning (PPML), most of the MPC protocols were only restricted to the computation over finite fields. The use of integer rings is well-motivated in practice due to their direct compatibility with hardware. In view of this practical application, a line of works [12,25,2,1,18] proposed the MPC protocol over $\mathbb{Z}_{2^k}$. Recently, Escudero and Soria-Vazquez [17] considered the non-commutative ring in the honest majority setting. They constructed an unconditionally secure MPC over non-commutative rings with black-box access to a ring containing an exceptional set[1], whose size is at least the number of parties. They also proposed an honest majority MPC protocol over the matrix ring $\mathcal{M}_{m\times m}(\mathbb{Z}_{2^k})$.

Inspired by [17], a natural question is can we design an MPC protocol over a non-commutative ring with only black-box access to the ring in the presence of $t \geq \frac{n}{2}$ corrupted parties? The answer is probably negative as the dishonest majority MPC protocols rely on some cryptographic assumptions. Moreover, while honest majority MPC protocols use the error-correction algorithm of Shamir secret sharing to detect and even correct the corruptions, the dishonest majority MPC protocols have to rely on the additive secret sharing scheme to protect the privacy of the data which has no room to detect the corruptions. Therefore, message authenticate codes (MACs) are commonly attached to the additive secret sharing scheme to detect the corruptions, which are highly related to the concrete structure of the non-commutative ring.

In view of the above reasons, we aim to construct a dishonest majority MPC over a specific family of the non-commutative ring, the matrix ring. Matrix plays an essential role in PPML, which allows distrustful parties to train and evaluate different machine learning models [24,22,19,23]. It was observed in [11] that securely multiplying two $m \times m$ matrices in SPDZ protocol requires at least $O(m^{2.8})$ authenticated Beaver triples, which is prohibitively expensive if a machine learning task needs a large number and sizes of matrix multiplication. Thus, an MPC protocol specialized for matrix operations may greatly improve the efficiency of PPML. Moreover, some other non-commutative rings could be represented in the form of matrix rings. For instance, the quaternion ring is another non-commutative ring with practical applications, which plays a central role in computer graphics and aerospace due to its competence in describing the rotation in three-dimensional space.

In this work, we present a variant of SPDZ protocol whose secret is defined over matrix rings. Different from the classic SPDZ protocol, the MAC and global key of our protocol are vectors of length m and the secret of our protocol is an $m \times m$ matrix. Thus, the size of our MAC is negligible compared to the size of

[1] A subset of a non-commutative ring where the difference between any two elements in this subset is invertible.

our secret. Utilizing the matrix structure, our MPC protocol uses vector oblivious linear evaluation (VOLE) and random vector oblivious linear evaluation (RVOLE) as functionalities to authenticate the sharing and create the sextuple for securely computing multiplication gate in the online phase. The advantage of using VOLE and RVOLE is that these functionalities allow the computation over both the finite fields and the integer rings with almost the same performance. Thus, our MPC protocol can evaluate the circuits over any finite field and integer ring. As a comparison, the matrix tuple constructed in [11] can only be defined over integer rings or prime fields of large size to meet the security parameter as they resort to homomorphic encryption (BFV scheme) in their preprocessing phase. Moreover, our MPC protocol is very efficient compared to the classic SPDZ protocol. The classic SPDZ protocol needs $O(m^3)$ Beaver triples for securely computing a multiplication gate which incurs $O(n^2m^3 \log q)$ bits of communication complexity while our MPC protocol only requires $O(n^2m^{2.5} \log q)$ bits of communication complexity in the preprocessing phase to prepare a sextuple for multiplication gate. [2] In the online phase, our MPC protocol requires $O(m^2n \log q)$ bits of communication complexity to securely compute a multiplication gate which is as efficient as the MPC protocol in [11]. The size of the secret sharing is half the size of the secret sharing scheme in [11].

1.1 Our contribution

MAC for matrix rings. To authenticate a matrix $M \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$, we choose a uniformly random vector $\boldsymbol{v} \in \mathcal{M}_{m\times 1}(\mathbb{F}_q)$ as the global key and use the matrix-vector product $M\boldsymbol{v}$ as the MAC of a matrix M. The intuition of this matrix-vector product is to reduce the size of MAC by applying the batch check, i.e., each component of the MAC is the inner product of a row of M and the global key $\boldsymbol{v}$. If the adversary aims to forge a fake authenticated secret sharing, he needs to choose a nonzero matrix $E \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$ and a MAC $\boldsymbol{\delta} \in \mathcal{M}_{m\times 1}(\mathbb{F}_q)$ such that $E\boldsymbol{v} = \boldsymbol{\delta}$. Since E is a nonzero matrix, we assume that the i-th row of E is a nonzero vector $\boldsymbol{e}_i^T$. Then, we have $\boldsymbol{e}_i^T\boldsymbol{v} = \delta_i$ where δ_i is the i-th component of $\boldsymbol{\delta}$. Since the global key $\boldsymbol{v}$ is distributed uniformly at random, the adversary succeeds with probability at most $1/q$. In comparison, the previous MPC protocol in [11] chooses a random element $\alpha \in \mathbb{F}_q$ as the global key and uses the scalar-matrix product αM as the corresponding MAC. Therefore, our MAC is m times smaller than theirs. The sharing of the matrix M in our protocol is defined as $\langle M\rangle = ([M], [\![\boldsymbol{v}]\!], [\![M\boldsymbol{v}]\!])$[3] where $[M]$ is the additive sharing of M and $[\![\boldsymbol{v}]\!]$, $[\![M\boldsymbol{v}]\!]$ are the additive sharing of $\boldsymbol{v}$ and $M\boldsymbol{v}$ respectively.

The use of VOLE. Our protocol uses the vector oblivious linear evaluation (VOLE for short) to compute the matrix-vector product. We make use of the matrix structure to optimize the generation of correlated randomness. In the computation of MAC, two parties need to obliviously compute the product of a

[2] We assume the matrix multiplication requires m^3 number of multiplications.
[3] We use $[\cdot]$ and , $[\![\cdot]\!]$ to represent the sharing of a matrix and vector.

matrix M with a column vector $\boldsymbol{v}$, i.e., $\boldsymbol{u} + \boldsymbol{w} = M\boldsymbol{v}$. Observe that $M\boldsymbol{v}$ can be decomposed into the sum of m vectors $v_i\boldsymbol{m}_i$ where v_i is the i-th component of $\boldsymbol{v}$ and $\boldsymbol{m}_i$ is the i-th column of M. Two parties can invoke VOLE m times to obtain the shares $\boldsymbol{u}_i, \boldsymbol{w}_i$ with $\boldsymbol{u}_i + \boldsymbol{v}_i = v_i\boldsymbol{m}_i$. In contrast, we have to invoke m^2 OLEs to obliviously compute $M\boldsymbol{v}$, which is usually more expensive than VOLE.

The use of RVOLE. To further reduce the communication complexity, we use the random vector oblivious linear evaluation (RVOLE for short) to compute the product of two random matrices[4] whose communication complexity is the most expensive in the preprocessing phase. The RVOLE used in this paper can be implemented by a pseudorandom correlated generator, which allows two parties to take a pair of short, correlated seeds, and expand them to produce a much larger correlated randomness. Thus, the invocation of RVOLE has sub-linear communication complexity.

Multiplication sextuple. The biggest challenge of MPC protocol over matrix rings is that the product of two matrices is not commutative. This prevents us from applying the Beaver triple straightforwardly. This problem also appears in [18]. Their solution is to use two types of secret sharings with left linearity and right linearity respectively and transform the type of secret sharing by consuming a double sharing, which is a pair of sharings associated with the same secret and different types. In our case, since our MAC has the form $X\boldsymbol{v}$, our secret sharing only allows left multiplication, i.e., all parties can only locally compute $A\langle M\rangle = \langle AM\rangle$. In this work, we propose a multiplication sextuple to circumvent this obstacle. Let $\langle X\rangle$ and $\langle Y\rangle$ be the sharings of matrix X and Y respectively. We prepare a sextuple $(\langle A\rangle, \langle A^T\rangle, \langle B\rangle, \langle C\rangle, \langle R\rangle, \langle R^T\rangle)$ where A, B, R are random matrices, A^T and R^T are the transpose of A and R, and $C = AB$. All parties partially open $\langle X\rangle - \langle A\rangle$ and $\langle Y\rangle - \langle B\rangle$ to D and E. The technique of Beaver triple requires all parties to locally compute $D\langle B\rangle + \langle A\rangle E + \langle C\rangle + DE$. However, as we mentioned above, it is impossible to locally compute the right multiplication $\langle A\rangle E$. To overcome this obstacle, all parties are required to locally compute $E^T\langle A^T\rangle - \langle R^T\rangle$ and partially open it to F by using the sharing $\langle A^T\rangle$ and $\langle R^T\rangle$. Then, all parties locally compute $F^T + \langle R\rangle = \langle AE\rangle$ by observing $F^T = (E^TA^T - R^T)^T = AE - R$. This completes the multiplication gate.

Function dependent preprocessing. The evaluation of a single multiplication gate in our MPC protocol needs two rounds and three broadcasts. Inspired by [7,18], we introduce function dependent preprocessing to improve the round and communication complexity. After the application of function dependent preprocessing, the evaluation of a multiplication gate only needs one round and two broadcasts. Since this improvement is not the focus of our paper, we take a brief overview of it in Section C.

[4] In our MPC protocol, the RVOLE is applied to the random matrix while VOLE is applied to the deterministic matrix.

Migration to $\mathcal{M}_{m\times m}(\mathbb{Z}_{p^k})$. The matrix in our MPC protocol can be defined over small fields and rings as well. Let us first consider the small fields and then generalize to the integer rings. The idea is to replace a global key of a vector in $\mathcal{M}_{m\times 1}(\mathbb{F}_q)$ with a global key of a matrix in $\mathcal{M}_{m\times \ell}(\mathbb{F}_q)$. The reason is that the adversary succeeds with probability $1/q$ if our MPC protocol is defined over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$. To reduce the error probability, we increase the size of the global key and MAC. Observe that $XV = \Delta$ where $V \in \mathcal{M}_{m\times \ell}(\mathbb{F}_q)$ is the MAC and $X \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$ is the secret. Therefore, each column of the global key is used to verify the correctness of the secret and we verify our secret X with ℓ equations instead of 1. The error probability will be reduced to $1/q^\ell$ while the size of MAC is still negligible compared to the size of our secret assuming m tends to infinity. In this sense, our MPC protocol can be defined over $\mathcal{M}_{m\times m}(\mathbb{Z}_p)$ with prime number p. Then, we can generalize our MPC protocol to a protocol over $\mathcal{M}_{m\times m}(\mathbb{Z}_{p^k})$ in a straightforward way. One can find the details in Section 6.

1.2 Overview of our technique

We first present our MPC protocol over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ with large q. Then, in Section 6, we show how to generalize this construction to a small field and ring case with slight modifications. As we have mentioned above, the authenticated sharing of our protocol is $\langle M\rangle = ([M], [\![\boldsymbol{v}]\!], [\![M\boldsymbol{v}]\!])$. We use a random vector $\boldsymbol{v}$ as our global key. The MAC of our matrix is the product of a matrix with the global key $\boldsymbol{v}$. The idea of our MAC comes from the batch check. A random vector can be used to verify the correctness of a vector of the same length by taking the inner product of these two vectors. Thus, to verify the correctness of an $m \times m$ matrix, we only need a MAC of size m. On the contrary, the classic SPDZ protocol requires MAC of size m^2 to verify an $m \times m$ matrix. Another merit of this sharing can be found in the use of VOLE and RVOLE which we have already discussed in the subsection 1.1.

In the preprocessing phase, our MPC protocol prepares sextuples of the form $(\langle A\rangle, \langle A^T\rangle, \langle B\rangle, \langle C\rangle, \langle R\rangle, \langle R^T\rangle)$ with random matrices $A, B, R \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$ and $C = AB$. We break this protocol into two procedures, π_{Mult} and π_{Double}. We also present a protocol Π_{Auth} to generate the authenticated sharing. Protocol Π_{Auth} uses functionality VOLE to create the MAC and takes the random linear combination to verify the correctness of sharings. The use of VOLE can be found in the previous subsection.

Procedure π_{Mult} produces a triple $(\langle A\rangle, \langle B\rangle, \langle C\rangle)$. We want to compute $[C]$ from $[A] = \left(A^{(1)}, \ldots, A^{(n)}\right)$ and $[B] = \left(B^{(1)}, \ldots, B^{(n)}\right)$. Observe that $C = AB = \left(\sum_{i=1}^n A^{(i)}\right)\left(\sum_{i=1}^n B^{(i)}\right)$. The additive sharing of cross terms $A^{(i)}B^{(j)}$ and $A^{(j)}B^{(i)}$ can be computed by P_i and P_j. The product of two $m \times m$ matrices can be decomposed into m matrix-vector multiplications, i.e., $AB = (A\boldsymbol{b}_1, \ldots, A\boldsymbol{b}_m)$. This implies that we only need to invoke m^2 times VOLE to complete this work. Moreover, to further reduce the communication complexity, we use RVOLE instead of VOLE for this job. We create seeds to generate the random matrix $A^{(i)}$ and reuse these seeds as inputs for the instances of RVOLE. Similar to

[20], we prepare τ authenticated sharings $\langle X_1\rangle, \ldots, \langle X_\tau\rangle$ and take the random linear combinations $\langle A\rangle = \sum_{i=1}^{\tau} r_i\langle X_i\rangle$ to prevent the leakage of bits of the matrices $X_1, \ldots, X_\tau$. Thus, instead of generating the sharing of AB, we need to generate the sharing of $X_1B, \ldots, X_\tau B$. We merge $X_1, \ldots, X_\tau$ into one matrix $X = (X_1^T|X_2^T \cdots |X_\tau^T)^T \in \mathcal{M}_{\tau m \times m}(\mathbb{F}_q)$ and invoke m^2 times RVOLE to obtain $[X_1B], \ldots, [X_\tau B]$. Then, we take random linear combinations to obtain two pairs $([A], [C]), ([A'], [C'])$ and apply protocol Π_{Auth} to compute the MAC of these sharings. By taking a random linear combination of the form $\chi\langle A\rangle - \langle A'\rangle$, we can verify the product relation and output the triple $(\langle A\rangle, \langle B\rangle, \langle C\rangle)$.

Procedure π_{Double} takes inputs $\langle A_i\rangle, i \in [2\ell]$ and outputs pairs of authenticated sharing $\langle A_i\rangle, \langle A_i^T\rangle, i \in [2\ell]$ for ℓ multiplication gates. The idea is to first locally compute $[A_i^T]$ from $[A_i]$ by applying the transpose to each share in $[A_i]$. Then, we apply protocol Π_{Auth} to create the authenticated sharing $\langle A_i^T\rangle$. To check the transpose relation, we generate a pair of authenticated sharing of random matrix A_0, A_0^T and sacrifice this pair by taking the random linear combination

$$\langle C\rangle = \sum_{i=1}^{2\ell} r_i\langle A_i\rangle + \langle A_0\rangle \quad \langle D\rangle = \sum_{i=1}^{2\ell} r_i\langle A_i^T\rangle + \langle A_0^T\rangle$$

It must hold that $C = D^T$. Then, this procedure will output pairs of authenticated sharing $\langle A_i\rangle, \langle A_i^T\rangle, i \in [2\ell]$.

In the online phase, our MPC protocol can securely compute the addition and multiplication gate. The addition gate can be locally computed without interaction. To compute the multiplication gate, we need a sextuple prepared in the preprocessing phase. This sextuple can help us to circumvent the obstacle that the product of two matrices is non-commutative. One can find the details in the subsection 1.1.

1.3 Related work

There are few MPC protocols handling the matrix operations. Escudero and Soria-Vazquez [17] presented an honest majority MPC protocol over matrix rings. One of the biggest challenges in their protocol is to construct Shamir secret sharing scheme over non-commutative rings. They constructed a subset of matrices as the evaluation points such that these matrices are commutative. Based on this subset of matrices, they presented the encoding and error correction algorithm for this Shamir secret sharing scheme. Since our MPC protocol is secure in the presence of dishonest majority, our building block is an additive secret sharing scheme. The sharing and reconstruction algorithm can be straightforwardly generalized from the commutative case. However, we need a MAC to verify the correctness of our sharing whose idea can be dated back to SPDZ protocol [14]. In our protocol, the global key and the MAC are vectors instead of elements. Thus, the MAC of our protocol is negligible compared to the size of the secret.

The most relevant work is due to [11] which presented a variant of SPDZ protocol over matrix rings. They mimic the classic SPDZ protocol to use a single

element as the global key to create the MAC of the matrix. Thus, the size of MAC in their protocol is as big as the secret. In the preprocessing phase, they apply the homomorphic matrix encryption [19] which is based on a variant of BFV scheme [9] to create the matrix triple. Their SPDZ protocol over matrix rings turns out to be very efficient compared to the classic SPDZ protocol handling the matrix operations as the entry-wise operations. However, the use of BFV scheme implies that their matrix must be defined over a ring $\mathbb{Z}_q$ with large enough q.[5] In comparison, our preprocessing phase uses the vector oblivious linear evaluation and the random oblivious linear evaluation as the functionalities, and thus our MPC protocol can be defined over any finite field or ring with almost the same performance. Moreover, the communication complexity of our MPC protocol is as efficient as theirs.

1.4 Organization of the paper

The paper is organized as follows. In Section 2, we present basic notations and definitions. In Section 3, we present the online phase of our MPC protocol. In Section 4, we present Protocol Π_{Auth} which outputs authenticated sharings. In Section 5, we present the preprocessing phase of our MPC protocol. In Section 6, we extend our MPC protocol to matrix rings over small fields and integer rings. In Section 7, we analyze the communication complexity of our MPC protocol and compare it with other dishonest majority MPC protocols over matrix rings. The missing proofs and protocols can be found in Section A.

2 Preliminaries

2.1 Basic Notation

We use the capital letter M to represent a matrix and bold small letter $\boldsymbol{v}$ to represent a column vector. The transpose of a matrix M is M^T and the transpose of a vector $\boldsymbol{v}$ is $\boldsymbol{v}^T$. Write $M = (m_{i,j})_{n\times n}$ and the j-th column of M is denoted by $\boldsymbol{m}_j = (m_{1,j}, \ldots, m_{n,j})$. For a vector $\boldsymbol{v}$, denote by v_i the i-th component of $\boldsymbol{v}$, i.e., $\boldsymbol{v} = (v_1, \ldots, v_n)^T$. Let $\mathcal{M}_{r\times c}(\mathbb{F}_q)$ be the collection of $r \times c$ matrices over $\mathbb{F}_q$.

Throughout the paper, the security parameter of MPC protocol is κ. Let $\mathbb{F}_q$ be the finite field of size q and $\mathbb{Z}_{p^k}$ be the ring of integers modulo p^k. We denote by $x \xleftarrow{\$} \mathcal{X}$ a variable x uniformly sampling from a finite set $\mathcal{X}$. Let $[N] = \{1, \cdots, N\}$.

2.2 Multiparty Computation

The set of parties in our MPC protocol is $\{P_1, \cdots, P_n\}$. We study the setting of dishonest majority, where at most $n-1$ parties are corrupted by the adversary. The adversary is static and malicious, which means that the set of corrupted

[5] In their experiment, $\log q = 128$.

parties is determined before the execution of protocol and corrupted parties can arbitrarily deviate from the protocol.

The security of our protocol is proved under Canetti's Universal Composability(UC) framework [10]. A protocol Π securely instantiates a functionality $\mathcal{F}$ if there exists a simulator that interacts with the adversary (or more formally, *environment*) such that he can distinguish the ideal world and real world with only negligible probability. The composability of UC framework enables us to construct our protocol in *hybrid* model, which means that protocol Π instantiates functionality $\mathcal{F}$ with access to another functionality $\mathcal{F}'$. In this case, Π instantiates $\mathcal{F}$ in the $\mathcal{F}'$-hybrid model. Different from a protocol Π which is associated with an ideal functionality and has simulation-based proof, we use π to represent a procedure, which acts as a subroutine of protocols, and has no related functionality or simulation-based proof.

We assume the private and authenticated channels between any pair of parties and a broadcast channel. Our MPC protocol achieves security with abort since the majority of parties are dishonest. In the ideal world, the functionality waits for a signal from the adversary before delivery of outputs. If the signal is Abort, all honest parties abort. Otherwise, the signal is OK, the functionality sends correct outputs to all honest parties. In the real world, when we say a party aborts, this party sends an Abort signal through the broadcast channel and all honest parties abort.

3 Online phase

We begin this section by introducing the authenticated secret sharing of a matrix, which is the building block of our MPC protocol. Then, we describe the functionalities required for our online phase. We present the Protocol Π_{online} to securely instantiate MPC functionality $\mathcal{F}_{MPC}$ in the ($\mathcal{F}_{Prep}$, $\mathcal{F}_{Coin}$)-hybrid model, where $\mathcal{F}_{Prep}$ generates correlated randomness in offline phase and $\mathcal{F}_{Coin}$ generates public random field elements. The implementation of $\mathcal{F}_{Prep}$ is presented in the Section 5.

3.1 Authenticated Secret Sharing

In the dishonest majority setting, additive secret sharing alone is not secure against malicious adversary. Similar to [13], we use a uniformly random global key to generate a MAC for each share to enhance security. The difference is that the global key and MACs are not elements in the matrix ring $\mathcal{M}_{m\times m}(\mathbb{F}_q)$. Instead, they are vectors of length m over $\mathbb{F}_q$.

Notations. We use $[\cdot]$ and $[\![\cdot]\!]$ to denote an additive secret sharing over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ and $\mathcal{M}_{m\times 1}(\mathbb{F}_q)$[6], respectively. An authenticated secret sharing $\langle X\rangle$ is a triple

[6] Here we use notion $\mathcal{M}_{m\times 1}(\mathbb{F}_q)$ instead of $\mathbb{F}_q^m$ in order to show that the global key and MACs can be generalized to matrix.

$([X], [\![\boldsymbol{v}]\!], [\![X\boldsymbol{v}]\!])$, where $X \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$ is the secret, $\boldsymbol{v} \xleftarrow{\$} \mathcal{M}_{m\times 1}(\mathbb{F}_q)$ is the global key and $X\boldsymbol{v} \in \mathcal{M}_{m\times 1}(\mathbb{F}_q)$ is the MAC of the secret. More precisely, $[X] = \left(X^{(1)}, \cdots, X^{(n)}\right)$, $[\![\boldsymbol{v}]\!] = \left(\boldsymbol{v}^{(1)}, \cdots, \boldsymbol{v}^{(n)}\right)$ and $([\![X\boldsymbol{v}]\!]) = \left(\boldsymbol{m}^{(1)}(X), \cdots, \boldsymbol{m}^{(n)}(X)\right)$ with

$$X = \sum_{i=1}^{n} X^{(i)}, \boldsymbol{v} = \sum_{i=1}^{n} \boldsymbol{v}^{(i)}, X\boldsymbol{v} = \sum_{i=1}^{n} \boldsymbol{m}^{(i)}(X).$$

where party P_i holds random share $X^{(i)}$ of secret X, key share $\boldsymbol{v}^{(i)}$ and MAC share $\boldsymbol{m}^{(i)}(X)$.

Local operations. For simplicity, we use "linear" to refer to "$\mathcal{M}_{m\times m}(\mathbb{F}_q)$-linear". Scheme $[\cdot]$ is both *left linear* and *right linear* due to distribute law of matrix rings. However, scheme $\langle\cdot\rangle$ is only *left linear*. Given an authenticated secret sharing $\langle X\rangle$ and a public matrix $A \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$, all parties could left multiply A to $[\![X\boldsymbol{v}]\!]$ to obtain $[\![AX\boldsymbol{v}]\!]$, but it is not possible to obtain $[\![XA\boldsymbol{v}]\!]$ with only local operations. To securely left multiply a matrix A with $\langle X\rangle$, all parties locally compute

$$A\langle X\rangle = \langle AX\rangle = ([AX], [\![\boldsymbol{v}]\!], [\![AX\boldsymbol{v}]\!])$$

with $[AX] = (AX^{(1)}, \ldots, AX^{(n)})$ and $[\![AX\boldsymbol{v}]\!] = (A\boldsymbol{m}^{(1)}(X), \ldots, A\boldsymbol{m}^{(n)}(X))$. To securely compute the sum of $\langle X\rangle$ and $\langle Y\rangle$, all parties locally compute

$$\langle X\rangle + \langle Y\rangle = \langle X+Y\rangle = ([X+Y], [\![\boldsymbol{v}]\!], [\![(X+Y)\boldsymbol{v}]\!])$$

with $[X+Y] = (X^{(1)}+Y^{(1)}, \ldots, X^{(n)}+Y^{(n)})$ and $[\![(X+Y)\boldsymbol{v}]\!] = \big(\boldsymbol{m}^{(1)}(X) + \boldsymbol{m}^{(i)}(Y), \ldots, \boldsymbol{m}^{(n)}(X) + \boldsymbol{m}^{(n)}(Y)\big)$. To securely add a public matrix A with $\langle X\rangle$, all parties locally compute

$$[X+A] = (X^{(1)}+A, X^{(2)}, \ldots, X^{(n)}), \boldsymbol{m}^{(i)}(X+A) = \boldsymbol{m}^{(i)}(X) + A\boldsymbol{v}^{(i)}$$

Let $\langle X+A\rangle = ([X+A], [\![\boldsymbol{v}]\!], [\![(X+A)\boldsymbol{v}]\!])$ be defined above. The affine operation can be found in Procedure π_{Aff} in Section A.

Opening and checking. To partially open an authenticate secret sharing $\langle Y\rangle = ([Y], [\![\boldsymbol{v}]\!], [\![Y\boldsymbol{v}]\!])$, all parties send their shares of $[Y]$ to P_1, who can reconstruct the secret and send the result Y' to other parties. To check the correctness of the opening, all parties could locally compute $[\![\boldsymbol{\sigma}]\!] = [\![Y\boldsymbol{v}]\!] - Y'[\![\boldsymbol{v}]\!]$, and broadcast the share of this value via a simultaneous message channel. The parties abort if the reconstructed value is not $\mathbf{0}$. The probability that a fake authenticated secret sharing passes the verification is $1/q$. These two procedures can be found in Section A.

Multiplication. In dishonest majority MPC protocols, correlated randomness generated in offline phase could assist the computation of multiplications. Beaver

triple [6] is a common technique in MPC protocols, which transforms execution of multiplications to broadcasts and linear operations. However, we can not adapt Beaver triple directly due to the lack of commutativity of matrix rings.

To multiply two authenticated sharings $\langle X \rangle$ and $\langle Y \rangle$, all parties prepare a Beaver triple $(\langle A \rangle, \langle B \rangle, \langle C \rangle)$ with $C = AB$ during offline phase. All parties partially open $D \leftarrow \langle X \rangle - \langle A \rangle$ and $E \leftarrow \langle Y \rangle - \langle B \rangle$. The sharing of $Z = XY$ could be represented as:

$$[Z] = [C] + D[B] + [A]E + DE$$
$$[\![Z\boldsymbol{v}]\!] = [\![C\boldsymbol{v}]\!] + D[\![B\boldsymbol{v}]\!] + [\![AE\boldsymbol{v}]\!] + DE[\![\boldsymbol{v}]\!]$$

We observe that all items except $[\![AE\boldsymbol{v}]\!]$ could be locally computed with linear operations. To compute MAC share $[\![AEv]\!]$, we follow the paradigm of "mask-open-unmask". We choose a random sharing $[R]$ as the mask of $[A]E$. However, when opening the masked value $[A]E - [R]$, we cannot guarantee the correctness due to the lack of MAC. Therefore, we prepare two additional authenticated sharings $(\langle A^T \rangle, \langle R^T \rangle)$ and partially open the transpose $\langle F \rangle = E^T \langle A^T \rangle - \langle R^T \rangle$ instead. Therefore, to execute a multiplication, all parties need to prepare a *multiplication sextuple* $(\langle A \rangle, \langle A^T \rangle, \langle B \rangle, \langle C \rangle, \langle R \rangle, \langle R^T \rangle)$ where $A, B, R \xleftarrow{\$} \mathcal{M}_{m\times m}(\mathbb{F}_q)$ and $C = AB$.

3.2 Required Functionalities

Here we present some functionalities. The functionality $\mathcal{F}_{MPC}$ enables the parties to input secrets in $\mathcal{M}_{m\times m}(\mathbb{F}_q)$, perform linear operations and multiplications on these values, and output results. The functionality $\mathcal{F}_{Prep}$ aims to prepare correlated randomness for $\mathcal{F}_{MPC}$.

Authentication functionality $\boldsymbol{\mathcal{F}_{Auth}}$. This functionality allows parties to generate the shares of global key $\boldsymbol{v}$ and transform an additive secret sharing $[X]$ to an authenticated secret sharing $\langle X \rangle$. Although we do not call $\mathcal{F}_{Auth}$ directly, $\mathcal{F}_{Auth}$ is contained in $\mathcal{F}_{Prep}$. $\mathcal{F}_{Auth}$ can be found in Functionality 1 below.

Functionality 1: $\mathcal{F}_{Auth}$

The functionality maintains a dictionary Val, which keeps a track of authenticated elements in $\mathcal{M}_{m\times m}(\mathbb{F}_q)$. Let $\mathcal{C}$ be the set of corrupted parties.

- **Initialize**: On receiving (Init) from all parties, sample random vector $\boldsymbol{v}^{(i)} \leftarrow \mathcal{M}_{m\times 1}(\mathbb{F}_q)$ for $i \notin \mathcal{C}$ and receive $\boldsymbol{v}^{(i)}$ from adversary for $i \in \mathcal{C}$. Store the global key $\boldsymbol{v} = \sum_{i=1}^{n} \boldsymbol{v}^{(i)}$ and send $\boldsymbol{v}^{(i)}$ to P_i.
- **Authenticate**: On receiving $(\mathsf{Auth}, [X])$ from each party P_i, where $[X]$ is an additive sharing over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$:
  1. Compute the MAC $\boldsymbol{m}(X) = X\boldsymbol{v}$.
  2. Wait for $\left\{\boldsymbol{m}^{(i)}(X)\right\}_{i\in\mathcal{C}}$ from adversary and sample $\left\{\boldsymbol{m}^{(i)}(X)\right\}_{i\notin\mathcal{C}}$ subject to $\sum_{i=1}^{n} \boldsymbol{m}^{(i)}(X) = \boldsymbol{m}(X)$.

3. Wait for message from adversary. If the message is OK, send $\boldsymbol{m}^{(i)}(X)$ to each party P_i. If the message is Abort, the functionality aborts.

Preprocessing functionality $\mathcal{F}_{\boldsymbol{Prep}}$. This functionality produces random sharings for input gates and multiplication sextuples for multiplication gates. $\mathcal{F}_{Prep}$ can be found in Functionality 2 below.

Functionality 2: $\mathcal{F}_{Prep}$

The functionality has all the same commands in $\mathcal{F}_{Auth}$, with following additional commands:

- **Input**: On input $(\mathsf{InputPrep}, P_i)$ from all parties, sample $R \xleftarrow{\$} \mathcal{M}_{m\times m}(\mathbb{F}_q)$ and generate its authenticated sharing $\langle R\rangle$. Output R to P_i and $\left(R^{(j)}, \boldsymbol{m}^{(j)}(R)\right)$ to P_j for all $j \neq i$.
- **Sextuple**: On input (Tuple) from all parties, sample $A, B, R \xleftarrow{\$} \mathcal{M}_{m\times m}(\mathbb{F}_q)$ and compute $C = AB$. Generate authenticated sharings $(\langle A\rangle, \langle A^T\rangle, \langle B\rangle, \langle C\rangle, \langle R\rangle, \langle R^T\rangle)$.

Multiparty computation functionality $\mathcal{F}_{\boldsymbol{MPC}}$. This functionality is the goal of our MPC protocol. $\mathcal{F}_{MPC}$ is can be found in Functionality 3 below.

Functionality 3: $\mathcal{F}_{MPC}$

The functionality maintains a dictionary Val, which keeps a track of authenticated elements in $\mathcal{M}_{m\times m}(\mathbb{F}_q)$.

- **Initialize**: On input (Init) from all parties, set the global key $[\![\boldsymbol{v}]\!]$ and obtain sufficient preprocessing data.
- **Input**: On input $(\mathsf{Input}, \mathtt{id}, X, P_i)$ from P_i and $(\mathsf{Input}, \mathtt{id}, P_i)$ from all other parties, store $\mathsf{Val}[\mathtt{id}] = X$.
- **Addition**: On input $(\mathsf{Add}, \mathtt{id}, \mathtt{id}_1, \mathtt{id}_2)$ from all parties, compute $Z = \mathsf{Val}[\mathtt{id}_1] + \mathsf{Val}[\mathtt{id}_2]$ and store $\mathsf{Val}[\mathtt{id}] = Z$.
- **Public matrix multiplication**: On input $(\mathsf{PubMul}, \mathtt{id}, A)$, compute $Z = A\mathsf{Val}[\mathtt{id}]$ and store $\mathsf{Val}[\mathtt{id}] = Z$.
- **Multiplication**: On input $(\mathsf{Mult}, \mathtt{id}, (\mathtt{id}_1, \mathtt{id}_2))$ from all parties, compute $Z = \mathsf{Val}[\mathtt{id}_1]\mathsf{Val}[\mathtt{id}_2]$ and store $\mathsf{Val}[\mathtt{id}] = Z$.
- **Check openings**: On input $(\mathsf{Check}, (\mathtt{id}_1, \cdots, \mathtt{id}_\ell), (X'_1, \cdots, X'_\ell))$ from all parties, wait for a signal for the adversary. If the adversary sends OK and $\mathsf{Val}[\mathtt{id}_j] = X'_j$ for $j \in [\ell]$, return OK to all honest parties. Otherwise, return Abort to all honest parties.
- **Output**: On input $(\mathsf{Output}, \mathtt{id})$ from all parties, the functionality retrieves $Y = \mathsf{Val}[\mathtt{id}]$ and sends Y to the adversary if $\mathsf{Val}[\mathtt{id}] \neq \emptyset$. If the adversary sends Abort then the functionality aborts, otherwise it delivers Y to all parties.

Coin tossing functionality $\mathcal{F}_{Coin}$. This functionality generates a uniformly random element in $\mathbb{F}_q$ for all parties.

Functionality 4: $\mathcal{F}_{Coin}$

Upon receiving (Coin) from all parties, sample $r \xleftarrow{\$} \mathbb{F}_q$ and send r to all parties.

3.3 Instantiation of $\mathcal{F}_{MPC}$

The protocol Π_{online} instantiates $\mathcal{F}_{MPC}$ in the ($\mathcal{F}_{Prep}$,$\mathcal{F}_{Coin}$)-hybrid model, with statistical security parameter κ. The random shares and multiplication sextuples produced in $\mathcal{F}_{Prep}$ will be used in Input and Mult commands, respectively.

Protocol 1: Π_{Online}

The parties maintain a dictionary Val for authenticated values.

- **Initialize**: The parties call $\mathcal{F}_{Prep}$ as follows:
  1. On input (Init) to get global key $[\![\boldsymbol{v}]\!]$
  2. On input (InputPrep, P_i) to prepare a random authenticated sharing $\langle R\rangle$ for each input gate, where the input provider P_i learns R
  3. On input (Tuple) to prepare a multiplication sextuple $(\langle A\rangle, \langle A^T\rangle, \langle B\rangle, \langle C\rangle, \langle R\rangle, \langle R^T\rangle)$ for each multiplication gate
- **Input**: If P_i receives (Input, id, X, P_i) and other parties receive (Input, id, P_i), execute following operations:
  1. P_i broadcasts $A = X - R$, where $\langle R\rangle$ is an unused input mask
  2. All parties locally compute $\langle X\rangle = \langle R\rangle + A$ and store Val[id] $= \langle X\rangle$.
- **Addition**: If all parties receive (Add, $\mathtt{id}, \mathtt{id}_1, \mathtt{id}_2$), retrieve $\langle X\rangle =$ Val[$\mathtt{id}_1$] and $\langle Y\rangle =$ Val[$\mathtt{id}_2$], locally compute $\langle Z\rangle = \langle X\rangle + \langle Y\rangle$ and set Val[id] $= \langle Z\rangle$.
- **Public matrix multiplication**: If all parties receive (PubMul, id, A), retrieve $\langle X\rangle =$ Val[id], locally compute $\langle Z\rangle = A\langle X\rangle$ and set Val[id] $= \langle Z\rangle$.
- **Multiplication**: If all parties receive (Mult, $\mathtt{id}, (\mathtt{id}_1, \mathtt{id}_2)$), retrieve $\langle X\rangle =$ Val[$\mathtt{id}_1$] and $\langle Y\rangle =$ Val[$\mathtt{id}_2$] and execute following operations:
  1. Choose an unused multiplication sextuple $(\langle A\rangle, \langle A^T\rangle, \langle B\rangle, \langle C\rangle, \langle R\rangle, \langle R^T\rangle)$.
  2. All parties locally compute $\langle D\rangle \leftarrow \langle X\rangle - \langle A\rangle$ and $\langle E\rangle \leftarrow \langle Y\rangle - \langle B\rangle$.
  3. All parties invoke $D \leftarrow \pi_{Open}(\langle D\rangle)$ and $E \leftarrow \pi_{Open}(\langle E\rangle)$.
  4. All parties locally compute $\langle F\rangle \leftarrow E^T\langle A^T\rangle - \langle R^T\rangle$ and invoke $F \leftarrow \pi_{Open}(\langle F\rangle)$
  5. All parties locally compute $\langle Z\rangle = \langle C\rangle + D\langle B\rangle + \langle R\rangle + DE + F^T$ and set Val[id] $= \langle Z\rangle$.
- **Check openings**: If all parties receive (Check, $(\mathtt{id}_1, \cdots, \mathtt{id}_\ell)$, $(X'_1, \cdots, X'_\ell)$), retrieve $\langle X_j\rangle =$ Val[$\mathtt{id}_j$] for $j \in [\ell]$ and execute following operations:
  1. Call $\mathcal{F}_{Coin}$ ℓ times to sample $r_1, \cdots, r_\ell \xleftarrow{\$} \mathbb{F}_q$.
  2. All parties locally compute $\langle Y\rangle \leftarrow \sum_{j=1}^{\ell} r_j\langle X_j\rangle$.
  3. All parties locally compute $Y' = \sum_{j=1}^{\ell} r_j X'_j$.

4. All parties invoke $\pi_{Check}(Y', \langle Y \rangle)$.

– **Output:** If all parties receive $(\mathsf{Output}, \mathtt{id})$ and retrieve $\langle Y \rangle = \mathsf{Val}[\mathtt{id}]$:
1. All parties invoke Check command to check all the opened values in the online phase so far.
2. If this does not abort, the parties partially open $\langle Y \rangle$ to obtains Y'.
3. All parties invoke $\pi_{Check}(Y', \langle Y \rangle)$. If this procedure passes, output Y'.

Theorem 1. *Protocol Π_{Online} securely implements $\mathcal{F}_{MPC}$ in the $(\mathcal{F}_{Prep}, \mathcal{F}_{Coin})$-hybrid model.*

Proof (Sketch). A full-fledged simulation-based proof is presented in Section B.1. Here we restrict ourselves to the core idea of the proof. For the case of Init command, it is easy to see that the shares of the global key are prepared for all parties on both Π_{Online} and $\mathcal{F}_{MPC}$. In the Input command, the value stored by $\mathcal{F}_{MPC}$ corresponds to the value stored by Π_{Online}, which can be seen authenticated through mask of the random share.

The case of Add and PubMul is easy since these steps only consist of local computations which can be simulated trivially. To analyze Mult command, we should take three values into consideration. The correctness of the multiplication step in $\mathcal{F}_{MPC}$ is easy to be verified. The parties will get a tuple $(\langle A \rangle, \langle A^T \rangle, \langle B \rangle, \langle C \rangle, \langle R \rangle, \langle R^T \rangle)$ before computing the product Z of two stored values $X, Y \in \mathcal{M}_{m \times m}(\mathbb{F}_q)$. Then the protocol opens $D \leftarrow X - A$, $E \leftarrow Y - B$ and $F \leftarrow E^T A^T - R^T$. The parties need to compute $Z = C + DB + AE + DE$, where AE can not be directly obtained since $\langle \cdot \rangle$ has no right linearity. Note that $F^T = AE - R$ and we could reconstruct $AE = F^T + R$.

The above assumes that D, E and F are opened correctly, but this can be assumed to be the case since, if this does not hold, this will be detected in when the Check command is issued. That is why we open $F \leftarrow E^T A^T - R^T$ instead of $AE - R$. Due to the left linearity of the scheme $\langle \cdot \rangle$, it is more straightforward to check the correctness of $E^T \langle A^T \rangle$ than $\langle A \rangle E$. Moreover, the adversary does not learn sensitive information since the values A, B, R^T perfectly mask the values X, Y and $E^T A^T$, respectively. Finally, in the Check and Output command, we can prove that a fake authenticated secret sharing will pass the verification with a negligible probability due to the following game which also appears in [14].

1. The challenger generates the secret key $\boldsymbol{v} \xleftarrow{\$} \mathcal{M}_{m \times 1}(\mathbb{F}_q)$ and MACs $\boldsymbol{\gamma}_i = X_i \boldsymbol{v}$ for $i \in [\ell]$ and sends $X_1, \ldots, X_\ell$ to the adversary.
2. The adversary sends back $X'_1, \ldots, X'_\ell$.
3. The challenger generates the random values $r_1, \ldots, r_\ell \in \mathbb{F}_q$.
4. The adversary provides an error $\boldsymbol{\delta} = (\delta_1, \ldots, \delta_m)^T$.
5. The adversary checks that $\{\sum_{i=1}^{\ell} r_i (X_i - X'_i)\} \boldsymbol{v} = \boldsymbol{\delta}$

The adversary wins the game if the check passes and exists $X_i - X'_i \neq 0$. The second step of the game reveals that corrupted players have the option to lie about the secret shares they opened during the execution of the protocol. $\boldsymbol{\delta}$

models the fact that the adversary is allowed to introduce errors on the MAC. Suppose $\sum_{i=1}^{\ell} r_i(X_i - X_i')$ is not an all-zero matrix and let the nonzero row be $(x_{a,1}, \ldots, x_{a,m})$. We have $\delta_a = \sum_{j=1}^{m} x_{a,j} v_j$. Since $\boldsymbol{v} = (v_1, \ldots, v_m)$ is kept secret from the adversary, the adversary wins the game with the probability at most $1/q$. Now we proceed to the case $\sum_{i=1}^{\ell} r_i(X_i - X_i') = 0$. Because $r_1, \ldots, r_\ell$ are random elements, the probability that $\sum_{i=1}^{\ell} r_i E_i = 0$ for not all-zero matrix E_i is at most $1/q$. Thus, the adversary wins this game with probability at most $1/q$.

4 Authentication

In this section, we show how to authenticate an additive secret sharing. We first introduce a cryptographic primitive VOLE and then show how to generate the MAC share by invoking the VOLE.

4.1 Required functionalities

Vector oblivious linear evaluation functionality $\mathcal{F}_{VOLE}$. We first define the essential functionality $\mathcal{F}_{VOLE}$ over $\mathbb{F}_q$ as Functionality 5. A VOLE is a two-party functionality between P_A and P_B, which takes as input a vector $\boldsymbol{x}$ from the sender P_A and a scalar v from the receiver P_B, then randomly samples a vector $\boldsymbol{u}$ and computes $\boldsymbol{w} = v\boldsymbol{x} + \boldsymbol{u}$. In our work we need to instantiate several instances of VOLE, therefore we associate a unique identifier *sid* to each instance[7]. The efficient instantiation of $\mathcal{F}_{VOLE}$ can be found in [3,4].

Functionality 5: $\mathcal{F}_{VOLE}^{sid}$

The functionality runs between sender P_A and receiver P_B. The **Initialize** step is run once first and the **Multiply** step could be run arbitrarily many times.

- **Initialize**: Upon receiving $v \in \mathbb{F}_q$ from P_B, store v.
- **Multiply**: Upon receiving $\boldsymbol{x} \in \mathbb{F}_q^m$ from P_A:
  1. Sample $\boldsymbol{u} \xleftarrow{\$} \mathbb{F}_q^m$. If P_A is corrupted, receive $\boldsymbol{u}$ from adversary.
  2. Compute $\boldsymbol{w} = v\boldsymbol{x} + \boldsymbol{u}$. If P_B is corrupted, receive $\boldsymbol{w}$ from adversary and recompute $\boldsymbol{u} = v\boldsymbol{x} - \boldsymbol{w}$.
  3. Output $\boldsymbol{u}$ to P_A and $\boldsymbol{w}$ to P_B.

4.2 Instantiation of $\mathcal{F}_{Auth}$

Now we proceed to generate MAC shares. Each party P_i randomly samples the global key share $\boldsymbol{v}^{(i)}$ when command Init is called. To authenticate a given share

[7] The unique identifier *sid* works only for a pair of parties and is not a global identifier in n-party setting.

$\left\{X^{(i)}\right\}_{i\in[n]}$, all parties aim to obtain additive sharing of $\left(\sum_{i=1}^{n} X^{(i)}\right)\left(\sum_{i=1}^{n} \boldsymbol{v}^{(i)}\right)$ as MAC shares. We observe that:

$$\left(\sum_{i=1}^{n} X^{(i)}\right)\left(\sum_{i=1}^{n} \boldsymbol{v}^{(i)}\right) = \sum_{i=1}^{n} X^{(i)}\boldsymbol{v}^{(i)} + \sum_{i\neq j} X^{(i)}\boldsymbol{v}^{(j)}$$

Each party P_i could locally compute the first item and each ordered pair (P_i, P_j) needs to interactively compute additive sharing of the second item, i.e., $\boldsymbol{u}^{(i,j)} + \boldsymbol{w}^{(j,i)} = X^{(i)}\boldsymbol{v}^{(j)}$. By setting $\boldsymbol{m}^{(i)}(X) = X^{(i)}\boldsymbol{v}^{(i)} + \sum_{j\neq i}\left(\boldsymbol{u}^{(i,j)} + \boldsymbol{w}^{(i,j)}\right)$, we conclude that $\sum_{i=1}^{n} \boldsymbol{m}^{(i)}(X) = X\boldsymbol{v}$, where $X = \sum_{i=1}^{n} X^{(i)}$ and $\boldsymbol{v}^{(i)} = \sum_{i=1}^{n} \boldsymbol{v}^{(i)}$.

Since matrix-vector multiplication is a natural generalization of scalar-vector multiplication, a pair (P_i, P_j) could obtain the additive sharing of $X^{(i)}\boldsymbol{v}^{(j)}$ via m invocations of VOLE. In k-th invocation $\mathcal{F}_{VOLE}^{k}$ for $k \in [m]$, P_i inputs the k-th column $\boldsymbol{x}_k^{(i)}$ of $X^{(i)}$ and P_j inputs the k-th component $v_k^{(j)}$ of global key share $\boldsymbol{v}^{(j)}$, then P_i receives $\boldsymbol{u}_k^{(i,j)}$ and P_j receives $\boldsymbol{w}_k^{(j,i)}$ such that $\boldsymbol{w}_k^{(j,i)} = v_k^{(j)}\boldsymbol{x}_k^{(i)} + \boldsymbol{u}_k^{(i,j)}$. By setting $\boldsymbol{u}^{(i,j)} = \sum_{k=1}^{m} -\boldsymbol{u}_k^{(i,j)}$ and $\boldsymbol{w}^{(j,i)} = \sum_{k=1}^{m} \boldsymbol{w}_k^{(j,i)}$, we obtain the additive sharing of $X^{(i)}\boldsymbol{v}^{(j)}$. It is easy to verify the correctness:

$$\begin{aligned}
\boldsymbol{u}^{(i,j)} + \boldsymbol{w}^{(j,i)} &= \sum_{k=1}^{m} -\boldsymbol{u}_k^{(i,j)} + \boldsymbol{w}_k^{(j,i)} \\
&= \sum_{k=1}^{m} -\boldsymbol{u}_k^{(i,j)} + v_k^{(j)}\boldsymbol{x}_k^{(i)} + \boldsymbol{u}_k^{(i,j)} \\
&= \sum_{k=1}^{m} v_k^{(j)}\boldsymbol{x}_k^{(i)}
\end{aligned}$$

It is insufficient to just apply VOLE to generate authenticated sharings in the presence of a malicious adversary. Because a corrupted party P_j may arbitrarily choose its inputs and use inconsistent vectors $\left(\boldsymbol{x}_1^{(j)}, \cdots, \boldsymbol{x}_m^{(j)}\right)$ or vector $\boldsymbol{v}^{(j)}$ to interact with different honest parties. To prevent such an attack, we introduce a consistency check which opens a random linear combination of authenticated secret sharings to detect the corruption. To avoid leakage caused by this opening, we sacrifice a random authenticated sharing as the mask. Although such a check can not guarantee the consistency of inputs in each invocation of $\mathcal{F}_{VOLE}$, it guarantees that the sum of errors toward an honest party is zero, which suffices to generate the correct MAC share as errors cancel out after the addition.

Combining VOLE with consistency check, we can obtain the authenticated sharings from the additive sharings. The authenticated sharings will be stored after the consistency check passes. Protocol Π_{Auth} is the instantiation of functionality $\mathcal{F}_{Auth}$.

Protocol 2: Π_{Auth}

- **Initialize**: If all parties receive (Init), each party P_i samples $\boldsymbol{v}^{(i)} \xleftarrow{\$} \mathcal{M}_{m\times 1}(\mathbb{F}_q)$ as global key share. For each ordered pair (P_i, P_j) and $k \in [m]$, P_i and P_j call the **Initialize** step of $\mathcal{F}_{VOLE}^k$, where P_j inputs $v_k^{(j)}$.
- **Authenticate**: If all parties receive (Auth, $[X_1], \ldots, [X_\ell]$):
  1. Each party P_i randomly samples a matrix $X_0^{(i)} \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$.
  2. For $h \in \{0\} \cup [\ell]$, write $X_h^{(i)} = (\boldsymbol{x}_{h,1}^{(i)}, \cdots, \boldsymbol{x}_{h,m}^{(i)})$:
     (a) For each ordered pair (P_i, P_j) and $k \in [m]$, P_i and P_j call the **Multiply** step of $\mathcal{F}_{VOLE}^k$, where P_i inputs $\boldsymbol{x}_{h,k}^{(i)}$.
     (b) P_i receives $\boldsymbol{u}_{h,k}^{(i,j)}$ and P_j receives $\boldsymbol{w}_{h,k}^{(j,i)}$ such that $\boldsymbol{w}_{h,k}^{(j,i)} = \boldsymbol{u}_{h,k}^{(i,j)} + v_k^{(j)} \boldsymbol{x}_{h,k}^{(i)}$.
     (c) Each party P_i sets $\boldsymbol{m}^{(i)}(X_h) = X_h^{(i)} \boldsymbol{v}^{(i)} + \sum_{j\neq i} \sum_{k\in[m]} (\boldsymbol{w}_{h,k}^{(i,j)} - \boldsymbol{u}_{h,k}^{(i,j)})$. Let $(X_h^{(i)}, \boldsymbol{v}^{(i)}, \boldsymbol{m}^{(i)}(X_h))$ as the P_i's share of $\langle X_h \rangle$.
  3. Parties call $\mathcal{F}_{Coin}$ ℓ times to obtain randomness $r_1, \cdots, r_\ell$.
  4. Parties locally compute $\langle Y \rangle = \langle X_0 \rangle + \sum_{h=1}^{\ell} r_h \langle X_h \rangle$.
  5. Parties invoke $Y' \leftarrow \pi_{Open}(\langle Y \rangle)$ and $\pi_{check}(Y', \langle Y \rangle)$ to check the correctness of opened value.
  6. If the check succeeds, output $\langle X_1 \rangle, \ldots, \langle X_\ell \rangle$.

Theorem 2. *Protocol Π_{Auth} securely implements $\mathcal{F}_{Auth}$ in the $(\mathcal{F}_{VOLE}, \mathcal{F}_{Coin})$-hybrid model.*

Proof. We analyze the consistency check in Π_{Auth} and defer the complete simulation-based security proof to Section B.2. There are two possible deviations in Π_{Auth}:

- A corrupted party P_j provides inconsistent global key share $\boldsymbol{v}^{(i)}$ with two different honest parties in the **Initialize** step.
- A corrupted party P_j provides inconsistent secret share $X_h^{(i)}$ for $h \in \{0\} \cup [\ell]$ with two different honest parties in the **Authentication** step.

In the command Auth, the adversary could introduce an arbitrarily additive error. For $h \in \{0\} \cup [\ell]$ and $k \in [m]$, let $\boldsymbol{x}_{h,k}^{(j,i)}, v_k^{(j,i)}$ be the *actual* input of P_j used in $\mathcal{F}_{VOLE}^k$ with an honest party P_i. We fix an honest party P_{i_0}, and define the *correct* inputs $\boldsymbol{x}_{h,k}^{(j)}, v_k^{(j)}$ to be equal to $\boldsymbol{x}_{h,k}^{(j,i_0)}, v_k^{(j,i_0)}$ respectively. Then we obtain the additive error between actual inputs and correct inputs:

$$\boldsymbol{\delta}_{h,k}^{(j,i)} = \boldsymbol{x}_{h,k}^{(j,i)} - \boldsymbol{x}_{h,k}^{(j)} \qquad \epsilon_k^{(j,i)} = v_k^{(j,i)} - v_k^{(j)}$$

for each $j \in \mathcal{C}, i \notin \mathcal{C}$. For an honest party P_j, it keeps inputs $\boldsymbol{x}_{h,k}^{(j,i)} = \boldsymbol{x}_{h,k}^{(j)}$ and $v_k^{(j,i)} = v_k^{(j)}$ for each $i \neq j$. Finally, we define that for $i, j \in \mathcal{C}$, the additive error is zero, i.e., $\boldsymbol{\delta}_{h,k}^{(j,i)} = \mathbf{0}$ and $\epsilon_k^{(j,i)} = 0$.

For $j \in \mathcal{C}, i \notin \mathcal{C}$, if P_j behaves as sender and P_i behaves as receiver, we have that

$$\sum_{k=1}^{m} \left(-\boldsymbol{u}_{h,k}^{(j,i)} + \boldsymbol{w}_{h,k}^{(i,j)} \right) = X_h^{(j)} \boldsymbol{v}^{(i)} + \Delta_h^{(j,i)} \boldsymbol{v}^{(i)}$$

where $\Delta_h^{(j,i)} = \left(\boldsymbol{\delta}_{h,1}^{(j,i)}, \cdots, \boldsymbol{\delta}_{h,m}^{(j,i)}\right)$. Similarly, reverse the role of P_i and P_j, we have that

$$\sum_{k=1}^{m}\left(-\boldsymbol{u}_{h,k}^{(i,j)} + \boldsymbol{w}_{h,k}^{(j,i)}\right) = X_h^{(i)}\boldsymbol{v}^{(j)} + X_h^{(i)}\boldsymbol{\epsilon}^{(j,i)}$$

where $\boldsymbol{\epsilon}^{(j,i)} = \left(\epsilon_1^{(j,i)}, \cdots, \epsilon_m^{(j,i)}\right)^T$.

Sum up the MAC share $\boldsymbol{m}^{(i)}(X_h)$, we can see the following result:

$$\begin{aligned}
\sum_{i=1}^{n}\boldsymbol{m}^{(i)}(X_h) &= \sum_{i=1}^{n}X_h^{(i)}\boldsymbol{v}^{(i)} + \sum_{j\neq i}\sum_{k=1}^{m}\left(-\boldsymbol{u}_{h,k}^{(i,j)} + \boldsymbol{w}_{h,k}^{(j,i)}\right) \\
&= \sum_{i=1}^{i}X_h^{(i)}\boldsymbol{v}^{(i)} + \sum_{j\neq i}X_h^{(i,j)}\boldsymbol{v}^{(j,i)} \\
&= X_h\boldsymbol{v} + \sum_{i\notin\mathcal{C}}\underbrace{\sum_{j\in\mathcal{C}}\Delta_h^{(j,i)}}_{\Delta_h^{(i)}}\boldsymbol{v}^{(i)} + \sum_{i\notin\mathcal{C}}X^{(i)}\underbrace{\sum_{j\in\mathcal{C}}\boldsymbol{\epsilon}^{(j,i)}}_{\boldsymbol{\epsilon}^{(i)}}
\end{aligned}$$

After the random linear combination with coefficients $(r_0 = 1, r_1, \cdots, r_\ell)$, we obtain the following MAC of variable Y:

$$\sum_{i=1}^{n}\boldsymbol{m}^{(i)}(Y) = Y\boldsymbol{v} + \sum_{i\notin\mathcal{C}}\sum_{h=0}^{\ell}r_h\Delta_h^{(i)}\boldsymbol{v}^{(i)} + \sum_{i\notin\mathcal{C}}\underbrace{\sum_{h=0}^{\ell}r_hX_h^{(i)}}_{Y^{(i)}}\boldsymbol{\epsilon}^{(i)}$$

Finally we proceed to check opening of Y. To pass the consistency, the adversary needs to introduce two errors $E = Y' - Y$ and $\boldsymbol{\gamma}$ such that:

$$\begin{aligned}
\sum_{i=1}^{n}\boldsymbol{m}^{(i)}(Y) + \boldsymbol{\gamma} - (Y+E)\boldsymbol{v} &= \mathbf{0} \\
\boldsymbol{\gamma} - E\boldsymbol{v} + \sum_{i\notin\mathcal{C}}\sum_{h=0}^{\ell}r_h\Delta_h^{(i)}\boldsymbol{v}^{(i)} + \sum_{i\notin C}Y^{(i)}\boldsymbol{\epsilon}^{(i)} &= \mathbf{0} \\
\sum_{i\notin\mathcal{C}}\left(\sum_{h=0}^{\ell}r_h\Delta_h^{(i)} - E\right)\boldsymbol{v}^{(i)} + \sum_{i\notin C}Y^{(i)}\boldsymbol{\epsilon}^{(i)} &= \sum_{i\in\mathcal{C}}E\boldsymbol{v}^{(i)} - \boldsymbol{\gamma}
\end{aligned}$$

We assert that if consistency check passes, then $\Delta_h^{(i)} = 0$ and $\boldsymbol{\epsilon}^{(i)} = \mathbf{0}$ with overwhelming probability. We prove this assertion with following two claims and defer their proofs in Section B.2.

Claim. If at least one $\boldsymbol{\epsilon}^{(i)} \neq \mathbf{0}$ for some $i \notin \mathcal{C}$, then consistency check passes with negligible probability.

Claim. If $\boldsymbol{\epsilon}^{(i)} = \mathbf{0}$ for all $i \notin \mathcal{C}$ and $\Delta_h^{(i)} \neq 0$ for some $i \notin \mathcal{C}$, then consistency check passes with negligible probability.

5 Offline phase

The preprocessing phase generates the authenticated random sharings $\langle R \rangle$ for the input gates and the multiplication sextuples $(\langle A \rangle, \langle A^T \rangle, \langle B \rangle, \langle C \rangle, \langle R \rangle, \langle R^T \rangle)$ for the multiplication gates. In this section, we focus on the production of the multiplication sextuples. The full-fledged protocol Π_{Prep} is described in Section A. To reduce the communication complexity, we introduce a variant of VOLE *programmable random VOLE*. For the convenience of presentation, the task of generating multiplication sextuple is divided into two parts: generating Beaver triples $(\langle A \rangle, \langle B \rangle, \langle C \rangle)$ and generating double sharings $(\langle A \rangle, \langle A^T \rangle), (\langle R \rangle, \langle R^T \rangle)$.

5.1 Required functionalities

Programmable Random VOLE functionality $\mathcal{F}_{RVOLE}$. A pseudorandom correlation generator (PCG) allows two parties to expand a pair of short, correlated seeds to a much larger amount of correlated randomness. Recently, efficient PCGs for RVOLE are based on several variants of learning parity with noise (LPN) assumptions [8,27,28]. While the communication complexity of original VOLE scales linearly in vector length, the communication complexity of PCG-based RVOLE is either square root of vector length (under primal LPN assumption) or logarithmic in vector length (under dual LPN assumption). In PCG-based RVOLE, the sender P_A sends a seed $s \in S$ instead of a whole vector $\boldsymbol{x}$, where S is the space of seed.

The property *programmability* was introduced to PCG-based RVOLE in [26], which allows the sender to reuse its seed s in different instances of $\mathcal{F}_{RVOLE}$. We model the programmability with function $Expand : S \to \mathbb{F}_q^a$, which deterministically expands the given random seed to a pseudorandom vector of given length a over $\mathbb{F}_q$. The functionality $\mathcal{F}_{RVOLE}^a$ is described as Functionality 6. The instantiation of $\mathcal{F}_{RVOLE}$ can be found in [26], which was adapted from [28].

Functionality 6: $\mathcal{F}_{RVOLE}^a$

$Expand : S \to \mathbb{F}_q^a$ is the deterministic expansion function with seed space S and output length a. The functionality runs between P_i and P_j.

Upon receiving $s \in S$ from P_i and $v \in \mathbb{F}_q$ from P_j:

1. Compute $\boldsymbol{x} = Expand(s)$.
2. Sample $\boldsymbol{u} \xleftarrow{\$} \mathbb{F}_q^a$. If P_A is corrupted, receive $\boldsymbol{u}$ from the adversary.
3. Compute $\boldsymbol{w} = v\boldsymbol{x} + \boldsymbol{u}$. If P_B is corrupted, receive $\boldsymbol{w}$ from adversary and recompute $\boldsymbol{u} = v\boldsymbol{x} - \boldsymbol{w}$.
4. Output $\boldsymbol{u}$ to P_i and $\boldsymbol{w}$ to P_j.

5.2 Generation of Beaver triple

The first step of generating Beaver triple is to securely compute matrix multiplication, which can be decomposed into some matrix-vector multiplications. We

encapsulate the two-party protocol between a random matrix $X \in \mathcal{M}_{a\times m}(\mathbb{F}_q)$ generated by seeds $(s_1, \cdots, s_m)$ and a fixed vector $\boldsymbol{v} \in \mathbb{F}_q^m$ as a procedure called random matrix-vector oblivious product evaluation (rMVOPE), which can be found in Procedure 3.

Procedure 3: π^a_{rMVOPE}

Let $Expand : S \to \mathbb{F}_q^a$ is the deterministic expansion function with seed space S and output length a. This is a two-party procedure between sender P_A and receiver P_B.

Receive $(s_1, \cdots, s_m) \in S^m$ from P_A and $\boldsymbol{v} \in \mathbb{F}_q^m$ from P_B:

1. For $k \in [m]$:
   (a) P_A and P_B invokes $\mathcal{F}^a_{RVOLE}$, where P_A inputs s_k and P_B inputs v_k.
   (b) P_A receives $\boldsymbol{u}_k$ and P_B receives $\boldsymbol{w}_k$
2. P_A sets $\boldsymbol{u} = -\sum_{k=1}^m \boldsymbol{u}_k$ and P_B sets $\boldsymbol{w} = \sum_{k=1}^m \boldsymbol{w}_k$.

Procedure π_{Mult} outputs the authenticated Beaver triples. We take a brief overview of this procedure. To generate a Beaver triple $(\langle A\rangle, \langle B\rangle, \langle C\rangle)$, our approach is similar to [20]. To check the correctness of $(\langle A\rangle, \langle B\rangle, \langle C\rangle)$, we sacrifice another Beaver triple $(\langle A'\rangle, \langle B\rangle, \langle C'\rangle)$. If all parties sample A, A' by expanding the seeds and directly invoke rMVOPE to obtain C, C', a corrupted party P_j may use inconsistent share $B^{(j)}$ toward different honest parties to guess some information of A and A'. To prevent such leakage, we generate τ copies $\{X_h, B, Z_h\}_{h\in[\tau]}$ and take $\mathbb{F}_q$-linear combinations with random coefficients $\{(r_h, r'_h)\}_{h\in[\tau]}$ to obtain A, A', C, C'. The random linear combinations yield universal hash functions to extract A', A', C, C' from several partially leaked values $\{(X_h, Z_h)\}_{h\in[\tau]}$.

Procedure 4: π_{Mult}

The procedure generates an authenticated triple $(\langle A\rangle, \langle B\rangle, \langle C\rangle)$ where $A, B \xleftarrow{\$} \mathcal{M}_{m\times m}(\mathbb{F}_q)$ and $C = AB$. The integer parameter $\tau = 4$ suggested in [20] specifies the number of input triples required to generate an authenticated triple.

– **Multiply:**
   1. Let $Expand : S \to \mathbb{F}_q^{\tau m}$. Each party P_i samples seeds $\left(s_1^{(i)}, \cdots, s_m^{(i)}\right)$ and obtains $X^{(i)} = \left(\boldsymbol{x}_1^{(i)}, \cdots, \boldsymbol{x}_m^{(i)}\right) \in \mathcal{M}_{\tau m\times m}(\mathbb{F}_q)$, where $\boldsymbol{x}_k^{(i)} = Expand\left(s_k^{(i)}\right)$ for $k \in [m]$.
   2. All parties invoke π_{Rand} to obtain an additive sharing $[B]$.
   3. For $k \in [m]$ and each ordered pair (P_i, P_j):
      (a) P_i and P_j invoke $\pi^{\tau m}_{rMVOPE}$, where P_i inputs $\left(s_1^{(i)}, \cdots, s_m^{(i)}\right)$ and P_j inputs $\boldsymbol{b}_k^{(j)}$
      (b) P_i receives $\boldsymbol{u}_k^{(i,j)}$ and P_j receives $\boldsymbol{w}_k^{(j,i)}$

4. Each party P_i sets $U^{(i,j)} = \left(\boldsymbol{u}_1^{(i,j)}, \cdots, \boldsymbol{u}_m^{(i,j)}\right)$ and $W^{(i,j)} = \left(\boldsymbol{w}_1^{(i,j)}, \cdots, \boldsymbol{w}_m^{(i,j)}\right)$.
5. Split the matrices into τ blocks, we write $U^{(i,j)} + W^{(j,i)} = X^{(i)}B^{(j)}$ as

$$\begin{pmatrix} U_1^{(i,j)} \\ \vdots \\ U_\tau^{(i,j)} \end{pmatrix} + \begin{pmatrix} W_1^{(j,i)} \\ \vdots \\ W_\tau^{(j,i)} \end{pmatrix} = \begin{pmatrix} X_1^{(i)} \\ \vdots \\ X_\tau^{(i)} \end{pmatrix} B^{(j)}$$

where $U_h^{(i,j)}, W_h^{(j,i)}, X_h^{(i)}, B^{(j)} \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$ for $h \in [\tau]$.
6. For $h \in [\tau]$, each party P_i computes $Z_h^{(i)} = X_h^{(i)}B^{(i)} + \sum_{j\neq i}\left(U_h^{(i,j)} + W_h^{(i,j)}\right)$.

– **Combine:**
1. All parties invoke $\mathcal{F}_{Coin}$ 2τ times to obtain $\{(r_h, r'_h)\}_{h\in[\tau]}$.
2. All parties locally compute

$$[A] = \sum_{h=1}^{\tau} r_h[X_h] \qquad [C] = \sum_{h=1}^{\tau} r_h[Z_h]$$

$$[A'] = \sum_{h=1}^{\tau} r'_h[X_h] \qquad [C'] = \sum_{h=1}^{\tau} r'_h[Z_h]$$

– **Authenticate**: All parties invoke $\mathcal{F}_{Auth}$ to obtain $\langle A\rangle, \langle A'\rangle, \langle B\rangle, \langle C\rangle$ and $\langle C'\rangle$.
– **Sacrifice**:
1. All parties invoke $\mathcal{F}_{Coin}$ to obtain χ.
2. All parties locally compute $\langle D\rangle = \chi\langle A\rangle - \langle A'\rangle$ and partially open $D \leftarrow \pi_{Open}(\langle D\rangle)$.
3. All parties locally compute $\langle E\rangle = \chi\langle C\rangle - \langle C'\rangle - D\langle B\rangle$ and partially open $E \leftarrow \pi_{Open}(\langle E\rangle)$.
4. If $E \neq 0$, then aborts.

– **Output**: If no party aborts, all parties output $(\langle A\rangle, \langle B\rangle, \langle C\rangle)$.

5.3 Generation of double sharing

To generate ℓ multiplication sextuples for securely computing ℓ multiplication gates, we need $2\ell + 1$ double sharings of the form $\langle A\rangle, \langle A^T\rangle$ with some random matrix A. Procedure π_{Double} receives the authenticated sharing $\langle A\rangle$ and output the pair of authenticated sharing $(\langle A\rangle, \langle A^T\rangle)$. We briefly explain the idea of this procedure. Observe that $[A^T]$ can be obtained by locally applying the transpose to each share of $[A]$. Then, we apply the $\mathcal{F}_{Auth}$ to obtain the authenticated sharing $\langle A^T\rangle$. We take random linear combinations of $2\ell + 1$ double sharing $\langle A_i\rangle$ and $\langle A_i^T\rangle$ respectively and partially open them to C and D. If there is no corruption, $C = D^T$ and the check passes. Otherwise, this check will pass with probability at most $1/q$. One can find π_{Double} in Procedure 5.

Procedure 5: π_{Double}

The procedure produces 2ℓ pairs of authenticated sharing $\langle A_i\rangle, \langle A_i^T\rangle, i \in [2\ell]$.
Double: Upon receiving (Double, $\langle A_1\rangle, \ldots, \langle A_{2\ell}\rangle$) from all parties:

1. All parties invoke π_{Rand} to obtain $[A_0]$.
2. All parties locally compute $[A_i^T]$ from $[A_i]$ for $i \in \{0\} \cup [2\ell]$ by taking the transpose of each share.
3. All parties invoke $\mathcal{F}_{Auth}$ with command (Auth, $[A_0], [A_0^T], [A_1^T], \ldots, [A_{2\ell}^T]$) to obtain the authenticated sharings $\langle A_0\rangle, \langle A_0^T\rangle, \langle A_1^T\rangle, \ldots, \langle A_{2\ell}^T\rangle$.
4. All parties call $\mathcal{F}_{Coin}$ 2ℓ times to obtain $r_1, \cdots, r_{2\ell}$.
5. All parties locally compute

$$\langle C\rangle = \sum_{i=1}^{2\ell} r_i \langle A_i\rangle + \langle A_0\rangle \quad \langle D\rangle = \sum_{i=1}^{2\ell} r_i \langle A_i^T\rangle + \langle A_0^T\rangle$$

6. All parties invoke π_{Open} to partially open C and D.
7. If $C \neq D^T$, then aborts.
8. All parties invoke π_{Check} to check the opened values.
9. If no party aborts, output 2ℓ pairs of authenticated sharings $(\langle A_i\rangle, \langle A_i^T\rangle), i \in [2\ell]$.

Putting together. Protocol Π_{Tuple} instantiates the functionality $\mathcal{F}_{Tuple}$ by invoking the procedures introduced above. π_{Mult} and π_{Double} are used to produce the authenticated sharings $(\langle A\rangle, \langle B\rangle, \langle C\rangle)$ and $(\langle A\rangle, \langle A^T\rangle), (\langle R\rangle, \langle R^T\rangle)$, respectively.

Protocol 6: Π_{Tuple}

This protocol produces ℓ authenticated sextuples $(\langle A\rangle, \langle A^T\rangle, \langle B\rangle, \langle C\rangle, \langle R\rangle, \langle R^T\rangle)$ with $C = AB$:

1. All parties invoke π_{Mult} ℓ times to produce $(\langle A_i\rangle, \langle B_i\rangle, \langle C_i\rangle)$ with $C_i = A_i B_i$ for $i \in [\ell]$.
2. All parties invoke π_{Rand} ℓ times to obtain $[R_1], \ldots, [R_\ell]$.
3. All parties call $\mathcal{F}_{Auth}$ with command (Auth, $[R_1], \ldots, [R_\ell]$) to obtain $\langle R_i\rangle$ for $i \in [\ell]$.
4. All parties invoke π_{Double} with command (Double, $\langle A_1\rangle, \ldots, \langle A_\ell\rangle, \langle R_1\rangle, \ldots, \langle R_\ell\rangle$) to obtain $(\langle A_i\rangle, \langle A_i^T\rangle)$ and $(\langle R_i\rangle, \langle R_i^T\rangle)$ for $i \in [\ell]$.
5. Output $(\langle A_i\rangle, \langle A_i^T\rangle, \langle B_i\rangle, \langle C_i\rangle, \langle R_i\rangle, \langle R_i^T\rangle)$ for $i \in [\ell]$.

Theorem 3. *Protocol Π_{Tuple} securely implements $\mathcal{F}_{Tuple}$ in the ($\mathcal{F}_{Auth}$, $\mathcal{F}_{RVOLE}$, $\mathcal{F}_{Coin}$)-hybrid model.*

Proof. Let $\mathcal{Z}$ be the environment, which we also refer to as the adversary capable of corrupting a set $\mathcal{C}$ containing at most $n-1$ parties. We construct a simulator $\mathcal{S}$ such that the real execution and ideal execution are indistinguishable to $\mathcal{Z}$.

Here we only prove the security of π_{Mult} and refer to Section B.3 for the full proof.

In procedure $\pi_{rMVOPE}^{\tau m}$, the sender only inputs m seeds. Therefore, the corrupted parties can only choose inconsistent seeds for different honest parties, which can not translate to an arbitrarily chosen additive error. However, for the convenience of analysis, we follow the idea of [26] and improve the competence of adversary to introduce an arbitrarily chosen additive error.

Simulating the Multiply step. The simulator $\mathcal{S}$ emulates the functionality $\mathcal{F}_{RVOLE}^{\tau m}$ in procedure $\pi_{rMVOPE}^{\tau m}$. For $j \in \mathcal{C}$ and $i \notin \mathcal{C}$, let $\left(s_{k,1}^{(j,i)}, \cdots, s_{k,m}^{(j,i)}\right)$ and $\boldsymbol{b}_k^{(j,i)}$ be the *actual* input in the k-th invocation of $\pi_{rMVOPE}^{\tau m}$ for $k \in [m]$. Fix an honest party P_{i_0} and define the *correct* input $\left(s_1^{(j)}, \cdots, s_m^{(j)}\right)$ and $\boldsymbol{b}_k^{(j)}$ to be equal to $\left(s_{1,1}^{(j,i_0)}, \cdots, s_{1,m}^{(j,i_0)}\right)$ and $\boldsymbol{b}_k^{(j,i_0)}$, respectively. For $i \notin \mathcal{C}$, $\mathcal{S}$ randomly samples $X^{(i)}, B^{(i)} \xleftarrow{\$} \mathcal{M}_{m\times m}(\mathbb{F}_q)$. For $j \in \mathcal{C}$, $\mathcal{S}$ receives $\left(s_{k,1}^{(j,i)}, \cdots, s_{k,m}^{(j,i)}\right)$ and $\boldsymbol{b}_k^{(j,i)}$ from the adversary. Then $\mathcal{S}$ receives $\left\{\boldsymbol{u}_k^{(j,i)}, \boldsymbol{w}_k^{(j,i)}\right\}_{j\in\mathcal{C}, i\notin\mathcal{C}}$ from the adversary and recomputes $\left\{\boldsymbol{u}_k^{(i,j)}, \boldsymbol{w}_k^{(i,j)}\right\}_{i\notin\mathcal{C}, j\in\mathcal{C}}$ accordingly. Finally, $\mathcal{S}$ honestly computes $Z^{(i)}$.

Simulating the Combine step. $\mathcal{S}$ emulates functionality $\mathcal{F}_{Coin}$ to obtain $\{(r_h, r'_h)\}_{h\in[\tau]}$ and executes local computations.

Simulating the Authentication step. $\mathcal{S}$ emulates functionality $\mathcal{F}_{Auth}$ with inputs from corrupted parties controlled by $\mathcal{Z}$. $\mathcal{S}$ authenticates additive sharings and we define E_{Auth}, E'_{Auth} to be the errors introduced in authentication. If E_{Auth}, E'_{Auth} is not zero, then parties authenticate values different from those in the previous step. If $\mathcal{Z}$ sends Abort to $\mathcal{F}_{Auth}$, $\mathcal{S}$ sends Abort to $\mathcal{F}_{Tuple}$.

Simulating the Sacrifice step. $\mathcal{S}$ samples $D \leftarrow \mathcal{M}_{m\times m}(\mathbb{F}_q)$ as $\chi A - A'$. If the triple is incorrect, $\mathcal{S}$ aborts; otherwise, $\mathcal{S}$ outputs it as a valid triple.

Indistinguishability. We argue that $\mathcal{Z}$ cannot distinguish real execution and simulated one. We will show that if no abort happens, the probability that adversary introduces some non-zero errors is negligible and the distribution of opened value is statistically close in both of the worlds.

Now we proceed to the introduced errors during **Multiply** step. Let $X_k^{(j,i)}$ be the matrix generated by seeds $\left(s_{k,1}^{(j,i)}, \cdots, s_{k,m}^{(j,i)}\right)$. In the k-th invocation of $\pi_{rMVOPE}^{\tau m}$, denote the errors as $\Delta_k^{(j,i)} = X_k^{(j,i)} - X^{(j)}$ and $\boldsymbol{\epsilon}_k^{(j,i)} = \boldsymbol{b}_k^{(j,i)} - \boldsymbol{b}^{(j)}$.

Following similar analysis in the proof of Theorem 2, we conclude that for $k \in [m], i \notin \mathcal{C}$ and $j \in \mathcal{C}$:

$$\sum_{i=1}^{n} \boldsymbol{z}_k^{(i)} = X\boldsymbol{b}_k + \underbrace{\sum_{i \notin \mathcal{C}} \Delta_k^{(i)} \boldsymbol{b}_k^{(i)}}_{\boldsymbol{\gamma}_k} + \sum_{i \notin \mathcal{C}} X^{(i)} \boldsymbol{\epsilon}_k^{(i)}$$

where $\Delta_k^{(i)} = \sum_{j \in \mathcal{C}} \Delta_k^{(j,i)}$ and $\boldsymbol{\epsilon}_k^{(i)} = \sum_{j \in \mathcal{C}} \boldsymbol{\epsilon}_k^{(j,i)}$. Putting m columns together, we have that:

$$Z = XB + (\boldsymbol{\gamma}_1, \cdots, \boldsymbol{\gamma}_m) + \sum_{i \notin \mathcal{C}} X^{(i)} \left(\boldsymbol{\epsilon}_1^{(i)}, \cdots, \boldsymbol{\epsilon}_m^{(i)} \right)$$

Let $\Gamma = (\boldsymbol{\gamma}_1, \cdots, \boldsymbol{\gamma}_m)$. Splitting the matrices into τ blocks, we have that:

$$\begin{pmatrix} Z_1 \\ \vdots \\ Z_\tau \end{pmatrix} = \begin{pmatrix} X_1 \\ \vdots \\ X_\tau \end{pmatrix} B + \begin{pmatrix} \Gamma_1 \\ \vdots \\ \Gamma_\tau \end{pmatrix} + \sum_{i \notin \mathcal{C}} \begin{pmatrix} X_1^{(i)} \\ \vdots \\ X_\tau^{(i)} \end{pmatrix} \left(\boldsymbol{\epsilon}_1^{(i)}, \cdots, \boldsymbol{\epsilon}_m^{(i)} \right)$$

Assume that each party (including the corrupted party) honestly takes linear combination in **Combine** step (The deviation in this step causes an additive error, which is absorbed by the errors E_{Auth}, E'_{Auth} in **Authentication** step). After these two steps, all parties obtain $\langle A \rangle, \langle A' \rangle, \langle B \rangle, \langle C \rangle, \langle C' \rangle$ where A, A', B, C, C' satisfy that:

$$C = AB + E_1 + E_2 + E_{Auth}$$
$$C' = A'B + E'_1 + E'_2 + E'_{Auth}$$

and

$$E_1 = \sum_{h=1}^{\tau} r_h \Gamma_h \qquad E_2 = \sum_{h=1}^{\tau} r_h \sum_{i \notin \mathcal{C}} A^{(i)} \left(\boldsymbol{\epsilon}_1^{(i)}, \cdots, \boldsymbol{\epsilon}_m^{(i)} \right)$$
$$E'_1 = \sum_{h=1}^{\tau} r'_h \Gamma_h \qquad E'_2 = \sum_{h=1}^{h} r'_h \sum_{i \notin \mathcal{C}} A'^{(i)} \left(\boldsymbol{\epsilon}_1^{(i)}, \cdots, \boldsymbol{\epsilon}_m^{(i)} \right)$$

If no abort happens in the **Sacrifice** step, we come to the following conclusions and defer their proofs to Section B.3.

Claim. If the sacrifice step passes, then $E = E_1 + E_2 + E_{Auth} = 0$ and $E' = E'_1 + E'_2 + E'_{Auth}$ with overwhelming probability.

Claim. If the sacrifice step passes, then $\{\boldsymbol{\epsilon}_k^{(i)}\}_{i \notin \mathcal{C}, k \in [m]}$ are zero with overwhelming probability.

Now we want to show that opened value D in the real execution is statistically close to uniform distribution in the simulated execution. Given that $D = \chi A - A'$,

it suffices to prove A' looks uniformly random to $\mathcal{Z}$ and thus can serve as a mask. The same analysis in [20] is applicable to our case since A' is the random linear combination of τ matrices whose entries can be treated as the random linear combinations of τ elements. Thus, the statistical distance of A' and uniform distribution is less than $2^{-\kappa}$.

If sacrifice step passes, we have that $E_{Auth} + E_1 = 0$. Furthermore, for k-th column:

$$-(E_{Auth})_k = \sum_{h=1}^{\tau} r_h \sum_{i \notin \mathcal{C}} \Delta_{k,h}^{(i)} \boldsymbol{b}_k^{(i)}, \text{where } \Delta_k^{(i)} = \begin{pmatrix} \Delta_{k,1}^{(i)} \\ \vdots \\ \Delta_{k,\tau}^{(i)} \end{pmatrix}$$

Since the distribution of E_1 is statistically close to the uniform distribution, the adversary can inject non-zero error E_{Auth} with negligible probability.

6 Extension to $\mathcal{M}_{m\times m}(\mathbb{Z}_{p^k})$

The protocol described in Section 3 and 5 is applicable to $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ with large enough q, i.e., $q \geq 2^\kappa$ so that the error probability can be reduced to $q^{-1} \leq 2^{-\kappa}$. We note that it is possible to modify our MPC protocol to evaluate the circuit over $\mathcal{M}_{m\times m}(\mathbb{Z}_{p^k})$ and $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ with small q. We only present the modification for integer rings. The same modification can be applied to small fields as well since the major challenge for both of them is to reduce the error probability.

We note that when our MPC protocol migrates from $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ to $\mathcal{M}_{m\times m}(\mathbb{Z}_{p^k})$, the error probability does not meet the security requirement anymore even if k is large enough. For example, given an authenticated secret sharing $\langle X \rangle = ([X], [\![\boldsymbol{v}]\!], [\![X\boldsymbol{v}]\!])$, where $A \in \mathcal{M}_{m\times m}(\mathbb{Z}_{p^k})$ and $\boldsymbol{v} \in \mathcal{M}_{m\times 1}(\mathbb{Z}_{p^k})$, the adversary $\mathcal{A}$ could inject an additional error E to forge an authenticated sharing $\langle A + E \rangle = ([A + E], [\![\boldsymbol{v}]\!], [\![X\boldsymbol{v}]\!])$, where $E = (p^{k-1})_{m\times m}$, i.e., each entry of E is p^{k-1}. The corrupted sharing passes verification with probability $1/p$ if $\sum_{i=1}^{m} \boldsymbol{v}[i]$ is divided by p. This problem also arises in the case of $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ with small q. To reduce soundness error, we propose the following modifications:

Authenticated Sharing. Instead of letting the global key in $\mathcal{M}_{m\times 1}(\mathbb{Z}_{p^k})$, we require that the global key is a matrix in $\mathcal{M}_{m\times \ell}(\mathbb{Z}_{p^k})$. This also implies that the MAC for each matrix is a matrix in $\mathcal{M}_{m\times \ell}(\mathbb{Z}_{p^k})$. One can treat the global key in $\mathcal{M}_{m\times \ell}(\mathbb{Z}_{p^k})$ as ℓ independent global keys in $\mathcal{M}_{m\times 1}(\mathbb{Z}_{p^k})$. Let $V \in \mathcal{M}_{m\times \ell}(\mathbb{Z}_{p^k})$ be the global key and assume that $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_\ell$ are the column vectors of V. Let E be the additional error injected by the adversary. To pass the verification, it must hold that $EV = X$ where $X \in \mathcal{M}_{m\times \ell}(\mathbb{Z}_{p^k})$ is the matrix known to the adversary. Let $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_\ell$ be the column vectors of X and we have $E\boldsymbol{v}_i = \boldsymbol{x}_i$ for $i \in [\ell]$. Since V is distributed uniformly at random, the adversary succeeds with probability at most $p^{-\ell}$. Therefore the soundness error is reduced to $p^{-\ell}$. If we

set $\ell = \frac{\kappa}{\log_2 p}$, the soundness error becomes $2^{-\kappa}$. Since the size of the matrix is much bigger than the MAC, the share size is almost the same as the previous one.

Linear Combinations. Taking the linear combinations of secret sharings is an efficient verification method to check the correctness of the sharings in batch which appears in Check command of online phase, and also the production of random and double sharing in the preprocessing phase. However, the soundness of this check becomes $1/p$ if our matrix is defined over $\mathcal{M}_{m\times m}(\mathbb{Z}_{p^k})$. We can repeat the linear combinations ℓ times to reduce the error probability to $p^{-\ell}$. Since each linear combination of random and double sharings needs to sacrifice one corresponding sharing, repeating ℓ times means that all parties need to prepare $\ell - 1$ additional sharings which can be amortized away due to this check in batch.

Combine. To prevent partial leakage of honest parties' private information, we need to use a universal hash function to extract randomness from τ copies. For a small field $\mathbb{F}_q$, we still utilize a random linear combination as a universal hash function. However, this does not work for ring $\mathbb{Z}_{p^k}$. To cope with this problem, we follow [12] to apply $r_1X_1 + \cdots + r_\tau X_\tau \mod p^k$ with some constriction. Different from [12], we restrict that $r_i \in \mathbb{Z}_p$ instead of the entries of X_i.

Sacrifice. Recall that to compute a triple $(\langle A\rangle, \langle B\rangle, \langle C\rangle)$, we need to sacrifice $(\langle A'\rangle, \langle B\rangle, \langle C'\rangle)$ to check its correctness. The sacrificing technique in $\mathcal{M}_{m\times m}(\mathbb{Z}_{p^k})$ can detect the corruptions with probability $1 - \frac{1}{p}$. To make this probability overwhelming, we have to sacrifice ℓ triples to verify the relation $C = AB$.

VOLE and RVOLE. We need to adapt VOLE and RVOLE for the integer ring setting. The existing VOLE protocols focus on $\mathbb{Z}_{2^k}$. For example, SPD$\mathbb{Z}_{2^k}$ [12] uses VOLE as functionality and Moz$\mathbb{Z}_{2^k}$arella [5] uses RVOLE over $\mathbb{Z}_{2^k}$ as functionality. Since PCG in RVOLE is based on LPN assumptions, the hardness of LPN assumptions should be re-estimated in the setting of $\mathbb{Z}_{2^k}$ [21].

7 Analysis

In this section we will analyze the communication complexity and share size of our MPC protocol over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ and $\mathcal{M}_{m\times m}(\mathbb{Z}_{p^k})$, respectively.

7.1 Analysis of MPC over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$

In this subsection, we present the communication complexity of our MPC protocol over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ assuming $q \geq 2^\kappa$. For small q, we refer the reader to the next subsection where we analyze the communication complexity of our MPC protocol over $\mathcal{M}_{m\times m}(\mathbb{Z}_{p^k})$. The analysis of the matrix over a small field is almost the same as the matrix over an integer ring.

Analysis of the online phase. At each step of partial opening, all parties send their shares to a specific party, then let this party reconstruct and broadcast the secret, thus the communication complexity is $2m^2(n-1)\log q$ bits. For each multiplication gate, all parties need to partially open three shares $\langle D\rangle, \langle E\rangle, \langle F\rangle$ and thus the communication complexity is $6m^2(n-1)\log q$ bits. Each input gate requires P_i to broadcast the difference between X and mask R, which communicates $m^2(n-1)\log q$ bits. For the output gate, the partial opening needs $2m^2(n-1)\log q$ bit of communication and verification needs $mn^2\log q$ bits of communication via simultaneous message channel. Another measure is share size, which is $m(m+1)n\log q$ bits, since $[\![\boldsymbol{v}]\!]$ remains unchanged in each authenticated sharing and we omit this item.

We analyze the communication complexity and share size of other MPC protocols and list the results below. We note that the matrix computed in [11] is defined over $\mathbb{Z}_{p^k}$ as they use BFV scheme in their preprocessing phase. The comparison with [11] is deferred to the integer ring. Since there are no dishonest majority MPC protocols over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$, we compare our protocol with SPDZ [14] to execute entry-wise computation. The variant of SPDZ protocols such as [20] has almost the same communication complexity in the online phase as m^3 Beaver triples are required to compute a matrix multiplication gate in the online phase.

	SPDZ [14]	our work	our work (FD)
comm	$4m^3(n-1)\log q$	$6m^2(n-1)\log q$	$4m^2(n-1)\log q$
share size	$2m^2\log q$	$m(m+1)\log q$	$m(m+1)\log q$

Table 1. The comparison of MPC protocols over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ in terms of share size and communication complexity of a multiplication gate (FD represents function dependent preprocessing)

Analysis of the preprocessing phase. The task of preprocessing is to generate random sharing and multiplication sextuple. The communication cost is dictated by Π_{Tuple} which produces the multiplication sextuples. As our preprocessing phase uses VOLE and RVOLE as the building blocks, we calculate the communication cost of preprocessing phase in the number of calls of the functionality $\mathcal{F}_{VOLE}$ and $\mathcal{F}_{RVOLE}$.

To generate a random authenticated sharing $\langle R\rangle$ for an input gate, where the secret R is known to P_i, P_i distributes the additive share $X^{(j)}$ to P_j and invokes $\mathcal{F}_{VOLE}$ with P_j. After producing $\ell+1$ such random sharings, P_i sacrifices the first one to invoke π_{Check} for consistency check. If ℓ is large enough, the cost of the consistency check can be amortized away. In this case, the preparation for an input gate only costs $n-1$ calls of $\mathcal{F}_{VOLE}$.

Protocol Π_{Tuple} produces ℓ sextuples by generating ℓ Beaver triples and 2ℓ double sharings. The procedure π_{Mult} makes m calls of $\pi^{\tau m}_{rMVOPE}$, 5 calls of

Π_{Auth} and 2 calls of π_{open} which finally leads to $m^2n(n-1)$ calls of $\mathcal{F}_{RVOLE}^{\tau m}$ and $5mn(n-1)$ calls of $\mathcal{F}_{VOLE}$ in total. To generate 2ℓ authenticated double sharings, the procedure π_{Double} invokes $\mathcal{F}_{Auth}$ $3\ell+2$ times and executes the consistency check, which causes $(3\ell+2)mn(n-1)$ calls of $\mathcal{F}_{VOLE}$. Thus, preparing a sextuple requires $m^2n(n-1)$ calls of $\mathcal{F}_{RVOLE}^{\tau m}$ and $8mn(n-1)$ calls of $\mathcal{F}_{VOLE}$. As shown in Section 5, the instantiation of 5.1, $\mathcal{F}_{VOLE}$ and $\mathcal{F}_{RVOLE}^{\tau m}$ incurs $O(m \log q)$ and $O(\sqrt{\tau m} \log q)$ bits of communication, respectively[8], therefore the overall communication complexity for a multiplication sextuple is $O(n^2 m^{2.5} \log q)$ bits.

7.2 Analysis of MPC over $\mathcal{M}_{m\times m}(\mathbb{Z}_{p^k})$

Analysis of the online phase. To migrate MPC protocol from $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ to $\mathcal{M}_{m\times m}(\mathbb{Z}_{p^k})$, we extend the global key and MACs from vectors of length m to $m \times \ell$ matrices, where $\ell = \frac{\kappa}{\log p}$. This modification does not affect the communication complexity but does affect the share size. In contrast, the homomorphic encryption scheme BFV in matrix triple [11] requires that plaintext modulus p^k is large enough. To make a fair comparison, we assume that p^k satisfies the security requirement of [11] and compare the communication complexity of the multiplication gate and the share size. Although our protocol needs slightly more communication than [11], our protocol has a smaller share size and can be defined over any integer ring. Moreover, the improvement of our MPC protocol by resorting to function dependent preprocessing can achieve the same communication complexity.

	matrix triple [11]	our work	our work (FD)
comm	$4m^2k(n-1)\log p$	$6m^2k(n-1)\log p$	$4m^2k(n-1)\log p$
share size	$2m^2k\log p$	$m^2k\log p + mk\kappa$	$m^2k\log p + mk\kappa$

Table 2. The comparison of MPC protocols over $\mathcal{M}_{m\times m}(\mathbb{Z}_{p^k})$ in terms of share size and communication complexity of a multiplication gate (FD represents function dependent preprocessing)

Analysis of the prerpocessing phase. The communication complexity of our preprocessing phase is almost the same as that of our MPC protocol over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$. In view of the changes from finite fields to integer rings introduced in Section 6, the overall communication complexity for computing a multiplication sextuple is $O(\kappa k n^2 m^{2.5})$ bits. Since the preprocessing phase in [11] takes a totally different approach by resorting to homomorphic encryption (BFV scheme) and zero knowledge proof, it is difficult to make a fair comparison.

[8] The RVOLE with communication complexity $O(\log \tau m \log q)$ bits is based on dual LPN assumptions. Our protocol requires RVOLE with programmability which are mostly based on primal LPN assumptions.

References

1. M. Abspoel, R. Cramer, I. Damgård, D. Escudero, M. Rambaud, C. Xing, and C. Yuan. Asymptotically good multiplicative LSSS over galois rings and applications to MPC over $\mathbb{Z}/p^k\mathbb{Z}$. In *ASIACRYPT 2020*, volume 12493 of *LNCS*, pages 151–180. Springer, 2020. doi:10.1007/978-3-030-64840-4_6.
2. M. Abspoel, R. Cramer, I. Damgård, D. Escudero, and C. Yuan. Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via galois rings. In *TCC 2019*, volume 11891 of *LNCS*, pages 471–501. Springer, 2019. doi:10.1007/978-3-030-36030-6_19.
3. B. Applebaum, I. Damgård, Y. Ishai, M. Nielsen, and L. Zichron. Secure arithmetic computation with constant computational overhead. In *CRYPTO 2017*, volume 10401 of *LNCS*, pages 223–254. Springer, 2017. doi:10.1007/978-3-319-63688-7_8.
4. B. Applebaum and N. Konstantini. Actively secure arithmetic computation and VOLE with constant computational overhead. In *EUROCRYPT 2023*, volume 14005 of *LNCS*, pages 190–219. Springer, 2023. doi:10.1007/978-3-031-30617-4_7.
5. C. Baum, L. Braun, A. Munch-Hansen, and P. Scholl. Moz$\mathbb{Z}_{2^k}$arella: Efficient vector-ole and zero-knowledge proofs over $\mathbb{Z}_{2^k}$. In *CRYPTO 2022*, volume 13510 of *LNCS*, pages 329–358. Springer, 2022. doi:10.1007/978-3-031-15985-5_12.
6. D. Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO '91*, volume 576 of *LNCS*, pages 420–432. Springer, 1991. doi:10.1007/3-540-46766-1_34.
7. A. Ben-Efraim, M. Nielsen, and E. Omri. Turbospeedz: Double your online spdz! improving SPDZ using function dependent preprocessing. In *ACNS 2019*, volume 11464 of *LNCS*, pages 530–549. Springer, 2019. doi:10.1007/978-3-030-21568-2_26.
8. E. Boyle, G. Couteau, N. Gilboa, and Y. Ishai. Compressing vector OLE. In *ACM CCS 2018*, pages 896–912. ACM, 2018. doi:10.1145/3243734.3243868.
9. Z. Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *CRYPTO 2012*, volume 7417 of *LNCS*, pages 868–886. Springer, 2012. doi:10.1007/978-3-642-32009-5_50.
10. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS 2001*, pages 136–145. IEEE Computer Society, 2001. doi:10.1109/SFCS.2001.959888.
11. H. Chen, M. Kim, I. P. Razenshteyn, D. Rotaru, Y. Song, and S. Wagh. Maliciously secure matrix multiplication with applications to private deep learning. In *ASIACRYPT 2020*, volume 12493 of *LNCS*, pages 31–59. Springer, 2020. doi:10.1007/978-3-030-64840-4_2.
12. R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing. Spd$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In *CRYPTO 2018*, volume 10992 of *LNCS*, pages 769–798. Springer, 2018. doi:10.1007/978-3-319-96881-0_26.
13. I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, 2013. doi:10.1007/978-3-642-40203-6_1.
14. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, 2012. doi:10.1007/978-3-642-32009-5_38.
15. D. Escudero, V. Goyal, A. Polychroniadou, and Y. Song. Turbopack: Honest majority MPC with constant online communication. In *ACM CCS 2022*, pages 951–964. ACM, 2022. doi:10.1145/3548606.3560633.

16. D. Escudero, V. Goyal, A. Polychroniadou, Y. Song, and C. Weng. Superpack: Dishonest majority MPC with constant online communication. In *EUROCRYPT 2023*, volume 14005 of *LNCS*, pages 220–250. Springer, 2023. doi:10.1007/978-3-031-30617-4_8.
17. D. Escudero and E. Soria-Vazquez. Efficient information-theoretic multi-party computation over non-commutative rings. In *CRYPTO 2021*, volume 12826 of *LNCS*, pages 335–364. Springer, 2021. doi:10.1007/978-3-030-84245-1_12.
18. D. Escudero, C. Xing, and C. Yuan. More efficient dishonest majority secure computation over $\mathbb{Z}_{2^k}$ via galois rings. In *CRYPTO 2022*, volume 13507 of *LNCS*, pages 383–412. Springer, 2022. doi:10.1007/978-3-031-15802-5_14.
19. X. Jiang, M. Kim, K. E. Lauter, and Y. Song. Secure outsourced matrix computation and application to neural networks. In *CCS 2018*, pages 1209–1222. ACM, 2018. doi:10.1145/3243734.3243837.
20. M. Keller, E. Orsini, and P. Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *ACM CCS 2016*, pages 830–842. ACM, 2016. doi:10.1145/2976749.2978357.
21. H. Liu, X. Wang, K. Yang, and Y. Yu. The hardness of LPN over any integer ring and field for PCG applications. *IACR Cryptol. ePrint Arch.*, page 712, 2022.
22. J. Liu, M. Juuti, Y. Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In *ACM CCS 2017*, pages 619–631. ACM, 2017. doi:10.1145/3133956.3134056.
23. P. Mohassel and P. Rindal. Aby^3: A mixed protocol framework for machine learning. In *ACM CCS 2018*, pages 35–52. ACM, 2018. doi:10.1145/3243734.3243760.
24. P. Mohassel and Y. Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38. IEEE Computer Society, 2017. doi:10.1109/SP.2017.12.
25. E. Orsini, N. P. Smart, and F. Vercauteren. Overdrive2k: Efficient secure MPC over $\mathbb{Z}_{2^k}$ from somewhat homomorphic encryption. In *CT-RSA 2020*, volume 12006 of *LNCS*, pages 254–283. Springer, 2020. doi:10.1007/978-3-030-40186-3_12.
26. R. Rachuri and P. Scholl. Le mans: Dynamic and fluid MPC for dishonest majority. In *CRYPTO 2022*, volume 13507 of *LNCS*, pages 719–749. Springer, 2022. doi:10.1007/978-3-031-15802-5_25.
27. P. Schoppmann, A. Gascón, L. Reichert, and M. Raykova. Distributed vector-ole: Improved constructions and implementation. In *ACM CCS 2019*, pages 1055–1072. ACM, 2019. doi:10.1145/3319535.3363228.
28. C. Weng, K. Yang, J. Katz, and X. Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1074–1091. IEEE, 2021. doi:10.1109/SP40001.2021.00056.

Supplementary Material

A Missing Functionalities and Protocols

Channels. This functionality models required communication channels.

Functionality 7: $\mathcal{F}_{Channels}$

The functionality proceeds as follows:

- **Pairwise**: On input (Message, x, P_i, P_j) from P_i, send x to P_j.
- **Broadcast**: On input (Broadcast, x, P_i) from P_i, send x to all parties.
- **Simultaneous**: On input (Simultaneous, x_i, P_i) from each party P_i, store this value. Do not send $\{x_i\}_{i\in[n]}$ to each party until all parties have provided inputs[a].

[a] This command aims to commit to input of each party.

Affine combinations. The parties could use π_{Aff} to locally compute the affine combination of $\langle\cdot\rangle$-share with coefficients $a_1, \cdots, a_\ell \in \mathbb{F}_q$.

Procedure 7: $\pi_{Aff}((\langle X_1\rangle, \cdots, \langle X_\ell\rangle), (a_1 \cdots, a_\ell))$

Given ℓ shared values $\langle X_j\rangle = (X_j^{(i)}, \boldsymbol{v}^{(i)}, \boldsymbol{m}^{(i)}(X_j))_{i\in[n]}$ for $j \in [\ell]$ and ℓ constant scalars $(a_1, \cdots, a_\ell)$, all parties can execute following operations to obtain shares of $Y = \sum_{j=1}^{\ell} a_j X_j$.

1. All parties locally compute

$$Y^{(i)} = \sum_{j=1}^{\ell} a_j X_j^{(i)}, \quad \boldsymbol{m}^{(i)}(Y) = \sum_{j=1}^{\ell} a_j \boldsymbol{m}^{(i)}(X_j)$$

2. The parties store the new shared value $\langle Y\rangle = (Y^{(i)}, \boldsymbol{v}^{(i)}, \boldsymbol{m}^{(i)}(Y))_{i\in[n]}$.

Opening and checking. The following procedures could allow the parties to partially open and check the correctness of opened values, respectively.

Procedure 8: $\pi_{Open}(\langle X\rangle)$

Given a share value $\langle X\rangle = (X^{(i)}, \boldsymbol{v}^{(i)}, \boldsymbol{m}^{(i)}(X))$:

1. All parties send their share $X^{(i)}$ to P_1
2. P_1 reconstructs $X = X^{(1)} + \cdots + X^{(n)}$ and broadcasts X to all parties.

Procedure 9: $\pi_{Check}(X', \langle X \rangle)$

Given an opened value X and a shared value $\langle X \rangle = (X^{(i)}, \boldsymbol{v}^{(i)}, \boldsymbol{m}^{(i)}(X))$:

1. All parties locally compute $\boldsymbol{\sigma}^{(i)} = \boldsymbol{m}^i(X) - X\boldsymbol{v}^{(i)}$ and broadcast this value via the simultaneous message channel.
2. All parties locally compute $\boldsymbol{\sigma} = \boldsymbol{\sigma}^{(1)} + \cdots + \boldsymbol{\sigma}^{(n)}$ and verify whether $\boldsymbol{z} = \boldsymbol{0}$. If the answer is no, abort.

Random additive secret sharing. This procedure generates a random additive secret sharing $[X]$.

Procedure 10: π_{Rand}

1. Each party P_i samples a random matrix $X^{(i)}$.
2. Output $[X] = (X^{(1)}, \ldots, X^{(n)})$ with $X = \sum_{i=1}^{n} X^{(i)}$.

Preprocessing protocol. Commands including **Initialize**, **Authenticate** and **Sextuple** can be done using essentially the same protocols as Π_{Auth} and Π_{Tuple}. Thus, it remains to complete this protocol by describing the **Input** command. In particular, P_i samples the random masks $R_0, R_1, \cdots, R_\ell \xleftarrow{\$} \mathcal{M}_{m\times m}(\mathbb{F}_q)$ and distributes random shares of $R_0, \ldots, R_\ell$ to other parties. Then all parties except P_i call functionality $\mathcal{F}_{VOLE}$ with P_i to generate the MAC of $\langle R_0 \rangle, \ldots, \langle R_\ell \rangle$. By use the same MAC checking procedure as in Π_{Auth}, we obtain the authenticated sharings $\langle R_1 \rangle, \ldots, \langle R_\ell \rangle$.

Protocol 11: Π_{Prep}

The protocol keeps a dictionary Val.

- **Initialize**: Same as in Π_{Auth}.
- **Authenticate**: Same as in Π_{Auth}.
- **Sextuple**: Same as in Π_{Tuple}.
- **Input**: On input ($\mathsf{InputPrep}$, P_i) from all parties do the following to create ℓ random authenticated mask:
  1. P_i randomly samples $R_0, R_1, \cdots, R_\ell \xleftarrow{\$} \mathcal{M}_{m\times m}(\mathbb{F}_q)$.
  2. For $h \in \{0\} \cup [\ell]$, P_i randomly samples $\{R_h^{(j)}\}_{j\in[n]}$ such that $\sum_{j=1}^{n} R_h^{(j)} = R_h$ and distributes $R_h^{(j)}$ to P_j.
  3. For $h \in \{0\} \cup [\ell]$, write $R_h = (\boldsymbol{r}_{h,1}, \cdots, \boldsymbol{r}_{h,m})$:
     (a) For $k \in [m]$ and each $j \neq i$, P_i and P_j call the **Multiply** step of $\mathcal{F}_{VOLE}^k$, where P_i inputs $\boldsymbol{r}_{h,k}$.
     (b) P_i receives $\boldsymbol{u}_{h,k}^{(i,j)}$ and P_j receives $\boldsymbol{v}_{h,k}^{(j,i)}$ such that $\boldsymbol{w}_{h,k}^{(j,i)} = \boldsymbol{u}_{h,k}^{(i,j)} + v_k^{(j)} \boldsymbol{r}_{h,k}^{(i)}$.
     (c) P_i sets $\boldsymbol{m}^{(i)}(R_h) = R_h \boldsymbol{v}^{(i)} - \sum_{k=1}^{m} \sum_{j\neq i} \boldsymbol{u}_{h,k}^{(i,j)}$ and P_j sets $\boldsymbol{m}^{(j)}(R_h) = \sum_{k=1}^{m} \boldsymbol{w}_{h,k}^{(j,i)}$.

4. Parties call $\mathcal{F}_{Coin}$ ℓ times to obtain randomness $\chi_1, \cdots, \chi_\ell$.
5. Parties locally compute $\langle Y \rangle = \langle R_0 \rangle + \sum_{h=1}^{\ell} \chi_h \langle R_h \rangle$.
6. Parties invoke $Y' \leftarrow \pi_{Open}(\langle Y \rangle)$ and $\pi_{check}(Y', \langle Y \rangle)$ to check the correctness of opened value.
7. If the check succeeds, output $\langle R_1 \rangle, \ldots, \langle R_\ell \rangle$.

B Missing Proofs

B.1 Proof of the Online Phase

Theorem 4 (Theorem 1,restated). *Protocol Π_{Online} securely implements $\mathcal{F}_{MPC}$ in the ($\mathcal{F}_{Prep}$, $\mathcal{F}_{Coin}$)-hybrid model.*

Proof. Let $\mathcal{Z}$ be an environment corrupting a set of at most $n-1$ parties. We assume that $\mathcal{Z}$ plays the role of both the distinguisher and the adversary, who simply forwards messages sent and received by corrupted parties in the protocol as directed by the environment.

Recall that the environment's view is the collection of all intermediate messages that corrupted players send and receive, plus the inputs and outputs of all players. We will describe a simulator $\mathcal{S}$ who has the access to the ideal functionality $\mathcal{F}_{Prep}$ and $\mathcal{F}_{Coin}$ and interacts with $\mathcal{Z}$ in such a way that the real interaction and the simulated interaction are indistinguishable to $\mathcal{Z}$. The simulator $\mathcal{S}$ works as follows.

Simulating Initialize and Input command. The simulator simply emulates the functionality $\mathcal{F}_{Prep}$ honestly. Then, $\mathcal{S}$ knows each MAC key $\boldsymbol{v}^{(i)}$ held by P_i. Also $\mathcal{S}$ distributes random shares to the corrupt parties for every input gate and the multiplication sextuples for every multiplication gate.

In the Input command, when a P_i is honest, $\mathcal{S}$ broadcasts a random element $R \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$. When a corrupted party P_i broadcasts ϵ, $\mathcal{S}$ extracts its input as $X = \epsilon + R$, where R is the random value that P_i should have used. Then the simulator stores the values as input to the $\mathcal{F}_{MPC}$.

Simulating Addition and Public matrix multiplication command. These steps only consist of local computations which can be simulated trivially, where $\mathcal{S}$ carries out honestly on behalf of the virtual honest parties.

Simulating Multiplication command. When the values D, E and F are opened for multiplication, $\mathcal{S}$ opens random shares on behalf of the honest parties.

Simulating the Output and Check openings command. $\mathcal{S}$ first receives the output Y from $\mathcal{F}_{MPC}$. Next, $\mathcal{S}$ executes Check command with the adversary, on behalf of the virtual honest parties. If the check fails, $\mathcal{S}$ sends Abort. If the above check passes, $\mathcal{S}$ modifies the honest parties shares it holds to be consistent with the output Y, as well as the MAC shares to be consistent with $Y\boldsymbol{v}$. Then $\mathcal{S}$ runs the π_{Check} with the adversary, on behalf of the honest parties.

Indistinguishability. Now we argue that $\mathcal{Z}$ cannot distinguish between real and ideal executions. It is clear for Init command, because $\mathcal{Z}$ gets random values in both executions. In Input command, the values broadcast by the honest parties are uniformly at random in both worlds. It is also the case Mult command, where the adversary receives honest parties' shares of three fresh random values. These shares are uniformly at random in both of the worlds. The MAC shares of these opened values are also uniformly at random in both of the worlds, which are the random sharings of a correct MAC with an error added by the adversary in Input command.

In Output command, the probability that the Check command and a single π_{Check} result in abort is the same in both executions. Meanwhile, if the first step of Check command passes, then the honest parties will reveal their shares in both executions. In the real execution, these shares are conditioned on adding up to the value computed in the protocol with the shares provided by the adversary, whereas in the simulated execution this sum is equal to the value output by the functionality. Due to the sketch proof above, we know that this check will pass except with probability $2/q$. It is the same as the last single π_{Check}, which will pass except with probability $1/q$. As a result, we can say that in both executions the values are the same, except with probability $3/q$.

B.2 Proof of Authentication

Claim. If at least one $\boldsymbol{\epsilon}^{(i)} \neq \mathbf{0}$ for some $i \notin \mathcal{C}$, then consistency check passes with negligible probability.

Proof. Assume for $i \notin \mathcal{C}$, $\boldsymbol{\epsilon}^{(i)} \neq \mathbf{0}$. Note that $Y^{(i)}$ is honestly generated and its distribution is uniformly random in $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ due to the random mask $X_0^{(i)}$. If the consistency check passes, $Y^{(i)}\boldsymbol{\epsilon}^{(i)} = \boldsymbol{\delta}$ for some $\boldsymbol{\delta}$ that is independent of $Y^{(i)}$, which happens with probability q^{-m}.

Claim. If $\boldsymbol{\epsilon}^{(i)} = \mathbf{0}$ for all $i \notin \mathcal{C}$ and $\Delta_h^{(i)} \neq 0$ for some $i \notin \mathcal{C}$, then consistency check passes with negligible probability.

Proof. If $E \neq 0$, the adversary passes consistency check only if $E\sum_{i\notin\mathcal{C}} \boldsymbol{v}^{(i)} = \boldsymbol{\delta}$ for some $\boldsymbol{\delta}$ that is independent of $\left\{\boldsymbol{v}^{(i)}\right\}_{i\notin\mathcal{C}}$, which happens with probability q^{-1}. If $E = 0$ and $\Delta_h^{(i)} \neq 0$ for some $i \notin \mathcal{C}, h \in \{0\} \cap [\ell]$, the adversary needs to make error $\Delta_h^{(i)}$ satisfy $\Delta_h^{(i)}\boldsymbol{v}^{(i)} = \boldsymbol{\delta}'$ for some $\boldsymbol{\delta}'$ that is independent of $\boldsymbol{v}^{(i)}$. Such attack succeeds with probability at most q^{-1}.

Theorem 5 (Theorem 2,restated). *Protocol Π_{Auth} securely implements $\mathcal{F}_{Auth}$ in the ($\mathcal{F}_{VOLE}$, $\mathcal{F}_{Coin}$)-hybrid model.*

Proof. We define a simulator $\mathcal{S}$ such that an environment $\mathcal{Z}$ can only distinguish a real execution interacting with the honest parties and an ideal execution with the simulator $\mathcal{S}$ with a negligible probability.

Simulating Initialize command. For $k \in [m]$, let $v_k^{(j,i)}$ be the input of a corrupted party P_j toward an honest party P_i during the **Initialize** step of $\mathcal{F}_{VOLE}^k$. $\mathcal{S}$ fixes an honest party P_{i_0} and sends $\boldsymbol{v}^{(j)} = \left(v_1^{(j,i_0)}, \cdots, v_m^{(j,i_0)}\right)$ to $\mathcal{F}_{Auth}$ as the global key share of P_j. For $i \notin \mathcal{C}$, $\mathcal{S}$ samples $\boldsymbol{v}^{(i)} \xleftarrow{\$} \mathbb{F}_q^m$.

Simulating Authenticate command.

1. For $i \notin \mathcal{C}$, randomly sample $X_0^{(i)} \xleftarrow{\$} \mathcal{M}_{m\times m}(\mathbb{F}_q)$.
2. For $h \in \{0\} \cup [\ell]$:
   (a) For all $j \in \mathcal{C}, i \notin \mathcal{C}$ and $k \in [m]$, $\mathcal{S}$ receives $\boldsymbol{x}_{h,k}^{(j,i)}, \boldsymbol{u}_{h,k}^{(j,i)}$ and $\boldsymbol{w}_{h,k}^{(j,i)}$ from the adversary.
   (b) Honestly compute the MAC share $\boldsymbol{m}^{(i)}(X_h)$ for $i \notin \mathcal{C}$ with the simulator of $\mathcal{F}_{VOLE}$.
3. Randomly sample and send $r_1, \cdots, r_\ell$ to the adversary.
4. Send adversary the honestly computed share $Y^{(i)}$ for $i \notin \mathcal{C}$ and receive $Y^{(j)}$ for $j \in \mathcal{C}$ from the adversary to reconstruct Y'.
5. Honestly compute $\boldsymbol{\sigma}^{(i)} = \boldsymbol{m}^{(i)}(Y) - Y'\boldsymbol{v}^{(i)}$ for $i \notin \mathcal{C}$ and receive $\boldsymbol{\sigma}^{(j)}$ from the adversary.
6. Execute the consistency check. If it fails, then send Abort to $\mathcal{F}_{Auth}$.
7. If no abort happens, for $j \in \mathcal{C}$ and $h \in [\ell]$, compute $\boldsymbol{m}^{(j)}(X_h)$ with $\boldsymbol{x}_{h,k}^{(j)}, \boldsymbol{v}^{(j)}, \boldsymbol{u}_{h,k}^{(j,i)}$ and $\boldsymbol{w}_{h,k}^{(j,i)}$, then send $(X_h^{(j)}, \boldsymbol{m}^{(j)}(X_h))$ to the adversary.

Indistinguishability. It is easy to observe that the transcript for messages inspected by the adversary has the identical distribution in ideal and real executions. In the previous analysis, we argue that if the adversary introduces additive errors that result in a fake authenticated sharing, the consistency check passes with negligible probability, therefore the probability of passing consistency check is almost identical in the two worlds. Finally, we show that the distribution of honest parties' MACs is identical in both worlds since $\mathcal{F}_{VOLE}$ outputs random vectors, which serve as a random mask.

B.3 Proof of Sextuple Generation

Claim. If the sacrifice step passes, then $E = E_1 + E_2 + E_{Auth} = 0$ and $E' = E_1' + E_2' + E_{Auth}'$ with overwhelming probability.

Proof. If the protocol does not abort in sacrifice step, then $\chi C - C' - DB = 0$. Since $C = AB + E$ and $C' = AB + E'$, we have that

$$\begin{aligned} \chi C - C' - DB &= 0 \\ \chi(AB + E) - (A'B + E') - (\chi A - A')B &= 0 \\ \chi E - E' &= 0 \end{aligned}$$

Such equation holds with probability q^{-1}.

Claim. If the sacrifice step passes, then $\{\boldsymbol{\epsilon}_k^{(i)}\}_{i\notin\mathcal{C},k\in[m]}$ are zero with overwhelming probability.

Proof. Due to the previous claim, if sacrifice step passes, then following equation holds

$$\begin{aligned} E_1 + E_2 + E_{Auth} &= 0 \\ -E_1 - E_{Auth} &= \sum_{i\notin\mathcal{C}} A^{(i)} \left(\boldsymbol{\epsilon}_1^{(i)}, \cdots, \boldsymbol{\epsilon}_m^{(i)}\right) \end{aligned}$$

where $\{A^{(i)}\}_{i\notin\mathcal{C}}$ is distributed uniformly at random and other items are independent of $\{A^{(i)}\}_{i\notin\mathcal{C}}$. Suppose that $\boldsymbol{\epsilon}_k^{(i)} \neq \mathbf{0}$ for some $k \in [m], i \notin \mathcal{C}$, then adversary needs to make $A^{(i)}\boldsymbol{\epsilon}_k^{(i)} = \boldsymbol{\delta}$ from some $\boldsymbol{\delta}$ is independent of $A^{(i)}$ to pass the sacrifice step, which happens with probability q^{-m}.

Theorem 6 (Theorem 3,restated). *Protocol Π_{Tuple} securely implements $\mathcal{F}_{Tuple}$ in the ($\mathcal{F}_{Auth}$, $\mathcal{F}_{RVOLE}$, $\mathcal{F}_{Coin}$)-hybrid model.*

Proof. Here we provide the supplementary proof of the security of π_{Double}.

Let $\mathcal{Z}$ be the environment, which we also refer to as adversary, corrupting a set $\mathcal{C}$ containing at most $n-1$ parties. We construct a simulator $\mathcal{S}$ such that the real execution and ideal execution is indistinguishable to $\mathcal{Z}$.

Simulating the Double step The simulator $\mathcal{S}$ emulates the functionality $\mathcal{F}_{Auth}$ with inputs from the adversary. Similarly, $\mathcal{S}$ just emulates the $\mathcal{F}_{Coin}$ to obtain $\{r_i\}_{i\in[2\ell]}$ and executes local computations. Note that every pair of the double sharings (A_i, A_i^T) for $i \in \{0\} \cup [2\ell]$ will be introduced errors in the two steps above, which we denote by (E_i, E_i'). Then, $\mathcal{S}$ runs the procedure π_{Check} on behalf of the virtual honest parties.

Indistinguishability Now we argue that $\mathcal{Z}$ cannot distinguish real execution and simulated one. Define $E_C = \sum_{i=0}^{2\ell} r_i E_i, E_D = \sum_{i=0}^{2\ell} r_i E_i'$, where $r_0 = 1$. The first check will pass if adversary make the sum of the weighted errors equal, that is to say $E_C = E_D$. To pass the second check, the key of the problem returns to the classic check if we denote errors of the MAC sharings added in the π_{Check} procedure by $\boldsymbol{\delta}_C, \boldsymbol{\delta}_D$, which satisfy that:

$$\begin{aligned} E_C\boldsymbol{v} &= \boldsymbol{\delta}_C \\ E_D\boldsymbol{v} &= \boldsymbol{\delta}_D \end{aligned}$$

Therefore adversary could introduce non-zero errors E_C, E_D with only negligible probability q^{-1}.

C Function Dependent Preprocessing

The downside of our protocol in Section 3 is that each multiplication gate requires 2 rounds of interactions while the original SPDZ protocol only needs one round. Recently, MPC protocols with function dependent preprocessing have been proposed in both honest majority [17,15] and dishonest majority setting [7,16]. By utilizing this idea, we can further reduce the round complexity and communication complexity of our MPC protocol. We briefly review the necessary changes for this improvement.

The value X is not represented by the authenticated sharing $\langle X \rangle$, but a pair $(\langle \Lambda_X \rangle, \Phi_X)$, where $\Lambda_X \xleftarrow{\$} \mathcal{M}_{m\times m}(\mathbb{F}_q)$ and the difference $\Phi_X = X - \Lambda_X$ is open to all parties. Note that Φ_X is masked with uniformly random Λ_X, therefore its exposure leaks no information about X.

Assume that all parties have two variants $(\langle \Lambda_X \rangle, \Phi_X)$ and $(\langle \Lambda_Y \rangle, \Phi_Y)$. For an addition gate, all parties just need to execute local additions to obtain $(\langle \Lambda_X + \Lambda_Y \rangle, \Phi_X + \Phi_Y)$ as the sharing of $X + Y$. For a multiplication gate, all parties choose a random mask $\langle \Lambda_Z \rangle$ and the main task is to compute the public difference Φ_Z. Following the analysis in Section 3, we could obtain:

$$\langle \Phi_Z \rangle = \Phi_X \Phi_Y + \langle \Lambda_X \Phi_Y \rangle + \Phi_X \langle \Lambda_Y \rangle + \langle \Lambda_X \Lambda_Y \rangle - \langle \Lambda_Z \rangle$$

We need to compute $\langle \Lambda_X \Phi_Y \rangle$ in the absence of right linearity and use the same approach to partially open $\Phi_Y^T \langle \Lambda_X^T \rangle - \langle R^T \rangle$, where $R \xleftarrow{\$} \mathcal{M}_{m\times m}(\mathbb{F}_q)$. Since Φ_Y is known to all parties in the function dependent model, this computation can be done locally. Thus, to compute a multiplication gate, in the preprocessing phase, we need to prepare $(\langle \Lambda_X \rangle, \langle \Lambda_X^T \rangle, \langle \Lambda_Y \rangle, \langle \Lambda_X \Lambda_Y \rangle, \langle R \rangle, \langle R^T \rangle, \langle \Lambda_Z \rangle)$, where $R \xleftarrow{\$} \mathcal{M}_{m\times m}(\mathbb{F}_q)$, $\Lambda_X, \Lambda_Y, \Lambda_Z$ are masked values aligned to X, Y, Z, respectively.

We define the functionality $\mathcal{F}_{FD-Prep}$ to describe the function dependent preprocessing as Functionality 8. We can slightly modify Π_{Prep} to instantiate this functionality. Based on $\mathcal{F}_{FD-Prep}$, we could instantiate $\mathcal{F}_{MPC}$ as Protocol 12. To avoid confusion with Π_{Online}, we denote this instantiation as $\Pi_{FD-Online}$.

Functionality 8: $\mathcal{F}_{FD-Prep}$

The functionality maintains a dictionary Val, which keeps a track of authenticated elements in $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ (Note that Val stores $\langle \Lambda_X \rangle$ instead of $\langle X \rangle$). This functionality has all the same commands in $\mathcal{F}_{Auth}$ with following additional commands:

- **Input**: On input $(\mathsf{InputPrep}, \mathtt{id}, P_i)$ from all parties, sample $\Lambda_X \xleftarrow{\$} \mathcal{M}_{m\times m}(\mathbb{F}_q)$, store $\mathsf{Val}[\mathtt{id}] = \Lambda_X$ and return Λ_X to P_i.
- **Addition**: On input $(\mathsf{AddPrep}, \mathtt{id}, (\mathtt{id}_1, \mathtt{id}_2))$ from all parties, compute $\Lambda_Z = \mathrm{Val}[\mathtt{id}_1] + \mathrm{Val}[\mathtt{id}_2]$ and store $\mathsf{Val}[\mathtt{id}] = \Lambda_Z$.

- **Public matrix multiplication**: On input ($\mathsf{PubMulPrep}$, $\mathtt{id}, A$) from all parties, compute $\Lambda_Z = A\mathsf{Val}[\mathtt{id}]$ and store $\mathsf{Val}[\mathtt{id}] = \Lambda_Z$.
- **Multiplication**: On input ($\mathsf{MultPrep}, \mathtt{id}, (\mathtt{id}_1, \mathtt{id}_2)$) from all parties, do following operations and return the tuple $\big(\langle\Lambda_X\rangle, \langle\Lambda_X^T\rangle, \langle\Lambda_Y\rangle, \langle\Lambda_X\Lambda_Y\rangle, \langle R\rangle, \langle R^T\rangle, \langle\Lambda_Z\rangle\big)$:
  - Set $\Lambda_X = \mathsf{Val}[\mathtt{id}_1]$ and $\Lambda_Y = \mathsf{Val}[\mathtt{id}_2]$
  - Generate authenticated sharings $\langle\Lambda_X^T\rangle$ and $\langle\Lambda_X\Lambda_Y\rangle$
  - Sample $\Lambda_Z \xleftarrow{\$} \mathcal{M}_{m\times m}(\mathbb{F}_q)$ and store $\mathsf{Val}[\mathtt{id}] = \Lambda_Z$.
  - Sample $R \xleftarrow{\$} \mathcal{M}_{m\times m}(\mathbb{F}_q)$ and obtain a double sharing $(\langle R\rangle, \langle R^T\rangle)$.

Protocol 12: $\Pi_{FD-Online}$

The parties maintain a dictionary Val for authenticated secret sharings of masking values.

- **Initialize**: Each party samples $\boldsymbol{v}^{(i)} \xleftarrow{\$} \mathcal{M}_{m\times m}(\mathbb{F}_q)$ and set $\mathsf{Val} = \emptyset$. Call $\mathcal{F}_{FD-Prep}$ with the circuit as input.
- **Input**: If P_i receives (Input, $\mathtt{id}, X, P_i$) and other parties receive (Input, $\mathtt{id}, P_i$), P_i retrieves mask Λ_X associated to X and broadcasts $\Phi_X = X - \Lambda_X$ to all parties.
- **Addition**: If all parties receive (Add, $\mathtt{id}, (\mathtt{id}_1, \mathtt{id}_2)$), retrieve public differences of two inputs Φ_X, Φ_Y and set difference of output as $\Phi_Z = \Phi_X + \Phi_Y$.
- **Public matrix multiplication**: If all parties receive (PubMul, $\mathtt{id}, A$) from all parties, retrieve difference of input Φ_X and update it as $A\Phi_X$.
- **Multiplication**: If all parties receive (Mult, $\mathtt{id}, (\mathtt{id}_1, \mathtt{id}_2)$), retrieve $\langle\Lambda_X\rangle = \mathsf{Val}[\mathtt{id}_1]$ and $\langle\Lambda_Y\rangle = \mathsf{Val}[\mathtt{id}_2]$ and corresponding differences Φ_X, Φ_Y. Do the following:
  1. Obtain the corresponding multiplication sextuple $(\langle\Lambda_X\rangle, \langle\Lambda_X^T\rangle, \langle\Lambda_Y\rangle, \langle\Lambda_X\Lambda_Y\rangle, \langle R\rangle, \langle R^T\rangle, \langle\Lambda_Z\rangle)$.
  2. All parties locally compute $\langle D\rangle = \langle\Lambda_X\Lambda_Y\rangle + \Phi_X\langle\Lambda_Y\rangle + \Phi_X\Phi_Y - \langle\Lambda_Z\rangle + \langle R\rangle$ and $\langle E\rangle = \Phi_Y^T\langle\Lambda_X^T\rangle - \langle R^T\rangle$.
  3. All parties invoke $D \leftarrow \pi_{Open}(\langle D\rangle)$ and $E \leftarrow \pi_{Open}(\langle E\rangle)$.
  4. Set $\Phi_Z = D + E^T$.
- **Partially Opening**: Same as in Π_{Online}.
- **Check Opening**: Same as in Π_{Online}.
- **Output**: When all parties output a variable Y, do the same as in Π_{Online} to open Λ_Y to all parties. Then reconstruct $Z = \Lambda_Y + \Phi_Y$.

Theorem 7. *Protocol $\Pi_{FD-Online}$ securely implements $\mathcal{F}_{MPC}$ in the ($\mathcal{F}_{FD-Prep}$, $\mathcal{F}_{Coin}$)-hybrid model.*