Dishonest Majority Multiparty Computation over Matrix Rings

Hongqing Liu, Chaoping Xing, Chen Yuan, Taoxu Zou

Shanghai Jiao Tong University, Shanghai, 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 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 to securely compute a matrix multiplication. We also show that the communication complexity of our MPC protocol is asymptotically as good as [16] which also presented a dishonest majority MPC protocol specialized for matrix operations, i.e., the communication complexity of securely computing a multiplication gate is $O(m^2n^2 \log q)$ in the preprocessing phase and $O(m^2 n \log q)$ in the online phase. The share size and the number of multiplications of our protocol are reduced by around 50% and 40% of [16], respectively. However, we take a completely different approach. The protocol in [16] uses a variant of BFV scheme to embed a whole matrix into a single ciphertext and then treats the matrix operation as the entry-wise operation in the ciphertext while our approach resorts to a variant of vector linear oblivious evaluation (VOLE) called the subfield VOLE [1] [33] which can securely compute the additive sharing of $v\boldsymbol{x}$ for $v \in \mathbb{F}_{q^b}, \boldsymbol{x} \in \mathbb{F}_q^a$ with sublinear communication complexity. Finally, we note that our MPC protocol can be easily extended to small fields.

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,

[1] In [33], there is a base VOLE which is also called subfield VOLE. The subfield VOLE in this paper is referred to the programmable VOLE $\Pi_{\mathsf{VOLE}}^{\mathsf{prog}}$ in [33] which silently generates correlated randomness from seeds.

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 from 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 MPC protocols over a ring are usually more complicated than those over a field. Before the advent of privacy preserving machine learning (PPML), most of the MPC protocols were 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 [18,31,3,2,24] proposed the MPC protocol over $\mathbb{Z}_{2^k}$. Recently, Escudero and Soria-Vazquez [23] 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[2], 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 [23], 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 [30,28,26,29]. It was observed in [16] 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

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

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 sublinear in the size of our secret assuming the size of our matrix is large enough. Utilizing the matrix structure, our MPC protocol uses vector oblivious linear evaluation (VOLE) and vector oblivious product evaluation (VOPE) as functionalities to authenticate the sharing and create the sextuple for securely computing multiplication gates in the online phase. The goal of VOPE is to compute the additive sharing of the product of two matrices which can be adapted from the subfield VOLE in [33] with slight modification. In the preprocessing phase, our MPC protocol needs $O(n^2m^2 \log q)$ bits of communication to prepare a sextuple for multiplication gate which has the same asymptotic performance as the protocol in [16]. In the online phase, our MPC protocol requires $O(m^2 n \log q)$ bits of communication complexity to securely compute a multiplication gate which is also as efficient as the MPC protocol in [16]. However, the size of the secret sharing is half the size of the secret sharing scheme in [16] and the number of multiplications in our protocol is reduced to $3m^3 + 3m^2$ while the protocol in [16] requires $5m^3 + m^2$ multiplication. We also compare the communication cost and computation cost of preprocessing in concrete parameter settings for $m = 128, 256, 512, 1024$. When m grows, the communication complexity of our protocol grows more slowly than [16]. For $m = 512, 1024$, the communication complexity of our protocol turns out to be smaller than [16]. Moreover, our experimental results imply that the running time of our VOLE-based preprocessing phase is 2.0x-24.2x faster than that of (fully) homomorphic encryption based preprocessing phase [16].

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 vector $\boldsymbol{\delta} \in \mathcal{M}_{m\times 1}(\mathbb{F}_q)$ 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 [16] 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 sharings 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 exploit 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 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 subfield VOLE. The subfield VOLE was proposed in [12] to securely compute the additive sharings of $v\boldsymbol{x}$ for $\boldsymbol{x} \in \mathbb{F}_q^a, v \in \mathbb{F}_{q^b}$ where v and $\boldsymbol{x}$ are the random element and random vector input by P_A and P_B respectively. To minimize the communication cost, the random vector $\boldsymbol{x}$ is expanded by a random seed while the random element v is chosen by P_B. Treating v as a vector $\boldsymbol{v} \in \mathbb{F}_q^b$, then $v\boldsymbol{x}$ becomes a product of two vectors $\boldsymbol{x}\boldsymbol{v}^T = (x_iv_j)_{a\times b}$. In this sense, the subfield VOLE can securely compute the additive sharing of $\boldsymbol{x}\boldsymbol{v}^T$ for $\boldsymbol{x} \in \mathbb{F}_q^a, \boldsymbol{v} \in \mathbb{F}_q^b$. We slightly modify the subfield VOLE in [33] to allow both parties to utilize short seeds to generate their random inputs. We call this modified subfield VOLE, random vector oblivious product evaluation (VOPE). Assuming $b = O(a)$, our VOPE has $O(a \log q)$ communication complexity, which is sublinear in output size $ab \log q$.

Computing the product of matrices. We propose the VOPE to compute the additive sharings of the product of two random matrices whose communication complexity is the dominant part of the preprocessing phase. Observe that one can decompose the product AB of two matrices $A, B \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$ into the sum of vector product $\boldsymbol{a}_i \otimes \boldsymbol{b}_i := \boldsymbol{a}_i\boldsymbol{b}_i^T$, $i \in [m]$ where $\boldsymbol{a}_i$ is the i-th column of A and $\boldsymbol{b}_i^T$ is the i-th row of B, i.e., $AB = \sum_{i=1}^m \boldsymbol{a}_i \otimes \boldsymbol{b}_i$. As mentioned above, our VOPE can produce the additive sharings of $\boldsymbol{a}_i \otimes \boldsymbol{b}_i$. Thus, it suffices to invoke m times of VOPE to obtain the additive sharing of AB.

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 [24]. 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

[3] We use $[\cdot]$ and , $[\![\cdot]\!]$ to represent the sharing of a matrix and vector, respectively.

sharing only allows left multiplication, i.e., all parties can only locally compute $A\langle M\rangle = \langle AM\rangle$. 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 $F^T = (E^T A^T - R^T)^T = AE - R$. 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 [9,22], 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 in the Supplementary Material.

Migration to small field $\mathbb{F}_q$. The matrix in our MPC protocol can be defined over small fields as well. 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 intuition 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 $XV = \varDelta$ 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 sublinear in the size of our secret assuming $m \gg \frac{\kappa}{\log_2 q}$. In this sense, our MPC protocol can be defined over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ with any prime power q. There are also some modifications for our MPC protocol to be applicable to $\mathcal{M}_{m\times m}(\mathbb{F}_q)$. The details can be found in Section D in the Supplementary Material.

1.2 Overview of Our Technique

We assume that our MPC protocol over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ with large q. 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 VOPE which we have already discussed in Section 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.

Procedure π_{Mult} produces a triple $(\langle A\rangle, \langle B\rangle, \langle C\rangle)$. We want to compute $[C]$ from $[A] = \left(A^{(1)}, \dots, A^{(n)}\right)$ and $[B] = \left(B^{(1)}, \dots, B^{(n)}\right)$. 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 the sum of m vector products as we mentioned above, i.e., $AB = \sum_{i=1}^{m} \boldsymbol{a}_i \otimes \boldsymbol{b}_i$ where $\boldsymbol{a}_i$ is the i-th column of A and $\boldsymbol{b}_i^T$ is the i-th row of B. This implies that we only need to invoke m times VOPE to complete this work. We create seeds to generate the random matrix $A^{(i)}$, $B^{(i)}$ and reuse these seeds as inputs for the instances of VOPE. The use of VOPE can be found in the previous section. We fix B and apply the above process twice to $([A], [B])$ and $([A'], [B])$ to prepare two pairs $([A], [C]), ([A'], [C'])$ with $C = AB, C' = A'B$. Then, we invoke 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 authenticated 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 Section 1.1.

1.3 Related Work

There are a few MPC protocols optimized for matrix operations. Escudero and Soria-Vazquez [23] 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 [20]. 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 [16] which presented a variant of SPDZ protocol over matrix rings $\mathcal{M}_{m\times m}(\mathbb{Z}_q)$, where $\mathbb{Z}_q$ is a large prime field. 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 multiplication [26] which is based on a variant of BFV scheme [14,25] 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.

In the preprocessing phase, we apply a variant of PCG-based subfield VOLE to securely multiply two random matrices. In [13], Boyle et al. proposed a PCG construction for matrix triple, which is adapted from the PCG for OLE under "splittable" ring-LPN assumption. However, their protocol generates a large batch of matrix triples of small-to-medium size (at most 16×16), while our protocol can deal with the matrices of large size (at least 128×128).[4]

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 analyze the communication complexity of our MPC protocol and compare it with other dishonest majority MPC protocols over matrix rings. The missing functionalities and protocols can be found in Section A in the Supplementary Material.

[4] In [13], they remarked "For larger matrix, more interactive approach such as the recent work based on homomorphic encryption [16] appears to be more practical".

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 $\boldsymbol{v}^T$. For a vector $\boldsymbol{v}$, denote by v_i the i-th component of $\boldsymbol{v}$, i.e., $\boldsymbol{v}^T = (v_1, \ldots, v_n)$. Let $\mathcal{M}_{a\times b}(\mathbb{F}_q)$ be the collection of $a \times b$ matrices over $\mathbb{F}_q$. For two column vectors $\boldsymbol{u} \in \mathbb{F}_q^a, \boldsymbol{v} \in \mathbb{F}_q^b$, we use $\boldsymbol{u} \otimes \boldsymbol{v} = \boldsymbol{u}\boldsymbol{v}^T \in \mathcal{M}_{a\times b}(\mathbb{F}_q)$ to represent their (outer) product, i.e., $\boldsymbol{u}\boldsymbol{v}^T = (u_i v_j)_{a\times b}$ where $\boldsymbol{u}^T = (u_1, \ldots, u_a)$ and $\boldsymbol{v}^T = (v_1, \ldots, v_b)$.

Throughout the paper, the security parameter of MPC protocol is κ. Let $\mathbb{F}_q$ be the finite field of size q and $\mathbb{F}_q^n$ be the vector space of n dimension. We denote by $x \overset{\$}{\leftarrow} \mathcal{X}$ 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 consider 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 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 [15]. 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 (unanimous) 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 by introducing the authenticated secret sharing of a matrix, which is the building block of our MPC protocol. Protocol Π_{online} securely implements MPC functionality $\mathcal{F}_{\mathsf{MPC}}$ in the $(\mathcal{F}_{\mathsf{Prep}}, \mathcal{F}_{\mathsf{Coin}})$-hybrid model, where $\mathcal{F}_{\mathsf{Prep}}$ generates correlated randomness in offline phase and $\mathcal{F}_{\mathsf{Coin}}$ generates public random field elements. The implementation of $\mathcal{F}_{\mathsf{Prep}}$ can be found in Section 5.

3.1 Authenticated Secret Sharing

In the dishonest majority setting, additive secret sharing alone is not resilient to the corruption caused by the malicious adversary. Similar to [19], we use a uniformly random global key to generate the MAC of the share. Our approach deviates from [19] by making the global key and MACs as a vector 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 $\mathcal{M}_{m\times 1}(\mathbb{F}_q)$[5], respectively. An authenticated secret sharing $\langle X\rangle$ is a triple $([X], [\![\boldsymbol{v}]\!], [\![X\boldsymbol{v}]\!])$, where $X \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$ is the secret, $\boldsymbol{v} \overset{\$}{\leftarrow} \mathcal{M}_{m\times 1}(\mathbb{F}_q)$ 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, $[X] = \big(X^{(1)}, \cdots, X^{(n)}\big)$, $[\![\boldsymbol{v}]\!] = \big(\boldsymbol{v}^{(1)}, \cdots, \boldsymbol{v}^{(n)}\big)$ and $([\![X\boldsymbol{v}]\!]) = \big(\boldsymbol{m}^{(1)}(X), \cdots, \boldsymbol{m}^{(n)}(X)\big)$ 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. 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

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

[5] 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.

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

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

Then, $\langle X+A \rangle = ([X+A], [\![\boldsymbol{v}]\!], [\![(X+A)\boldsymbol{v}]\!])$ is the authenticated secret sharing of $X+A$. The affine operation can be found in procedure π_{Aff} in Section A in the Supplementary Material.

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 verify the opened value Y', all parties locally compute $[\![\boldsymbol{\sigma}]\!] = [\![Y\boldsymbol{v}]\!] - Y'[\![\boldsymbol{v}]\!]$, and broadcast the shares of this value via a simultaneous message channel. The parties abort if the reconstructed value σ 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 in the Supplementary Material.

Multiplication. In dishonest majority MPC protocols, correlated randomness generated in offline phase could assist the computation of multiplications. Beaver triple [8] 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 non-commutative property of matrix ring.

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 the preprocessing 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

The functionality $\mathcal{F}_{\mathsf{MPC}}$ enables the parties to securely share their inputs, perform linear operations and multiplications, and output the result. The functionality $\mathcal{F}_{\mathsf{Prep}}$ is used to prepare correlated randomness for $\mathcal{F}_{\mathsf{MPC}}$.

Authenticating functionality $\mathcal{F}_{\mathsf{Auth}}$. 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 $\mathcal{F}_{\mathsf{Auth}}$ directly, $\mathcal{F}_{\mathsf{Auth}}$ is contained in $\mathcal{F}_{\mathsf{Prep}}$.

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

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 $\mathcal{F}_{\mathsf{Prep}}$. This functionality produces random sharings for input gates and multiplication sextuples for multiplication gates.

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

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

- **Input**: On input $(\mathsf{InputPrep}, P_i)$ from all parties, sample $R \overset{\$}{\leftarrow} \mathcal{M}_{m\times m}(\mathbb{F}_q)$ and generate its authenticated sharing $\langle R \rangle$ such that for $j \in \mathcal{C}$, $\left(R^{(j)}, \boldsymbol{m}^{(j)}(R)\right)$ is chosen by the adversary. Output R to P_i and $\left(R^{(j)}, \boldsymbol{m}^{(j)}(R)\right)$ to P_j for all $j \notin \mathcal{C} \cup \{i\}$.
- **Sextuple**: On input (Tuple) from all parties, sample $A, B, R \overset{\$}{\leftarrow} \mathcal{M}_{m\times m}(\mathbb{F}_q)$ 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 $\mathcal{F}_{\mathsf{MPC}}$. This functionality provides all the necessary operations for our MPC protocol.

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

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 $[\![\boldsymbol{v}]\!]$.

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

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

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

- If the adversary returns OK, send r to all honest parties.
- If the adversary returns Abort, send Abort to all honest parties.

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

The protocol Π_{online} instantiates $\mathcal{F}_{\mathsf{MPC}}$ in the $(\mathcal{F}_{\mathsf{Prep}}, \mathcal{F}_{\mathsf{Coin}})$-hybrid model, with statistical security parameter κ. The random shares and multiplication sextuples produced in $\mathcal{F}_{\mathsf{Prep}}$ 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 $\mathcal{F}_{\mathsf{Prep}}$ as follows:
  1. On input (Init) to get global key $[\![\boldsymbol{v}]\!]$.
  2. On input $(\mathsf{InputPrep}, P_i)$ 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 $(\mathsf{Input}, \mathtt{id}, X, P_i)$ and other parties receive $(\mathsf{Input}, \mathtt{id}, P_i)$, 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 $\mathsf{Val}[\mathtt{id}] = \langle X\rangle$.
- **Addition**: If all parties receive (Add, $\mathtt{id}, \mathtt{id}_1, \mathtt{id}_2$), retrieve $\langle X\rangle = \mathsf{Val}[\mathtt{id}_1]$ and $\langle Y\rangle = \mathsf{Val}[\mathtt{id}_2]$, locally compute $\langle Z\rangle = \langle X\rangle + \langle Y\rangle$ and set $\mathsf{Val}[\mathtt{id}] = \langle Z\rangle$.
- **Public matrix multiplication**: If all parties receive (PubMul, $\mathtt{id}, A$), retrieve $\langle X\rangle = \mathsf{Val}[\mathtt{id}]$, locally compute $\langle Z\rangle = A\langle X\rangle$ and set $\mathsf{Val}[\mathtt{id}] = \langle Z\rangle$.
- **Multiplication**: If all parties receive (Mult, $\mathtt{id}, \mathtt{id}_1, \mathtt{id}_2$), retrieve $\langle X\rangle = \mathsf{Val}[\mathtt{id}_1]$ and $\langle Y\rangle = \mathsf{Val}[\mathtt{id}_2]$ 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 partially open $D \leftarrow \pi_{\mathsf{Open}}(\langle D\rangle)$ and $E \leftarrow \pi_{\mathsf{Open}}(\langle E\rangle)$.
   4. All parties locally compute $\langle F\rangle \leftarrow E^T\langle A^T\rangle - \langle R^T\rangle$ and partially open $F \leftarrow \pi_{\mathsf{Open}}(\langle F\rangle)$
   5. All parties locally compute $\langle Z\rangle = \langle C\rangle + D\langle B\rangle + \langle R\rangle + DE + F^T$ and set $\mathsf{Val}[\mathtt{id}] = \langle Z\rangle$.
- **Check openings**: If all parties receive (Check, $(\mathtt{id}_1, \cdots, \mathtt{id}_\ell)$, $(X'_1, \cdots, X'_\ell)$), retrieve $\langle X_j\rangle = \mathsf{Val}[\mathtt{id}_j]$ for $j \in [\ell]$ and execute following operations:
   1. Call $\mathcal{F}_{\mathsf{Coin}}$ 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 $\pi_{\mathsf{Check}}(Y', \langle Y\rangle)$.
- **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 obtain Y'.
   3. All parties invoke $\pi_{\mathsf{Check}}(Y', \langle Y\rangle)$. If this procedure passes, output Y'.

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

Proof. A full-fledged simulation-based proof is presented in Section B.1 in the Supplementary Material. 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 $\mathcal{F}_{\mathsf{MPC}}$. In the Input command, the value stored by $\mathcal{F}_{\mathsf{MPC}}$ corresponds to the value stored by Π_{Online}, which can be seen authenticated through the mask of a 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 $\mathcal{F}_{\mathsf{MPC}}$ is easy to be verified. The parties obtain 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 $X, Y \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$. The parties first partially opens $D \leftarrow X - A$ and $E \leftarrow Y - B$, and then compute locally $[Z] = [C] + D[B] + [A]E + DE$, which is equivalent to $[Z] = [XY]$. The third value $F^T \leftarrow E^T A^T - R^T$ is opened to compute the MAC of Z. The parties can locally compute $[\![Z\boldsymbol{v}]\!] = [\![C\boldsymbol{v}]\!] + D[\![B\boldsymbol{v}]\!] + F[\![\boldsymbol{v}]\!] + [\![R\boldsymbol{v}]\!] + DE[\![\boldsymbol{v}]\!]$. We can verify that the formula is equivalent to $[\![Z\boldsymbol{v}]\!] = [\![XY\boldsymbol{v}]\!]$. Note that each time we partially open a value, we compute its MAC. This MAC will be used in the Check command to check the correctness of this opened value. The privacy argument is clear as we always mask our secret with a random matrix when we want to do partially opening.

In command Trans, the security follows from the fact that $\langle R\rangle$ acts as a random mask of $\langle X\rangle$ and therefores opening of M leaks no information about $\langle X\rangle$. The correctness of this opening is guaranteed by Check command. Finally, in the Check and Output command, we can prove that a corrupted authenticated secret sharing will pass the verification with a negligible probability due to the following game which first appears in [20].

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 $X_1, \dots, X_\ell$ to the adversary.
2. The adversary sends back $X'_1, \dots, X'_\ell$.
3. The challenger generates the random values $r_1, \dots, r_\ell \in \mathbb{F}_q$.
4. The adversary provides an error $\boldsymbol{\delta} = (\delta_1, \dots, \delta_m)^T$.
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 parties have the option to lie about the secret shares that 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 $(x_{a,1}, \dots, x_{a,m})$. We have $\delta_a = \sum_{j=1}^{m} x_{a,j}v_j$. Since $\boldsymbol{v} = (v_1, \dots, v_m)^T$ 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 $r_1, \dots, r_\ell$ are random elements, the probability that $\sum_{i=1}^{\ell} r_iE_i = 0$ 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 $\mathcal{F}_{\mathsf{VOLE}}$. 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 $\boldsymbol{w}$ and computes $\boldsymbol{u} = v\boldsymbol{x} + \boldsymbol{w}$. We need to invoke VOLE multiple times and thus we attach a unique identifier sid to each instance[6]. The efficient instantiation of $\mathcal{F}_{\mathsf{VOLE}}$ can be found in [4,5].

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

The functionality runs between sender P_A and receiver P_B. The **Initialize** step runs once at the beginning and the **Multiply** step could run multiple 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 $\boldsymbol{w} \xleftarrow{\$} \mathbb{F}_q^m$. If P_B is corrupted, receive $\boldsymbol{w}$ from adversary.
  2. Compute $\boldsymbol{u} = v\boldsymbol{x} + \boldsymbol{w}$. If P_A is corrupted, receive $\boldsymbol{u}$ from adversary and recompute $\boldsymbol{w} = \boldsymbol{u} - v\boldsymbol{x}$.
  3. Output $\boldsymbol{u}$ to P_A and $\boldsymbol{w}$ to P_B.

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

Now we proceed to the generation of MAC shares. Each party P_i randomly samples the global key share $\boldsymbol{v}^{(i)}$ when command Init is invoked. To authenticate a given share $\left\{X^{(i)}\right\}_{i\in[n]}$, all parties jointly compute the additive sharing of $\left(\sum_{i=1}^n X^{(i)}\right)\left(\sum_{i=1}^n \boldsymbol{v}^{(i)}\right)$. 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 can locally compute $X^{(i)}\boldsymbol{v}^{(i)}$ and each ordered pair (P_i, P_j) needs to interactively compute additive sharing of $X^{(i)}\boldsymbol{v}^{(j)}$, i.e., $\boldsymbol{u}^{(i,j)} + \boldsymbol{w}^{(j,i)} = X^{(i)}\boldsymbol{v}^{(j)}$, where P_i and P_j receives $\boldsymbol{u}^{(i,j)}$ and $\boldsymbol{w}^{(j,i)}$, respectively. 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 have $\sum_{i=1}^n \boldsymbol{m}^{(i)}(X) = X\boldsymbol{v}$, where $X = \sum_{i=1}^n X^{(i)}$ and $\boldsymbol{v}^{(i)} = \sum_{i=1}^n \boldsymbol{v}^{(i)}$, therefore $\boldsymbol{m}^{(i)}(X)$ is the MAC share of P_i.

Since matrix-vector multiplication is a natural generalization of scalar-vector multiplication, a pair (P_i, P_j) can generate the additive sharing of $X^{(i)}\boldsymbol{v}^{(j)}$ by invoking m VOLE instances. In the k-th invocation of $\mathcal{F}^k_{\mathsf{VOLE}}$, 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 $\boldsymbol{v}^{(j)}$. According to the definition of VOLE, P_i receives $\boldsymbol{u}_k^{(i,j)}$ and P_j receives $\boldsymbol{w}_k^{(j,i)}$ such that $\boldsymbol{u}_k^{(j,i)} = v_k^{(j)}\boldsymbol{x}_k^{(i)} + \boldsymbol{w}_k^{(i,j)}$. By setting $\boldsymbol{u}^{(i,j)} = \sum_{k=1}^m \boldsymbol{u}_k^{(i,j)}$ and $\boldsymbol{w}^{(j,i)} = -\sum_{k=1}^m \boldsymbol{w}_k^{(j,i)}$, P_i and P_j jointly generate the additive sharing of $X^{(i)}\boldsymbol{v}^{(j)}$. It is easy to verify the correctness.

[6] The unique identifier sid is locally shared among a pair of parties and thus is not a global identifier in n-party setting.

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

Invoking VOLE alone is not sufficient to generate authenticated sharings in the presence of a malicious adversary. Because a corrupted party P_j may 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 attack, we introduce a consistency check which partially open 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 to mask the opened value. Although such a check can not guarantee the consistency of inputs in each invocation of $\mathcal{F}_{\mathsf{VOLE}}$, 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, all parties can obtain the authenticated sharings. Protocol Π_{Auth} is the instantiation of functionality $\mathcal{F}_{Auth}$ which outputs the authenticated sharings.

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 $\mathcal{F}_{\mathsf{VOLE}}^k$, where P_j inputs $v_k^{(j)}$.
- **Authenticate**: If all parties receive $(\mathsf{Auth}, [X_1], \ldots, [X_\ell])$:
  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 $\mathcal{F}_{\mathsf{VOLE}}^k$, 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 $\boldsymbol{u}_{h,k}^{(i,j)} = \boldsymbol{w}_{h,k}^{(j,i)} + v_k^{(j)} \boldsymbol{x}_{h,k}^{(i)}$.
     (c) Each party P_i sets $\boldsymbol{m}^{(i)}(X_h) = X_h^{(i)} \boldsymbol{v}^{(i)} + \sum_{j\neq i} \sum_{k\in[m]} (\boldsymbol{u}_{h,k}^{(i,j)} - \boldsymbol{w}_{h,k}^{(i,j)})$. Let $\left(X_h^{(i)}, \boldsymbol{v}^{(i)}, \boldsymbol{m}^{(i)}(X_h)\right)$ as the P_i's share of $\langle X_h \rangle$.
  3. Parties call $\mathcal{F}_{\mathsf{Coin}}$ 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 $Y' \leftarrow \pi_{\mathsf{Open}}(\langle Y \rangle)$ and $\pi_{\mathsf{check}}(Y', \langle Y \rangle)$ 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 $\mathcal{F}_{\mathsf{Auth}}$ in the ($\mathcal{F}_{\mathsf{VOLE}}$, $\mathcal{F}_{\mathsf{Coin}}$)-hybrid model.*

Proof. We analyze the consistency check in Π_{Auth} and defer the complete simulation-based security proof to Section B.2 in the Supplementary Material. 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 $\mathcal{F}_{\mathsf{VOLE}}^k$ 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., $\boldsymbol{\delta}_{h,k}^{(j,i)} = \boldsymbol{0}$ 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:

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

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:

$$\sum_{i=1}^{n} \boldsymbol{m}^{(i)}(Y) + \boldsymbol{\gamma} - (Y + E)\boldsymbol{v} = \mathbf{0}$$

$$\boldsymbol{\gamma} - E\boldsymbol{v} + \sum_{i \notin \mathcal{C}} \sum_{h=0}^{\ell} r_h \Delta_h^{(i)} \boldsymbol{v}^{(i)} + \sum_{i \notin C} Y^{(i)} \boldsymbol{\epsilon}^{(i)} = \mathbf{0}$$

$$\sum_{i \notin \mathcal{C}} \left(\sum_{h=0}^{\ell} r_h \Delta_h^{(i)} - E \right) \boldsymbol{v}^{(i)} + \sum_{i \notin C} Y^{(i)} \boldsymbol{\epsilon}^{(i)} = \sum_{i \in \mathcal{C}} E\boldsymbol{v}^{(i)} - \boldsymbol{\gamma}$$

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 in the Supplementary Material.

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 Preprocessing 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 multiplication sextuples. In Section A in the Supplementary Material, we describe the protocol Π_{Prep} for full-fledged preprocessing phase. To reduce the communication complexity of generating matrix triple, we introduce a variant of subfield VOLE called *vector oblivious product evaluation.* The process of generating multiplication sextuple is divided into two parts: the generation of Beaver triples $(\langle A \rangle, \langle B \rangle, \langle C \rangle)$ and double sharings $(\langle A \rangle, \langle A^T \rangle), (\langle R \rangle, \langle R^T \rangle)$.

5.1 Vector Oblivious Product Evaluation

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 relying on several variants of learning parity with noise (LPN) assumptions were used to construct random VOLE (RVOLE) [10,35,36,17,11,34]. While the communication complexity of original VOLE scales linearly in vector length, the communication complexity of PCG-based RVOLE is either the square root of vector length (under primal LPN assumption) [10,35,36] or logarithmic in vector length (under dual LPN assumption) [10,17,11,34]. In this work, we leverage the dual LPN assumption to reduce the communication cost.

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 seeds. The property *programmability* was introduced to PCG-based RVOLE in [12], which allows the sender to reuse its seed s in different instances of RVOLE. We model the programmability with function $\mathsf{Expand} : S \to \mathbb{F}_q^a$, which deterministically expands the given random seed to a pseudorandom vector of given length a over $\mathbb{F}_q$.

Boyle et al. [12], proposed a variant of RVOLE, called subfield VOLE, which can securely compute $\boldsymbol{u} = v\boldsymbol{x} + \boldsymbol{w}$, where $\boldsymbol{x} \in \mathbb{F}_q^a, v \in \mathbb{F}_{q^b}, \boldsymbol{u}, \boldsymbol{w} \in \mathbb{F}_{q^b}^a$. In a subfield VOLE instance between P_A and P_B, $\boldsymbol{x} \in \mathbb{F}_q^a$ is generated from a seed $s \in S$ chosen by P_A and $v \in \mathbb{F}_{q^b}$ is directly chosen by P_B. Thus, subfield VOLE could be regarded as a PCG for product of vectors, i.e., rewrite $v \in \mathbb{F}_{q^b}$ as $\boldsymbol{v} \in \mathbb{F}_q^b$ and the subfield VOLE actually computes the additive sharing of $\boldsymbol{x} \otimes \boldsymbol{v} \in \mathcal{M}_{a\times b}(\mathbb{F}_q)$. Since we have already shown that the product of two $m \times m$ matrices can be decomposed into the sums of products of the form $\boldsymbol{x} \otimes \boldsymbol{v} \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$, it suffices to invoke subfield VOLE m times to compute the matrix multiplication.

However, in subfield VOLE, the input $v \in \mathbb{F}_{q^b}$ is chosen uniformly at random which means the input size of P_B is $b \log q$ bits. Note that in our setting, a and b are of almost the same size $\Omega(m)$ which means it is necessary to minimize the input size from both sides. Thus, we modify this subfield VOLE by generating a pseudorandom element $v \in \mathbb{F}_{q^b}$ from a seed. We call this modified subfield VOLE vector oblivious product evaluation (VOPE). The functionality $\mathcal{F}_{\mathsf{VOPE}}^{a,b}$ can be found in Functionality 6. The instantiation of $\mathcal{F}_{\mathsf{VOPE}}^{a,b}$ is given in Section E.1 in the Supplementary Material, which is adapted from [33].

Functionality 6: $\mathcal{F}_{\mathsf{VOPE}}^{a,b}$

Let $\mathsf{Expand} : S \to \mathbb{F}_q^a, \mathsf{Expand}' : S' \to \mathbb{F}_q^b$ be the deterministic expansion functions with seed space S, S' and output length a, b, respectively. The functionality runs between sender P_A and receiver P_B.

Upon receiving $s \in S$ from P_A and $s' \in S'$ from P_B:

1. Compute $\boldsymbol{x} = \mathsf{Expand}(s), \boldsymbol{v} = \mathsf{Expand}'(s')$.
2. Sample $W \xleftarrow{\$} \mathcal{M}_{a,b}(\mathbb{F}_q)$. If P_B is corrupted, receive W from the adversary.
3. Compute $U = \boldsymbol{x} \otimes \boldsymbol{v} - W$. If P_A is corrupted, receive U from adversary and recompute $W = \boldsymbol{x} \otimes \boldsymbol{v} - U$.
4. Output U to P_A and W to P_B.

5.2 Generation of Beaver Triple

To simplify our proof, recall that we define $\boldsymbol{u} \otimes \boldsymbol{v} = \boldsymbol{u}\boldsymbol{v}^T$. The first step of generating Beaver triple is to securely compute matrix multiplication, which can be decomposed into some tensor products of vectors. Assume that there are two random matrices $A \in \mathcal{M}_{m_1 \times m_2}(\mathbb{F}_q), B \in \mathcal{M}_{m_2 \times m_3}(\mathbb{F}_q)$. Let $\boldsymbol{a}_i$ be the i-th column of A and $\boldsymbol{b}_i$ be the i-th row of B. Then, we have $AB = \sum_{i=1}^{m_2} \boldsymbol{a}_i \otimes \boldsymbol{b}_i$. This implies that it suffices to compute m_2 products $C_i = \boldsymbol{a}_i \otimes \boldsymbol{b}_i \in \mathcal{M}_{m_1 \times m_3}(\mathbb{F}_q)$, and then add them together to obtain $AB = C = \sum_{i \in [m_2]} C_i$.

Procedure π_{Mult} outputs the authenticated Beaver triples. Note that seeds s, s' will be reused several times for different pairs of parties. A corrupted party could cause the inconsistency of seeds towards different honest parties. Therefore, we add a consistency check at the end of π_{Mult}: To check the correctness of $(\langle A \rangle, \langle B \rangle, \langle C \rangle)$, we sacrifice another Beaver triple $(\langle A' \rangle, \langle B \rangle, \langle C' \rangle)$.

Procedure 3: π_{Mult}

Let $\mathsf{Expand} : S \to \mathbb{F}_q^{2m}, \mathsf{Expand}' : S' \to \mathbb{F}_q^m$ be the deterministic expansion functions with seed space S, S' and output length a, b, respectively. 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 $C = AB$.

- **Multiply:**
  1. Each party P_i samples seeds $\left(s_1^{(i)}, \cdots, s_m^{(i)}\right) \in S^m, \left({s'_1}^{(i)}, \cdots, {s'_m}^{(i)}\right) \in S'^m$ and obtains $\hat{A}^{(i)} = \left(\hat{\boldsymbol{a}}_1^{(i)}, \cdots, \hat{\boldsymbol{a}}_m^{(i)}\right) \in \mathcal{M}_{2m \times m}(\mathbb{F}_q), B^{(i)} = \left(\boldsymbol{b}_1^{(i)}, \cdots, \boldsymbol{b}_m^{(i)}\right)^T$, where $\hat{\boldsymbol{a}}_k^{(i)} = \mathsf{Expand}\left(s_k^{(i)}\right), \boldsymbol{b}_k^{(i)} = \mathsf{Expand}'({s'_k}^{(i)})$ for $k \in [m]$.
  2. For $k \in [m]$ and each ordered pair (P_i, P_j):
     (a) P_i and P_j invoke $\mathcal{F}_{\mathsf{VOPE}}^{2m,m}$, where P_i inputs $s_k^{(i)}$ and P_j inputs ${s'_k}^{(j)}$.
     (b) P_i receives $U_k^{(i,j)}$ and P_j receives $W_k^{(j,i)}$
  3. Each party P_i locally computes
  $$\hat{C}^{(i)} = \hat{A}^{(i)} B^{(i)} + \sum_{j \neq i} \sum_{k \in [m]} \left(U_k^{(i,j)} + W_k^{(i,j)}\right)$$
  4. Each party P_i rewrites: $\hat{A}^{(i)} = \begin{pmatrix} A^{(i)} \\ A'^{(i)} \end{pmatrix}, \hat{C}^{(i)} = \begin{pmatrix} C^{(i)} \\ C'^{(i)} \end{pmatrix}$ and obtain $[A], [A'], [B], [C], [C']$.
- **Authenticate**: All parties invoke $\mathcal{F}_{\mathsf{Auth}}$ to obtain $\langle A \rangle, \langle A' \rangle, \langle B \rangle, \langle C \rangle$ and $\langle C' \rangle$.
- **Sacrifice**:
  1. All parties invoke $\mathcal{F}_{\mathsf{Coin}}$ to obtain a random element χ.
  2. All parties locally compute $\langle D \rangle = \chi \langle A \rangle - \langle A' \rangle$ and partially open $D \leftarrow \pi_{\mathsf{Open}}(\langle D \rangle)$.
  3. All parties locally compute $\langle E \rangle = \chi \langle C \rangle - \langle C' \rangle - D \langle B \rangle$ and partially open $E \leftarrow \pi_{\mathsf{Open}}(\langle E \rangle)$.
  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 sharings $\langle A \rangle$ 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 $\mathcal{F}_{\mathsf{Auth}}$ to obtain the authenticated sharing $\langle A^T \rangle$. Take random linear combinations of $2\ell + 1$ double sharings $(\langle A_i \rangle, \langle A_i^T \rangle)$ 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 $1/q$.

Procedure 4: π_{Double}

Let n_D denote th number of double sharings. The procedure produces n_D pairs of authenticated sharing $\langle A_i \rangle, \langle A_i^T \rangle, i \in [n_D]$.
Double: Upon receiving (Double, $\langle A_1 \rangle, \ldots, \langle A_{n_D} \rangle$) 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 $i \in \{0\} \cup [n_D]$ by taking the transpose of each share.
3. All parties invoke $\mathcal{F}_{\mathsf{Auth}}$ with command (Auth, $[A_0], [A_0^T], [A_1^T], \ldots, [A_{n_D}^T]$) to obtain the authenticated sharings $\langle A_0 \rangle, \langle A_0^T \rangle, \langle A_1^T \rangle, \ldots, \langle A_{n_D}^T \rangle$.
4. All parties call $\mathcal{F}_{\mathsf{Coin}}$ n_D times to obtain $r_1, \cdots, r_{n_D}$.
5. All parties locally compute

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

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 n_D pairs of authenticated sharings $(\langle A_i \rangle, \langle A_i^T \rangle), i \in [n_D]$.

Putting together. Protocol Π_{Sextuple} instantiates the functionality $\mathcal{F}_{\mathsf{Sextuple}}$ 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 5: Π_{Sextuple}

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 invoke $\mathcal{F}_{\mathsf{Auth}}$ with command $(\mathsf{Auth}, [R_1], \ldots, [R_\ell])$ to obtain $\langle R_i \rangle$ for $i \in [\ell]$.
4. All parties set $n_D = 2\ell$ and invoke π_{Double} with command $(\mathsf{Double}, \langle A_1 \rangle, \ldots, \langle A_\ell \rangle, \langle R_1 \rangle, \ldots, \langle R_\ell \rangle)$ 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_{Sextuple} securely implements $\mathcal{F}_{\mathsf{Sextuple}}$ in the ($\mathcal{F}_{\mathsf{Auth}}$, $\mathcal{F}_{\mathsf{VOPE}}^{2m,m}$, $\mathcal{F}_{\mathsf{Coin}}$)-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 in the Supplementary Material for the full proof.

In functionality $\mathcal{F}_{\mathsf{VOPE}}^{2m,m}$ between P_i and P_j, both P_i and P_j only input their 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 [33] and improve the ability of adversary to introduce an arbitrarily chosen additive error.

Simulating the Multiply step. The simulator $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\mathsf{VOPE}}^{2m,m}$. For $j \in \mathcal{C}$ and $i \notin \mathcal{C}$, let $s_k^{(j,i)}$ and ${s'}_k^{(j,i)}$ be the *actual* input in the k-th invocation of $\mathcal{F}_{\mathsf{VOPE}}^{2m,m}$ for $k \in [m]$. Fix an honest party P_{i_0} and define the *correct* input $s_k^{(j)}$ and ${s'}_k^{(j)}$ to be equal to $s_k^{(j,i_0)}$ and ${s'}_k^{(j,i_0)}$, respectively. For $i \notin \mathcal{C}$, $\mathcal{S}$ randomly samples $\hat{A}^{(i)} \xleftarrow{\$} \mathcal{M}_{\tau m \times m}(\mathbb{F}_q), B^{(i)} \xleftarrow{\$} \mathcal{M}_{m \times m}(\mathbb{F}_q)$. For $j \in \mathcal{C}$, $\mathcal{S}$ receives ${s_k}^{(j,i)}$ and ${s'}_k^{(j,i)}$ from the adversary, where $i \notin \mathcal{C}, k \in [m]$. Then $\mathcal{S}$ receives $\left\{U_k^{(j,i)}, W_k^{(j,i)}\right\}_{j \in \mathcal{C}, i \notin \mathcal{C}}$ from the adversary and recomputes $\left\{U_k^{(i,j)}, W_k^{(i,j)}\right\}_{i \notin \mathcal{C}, j \in \mathcal{C}}$ accordingly. Finally, $\mathcal{S}$ honestly computes $\hat{C}^{(i)}$.

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

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 $\hat{A}^{(j,i)}$ and $B^{(j,i)}$ be the matrices generated by seeds $\left(s_1^{(j,i)}, \cdots, s_m^{(j,i)}\right)$ and $\left({s'_1}^{(j,i)}, \cdots, {s'_m}^{(j,i)}\right)$, respectively. In the k-th invocation of $\mathcal{F}_{\mathsf{VOPE}}^{2m,m}$, denote the errors as $\hat{\boldsymbol{\delta}}_k^{(j,i)} = \hat{\boldsymbol{a}}_k^{(j,i)} - \hat{\boldsymbol{a}}_k^{(j)}$ and $\boldsymbol{\gamma}_k^{(j,i)} = \boldsymbol{b}_k^{(j,i)} - \boldsymbol{b}_k^{(j)}$. Then we conclude that for $k \in [m], i \notin \mathcal{C}$ and $j \in \mathcal{C}$:

$$U_k^{(i,j)} + W_k^{(j,i)} = \hat{\boldsymbol{a}}_k^{(i)} \otimes \boldsymbol{b}^{(j)} + \hat{\boldsymbol{a}}_k^{(i)} \otimes \boldsymbol{\gamma}_k^{(j,i)}$$
$$U_k^{(j,i)} + W_k^{(i,j)} = \hat{\boldsymbol{a}}_k^{(j)} \otimes \boldsymbol{b}^{(i)} + \hat{\boldsymbol{\delta}}_k^{(j,i)} \otimes \boldsymbol{b}_k^{(i)}$$

Following similar analysis in the proof of Theorem 2, we define $\hat{\Delta}^{(j,i)} = \left(\hat{\boldsymbol{\delta}}_1^{(j,i)}, \cdots, \hat{\boldsymbol{\delta}}_m^{(j,i)}\right)$, $\Gamma^{(j,i)} = \left(\boldsymbol{\gamma}_1^{(j,i)}, \cdots, \boldsymbol{\gamma}_m^{(j,i)}\right)^T$ and compute $\hat{C}$ as

$$\begin{aligned}\hat{C} = \sum_{i\in[n]} \hat{C}^{(i)} &= \hat{A}B + \sum_{i\notin\mathcal{C}} \sum_{j\in\mathcal{C}} \sum_{k\in[m]} \left(\hat{\boldsymbol{a}}_k^{(i)} \otimes \boldsymbol{\epsilon}_k^{(j,i)} + \hat{\boldsymbol{\delta}}_k^{(j,i)} \otimes \boldsymbol{b}_k^{(i)}\right) \\ &= \hat{A}B + \sum_{i\notin\mathcal{C}} \sum_{j\in\mathcal{C}} \hat{A}^{(i)}\Gamma^{(j,i)} + \hat{\Delta}^{(j,i)}B^{(i)} \\ &= \hat{A}B + \sum_{i\notin\mathcal{C}} \hat{A}^{(i)}\Gamma^{(i)} + \hat{\Delta}^{(i)}B^{(i)}\end{aligned}$$

where $\hat{\Delta}^{(i)} = \sum_{j\in\mathcal{C}} \hat{\Delta}^{(j,i)}$ and $\Gamma^{(i)} = \sum_{j\in\mathcal{C}} \Gamma^{(j,i)}$. Splitting the matrices into 2 blocks, we have that:

$$\begin{pmatrix} C \\ C' \end{pmatrix} = \begin{pmatrix} A \\ A' \end{pmatrix} B + \sum_{i\notin\mathcal{C}} \begin{pmatrix} A^{(i)} \\ {A'}^{(i)} \end{pmatrix} \Gamma^{(i)} + \begin{pmatrix} \Delta^{(i)} \\ {\Delta'}^{(i)} \end{pmatrix} B^{(i)}$$

After the **Authentication** step, all parties obtain $\langle A\rangle, \langle A'\rangle, \langle B\rangle, \langle C\rangle, \langle C'\rangle$. Assume that the adversary introduces additive error E_{Auth}, E'_{Auth} in this step, then 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

$$E_1 = \sum_{i\notin\mathcal{C}} A^{(i)}\Gamma^{(i)} \qquad E_2 = \sum_{i\notin\mathcal{C}} \Delta^{(i)}B^{(i)}$$
$$E'_1 = \sum_{i\notin\mathcal{C}} {A'}^{(i)}\Gamma^{(i)} \qquad E_2 = \sum_{i\notin\mathcal{C}} {\Delta'}^{(i)}B^{(i)}$$

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

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 $\{\Gamma^{(i)}, \Delta^{(i)}, {\Delta'}^{(i)}\}_{i \notin C}$ are zero with overwhelming probability.

Finally, we want to show that the opened value D in the real execution is computationally indistinguishable from the uniform matrix 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. Each column $\boldsymbol{a}'_i$ of A' is part of output of expansion function Expand, therefore we want to show that Expand acts as a PRG. The concrete construction of Expand is given in Section E.1 in the Supplementary Material, and the pseudorandomness of output is guaranteed by dual LPN assumption.

6 Analysis

In this section, we analyze the communication and computation cost of our MPC protocol over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ assuming $q \geq 2^\kappa$. The computation complexity is measured by the number of multiplications.

6.1 Analysis of the online phase

First, we consider communication complexity. At each step of partial opening a matrix, 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 transformation operation, all parties need to partially open one share and thus the communication complexity is $2m^2(n-1)\log q$. 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.

Now we proceed to analyze the computation complexity for each multiplication gate. According to Mult command in Π_{Online}, all parties execute the following computation: $E^T[A^T], E^T[\![A^T\boldsymbol{v}]\!], D[B], D[\![B\boldsymbol{v}]\!], DE, DE[\![\boldsymbol{v}]\!]$. Since left multiplication requires m^3 and m^2 multiplications in scheme $[\cdot]$ and $[\![\cdot]\!]$ respectively, the overall computation complexity is $3m^3 + 3m^2$ multiplications.

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, computation complexity and share size of other MPC protocols and list the results in Table 1. Here FI and FD refer to the online communication with function-independent and function-dependent preprocessing, respectively. Although our protocol needs slightly more communication than [16], our protocol has the smallest share size and computation complexity among these protocols. Moreover, the improvement of our MPC protocol by resorting to function-dependent preprocessing can achieve the same communication complexity as [16].

	communication	share size	# multiplications
SPDZ [20]	$4m^3(n-1)\log q$	$2m^2 \log q$	$6m^3$
matrix triple [16]	$4m^2(n-1)\log q$	$2m^2 \log q$	$5m^3 + m^2$
This work (FI)	$6m^2(n-1)\log q$	$m(m+1)\log q$	$3m^3 + 3m^2$
This work (FD)	$4m^2(n-1)\log q$	$m(m+1)\log q$	$3m^3 + 3m^2$

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

6.2 Analysis of the preprocessing phase

The task of preprocessing is to generate random sharings and multiplication sextuples. The communication cost is mainly caused by Π_{Sextuple} which produces the multiplication sextuples. As our preprocessing phase uses VOLE and VOPE as the building blocks, we calculate the communication cost of preprocessing phase in terms of the calls of the functionality $\mathcal{F}_{\mathsf{VOLE}}$ and $\mathcal{F}_{\mathsf{VOPE}}$.

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 $R^{(j)}$ to P_j and invokes $\mathcal{F}_{\mathsf{VOLE}}$ with P_j. After producing $\ell+1$ such random sharings, all parties invoke π_{Check} to check the consistency of these sharings. If ℓ is large enough, the communication cost of the consistency check can be amortized away. In this case, the preparation for an input gate requires $n-1$ calls of $\mathcal{F}_{\mathsf{VOLE}}$.

Protocol Π_{Sextuple} produces ℓ sextuples by generating ℓ Beaver triples and 2ℓ double sharings. During this process, the communication cost is caused by ℓ calls of Π_{Auth} and the invocation of procedure π_{Mult} and π_{Double}. Procedure π_{Mult} generates a multiplication triple $(\langle A\rangle, \langle B\rangle, \langle C\rangle)$ by making $mn(n-1)$ calls of $\mathcal{F}^{2m,m}_{\mathsf{VOPE}}$, 5 calls of Π_{Auth} and 2 calls of π_{open}. Procedure π_{Double} generates 2ℓ authenticated sharings $\langle A\rangle, \langle A^T\rangle$ by making $2\ell+2$ calls of Π_{Auth} and 2 calls of π_{open}. In summary, generating a sextuple requires $mn(n-1)$ calls of $\mathcal{F}^{2m,m}_{\mathsf{VOPE}}$ and $8mn(n-1)$ calls of $\mathcal{F}_{\mathsf{VOLE}}$ assuming ℓ is large enough.

The communication cost of $\mathcal{F}_{\mathsf{VOLE}}$ scales linearly in the length of the vector, which incurs $O(m\log q)$ bits of communication. The analysis of $\mathcal{F}^{2m,m}_{\mathsf{VOPE}}$ depends on the dual LPN parameters. Given the dual LPN parameter $(2m, 2cm, t)$, $\mathcal{F}^{2m,m}_{\mathsf{VOPE}}$ requires t invocations of $\mathcal{F}^{m}_{\mathsf{rsVOLE}}$ (which is sublinear in m), $t\log\frac{2cm}{t}$ invocations of κ-bit OT and exchange of $t(1+m)$ field elements, which result in

$O(m \log q)$ bits of communication. (Note that $t = O(1)$ which does not grow with m.)

Now we proceed to the analysis of the concrete communication cost. We pick the parameters in [16] for a comparison. For a matrix ring $\mathcal{M}_{128\times128}(\mathbb{F}_q)$ where the prime number q satisfies $\log q \approx 128$, [16] shows that each party communicates 12.46MB to generate a matrix triple for the multiplication gate. Our protocol requires 2^7 invocations of $\mathcal{F}_{\mathsf{VOPE}}^{2^8,2^7}$ and 2^{10} invocations of $\mathcal{F}_{\mathsf{VOLE}}^{2^7}$. We choose the security parameter to be 80 bits and then obtain the corresponding dual LPN parameters in [27]. The detailed calculation of communication cost of $\mathcal{F}_{\mathsf{VOPE}}$ and $\mathcal{F}_{\mathsf{VOLE}}$ is deferred to Section E.2 in the Supplementary Material.

Table 2 demonstrates the communication cost of our protocol, the protocol relying on the random VOLE [10], the protocol relying on subfield VOLE [33] and the protocol relying on the homomorphic encryption [16] to prepare the correlated randomness for computing the multiplication gate. The "random VOLE" protocol computes random matrix multiplication with m^2 random VOLE instances [10], and the "subfield VOLE" protocol invokes m times of subfield VOLE in [33], where the extension field is defined as $\mathbb{F}_{q^m}$. One can see that the communication cost of our protocol grows more slowly than [16]. The reason is that the amortized communication cost of PCG-based VOLE decreases with the size of m.

m	random VOLE	subfield VOLE	This work	HE [16]
128	83.5 MB	34.8 MB	19.0 MB	12.5 MB
256	362 MB	138 MB	60 MB	50 MB
512	1453 MB	518 MB	198 MB	199 MB
1024	6000 MB	2004 MB	739 MB	797 MB

Table 2. The communication cost to prepare correlated randomness for computing a multiplication gate.

6.3 Experimental result

Online phase We implement the online phase of different MPC protocols over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ in C++ with the multiple precision integer arithmetic provided by MPIR library [6]. All experiments were carried out on a server equipped with an Intel Xeon Gold 5220R processor and 128GB RAM. We apply Linux tc command to emulate a real network environment and simulate the LAN network with 1Gbps bandwidth, 1ms latency. Table 3 compares the performances of computing a multiplication gate for each MPC protocol, which shows that our approach is around 1.38x-1.85x faster than [16].

m	SPDZ [20]	matrix triple [16]	This work
128	1.3 sec	96 ms	52 ms
256	9.5 sec	559 ms	329 ms
512	77.9 sec	5.0 sec	3.2 sec
1024	633 sec	42.5 sec	30.9 sec

Table 3. Runtime to compute a multiplication gate in the online phase.

Preprocessing phase We present the benchmarks of VOLE-based preprocessing protocols to generate the correlated randomness for a multiplication gate, in which secure random matrix multiplication is the bottleneck of the computation. All VOLE-based preprocessing protocols rely on PCG techniques, which expand a pair of short seeds to long correlated randomness. We apply the PCG implementation of libOTe [32] to estimate the runtime of the expansion step, which is based on quasi-cyclic codes in [12]. To estimate the efficiency of generating seeds, we calculate the required number of VOLEs and OTs and benchmark the runtime of VOLE and OT with Lattigo [1] and libOTe [32] libraries, respectively. The cost of VOLE is estimated by running the ring-LWE based OLE protocol in [7].

Table 4 provides total estimated runtime on secure random matrix multiplication in the LAN setting. To make a fair comparison with [16], all VOLE-based protocols are tested with 16 threads. As can be seen from the table, our preprocessing phase achieves a 1.44x-24.17x speedup compared to [16] with the same thread number. It is noteworthy that [16] requires a key generation and a one-time setup (when $m = 128$, these operations take around 83 seconds and 14.5 seconds respectively), while our protocol does not rely on a heavy setup. We provide a full-fledged experiment result of VOLE-based preprocessing in Section E.3 in the Supplementary Material.

m	random VOLE	subfield VOLE	This work	HE [16]
128	3.6 sec	2.9 sec	4.1 sec	5.9 sec
256	13.9 sec	10.1 sec	8.2 sec	25.5 sec
512	56.4 sec	38.2 sec	16.8 sec	2.3 min
1024	4.1 min	2.5 min	36.0 sec	14.5 min

Table 4. Benchmark: Runtime to prepare correlated randomness for computing a multiplication gate, measured with 16 threads.

Acknowledgement

The authors would like to thank Jiawei Ni for her assistance with implementation. We are also grateful for valuable suggestions from anonymous reviews in Asiacrypt 2024. The work was supported in part by the National Key Research and Development (R&D) Program of China under Grant 2022YFA1004900 and in part by the National Natural Science Foundation of China under Grants 12031011, 12361141818, and 12101404. This work was also supported in part by Ant Group through CCF-Ant Research Fund CCF-AFSG RF20230306.

References

1. Lattigo v5. Online: `https://github.com/tuneinsight/lattigo` (Nov 2023), ePFL-LDS, Tune Insight SA
2. Abspoel, M., Cramer, R., Damgård, I., Escudero, D., Rambaud, M., Xing, C., Yuan, C.: Asymptotically good multiplicative LSSS over galois rings and applications to MPC over $\mathbb{Z}/p^k\mathbb{Z}$. In: ASIACRYPT 2020. LNCS, vol. 12493, pp. 151–180. Springer (2020)
3. Abspoel, M., Cramer, R., Damgård, I., Escudero, D., Yuan, C.: Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via galois rings. In: TCC 2019. LNCS, vol. 11891, pp. 471–501. Springer (2019)
4. Applebaum, B., Damgård, I., Ishai, Y., Nielsen, M., Zichron, L.: Secure arithmetic computation with constant computational overhead. In: CRYPTO 2017. LNCS, vol. 10401, pp. 223–254. Springer (2017)
5. Applebaum, B., Konstantini, N.: Actively secure arithmetic computation and VOLE with constant computational overhead. In: EUROCRYPT 2023. LNCS, vol. 14005, pp. 190–219. Springer (2023)
6. B. Gladman, W.H., J. Moxham, e.a.: MPIR: Multiple Precision Integers and Rationals (2015), version 2.7.0, `http://mpir.org`
7. Baum, C., Escudero, D., Pedrouzo-Ulloa, A., Scholl, P., Troncoso-Pastoriza, J.R.: Efficient protocols for oblivious linear function evaluation from ring-lwe. J. Comput. Secur. **30**(1), 39–78 (2022)
8. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: CRYPTO '91. LNCS, vol. 576, pp. 420–432. Springer (1991)
9. Ben-Efraim, A., Nielsen, M., Omri, E.: Turbospeedz: Double your online spdz! improving SPDZ using function dependent preprocessing. In: ACNS 2019. LNCS, vol. 11464, pp. 530–549. Springer (2019)
10. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y.: Compressing vector OLE. In: ACM CCS 2018. pp. 896–912. ACM (2018)
11. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Resch, N., Scholl, P.: Correlated pseudorandomness from expand-accumulate codes. In: CRYPTO 2022. LNCS, vol. 13508, pp. 603–633. Springer (2022)
12. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators: Silent OT extension and more. In: CRYPTO 2019. LNCS, vol. 11694, pp. 489–518. Springer (2019)
13. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators from ring-lpn. In: CRYPTO 2020. LNCS, vol. 12171, pp. 387–416. Springer (2020)

14. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical gapsvp. In: CRYPTO 2012. LNCS, vol. 7417, pp. 868–886. Springer (2012)
15. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: FOCS 2001. pp. 136–145. IEEE Computer Society (2001)
16. Chen, H., Kim, M., Razenshteyn, I.P., Rotaru, D., Song, Y., Wagh, S.: Maliciously secure matrix multiplication with applications to private deep learning. In: ASIACRYPT 2020. LNCS, vol. 12493, pp. 31–59. Springer (2020)
17. Couteau, G., Rindal, P., Raghuraman, S.: Silver: Silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes. In: CRYPTO 2021. LNCS, vol. 12827, pp. 502–534. Springer (2021)
18. Cramer, R., Damgård, I., Escudero, D., Scholl, P., Xing, C.: Spd$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In: CRYPTO 2018. LNCS, vol. 10992, pp. 769–798. Springer (2018)
19. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In: ESORICS 2013. LNCS, vol. 8134, pp. 1–18. Springer (2013)
20. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer (2012)
21. Escudero, D., Goyal, V., Polychroniadou, A., Song, Y.: Turbopack: Honest majority MPC with constant online communication. In: ACM CCS 2022. pp. 951–964. ACM (2022)
22. Escudero, D., Goyal, V., Polychroniadou, A., Song, Y., Weng, C.: Superpack: Dishonest majority MPC with constant online communication. In: EUROCRYPT 2023. LNCS, vol. 14005, pp. 220–250. Springer (2023)
23. Escudero, D., Soria-Vazquez, E.: Efficient information-theoretic multi-party computation over non-commutative rings. In: CRYPTO 2021. LNCS, vol. 12826, pp. 335–364. Springer (2021)
24. Escudero, D., Xing, C., Yuan, C.: More efficient dishonest majority secure computation over $\mathbb{Z}_{2^k}$ via galois rings. In: CRYPTO 2022. LNCS, vol. 13507, pp. 383–412. Springer (2022)
25. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. IACR Cryptol. ePrint Arch. p. 144 (2012), `http://eprint.iacr.org/2012/144`
26. Jiang, X., Kim, M., Lauter, K.E., Song, Y.: Secure outsourced matrix computation and application to neural networks. In: CCS 2018. pp. 1209–1222. ACM (2018)
27. Liu, H., Wang, X., Yang, K., Yu, Y.: The hardness of LPN over any integer ring and field for PCG applications. IACR Cryptol. ePrint Arch. p. 712 (2022), `https://eprint.iacr.org/2022/712`
28. Liu, J., Juuti, M., Lu, Y., Asokan, N.: Oblivious neural network predictions via minionn transformations. In: ACM CCS 2017. pp. 619–631. ACM (2017)
29. Mohassel, P., Rindal, P.: Aby^3: A mixed protocol framework for machine learning. In: ACM CCS 2018. pp. 35–52. ACM (2018)
30. Mohassel, P., Zhang, Y.: Secureml: A system for scalable privacy-preserving machine learning. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 19–38. IEEE Computer Society (2017)
31. Orsini, E., Smart, N.P., Vercauteren, F.: Overdrive2k: Efficient secure MPC over $\mathbb{Z}_{2^k}$ from somewhat homomorphic encryption. In: CT-RSA 2020. LNCS, vol. 12006, pp. 254–283. Springer (2020)
32. Peter Rindal, L.R.: libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. `https://github.com/osu-crypto/libOTe`

33. Rachuri, R., Scholl, P.: Le mans: Dynamic and fluid MPC for dishonest majority. In: CRYPTO 2022. LNCS, vol. 13507, pp. 719–749. Springer (2022)
34. Raghuraman, S., Rindal, P., Tanguy, T.: Expand-convolute codes for pseudorandom correlation generators from LPN. In: CRYPTO 2023. LNCS, vol. 14084, pp. 602–632. Springer (2023)
35. Schoppmann, P., Gascón, A., Reichert, L., Raykova, M.: Distributed vector-ole: Improved constructions and implementation. In: ACM CCS 2019. pp. 1055–1072. ACM (2019)
36. Weng, C., Yang, K., Katz, J., Wang, X.: Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1074–1091. IEEE (2021)

Supplementary Material

A Missing Functionalities and Protocols

Channels. This functionality models required communication channels.

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

The functionality proceeds as follows:

- **Pairwise**: On input $(\mathsf{Message}, x, P_i, P_j)$ from P_i, send x to P_j.
- **Broadcast**: On input $(\mathsf{Broadcast}, x, P_i)$ from P_i, send x to all parties.
- **Simultaneous**: On input $(\mathsf{Simultaneous}, x_i, P_i)$ 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 6: $\pi_{\mathsf{Aff}}((\langle X_1\rangle, \cdots, \langle X_\ell\rangle), (a_1 \cdots, a_\ell))$

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 7: $\pi_{\mathsf{Open}}(\langle X\rangle)$

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 8: $\pi_{\mathsf{Check}}(X', \langle X \rangle)$

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 $\boldsymbol{\sigma} = \mathbf{0}$. If the answer is no, abort.

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

Procedure 9: 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 $\mathsf{Initialize}$, $\mathsf{Authenticate}$ and $\mathsf{Sextuple}$ can be done using essentially the same protocols as Π_{Auth} and Π_{Sextuple}. The $\mathsf{TransTuple}$ command can be done with procedure $\Pi_{\mathsf{TransTuple}}$. 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 $\mathcal{F}_{\mathsf{VOLE}}$ 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 10: 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_{Sextuple}.
– **TransTuple**: Same as in $\Pi_{\mathsf{TransTuple}}$.
– **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 $\mathcal{F}_{\mathsf{VOLE}}^{k}$, 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 $\boldsymbol{u}_{h,k}^{(j,i)} = \boldsymbol{w}_{h,k}^{(i,j)} + v_k^{(j)} \boldsymbol{r}_{h,k}^{(i)}$.

(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 $\mathcal{F}_{\mathsf{Coin}}$ 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 $Y' \leftarrow \pi_{\mathsf{Open}}(\langle Y\rangle)$ and $\pi_{\mathsf{check}}(Y', \langle Y\rangle)$ 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 $\mathcal{F}_{\mathsf{MPC}}$ in the ($\mathcal{F}_{\mathsf{Prep}}$, $\mathcal{F}_{\mathsf{Coin}}$)-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 $\mathcal{F}_{\mathsf{Prep}}$ and $\mathcal{F}_{\mathsf{Coin}}$ 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 $\mathcal{F}_{\mathsf{Prep}}$ 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 $\mathcal{F}_{\mathsf{MPC}}$.

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 $\mathcal{F}_{\mathsf{MPC}}$. 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 and Trans command, where the adversary receives honest parties' shares of 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 procedure 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 $\mathcal{F}_{\mathsf{Auth}}$ in the ($\mathcal{F}_{\mathsf{VOLE}}$, $\mathcal{F}_{\mathsf{Coin}}$)-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 $\mathcal{F}_{\mathsf{VOLE}}^k$. $\mathcal{S}$ 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 $\mathcal{F}_{\mathsf{Auth}}$ 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 $\mathcal{F}_{\mathsf{VOLE}}$.
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 $\mathcal{F}_{\mathsf{Auth}}$.
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 $\mathcal{F}_{\mathsf{VOLE}}$ 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 $\{\Gamma^{(i)}, \Delta^{(i)}, {\Delta'}^{(i)}\}_{i \notin \mathcal{C}}$ are zero with overwhelming probability.

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

$$\begin{aligned}E_1 + E_2 + E_{Auth} &= 0\\ \sum_{i \notin \mathcal{C}} \left(A^{(i)}E^{(i)} + \Delta^{(i)}B^{(i)}\right) &= -E_{Auth}\end{aligned}$$

where $\{A^{(i)}, B^{(i)}\}_{i \notin \mathcal{C}}$ are distributed uniformly at random and other items are independent of $\{A^{(i)}, B^{(i)}\}_{i \notin \mathcal{C}}$. Suppose that $\Delta^{(i)}$ is not a zero matrix for some $i \notin \mathcal{C}$, then adversary needs to make $\Delta^{(i)}B^{(i)} = X$ for some matrix X independent of $B^{(i)}$ to pass the sacrifice step, which happens with probability q^{-m}. The same analysis works for ${\Delta'}^{(i)}$ and $\Gamma^{(i)}$.

Theorem 6 (Theorem 3, restated). *Protocol Π_{Sextuple} securely implements $\mathcal{F}_{\mathsf{Sextuple}}$ in the ($\mathcal{F}_{\mathsf{Auth}}$, $\mathcal{F}_{\mathsf{VOPE}}^{2m,m}$, $\mathcal{F}_{\mathsf{Coin}}$)-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 $\mathcal{F}_{\mathsf{Auth}}$ with inputs from the adversary. Similarly, $\mathcal{S}$ just emulates the $\mathcal{F}_{\mathsf{Coin}}$ 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 [23,21] and dishonest majority setting [9,22]. 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 $\mathcal{F}_{\mathsf{FD-Prep}}$ to describe the function-dependent preprocessing as Functionality 8. We can slightly modify Π_{Prep} to instantiate this functionality. Based on $\mathcal{F}_{\mathsf{FD-Prep}}$, we could instantiate $\mathcal{F}_{\mathsf{MPC}}$ as Protocol 12. To avoid confusion with Π_{Online}, we denote this instantiation as $\Pi_{\mathsf{FD-Online}}$.

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

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 $\mathcal{F}_{\mathsf{Auth}}$ 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 $(\langle \Lambda_X \rangle, \langle \Lambda_X^T \rangle, \langle \Lambda_Y \rangle, \langle \Lambda_X \Lambda_Y \rangle, \langle R \rangle, \langle R^T \rangle, \langle \Lambda_Z \rangle)$:
  - 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 11: $\Pi_{\mathsf{FD-Online}}$

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 $\mathcal{F}_{\mathsf{FD-Prep}}$ 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 tuple $(\langle \Lambda_X \rangle, \langle \Lambda_X^T \rangle, \langle \Lambda_Y \rangle, \langle \Lambda_X \Lambda_Y \rangle, \langle R \rangle, \langle R^T \rangle, \langle \Lambda_Z \rangle)$.
  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 $D \leftarrow \pi_{\mathsf{Open}}(\langle D \rangle)$ and $E \leftarrow \pi_{\mathsf{Open}}(\langle E \rangle)$.
  4. Set $\Phi_Z = D + E^T$.
– **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 $\Pi_{\mathsf{FD-Online}}$ securely implements $\mathcal{F}_{\mathsf{MPC}}$ in the ($\mathcal{F}_{\mathsf{FD-Prep}}$, $\mathcal{F}_{\mathsf{Coin}}$)-hybrid model.*

Proof. The security proof is similar to the Π_{Online}.

D Extension to Small Fields

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 $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ with small q. We only present the modification.

Authenticated Sharing. Instead of letting the global key in $\mathcal{M}_{m\times 1}(\mathbb{F}_q)$, we require that the global key is a matrix in $\mathcal{M}_{m\times \ell}(\mathbb{F}_q)$. This also implies that the MAC for each matrix is a matrix in $\mathcal{M}_{m\times \ell}(\mathbb{F}_q)$. One can treat the global key in $\mathcal{M}_{m\times \ell}(\mathbb{F}_q)$ as ℓ independent global keys in $\mathcal{M}_{m\times 1}(\mathbb{F}_q)$. Let $V \in \mathcal{M}_{m\times \ell}(\mathbb{F}_q)$ be the global key and assume that $\boldsymbol{v}_1, \dots, \boldsymbol{v}_\ell$ 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 $X \in \mathcal{M}_{m\times \ell}(\mathbb{F}_q)$ is the matrix known to the adversary. Let $\boldsymbol{x}_1, \dots, \boldsymbol{x}_\ell$ 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 $q^{-\ell}$. Therefore the soundness error is reduced to $q^{-\ell}$. If we set $\ell = \frac{\kappa}{\log_2 q}$, 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 error of this check becomes $1/q$ if our matrix is defined over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$. We can repeat the linear combinations ℓ times to reduce the error probability to $q^{-\ell}$. 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.

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 $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ can detect the corruptions with probability $1 - \frac{1}{q}$. To make this probability overwhelming, we have to sacrifice ℓ triples to verify the relation $C = AB$.

VOLE and subfield VOLE. Since the implementation of our VOLE and subfield VOLE are based on the dual LPN assumption, both of them can be defined over small field as well.

E Deteails for Preprocessing

E.1 Instantiation of VOPE

Following the line of [12,33], we first propose the protocol Π_{spVOPE} that securely implements the single point VOPE (spVOPE), and then obtain Π_{VOPE} by invoking $\mathcal{F}_{\mathsf{spVOPE}}$ multiple times. The major difference between random subfield

VOLE in [33] and our VOPE is that the subfield VOLE requires that one-side input is generated by a seed and the input of another side is given deterministically; while our VOPE requires that the inputs of both sides are expanded from seeds. As a building block, we choose a new functionality $\mathcal{F}^b_{\mathsf{rsVOLE}}$ instead of functionality $\mathcal{F}^b_{\mathsf{sVOLE}}$ in [33], i.e., we allow P_A to input a random seed to generate a pseudorandom vector $\boldsymbol{x} \in \mathbb{F}_q^m$ and the input $v \in \mathbb{F}_q$ from P_B can be chosen arbitrarily. This is exactly described by the functionality $\mathcal{F}^{\mathsf{prog}}_{\mathsf{VOLE}}$ in [33] by setting $r = 1$. Thus, the protocol $\Pi^{\mathsf{prog}}_{\mathsf{VOLE}}$ in [33] can securely implement our functionality $\mathcal{F}^b_{\mathsf{rsVOLE}}$.

Functionality 9: $\mathcal{F}^b_{\mathsf{rsVOLE}}$

The functionality runs between sender P_A and receiver P_B.
Let $\mathsf{Expand}' : S' \to \mathbb{F}_q^b$ be the expansion function with seed space S' and output length b.
Initialize: On receiving (Init) from P_A and (Init, s') from P_B. Note that Init command is only invoked once.
Extend: On receiving (Extend) from P_A, P_B.

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

Next, we briefly review single point subfield VOLE (spsVOLE) protocol $\Pi^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ in [33], which is very similar to our protocol $\Pi^{a,b}_{\mathsf{spVOPE}}$. The goal of spsVOLE is to compute the additive sharing of $v\boldsymbol{e}$, where weight-1 vector $\boldsymbol{e} \in \mathbb{F}_q^a$ and $v \in \mathbb{F}_{q^b}$ are provided by P_A and P_B respectively. During this protocol, P_A and P_B invoke $1 + b$ times of $\mathcal{F}_{\mathsf{sVOLE}}$ and $\lceil \log a \rceil$ times of κ-bit OTs. Note that when b is large, $\mathcal{F}_{\mathsf{sVOLE}}$ is the dominant part of communication, which takes $O(b^2 \log q)$ communication in total.

$\Pi^{a,b}_{\mathsf{spVOPE}}$ is adapted from protocol $\Pi^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ in [33] with slight modification. The major difference is that we use functionality $\mathcal{F}^b_{\mathsf{rsVOLE}}$ instead of $\mathcal{F}_{\mathsf{sVOLE}}$ in [33]. This difference is due to the fact that the input of P_B in our protocol is expanded from a seed instead of a truly random element sampled by P_B. Since b is a large number in our protocol, this adaption can greatly save the communication cost.

Functionality 10: $\mathcal{F}^{a,b}_{\mathsf{spVOPE}}$

The functionality runs between sender P_A and receiver P_B.
Let $\mathsf{Expand}' : S' \to \mathbb{F}_q^b$ be the deterministic expansion functions with seed space S' and output length b.
Initialize: Upon receiving (Init) from P_A and (Init, s') from P_B.
Extend: Upon receiving $(\mathsf{Extend}, \alpha, \beta)$ from P_A and (Extend) from P_B, where $\alpha \in [a], \beta \in \mathbb{F}_q$:

1. Compute $\boldsymbol{v} = \mathsf{Expand}'(s')$ and sample $W \xleftarrow{\$} \mathcal{M}_{a\times b}(\mathbb{F}_q)$. If P_B is corrupted, receive W from the adversary.
2. Set $\boldsymbol{e} \in \mathbb{F}_q^a$ such that $e_\alpha = \beta$ and $e_i = 0$ for $i \neq \alpha$. Compute $U = \boldsymbol{e} \otimes \boldsymbol{v} + W$. If P_A is corrupted, receive U from the adversary and recompute $W = U - \boldsymbol{e} \otimes \boldsymbol{v}$.
3. If P_B is corrupted, receive a set $I \subset [a]$ from adversary. If $\alpha \in I$, send $\mathsf{Success}$ to P_B and continue. Otherwise, send Abort to both parties and abort.
4. Output $(\boldsymbol{e}, U)$ to P_A and W to P_B.

Protocol 12: $\Pi^{a,b}_{\mathsf{spVOPE}}$

This protocol runs between sender P_A and receiver P_B.
Let $\mathsf{Expand}' : S' \to \mathbb{F}_q^b$ be the deterministic expansion function with seed space S' and output length b.
Initialize: This step is only executed once. P_A sends (Init) and P_B sends (Init, s') to $\mathcal{F}^b_{\mathsf{sVOLE}}$.
Extend: This step can be executed several times.

1. P_A and P_B send (Extend) to $\mathcal{F}^b_{\mathsf{rsVOLE}}$, which returns $(x, \boldsymbol{z}) \in \mathbb{F}_q \times \mathbb{F}_q^b$ to P_A and $\boldsymbol{y} \in \mathbb{F}_q^b$ to P_B.
2. P_A sends $x' = \beta - x \in \mathbb{F}_q$ to P_B, then P_B computes $\boldsymbol{v} = \mathsf{Expand}'(s') \in \mathbb{F}_q^b$ and $\boldsymbol{\gamma} = \boldsymbol{z} - x'\boldsymbol{v}$. P_A defines $\boldsymbol{e} \in \mathbb{F}_q^a$ as the single point vector such that $e_\alpha = \beta$.
3. P_B samples the seed of GGM tree $s \xleftarrow{\$} \{0,1\}^\kappa$ and runs $\mathsf{GGM}(1^a, s)$ to obtain $\big(\{\boldsymbol{w}_j\}_{j\in[a]}, \{(K_0^i, K_1^i)\}_{i\in[h]}\big)$, where $\boldsymbol{w}_j \in \mathbb{F}_q^b$ and $h = \lceil \log a \rceil$. P_B sets $W = (\boldsymbol{w}_1, \cdots, \boldsymbol{w}_a)^T \in \mathcal{M}_{a\times b}(\mathbb{F}_q)$. P_A lets $\overline{\alpha_i}$ be the compliment of the i-th bit of bit representation of α. For $i \in [h]$, P_A sends $\overline{\alpha_i} \in \{0,1\}$ to $\mathcal{F}_{\mathsf{OT}}$ and P_B sends (K_0^i, K_1^i) to $\mathcal{F}_{\mathsf{OT}}$. P_A receives $K_{\overline{\alpha_i}}{}^i$, which then runs $\{\boldsymbol{w}_j\}_{j\neq\alpha} = \mathsf{GGM}'(\alpha, \{K^i_{\overline{\alpha_i}}\}_{i\in[h]})$.
4. P_B sends $\boldsymbol{d} = \boldsymbol{\gamma} - \sum_{i\in[a]} \boldsymbol{w}_i$ to P_A. Then, P_A defines $U = (\boldsymbol{u}_1, \cdots, \boldsymbol{u}_a)^T \in \mathcal{M}_{a\times b}(\mathbb{F}_q)$ such that for $i \in [a]$,

$$\boldsymbol{u}_i = \begin{cases} \boldsymbol{w}_i & i \neq \alpha \\ \boldsymbol{z} - (\boldsymbol{d} + \sum_{i\neq\alpha} \boldsymbol{w}_i) & i = \alpha \end{cases}$$

Consistency check:

1. P_A and P_B send (Extend) to $\mathcal{F}^b_{\mathsf{rsVOLE}}$, which returns $(x^*, \boldsymbol{z}^*) \in \mathbb{F}_q \times \mathbb{F}_q^b$ to P_A and $\boldsymbol{y}^* \in \mathbb{F}_q^b$ to P_B.
2. P_A and P_B invokes $\mathcal{F}_{\mathsf{Coin}}$ to sample $r_i \in \mathbb{F}_q$ for $i \in [a]$. P_A sends $x'' = r_\alpha \cdot \beta - x^* \in \mathbb{F}_q^b$ to P_B.
3. Let $\boldsymbol{r} = (r_1, \cdots, r_a)^T \in \mathbb{F}_q^a$. P_A computes $V_A = U^T\boldsymbol{r} - \boldsymbol{z}^* \in \mathbb{F}_q^b$ and P_B computes $V_B = W^T\boldsymbol{r} + x''\boldsymbol{v} - \boldsymbol{y}^* \in \mathbb{F}_q^b$. Then P_A sends V_A to $\mathcal{F}_{\mathsf{EQ}}$ and P_B sends V_B to $\mathcal{F}_{\mathsf{EQ}}$. If either party receives False or Abort from $\mathcal{F}_{\mathsf{EQ}}$, it aborts.
4. P_A outputs $(\boldsymbol{e}, U) \in \mathbb{F}_q^a \times \mathcal{M}_{a\times b}(\mathbb{F}_q)$ and P_B outputs $W \in \mathcal{M}_{a\times b}(\mathbb{F}_q)$.

Theorem 8. *Assume that* Expand' *is a pseudorandom generator, protocol* $\mathit{\Pi}^{a,b}_{\mathsf{spVOPE}}$ *securely implements* $\mathcal{F}^{a,b}_{\mathsf{spVOPE}}$ *in the* $(\mathcal{F}^{b}_{\mathsf{rsVOLE}}, \mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{EQ}}, \mathcal{F}_{\mathsf{Coin}})$*-hybrid model.*

Proof. The security proof is almost the same to $\mathit{\Pi}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ in [33]. We only list the difference from $\mathit{\Pi}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$:

1. We represent the element in $\mathbb{F}_{q^b}$ as a vector in $\mathbb{F}_q^b$.
2. In $\mathit{\Pi}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$, P_B inputs a vector $\boldsymbol{v} \in \mathbb{F}_q^b$ to the functionality $\mathcal{F}_{\mathsf{sVOLE}}$ which returns the additive sharings of $x\boldsymbol{v}$. In our $\mathit{\Pi}^{a,b}_{\mathsf{spVOPE}}$ protocol, P_B inputs a seed s' to $\mathcal{F}^{b}_{\mathsf{rsVOLE}}$ which also returns the additive sharings of $x\boldsymbol{v}$ where $\boldsymbol{v}$ is generated by the seed s'.
3. In $\mathit{\Pi}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$, the consistency check is carried out over extension field $\mathbb{F}_{q^b}$ which yields the error probability q^{-b}. In our protocol, since $q \geq 2^{\kappa}$, our consistency check is carried out $\mathbb{F}_q$ which yields the error probability $1/q$. This modification reduces the number of calls $\mathcal{F}^{a,b}_{\mathsf{rsVOLE}}$ from b to 1 in the consistency check step.
4. In $\mathit{\Pi}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$, the challenges $\{r_i\}_{i\in[a]}$ are sampled by P_A and then sent to P_B. In our protocol, P_A and P_B call $\mathcal{F}_{\mathsf{Coin}}$ to produce challenges. Given a pair of shared seeds, P_A and P_B could invoke $\mathcal{F}_{\mathsf{Coin}}$ without interaction.

Now we discuss two expansion functions in $\mathit{\Pi}_{\mathsf{VOPE}}$, which are based on the following dual LPN assumption.

Definition 1 (Dual LPN assumption). *Let* $H \in \mathcal{M}_{m\times n}(\mathbb{F}_q)$ *and consider following game* $G_b(\kappa)$ *with a PPT adversary* $\mathcal{A}$*, parameterized by a bit* b *and the security parameter* κ*:*

1. *Sample a random,* t*-regular vector* $\boldsymbol{e} \in \mathbb{F}_q^n$*, i.e.,* $\boldsymbol{e}$ *is the concatenation of* t *vectors* $\boldsymbol{e}_1, \cdots, \boldsymbol{e}_t$*, wherein each* $\boldsymbol{e}_i$ *is a sparse vector of Hamming weight* 1.
2. *If* $b = 1$*, let* $\boldsymbol{y} = H\boldsymbol{e}$*, otherwise sample* $\boldsymbol{y} \xleftarrow{\$} \mathbb{F}_q^m$*.*
3. *Send* $\boldsymbol{y}$ *to* $\mathcal{A}$ *and receive a bit* b'*.*

If $\mathcal{A}$ *has a negligible advantage to distinguish* $G_0(\kappa)$ *and* $G_1(\kappa)$*, then dual LPN assumption holds. A tuple* (m, n, t) *is called a dual-LPN parameter. The parameter* $c = n/m$ *is called a compression parameter.*

Let c be the compression parameter and choose two dual-LPN parameters $(a, ca, t), (b, cb, t')$. Given two fixed matrices $H \in \mathcal{M}_{a\times ca}(\mathbb{F}_q), H' \in \mathcal{M}_{b\times cb}(\mathbb{F}_q)$, then we could define two expansion functions as follows:

$$\mathsf{Expand}' : S \to \mathbb{F}_q^a, \quad \mathsf{Expand}(\boldsymbol{e}) = H\boldsymbol{e}$$
$$\mathsf{Expand}' : S' \to \mathbb{F}_q^b, \quad \mathsf{Expand}'(\boldsymbol{e}') = H'\boldsymbol{e}'$$

where $S(S')$ is the collections of t-regular(t'-regular) vector of length $ca(cb)$, respectively.

Given dual LPN assumption in Definition 1 and functionality $\mathcal{F}^{ca/t,b}_{\mathsf{spVOPE}}$, we are able to present the protocol $\mathit{\Pi}^{a,b}_{\mathsf{VOPE}}$.

Protocol 13: $\Pi^{a,b}_{\mathsf{VOPE}}$

Given two dual LPN parameters (c, ca, t) and (b, cb, t'), we could instantiate two expansion functions Expand and Expand' as above. Let $H \in \mathcal{M}_{a\times ca}(\mathbb{F}_q)$ be the matrix in the (a, ca, t) dual LPN assumption.

1. P_A sends (Init) and P_B sends (Init, s') to $\mathcal{F}^{ca/t,b}_{\mathsf{spVOPE}}$, where s' describes a t'-regular vector of length cb.
2. P_A inputs $s \in S$, which describes a t-regular vector $\boldsymbol{e}$ of length ca. $\boldsymbol{e}$ is the concatination of t vectors $\{\boldsymbol{e}_i\}_{i\in[t]}$, where the α_i-th component of each $\boldsymbol{e}_i$ is β_i and others are 0.
3. For $i \in [t]$, P_A and P_B send $(\mathsf{Extend}, \alpha_i, \beta_i), (\mathsf{Extend})$ to $\mathcal{F}^{ca/t,b}_{\mathsf{spVOPE}}$ and receive $Y_i, Z_i \in \mathcal{M}_{ca/t\times b}(\mathbb{F}_q)$, respectively.
4. P_A sets $Y \in \mathcal{M}_{ca\times b}(\mathbb{F}_q)$ and P_B sets $Z \in \mathcal{M}_{ca\times b}(\mathbb{F}_q)$ such that

$$Y = \begin{pmatrix} Y_1 \\ \vdots \\ Y_t \end{pmatrix}, Z = \begin{pmatrix} Z_1 \\ \vdots \\ Z_t \end{pmatrix}$$

5. P_A outputs $U = HY \in \mathcal{M}_{a\times b}(\mathbb{F}_q)$ and P_B outputs $W = -HZ \in \mathcal{M}_{a\times b}(\mathbb{F}_q)$.

Theorem 9. *Protocol $\Pi^{a,b}_{\mathsf{VOPE}}$ securely implements functionality $\mathcal{F}^{a,b}_{\mathsf{VOPE}}$ in the $\mathcal{F}^{ca/t,b}_{\mathsf{spVOPE}}$-hybrid model under (a, ca, t) and (b, cb, t') dual LPN assumption.*

Proof. Observe that P_A and P_B do not interact in Π_{VOPE} except that they jointly invoke $\mathcal{F}_{\mathsf{spVOPE}}$. We construct a simulator $\mathcal{S}$ that emulates functionality $\mathcal{F}^{ca/t,b}_{\mathsf{spVOPE}}$. The security proof is adapted from [36].

Corrupted P_A : During the initialization, $\mathcal{S}$ randomly samples $\boldsymbol{v} \in \mathbb{F}_q^b$. For $i \in [t]$, S emulates $\mathcal{F}^{ca/t,b}_{\mathsf{spVOPE}}$ and receives the inputs $\boldsymbol{e}_i \in \mathbb{F}_q^{ca/t}$ (with Hamming weight at most 1) and $Y_i \in \mathcal{M}_{ca/t\times b}(\mathbb{F}_q)$ from the adversary $\mathcal{A}$. Then $\mathcal{S}$ sets $\boldsymbol{e}, Y$ and computes $\boldsymbol{x} = H\boldsymbol{e}$ and $U = HY$.

Corrupted P_B : During the initialization, $\mathcal{S}$ records the vector $\boldsymbol{v} \in \mathbb{F}_q^b$ that $\mathcal{A}$ sends to $\mathcal{F}^{ca/t,b}_{\mathsf{spVOPE}}$. For $i \in [t]$, $\mathcal{S}$ receives the inputs $Z_i \in \mathcal{M}_{ca/t\times b}(\mathbb{F}_q)$and $I_i \subset [a]$ that $\mathcal{A}$ sends to $\mathcal{F}^{ca/t,b}_{\mathsf{spVOPE}}$. Then $\mathcal{S}$ randomly samples t weight-1 vectors $\{\boldsymbol{e}_i\}_{i\in[t]}$ and records the non-zero entries $\{\alpha_i\}_{i\in[t]}$. If $\alpha_i \in I_i$ for all i, then $\mathcal{S}$ continues; otherwise, it aborts. Finally, $\mathcal{S}$ sets $\mathcal{Z}$ and computes $\boldsymbol{x} = H\boldsymbol{e}, W = -HZ$.

The view of $\mathcal{A}$ is simulated perfectly, and in both real world and ideal world, the outputs of P_A and P_B satisfy that $U + W = \boldsymbol{x} \otimes \boldsymbol{v}$. The only difference is that in the ideal world $\boldsymbol{x}, \boldsymbol{v}$ are uniform, while in the real world they are computed from uniform seeds s and s', respectively. If (a, ca, t) and (b, cb, t') dual LPN assumptions hold, this difference is indistinguishable between ideal and real worlds.

E.2 Concrete Communication Cost of Preprocessing

In this subsection, we focus on concrete communication cost of Π_{Sextuple}, which depends on the instantiation of two functionalities: $\mathcal{F}^{m}_{\mathsf{VOLE}}$ and $\mathcal{F}^{2m,m}_{\mathsf{VOPE}}$. Here we set $m = 128, \kappa = 80$.

Analysis of $\mathcal{F}^{m}_{\mathsf{VOLE}}$ We follow the line of [10] to convert RVOLE to VOLE. Since the scalar of each VOLE is fixed as an element of share of global key, we could set the length of RVOLE as a large number such that the amortized communication is negligible. Then we only consider the communication complexity of the conversion: the sender and receiver sends a vector of length m to each other. In our setting, the size of the field element is 128 bit and length m is 2^7, therefore each invocation requires 4KB.

Analysis of $\mathcal{F}^{2m,m}_{\mathsf{VOPE}}$ Given the dual LPN parameters $(2m, 2cm, t)$, recall that communication cost of $\mathcal{F}^{2m,m}_{\mathsf{VOPE}}$ consists of t length-m random subfield VOLE, $t \log \frac{2cm}{t}$ κ-bit OT and $(1+m)t$ field elements. As we mentioned in Section E.1, we instantiate $\mathcal{F}^{m}_{\mathsf{rsVOLE}}$ with protocol $\Pi^{\mathsf{prog}}_{\mathsf{VOLE}}$ in [33]. With this instantiation and another dual LPN parameter (m, cm, t'), communication compleixty of t $\mathcal{F}^{m}_{\mathsf{rsVOLE}}$ instances could be computed as t invocations of length-t' VOLE, $t' \log \frac{cm}{t'}$ invocations of κ-bit OT and $t(1+t')$ field elements.

We follow the suggestion of [27] to calculate the dual LPN parameter, i.e., $(2m, 2cm, t) = (2^8, 2^{10}, 28)$ and $(m, cm, t') = (2^7, 2^9, 29)$. Then, each $\mathcal{F}^{2m,m}_{\mathsf{VOPE}}$ needs around 115.2 KB of communication.

E.3 Concrete Runtime of Preprocessing

In this subsection, we elaborate on the concrete runtime of preprocessing. We first choose parameters and estimate the required number of fundamental cryptographic primitives (OT and OLE), and benchmark the corresponding runtime.

In libOTe [32], the weight t of regular error vector $\boldsymbol{e}$ in dual LPN assumption has to be divisible by 8, thus we set $t = 32$ for all $m = 128, 256, 512, 1024$. Fixing weight t, we display the number of OLE and OT instances to prepare for a multiplication gate in Table 5. From Table 5, it is obvious that subfield VOLE reduces the number of OT instances, and VOPE transforms some OLE instances into OT instances.

m	random VOLE		subfield VOLE		VOPE	
	OLE	OT	OLE	OT	OLE	OT
128	524288	2621440	524288	20480	131072	36864
256	2097152	12582912	2097152	49152	262144	90112
512	8388608	58720256	8388608	114688	524288	216736
1024	33554432	268435456	33554432	262144	1048576	491520

Table 5. The number of OLE and OT instances to prepare for a multiplication gate for all VOLE-based preprocessing.

Table 6 provides the details about the microbenchmark of runtime to prepare for a multiplication gate. In the third and fourth columns, the runtime of OLE and OT instances in $\Pi^{2m,m}_{\mathsf{spVOPE}}$ is estimated with Lattigo [1] and LibOTe [32], respectively. The runtime of expansion in step 5, $\mathcal{F}^{2m,m}_{\mathsf{VOPE}}$ is displayed in the fifth column "Extend", which is instantiated with quasi-cyclic code in [12]. The columns labeled "Com" and "AddMacs" refer to the runtime of communication and authentication, respectively. Expansion is the most computationally intense operation in VOLE-based preprocessing, which is optimized by using 16 threads.

m	Protocol	BaseVOLE	OT	Extend	Com	AddMacs	All
128	random VOLE	2.115	0.404	0.163	0.622	0.269	3.573
	subfield VOLE		0.029		0.278		2.854
	VOPE	0.51	0.034	3.15	0.151		4.114
256	random VOLE	8.143	2.232	0.269	2.714	0.529	13.887
	subfield VOLE		0.034		1.109		10.084
	VOPE	1.017	0.04	6.105	0.484		8.175
512	random VOLE	32.57	11.441	0.489	11.043	0.918	56.463
	subfield VOLE		0.043		4.151		38.173
	VOPE	2.034	0.058	12.12	1.629		16.759
1024	random VOLE	130.288	65.637	0.96	45.943	1.814	244.642
	subfield VOLE		0.064		16.033		149.159
	VOPE	4.068	0.102	24.121	5.908		36.013

Table 6. Microbenmarks: Runtime to prepare correlated randomness for computing a multiplication gate, all timing measured in seconds.