CompactTag: Minimizing Computation Overheads in Actively-Secure MPC for Deep Neural Networks

Yongqin Wang
University of Southern California
yongqin@usc.edu

Pratik Sarkar
Supra Research
pratik93@bu.edu

Nishat Koti
Indian Institute of Science Bangalore
kotis@iisc.ac.in

Arpita Patra
Indian Institute of Science Bangalore
arpita@iisc.ac.in

Murali Annavaram
University of Southern California
annavara@usc.edu

Abstract

Secure Multiparty Computation (MPC) protocols enable secure evaluation of a circuit by several parties, even in the presence of an adversary who maliciously corrupts all but one of the parties. These MPC protocols are constructed using the well-known secret-sharing-based paradigm (SPDZ and SPD$\mathbb{Z}_{2^k}$), where the protocols ensure security against a malicious adversary by computing Message Authentication Code (MAC) tags on the input shares and then evaluating the circuit with these input shares and tags. However, this tag computation adds a significant runtime overhead, particularly for machine learning (ML) applications with computationally intensive linear layers, such as convolutions and fully connected layers.

To alleviate the tag computation overhead, we introduce CompactTag, a lightweight algorithm for generating MAC tags specifically tailored for linear layers in ML. Linear layer operations in ML, including convolutions, can be transformed into Toeplitz matrix multiplications. For the multiplication of two matrices with dimensions $\mathsf{T}_1 \times \mathsf{T}_2$ and $\mathsf{T}_2 \times \mathsf{T}_3$ respectively, SPD$\mathbb{Z}_{2^k}$ required $O(\mathsf{T}_1 \cdot \mathsf{T}_2 \cdot \mathsf{T}_3)$ local multiplications for the tag computation. In contrast, CompactTag only requires $O(\mathsf{T}_1 \cdot \mathsf{T}_2 + \mathsf{T}_1 \cdot \mathsf{T}_3 + \mathsf{T}_2 \cdot \mathsf{T}_3)$ local multiplications, resulting in a substantial performance boost for various ML models.

We empirically compared our protocol to the SPD$\mathbb{Z}_{2^k}$ protocol for various ML circuits, including ResNet Training-Inference, Transformer Training-Inference, and VGG16 Training-Inference. SPD$\mathbb{Z}_{2^k}$ dedicated around 30% of its online runtime for tag computation. CompactTag speeds up this tag computation bottleneck by up to 23×, resulting in up to 1.47× total online phase runtime speedups for various ML workloads.

1 Introduction

Machine learning (ML) has gained significant importance, particularly in fields like finance, healthcare, and retail advertising, thanks to its ability to handle large amounts of data. Increasingly, ML providers rely on cloud-based servers to provide model training and serving results. However, cloud-based systems are vulnerable to data and model leaks. To protect this data, privacy-preserving technologies like secure multiparty computation (MPC) have been developed. MPC-based secure training allows clients to share their data with multiple mutually distrustful servers, enabling them to collaboratively train a model without revealing specific data details. The trained model can then be stored in a secret-shared format to enable secure inference. The client shares its data with servers using secret sharing, and the servers execute the model on the confidential input to obtain the final output.

When choosing an MPC protocol for Secure ML, two key factors are the threat model setting and scalability. The most robust threat model setting is the dishonest majority, where an adversary can maliciously corrupt a majority of participating MPC parties and manipulate their behavior. It is assumed that in this setting, the adversary can't access honest parties' inputs, where such honest parties are only a minority. This setting is highly robust and doesn't rely on any other trust assumptions regarding non-collusion between the parties or usage of secure hardware. Scalability is another essential factor, as protocols should easily accommodate a growing number of participants without hampering the protocol's efficiency or necessitating major protocol modifications.

Our setting: In this work, we study the *n-party actively secure dishonest majority* setting, which ensures security against an adversary that can maliciously corrupt a subset $t < n$ of parties. This setting has been examined using authenticated garbling-based approaches [47, 50] which demands constant rounds of interaction. This setting has also been studied in the literature through the SPDZ line of works [14, 12, 28], which are faster in practice but require linear (in the multiplicative depth of the circuit) rounds of interaction. In this work, we introduce CompactTag, an optimization that stays within the SPDZ framework and is compatible with all the existing works that use SPDZ.

SPDZ framework: Within the SPDZ framework, parties parse the desired function as an arithmetic circuit comprising of addition and multiplication gates. The protocol follows the offline-online paradigm, where the online phase aims for rapid execution, benefiting from the bulk of computation taking place in the offline phase. More precisely, in the offline phase, parties generate *correlated data* for each multiplication gate and input gate. During the online phase, each party secret shares its inputs among all parties and uses the offline *correlated data* to create Message Authentication Code (MAC) tags on their input shares. These secret shares and tags possess additive homomorphic properties. Consequently, computing an addition gate is straightforward since parties can locally add their shares and tags. For multiplication gates, parties employ the *correlated data* to multiply the shares(of the secret) and MAC tags. In the final output phase, parties verify the output against the output tag to detect any malicious behavior.

Motivation: The computation of MAC tags for multiplication gates introduces significant overhead during the online phase. Deep neural networks (DNNs) rely extensively on convolutions and fully connected layers, which are expensive to perform. Empirical observations in the state-of-the-art SPD$\mathbb{Z}_{2^k}$ reveal that tag computation can account for 30% of the online runtime. To illustrate, multiplying two matrices with dimensions $\mathsf{T}_1 \times \mathsf{T}_2$ and $\mathsf{T}_2 \times \mathsf{T}_3$ in SPD$\mathbb{Z}_{2^k}$ requires $O(\mathsf{T}_1 \cdot \mathsf{T}_2 \cdot \mathsf{T}_3)$ local multiplications for the tag computation. This study aims to investigate the feasibility of reducing this cubic computation overhead to quadratic. Further, computation complexity directly impacts the power usage of a system. Thus, reducing computation would directly help reduce the power consumption of secure ML algorithms, where the latter is a growing concern [2]. Optimizing the power usage allows attaining substantial savings in monetary cost, promotes scalability, and plays an important role in reducing our carbon footprint.

1.1 Our contributions

This paper introduces an improved method for tag computation when performing matrix multiplications for actively secure dishonest majority MPC.

Optimized tag generation: We introduce CompactTag, an optimization within the SPD$\mathbb{Z}_{2^k}$ framework that efficiently generates tags when performing matrix multiplications. In contrast to the existing protocol [12] for matrix multiplication, which involves replicating the computationally expensive local multiplications of secret shared values on the tags, the CompactTag algorithm utilizes alternative lightweight techniques for tag generation. Specifically, when multiplying two matrices with dimensions $\mathsf{T}_1 \times \mathsf{T}_2$ and $\mathsf{T}_2 \times \mathsf{T}_3$, the tag computation in CompactTag requires only quadratic computation, i.e. $O(\mathsf{T}_1 \cdot \mathsf{T}_2 + \mathsf{T}_1 \cdot \mathsf{T}_3 + \mathsf{T}_2 \cdot \mathsf{T}_3)$ local multiplications, whereas previous protocols required cubic computation, leading to concrete asymptotic improvements in the online phase. The offline phase of CompactTag remains the same as SPD$\mathbb{Z}_{2^k}$ without any modifications. Thus, CompactTag allows to improve the computational efficiency of matrix multiplication without inflating the communication of the online phase. Moreover, CompactTag is also compatible with prime order fields and can be effortlessly applied in the original SPDZ framework [14].

Experimental results: Our improvements in asymptotic performance directly manifest as demonstrated in our extensive experiments. For this we have implemented CompactTag using the CrypTen [29] library. Our implementations of CompactTag exploit GPUs to accelerate the online phase by leveraging Crypten's GPU support. To analyze the performance of CompactTag, we evaluate CompactTag on different ML models and compare it against SPD$\mathbb{Z}_{2^k}$. We elaborate on this below:

- *ML models considered:* We consider three popular DNN models: 1) VGG16, 2) ResNet, and 3) Transformer, and benchmark the training as well as inference phases.
- *Tag computation speedup:* As opposed to SPD$\mathbb{Z}_{2^k}$, where the tag computation takes up 30% of the online runtime, it is only 2% when using CompactTag. This reduction in tag computation time results from CompactTag's lightweight tag generation, which reduces tag computation by up to 23×.
- *Online computation speedup:* The reduction in tag computation time brings in savings of up to 1.47× in the overall online runtime in comparison to SPD$\mathbb{Z}_{2^k}$.
- *Power usage:* CompactTag reduces the power consumption of the SPD$\mathbb{Z}_{2^k}$ framework up to 36%.
- *Scalability:* We also observe that our overall speedup over SPD$\mathbb{Z}_{2^k}$ increases when we increase the interconnection bandwidth between parties or increase the ring size from 32 bits to 64 bits. Furthermore, our tag-computation speedup over SPD$\mathbb{Z}_{2^k}$ remains almost the same for an increasing number of parties and does not decrease, thus showcasing the clear scalability of CompactTag.

1.2 Related work

SPDZ [14], MASCOT [27], and Overdrive [28] are actively secure dishonest majority MPC over finite fields. The work of [4] also proposed improvements by considering function-dependent preprocessing phases. CompactTag is compatible with all these works by replacing ring-based sub-protocols in CompactTag with finite field-based sub-protocols. The work of [13] proposed optimizations for equality testing, comparison, and truncation that work over the ring of integers modulo 2^k. The work of [16] proposed MPC optimizations for circuits that involve both arithmetic and boolean operations.

There is a recent line of works [17, 20] in the weaker dishonest majority setting where a constant fraction ε (where $\frac{1}{2} < \varepsilon < 1$) of the parties are corrupted by the adversary. In their setting, they obtain efficient protocols based on packed secret sharing. Our protocol considers the stronger and more robust setting where the adversary can corrupt up to all but one party.

A plethora of works focus on optimizations for the online phase [10, 26, 18, 49, 29, 48], but those works mostly focus on the passive secure 2PC protocols whereas CompactTag is designed specifically for actively secure dishonest majority MPC. The work of [34] proposes a system-level optimization of Softmax specifically for MPC, but it could be applicable for actively secure protocols as well.

There are privacy-preserving ML protocols in the two-party [39, 22, 21, 25] and a small number of parties [36, 31, 40, 30] setting. The recent works of [22, 21, 25] proposed customized secure training and inference protocols in the two-party setting using functional secret sharing based on GPUs. We focus on the generalized n-party setting.

The works of [9, 33] consider the case where a server holds the training model and a client wants to perform a secure inference using its secret data. We note that CompactTag (or even the SPDZ framework) considers a different scenario where none of the participating parties have access to the entire model or the data and hence our threat model is fundamentally different and likely to be more practical in industrial deployments.

Some works focus on secure hardware [46, 37, 23] or memory access patterns [19, 44, 41, 42]. Those works fall outside the scope of MPC.

1.3 Paper organization

In Section 2, we present our notations and discuss the SPD$\mathbb{Z}_{2^k}$ framework. In Section 3, we provide an overview of CompactTag and the formal protocol details. Finally, we provide our experimental results in Section 4. Appendix B provides the security analysis of CompactTag.

2 Preliminaries

We provide the notations and the necessary background in secret-sharing and SPD$\mathbb{Z}_{2^k}$ in this section.

2.1 Security model

We design protocols in the n-party dishonest majority setting. Let $\mathcal{P} = \{\mathsf{P}_1, \mathsf{P}_2, \ldots, \mathsf{P}_n\}$ denote the set of n parties involved in the computation. These parties are connected via pairwise private and authenticated channels. The parties also have access to a broadcast channel. We assume the presence of a probabilistic polynomial time malicious adversary $\mathcal{A}$ that can corrupt up to $t < n$ parties in $\mathcal{P}$. Our protocols are secure in the standard real/ideal-world simulation paradigm [5].

2.2 Notation

We use the following notations in the paper. $\mathbb{Z}_{2^k}$ denotes the ring of integers $\{0, 1, \ldots, 2^k - 1\}$ with addition and multiplication operations performed modulo 2^k. Boldfont capital letters such as $\mathbf{M}$ denote a matrix whereas boldfont small letters denote a vector such as $\mathbf{v}$. $\mathbf{M} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$ denotes a matrix of dimension $\mathsf{T}_1 \times \mathsf{T}_2$ where each element $\mathbf{M}_{pq}$ in $\mathbf{M}$, for $p \in \{1, \ldots, \mathsf{T}_1\}, q \in \{1, \ldots, \mathsf{T}_2\}$, belongs to the ring $\mathbb{Z}_{2^k}$. $\mathbf{X} \odot \mathbf{Y}$ denotes a matrix multiplication operation between matrices $\mathbf{X} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$ and $\mathbf{Y} \in \mathbb{Z}_{2^k}^{\mathsf{T}_2 \times \mathsf{T}_3}$. $\mathbf{X} * \mathbf{Y}$ denotes element-wise multiplication operation between elements of the matrices $\mathbf{X} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$ and $\mathbf{Y} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$. $\mathbf{X} + \mathbf{Y}$ denotes element-wise addition operation between elements of the matrices $\mathbf{X}$ and $\mathbf{Y}$. κ denotes the computation security parameter. $(\mathbf{M})^{\mathsf{c}}$ denotes that the matrix $\mathbf{M} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$ has been compressed along one dimension, say along the columns (by linearly combining all the columns under random linear combiners), to obtain $(\mathbf{M})^{\mathsf{c}} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times 1}$. We use $\mathsf{x} =_k \mathsf{y}$ to denote $\mathsf{x} = \mathsf{y} \mod 2^k$. We use the notation $[\mathsf{v}]$ to denote that a value $\mathsf{v} \in \mathbb{Z}_{2^k}$ is additively shared, i.e., there exist values $[\mathsf{v}]_i \in \mathbb{Z}_{2^k}$ for $i \in \{1, \ldots, n\}$ such that $\mathsf{v} =_k \sum_i [\mathsf{v}]_i$.

2.3 Commitment scheme

Let $\mathsf{COM}(x)$ denote the commitment of a value x in the Universally-Composable(UC) model [6]. The commitment scheme $\mathsf{COM}(x)$ possesses two properties; *hiding* and *binding*. The former ensures privacy of the value x given just its commitment $\mathsf{COM}(x)$, while the latter prevents a corrupt party from opening the commitment to a different value $x' \neq x$. In addition, UC-secure commitments [7, 8] require a simulator (for a corrupt committer) to extract the message committed by a corrupt committer. Also, it enables a simulator (for an honest committer) to commit to 0 and later open it to any valid message by using the trapdoor. This is abstracted via the ideal functionality $\mathcal{F}_{\mathsf{NICOM}}$ for non-interactive commitments in [8]. A practical realization of a commitment scheme can be via a hash function $\mathsf{H}(\cdot)$ given below, whose security can be proven in the programmable random-oracle model (ROM)—for $(c, o) = (\mathsf{H}(x||r), x||r) = \mathsf{COM}(x; r)$.

2.4 Authenticated secret-sharing semantics

We design our protocols using the authenticated additive sharing semantics described in [12]. Here, not only each secret is additively shared among the parties in $\mathcal{P}$, but there also there exists a message authentication code (MAC) (also referred to as a *tag*) on each secret, generated under a global MAC key. The tag and the global MAC key are also additively shared among the parties in $\mathcal{P}$. We will elaborate on this next.

Let s be the statistical security parameter. Let $\Delta \in \mathbb{Z}_{2^s}$ denote a global key that is additively shared among the parties, i.e., $\mathsf{P}_i \in \mathcal{P}$ holds $[\Delta]_i \in \mathbb{Z}_{2^s}$ such that $\Delta =_s \sum_i [\Delta]_i$. Let $\mathsf{v} \in \mathbb{Z}_{2^k}$ denote the secret value that has to be authenticated shared. Let $[\mathsf{v}]_i \in \mathbb{Z}_{2^{k+s}}$ denote the additive shares of $\mathsf{v} \in \mathbb{Z}_{2^k}$ in a larger ring $\mathbb{Z}_{2^{k+s}}$, i.e. $\tilde{\mathsf{v}} =_{k+s} \sum_i [\mathsf{v}]_i$ and $\mathsf{v} =_k \tilde{\mathsf{v}}$. Let $\mathsf{tag}_\mathsf{v} =_{k+s} \Delta \cdot \tilde{\mathsf{v}}$ denote the tag where tag_v is also $[\cdot]$-shared over $\mathbb{Z}_{2^{k+s}}$ among parties in $\mathcal{P}$. We say that v is authenticated secret shared or $[\![\cdot]\!]$-shared if $\mathsf{P}_i \in \mathcal{P}$ holds $[\mathsf{v}]_i \in \mathbb{Z}_{2^{k+s}}, [\mathsf{tag}_\mathsf{v}]_i \in \mathbb{Z}_{2^{k+s}}, [\Delta]_i \in \mathbb{Z}_{2^s}$. We thus denote $[\![\mathsf{v}]\!]$ as the tuple $([\mathsf{v}], [\mathsf{tag}_\mathsf{v}], [\Delta])$. Observe that

$$\sum_{i=1}^{n} [\mathsf{tag}_\mathsf{v}]_i =_{k+s} (\sum_{i=1}^{n} [\mathsf{v}]_i) \cdot (\sum_{i=1}^{n} [\Delta]_i) \qquad (1)$$

Notice that $[\![\cdot]\!]$-sharing scheme satisfies the linearity property, i.e., given constants $c_1, \ldots, c_m \in \mathbb{Z}_{2^{k+s}}$ and $[\![\mathsf{x}_1]\!], \ldots, [\![\mathsf{x}_m]\!]$, parties can compute $[\![c_1\mathsf{x}_1 + \ldots + c_t\mathsf{x}_m]\!]$ non-interactively by locally multiplying the constant with the additive shares of $\mathsf{x}_i, \mathsf{tag}_{\mathsf{x}_i}$ for $i \in \{1, \ldots, m\}$ that they possess and adding up these resultant values.

We say that a matrix $\mathbf{M} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$ is $[\![\cdot]\!]$-shared if every element in $\mathbf{M}$ is $[\![\cdot]\!]$-shared. We denote this as $[\![\mathbf{M}]\!] = ([\mathbf{M}], [\mathsf{tag}_\mathbf{M}], [\Delta])$ where $\mathsf{tag}_\mathbf{M}$ is the matrix comprising of tags on each element in $\mathbf{M}$. Further, for constants $c_1, c_2 \in \mathbb{Z}_{2^k}$ and matrices $\mathbf{X}, \mathbf{Y} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$, we use the notation $c_1[\![\mathbf{X}]\!] + c_2[\![\mathbf{Y}]\!]$ to denote the operation of multiplying each element in $[\mathbf{X}]$ with c_1 and each element in $[\mathbf{Y}]$ with c_2, followed by element-wise addition of the $[\cdot]$-shared matrices (with the same operation performed on the tags as well), i.e., $c_1[\![\mathbf{X}]\!] + c_2[\![\mathbf{Y}]\!] = (c_1[\mathbf{X}] + c_2[\mathbf{Y}], c_1[\mathsf{tag}_\mathbf{X}] + c_2[\mathsf{tag}_\mathbf{Y}], [\Delta])$.

2.5 Overview of SPD$\mathbb{Z}_{2^k}$ [12]

We next give a brief overview of some primitives from SPD$\mathbb{Z}_{2^k}$ that we rely on for our construction. We refer an interested reader to [12] for further details.

Authenticated secret sharing: To design an actively secure protocol, SPD$\mathbb{Z}_{2^k}$ uses secure tags to detect any cheating that occurs during the protocol run. All inputs and the intermediate values that are computed as part of the function evaluation are authenticated secret shared, as described in section 2.4.

Authenticating an additively shared secret: We rely on the ideal functionality $\mathcal{F}_{\mathsf{MAC}}$ (see Appendix A.2) from [12]. During the initialization phase, it samples and distributes additive shares ($[\cdot]$-shares) of the global MAC key, Δ. Post this initialization, $\mathcal{F}_{\mathsf{MAC}}$ uses the global Δ to generate additive shares of a tag (MAC) on a secret that is additively shared among the parties. For this, it takes $[\cdot]$-shares of a secret $\mathsf{v} \in \mathbb{Z}_{2^k}$ as input and generates $[\cdot]$-shares of its tag.

Sampling public constants: We rely on the coin-tossing functionality $\mathcal{F}_{\mathsf{Rand}}$ (see Appendix A.1 for details) from [12], which randomly samples an element from the ring and makes it available to all parties.

Reconstructing $[\![\cdot]\!]$-shared values and checking their tags: At a high level, to reconstruct a $[\![\cdot]\!]$-shared value $\mathsf{v} \in \mathbb{Z}_{2^k}$, each party broadcasts its share $[\mathsf{v}]_i$. Then, everyone computes $\tilde{\mathsf{v}} =_{k+s} \sum_i [\mathsf{v}]_i$ and checks if $\Delta \cdot \tilde{\mathsf{v}}$ equals tag_v modulo 2^{k+s} without revealing Δ. Note that to ensure privacy, as discussed in [12], the higher order s bits of $\tilde{\mathsf{v}}$ are masked before reconstruction. This is because v may be a linear combination of other $[\![\cdot]\!]$-shared values, and the higher order s bits of v may leak information about overflows that occurred during the linear combinations. For reconstructing v, parties perform:

1. Generate $[\![\cdot]\!]$-shares of a random $\mathsf{r} \in \mathbb{Z}_{2^s}$ using $\mathcal{F}_{\mathsf{MAC}}$.
2. Compute $[\![\mathsf{w}]\!] = [\![\mathsf{v}]\!] + 2^k \cdot [\![\mathsf{r}]\!]$.
3. Each $\mathsf{P}_i \in \mathcal{P}$ broadcasts $[\mathsf{w}]_i$, and reconstructs $\mathsf{w} =_{k+s} \sum_{i=1}^{n} [\mathsf{w}]_i$.
4. Each P_i commits to $[\mathsf{cs}]_i =_{k+s} [\mathsf{tag}_\mathsf{w}]_i - \mathsf{w} \cdot [\Delta]_i$ using commitment randomness via $\mathcal{F}_{\mathsf{NICOM}}$ and broadcast the commitments.
5. After obtaining all the broadcasted commitments, all parties open their commitments and check if $\mathsf{cs} =_{k+s} \sum_{i=1}^{n} [\mathsf{cs}]_i$ equals 0.
6. If $\mathsf{cs} =_{k+s} 0$, parties take $\mathsf{v} =_k \mathsf{w}$ as the output, otherwise abort.

As discussed in [12], the above approach has a failure probability of 2^{-s}.

Batch reconstruction: For ML workloads that may require reconstructing a huge number of secret values, it is useful to check for the correctness of a large batch of values at the end of the protocol in a single shot rather than checking for each value separately. For this, SPD$\mathbb{Z}_{2^k}$ proposes a batch reconstruction procedure for reconstructing and checking m $[\![\cdot]\!]$-shared values in a single shot using random linear combinations. Batch reconstruction allows parties to reduce the communication as well as round complexity in comparison to performing a single check for each of the m shared values. The detailed procedure, Π_{BatchRec}, is described in Figure 1. Since our constructions (discussed in Section 3) will operate on matrices and we compare our technique with this method, this procedure is described to take as input a matrix comprising of $\mathsf{T}_1 \cdot \mathsf{T}_2$ elements (instead of considering m values) and the goal is to reconstruct all the entries in $\mathbf{M}$.

Note that even when dealing with a matrix of values, the Π_{BatchRec} procedure requires parties to commit to just one value (cs) during the tag check process. This contrasts with the multiple values that would be needed if one were to use the single value reconstruction method mentioned earlier. In this way, Π_{BatchRec} reduces the communication complexity when reconstructing multiple values. Finally, as discussed

Procedure Π_{BatchRec}

INPUT: $[\![\mathbf{M}]\!] = ([\mathbf{M}], [\mathsf{tag}_{\mathbf{M}}], [\Delta])$ for $\mathbf{M} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$.

OUTPUT: $\mathbf{M}$.

Preprocessing phase: Generate $[\![\cdot]\!]$-shares (over $\mathbb{Z}_{2^{k+s}}$) of an all-zero matrix, $\mathbf{R} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$ using $\mathcal{F}_{\mathsf{MAC}}$ (see Section 2.5) such that each element $\mathbf{R}_{pq} \in \mathbf{R}$ for $p \in \{1, \ldots, \mathsf{T}_1\}, q \in \{1, \ldots, \mathsf{T}_2\}$ satisfies $\mathbf{R}_{pq} =_k \sum_i [\mathbf{R}_{pq}]_i =_k 0$.

Online phase:

// Open

To reconstruct the matrix $\mathbf{M}$, parties do the following:

1. Compute $[\![\mathbf{W}]\!] = [\![\mathbf{M}]\!] + [\![\mathbf{R}]\!]$.
2. Parse $[\![\mathbf{W}]\!] = ([\mathbf{W}], [\mathsf{tag}_{\mathbf{W}}], [\Delta])$.
3. Broadcast shares of $[\mathbf{W}]$, and compute $\mathbf{W} =_{k+s} \sum_{i=1}^{n} [\mathbf{W}]_i$.

// Tag check

To check for the correctness of the reconstructed $\mathbf{M}$, parties do the following.

1. Sample a public random matrix $\chi \in \mathbb{Z}_{2^s}^{\mathsf{T}_1 \times \mathsf{T}_2}$ by invoking $\mathcal{F}_{\mathsf{Rand}}$.
2. Compute $[\mathbf{CS}] =_{k+s} [\mathsf{tag}_{\mathbf{W}}] * \chi - [\Delta]_i \cdot (\mathbf{W} * \chi)$.
3. Compute and commit to $[\mathsf{cs}] =_{k+s} \sum_{i=1}^{\mathsf{T}_1} \sum_{j=1}^{\mathsf{T}_2} [\mathbf{CS}_{i,j}]$ using the commitment randomness via $\mathcal{F}_{\mathsf{NICOM}}$ and broadcast the commitments.
4. After obtaining all the broadcasted commitments, all parties open their commitments and compute $\mathsf{cs} =_{k+s} \sum_i^n [\mathsf{cs}]_i$.
5. Check if $\mathsf{cs} =_{k+s} 0$. If the check passes, output $\mathbf{M} =_k \mathbf{W}$, else abort.

Figure 1: Procedure for reconstructing all elements in $\mathbf{M} \in \mathbb{Z}_{2^{k+s}}^{\mathsf{T}_1 \times \mathsf{T}_2}$ as a batch and checking their tags in SPD$\mathbb{Z}_{2^k}$.

in [12], Π_{BatchRec} fails with a small probability which is upper bounded by $2^{-s+\log(s+1)}$.

Matrix multiplication: Given $[\![\cdot]\!]$-shares of matrices $\mathbf{X} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$ and $\mathbf{Y} \in \mathbb{Z}_{2^k}^{\mathsf{T}_2 \times \mathsf{T}_3}$, consider the case of computing $[\![\cdot]\!]$-shares of $\mathbf{Z} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_3}$ where $\mathbf{Z} = \mathbf{X} \odot \mathbf{Y}$. To achieve this, the literature [29, 12, 13, 48] has relied on using matrix Beaver triples that are $[\![\cdot]\!]$-shared. Let $(\mathbf{A}, \mathbf{B}, \mathbf{C})$ be such a triple where $\mathbf{A} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$, $\mathbf{B} \in \mathbb{Z}_{2^k}^{\mathsf{T}_2 \times \mathsf{T}_3}$, $\mathbf{C} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_3}$ and $\mathbf{C} = \mathbf{A} \odot \mathbf{B}$. The preprocessing phase of multiplication involves generating $[\![\cdot]\!]$-shares of $\mathbf{A}, \mathbf{B}, \mathbf{C}$, which are used in the online phase to generate $[\![\cdot]\!]$-shares of $\mathbf{Z}$. This involves reconstructing $\mathbf{E} = (\mathbf{X} - \mathbf{A})$ and $\mathbf{U} = (\mathbf{Y} - \mathbf{B})$ and computing $[\mathbf{Z}]$ and $[\mathsf{tag}_{\mathbf{Z}}]$ as

Procedure Π_{MatMul}

INPUT: $[\![\mathbf{X}]\!] = ([\mathbf{X}], [\mathsf{tag}_{\mathbf{X}}], [\Delta])$, $[\![\mathbf{Y}]\!] = ([\mathbf{Y}], [\mathsf{tag}_{\mathbf{Y}}], [\Delta])$ for $\mathbf{X} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$, and $\mathbf{Y} \in \mathbb{Z}_{2^k}^{\mathsf{T}_2 \times \mathsf{T}_3}$.

OUTPUT: $[\![\mathbf{Z}]\!]$ where $\mathbf{Z} = \mathbf{X} \odot \mathbf{Y}$.

Preprocessing phase: Generate $[\![\cdot]\!]$-shares of Beaver matrix triple $\mathbf{A} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$, $\mathbf{B} \in \mathbb{Z}_{2^k}^{\mathsf{T}_2 \times \mathsf{T}_3}$, and $\mathbf{C} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_3}$ such that $\mathbf{C} = \mathbf{A} \odot \mathbf{B}$ using the technique described in [12, 13].

Online phase:

1. Parties execute open phase of Π_{BatchRec} (Figure 1) to reconstruct $\mathbf{E} = \mathbf{X} - \mathbf{A}$ and $\mathbf{U} = \mathbf{Y} - \mathbf{B}$.
2. Parties locally compute
   - $[\mathbf{Z}] = [\mathbf{C}] + \mathbf{E} \odot [\mathbf{B}] + [\mathbf{A}] \odot \mathbf{U}$, and
   - $[\mathsf{tag}_{\mathbf{Z}}] = [\mathsf{tag}_{\mathbf{C}}] + \mathbf{E} \odot [\mathsf{tag}_{\mathbf{B}}] + [\mathsf{tag}_{\mathbf{A}}] \odot \mathbf{U}$
3. P_1 locally computes $[\mathbf{Z}]_1 = [\mathbf{Z}]_1 + \mathbf{E} \odot \mathbf{U}$ and $[\mathsf{tag}_{\mathbf{Z}}]_1 = [\mathsf{tag}_{\mathbf{Z}}]_1 + [\Delta]_1 \cdot (\mathbf{E} \odot \mathbf{U})$

// Verification

1. Parties use the tag check phase from Π_{BatchRec} (Figure 1) to check the validity of opened matrices $\mathbf{E}$ and $\mathbf{U}$.
2. If the previous step does not abort, output $[\![\mathbf{Z}]\!] = ([\mathbf{Z}], [\mathsf{tag}_{\mathbf{Z}}], [\Delta])$.

Figure 2: Procedure of matrix multiplication in SPD$\mathbb{Z}_{2^k}$.

follows.

$$[\mathbf{Z}] = [\mathbf{C}] + \mathbf{E} \odot [\mathbf{B}] + [\mathbf{A}] \odot \mathbf{U} + \mathbf{E} \odot \mathbf{U} \tag{2}$$

$$[\mathsf{tag}_{\mathbf{Z}}] = [\mathsf{tag}_{\mathbf{C}}] + \mathbf{E} \odot [\mathsf{tag}_{\mathbf{B}}] + [\mathsf{tag}_{\mathbf{A}}] \odot \mathbf{U} + [\Delta](\mathbf{E} \odot \mathbf{U}) \tag{3}$$

The procedure Π_{MatMul} for matrix multiplication is described in Figure 2.

2.6 Secure ML operations

We next discuss the primitives that are relied on to realize secure ML operations. Note that in ML, convolution and fully connected layers (matrix multiplications) are considered Linear layers.

Convolution: Convolution operations are also widely used in ML applications. Like fully connected layers, convolution layer is also a linear layer, and it can even be written in the form of matrix multiplication as discussed in [1, 31]. For example, consider convolution with a kernel $f \times f$ over a $w \times h$ input with $p \times p$ padding using $s \times s$ stride, and the convolution has i input channels and o output channels. This convolution can be reduced to a matrix multiplication between $\mathsf{T}_1 \times \mathsf{T}_2$ and $\mathsf{T}_2 \times \mathsf{T}_3$ matrices, where

$$\mathsf{T}_1 = \frac{(w-f+2p)(h-f+2p)}{s^2},\ \mathsf{T}_2 = i \cdot f \cdot f,\ \mathsf{T}_3 = o \quad (4)$$

Fixed-point arithmetic: Operands for ML workloads are real numbers. Fixed point arithmetic provides an efficient and accurate method for representing real numbers over the ring algebraic structure, thereby performing FPA operations on the encoded values over rings [29, 45, 49, 48, 34, 36, 13]. Here, a fractional value is represented as a k-bit number in signed 2's complement notation, where the most significant bit is the sign bit, f least significant bits denote the fractional part (also known as precision bits). Operations are performed on the k-bit integer, treated as an element of $\mathbb{Z}_{2^k}$, modulo 2^k. We refer an interested reader to [29] for further details.

Truncation: For ML workloads, fixed-point encoding is commonly used to represent real numbers. When using fixed-point representation (f bits are used for precision) to perform multiplication, the trailing f bits of the result of multiplication must be discarded for correctness. The most direct approach to drop the last f bits given $[\mathsf{v}]$, is to have each party $\mathsf{P}_i \in \mathcal{P}$ to locally divide $[\mathsf{v}]_i$ by $l = 2^{\mathsf{f}}$. However, this method would produce errors if the sum of the shares $[\mathsf{v}]_i$ wraps around the ring $\mathbb{Z}_{2^{k+s}}$, as described in [29]. Elaborately, let θ_{v} denote the number of times the sum of $[\mathsf{v}]_i$ wraps around $\mathbb{Z}_{2^{k+s}}$, i.e., $\mathsf{v} = \sum_i [\mathsf{v}]_i - \theta_{\mathsf{v}} 2^{k+s}$. It becomes evident that this approach would fail when $\theta_{\mathsf{v}} \neq 0$, because each party $\mathsf{P}_i \in \mathcal{P}$ will locally truncate its share $[\mathsf{v}]_i$ which results in them having a sharing of $\mathsf{v}' =_k \sum_i \frac{[\mathsf{v}]_i}{l}$. However, parties should instead hold a sharing of $\frac{\mathsf{v}}{l} = \sum_i \frac{[\mathsf{v}]_i}{l} - \frac{\theta_{\mathsf{v}}}{l} 2^{k+s}$, and observe that $\mathsf{v}' \neq \frac{\mathsf{v}}{l}$. Hence, a different approach is required. We rely on the approach described in [13], which is detailed in Figure 3 (described with respect to truncating each element in a given matrix). At a high level, to truncate $\mathsf{v} \in \mathbb{Z}_{2^{k+s}}$ which is additively shared, this approach relies on reconstructing $\mathsf{r} - \mathsf{v}$ for a random value $\mathsf{r} \in \mathbb{Z}_{2^{k+s}}$. This is followed by truncating $\mathsf{r} - \mathsf{v}$ in clear to obtain its truncated version, denoted as $(\mathsf{r} - \mathsf{v})^{\mathsf{f}}$. This value is then subtracted from the truncated version of r, denoted as r^{f}, to obtain the truncated v, denoted as v^{f}.

Non-linear Functions: Besides linear functions, ML workloads also include many non-linear functions like ReLU, Softmax, and MaxPooling. For these, we rely on the procedures described in the work of [13]. We refer an interested reader to [13] for further details.

3 CompactTag

In this section, we will describe our improved solution, CompactTag, that facilitates efficient matrix multiplication protocol. Recall from Section 2.5 that when performing matrix multiplication to compute $[\![\cdot]\!]$-shares of $\mathbf{Z} = \mathbf{X} \odot \mathbf{Y}$, where $\mathbf{X} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$ and $\mathbf{Y} \in \mathbb{Z}_{2^k}^{\mathsf{T}_2 \times \mathsf{T}_3}$, one needs to compute $[\cdot]$-shares of $\mathbf{Z}$ as well as $[\cdot]$-shares of $\mathsf{tag}_{\mathbf{Z}}$, as described in Eq. 2, 3. Note that the tags are computed solely for the purpose of detecting malicious entities perturbing the actual computations and thus entail performing $O(\mathsf{T}_1 \cdot \mathsf{T}_2 \cdot \mathsf{T}_3)$ local multiplications. For ML workloads that deal with huge matrices with dimensions in the order of thousands, this tag computation adds a significant amount of computational overhead to achieve active security. We design a solution that reduces the tag overhead without affecting the round or communication complexity of the other components in the evaluation.

We first present a high-level insight of our work followed by details. Instead of computing $[\mathsf{tag}_{\mathbf{Z}}]$ as described in Eq. 3, we compute it optimistically by invoking the procedure Π_{OptMAC} which only requires performing $\mathsf{T}_1 \cdot \mathsf{T}_3$ local multiplications. However, the optimistic computation relies on publicly reconstructing a matrix, which can be potentially altered by an adversary, resulting in an incorrect $[\mathsf{tag}_{\mathbf{Z}}]$. To verify the correctness of this optimistically computed tag, we compute another *compact* tag on $\mathbf{Z}$, denoted as $(\mathsf{tag}_{\mathbf{Z}})^{\mathsf{c}}$. The optimistically computed tag is then verified for correctness using the compact tag. Note that computing the compact tag coupled with its verification requires $O(\mathsf{T}_1 \cdot \mathsf{T}_2 + \mathsf{T}_1 \cdot \mathsf{T}_3 + \mathsf{T}_2 \cdot \mathsf{T}_3)$ local multiplications. In this way, the total number of local

Procedure $\Pi_{\mathsf{Truncation}}$

INPUT: $[\![\mathbf{M}]\!] = ([\mathbf{M}], [\mathsf{tag}_{\mathbf{M}}], [\Delta])$ for $\mathbf{M} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$.

OUTPUT: $[\![\mathbf{M}^{\mathsf{f}}]\!]$ where elements in matrix $\mathbf{M}^{\mathsf{f}}$ are the truncated versions of the elements in $\mathbf{M}$, i.e. $(\mathbf{M}_{pq})^{\mathsf{f}} = \frac{\mathbf{M}_{pq}}{2^{\mathsf{f}}}$ for $p \in \{1, \ldots, \mathsf{T}_1\}, q \in \{1, \ldots, \mathsf{T}_2\}$.

Preprocessing phase: Generate $[\![\cdot]\!]$-shares of a random matrix $\mathbf{R} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$ as well as $\mathbf{R}^{\mathsf{f}} = \frac{\mathbf{R}}{2^{\mathsf{f}}}$ as per [13].

Online phase: Parties do the following.

1. Compute $[\mathbf{D}] = [\mathbf{R}] - [\mathbf{M}]$ and execute the open phase of Π_{BatchRec} (Figure 1) to reconstruct $\mathbf{D}$.
2. Set $[\mathbf{M}^{\mathsf{f}}] = [\mathbf{R}^{\mathsf{f}}] - \frac{\mathbf{D}}{2^{\mathsf{f}}}$ and $[\mathsf{tag}_{\mathbf{M}^{\mathsf{f}}}] = [\mathsf{tag}_{\mathbf{R}^{\mathsf{f}}}] - [\Delta] \cdot \frac{\mathbf{D}}{2^{\mathsf{f}}}$.

// Verification

1. Parties use the tag check phase from Π_{BatchRec} (Figure 1) to check the validity of opened value $\mathbf{D}$.
2. If the previous step does not abort, output $[\![\mathbf{M}^{\mathsf{f}}]\!] = ([\mathbf{M}^{\mathsf{f}}], [\mathsf{tag}_{\mathbf{M}^{\mathsf{f}}}], [\Delta])$.

Figure 3: Procedure for truncating the last f bits each element in a $[\![\cdot]\!]$-shared matrix $\mathbf{M}$ [13].

multiplications required to generate $[\cdot]$-shares of tag on $\mathbf{Z}$ is reduced to $O(\mathsf{T}_1 \cdot \mathsf{T}_2 + \mathsf{T}_1 \cdot \mathsf{T}_3 + \mathsf{T}_2 \cdot \mathsf{T}_3)$ as opposed to $O(\mathsf{T}_1 \cdot \mathsf{T}_2 \cdot \mathsf{T}_3)$ in the standard approach. This reduction in the computation cost of tag generation is the key to improving performance in ML algorithms (Section 4).

The rest of this section is structured as follows: we first discuss the protocol Π_{Compress} that compresses a given matrix along one dimension in Section 3.1. This protocol will aid in generating the compact tag. This is followed by describing the optimistic tag generation protocol Π_{OptMAC} in Section 3.2. Finally, we discuss the improved matrix multiplication protocol, CompactTag, in Section 3.3 and 3.4, which has a reduced computation overhead for tag generation.

3.1 Compressing a matrix

We use procedure Π_{Compress} to compress a large 2-dimensional matrix into a compact form by reducing it along one dimension. Specifically, Π_{Compress} takes a 2-dimensional matrix, say $\mathbf{M} \in \mathbb{Z}_{2^{k+s}}^{\mathsf{T}_1 \times \mathsf{T}_2}$ as input and outputs a compressed 1-dimensional matrix, denoted as $(\mathbf{M})^{\mathsf{c}} \in \mathbb{Z}_{2^{k+s}}^{\mathsf{T}_1 \times 1}$. It also takes a matrix of public constants $\chi \in \mathbb{Z}_{2^s}^{\mathsf{T}_2 \times 1}$ as an input, which is known to all the parties in $\mathcal{P}$. χ can be obtained by invoking $\mathcal{F}_{\mathsf{Rand}}$ (see Section 2.5). All parties then compute $\mathbf{M} \odot \chi$ to obtain a linear combination of the columns of $\mathbf{M}$, which has the effect of compressing $\mathbf{M}$ and reducing it along one dimension. The formal protocol appears in Figure 4. Note that the input matrix $\mathbf{M}$ may be a publicly known matrix (matrix values known to all parties) or a secret-sharing of a matrix. For an input matrix of dimension $\mathsf{T}_1 \times \mathsf{T}_2$, Π_{Compress} requires performing $\mathsf{T}_1 \cdot \mathsf{T}_2$ local multiplications and reduces the matrix dimension by a factor of T_2. This compression is the first step in reducing the tag computation complexity.

Procedure Π_{Compress}

INPUT: $\mathbf{M} \in \mathbb{Z}_{2^{k+s}}^{\mathsf{T}_1 \times \mathsf{T}_2}$, and a public matrix $\chi \in \mathbb{Z}_{2^s}^{\mathsf{T}_2 \times 1}$.

OUTPUT: $(\mathbf{M})^{\mathsf{c}} \in \mathbb{Z}_{2^{k+s}}^{\mathsf{T}_1 \times 1}$.

Online phase:

// Compress

1. Compute $(\mathbf{M})^{\mathsf{c}} = \mathbf{M} \odot \chi$.

Figure 4: Procedure of compressing a matrix.

3.2 Optimistic tag generation

Given a $[\cdot]$-shared matrix $\mathbf{M} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$, Π_{OptMAC} (Figure 5) optimistically generates $[\cdot]$-shares of a tag on $\mathbf{M}$. For this, $[\![\cdot]\!]$-shares of a random matrix $\mathbf{R} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$ are generated in the preprocessing phase, which is used to optimistically generate $[\cdot]$-shares of tag on $\mathbf{M}$ in the online phase. The approach is for parties to reconstruct $\mathbf{D} = \mathbf{R} - \mathbf{M}$. Parties then use $[\mathsf{tag}_{\mathbf{R}}]$ and $[\mathsf{tag}_{\mathbf{D}}] = [\Delta] \cdot \mathbf{D}$ to generate $[\mathsf{tag}_{\mathbf{M}}] = [\mathsf{tag}_{\mathbf{R}}] - [\mathsf{tag}_{\mathbf{D}}]$ using the linearity of $[\cdot]$-sharing. Note that Π_{OptMAC} does not provide security against malicious adversaries because the reconstruction of $\mathbf{D}$ will not be verified at this point. Hence, we call this step *optimistic* tag generation. An active adversary can introduce errors when reconstructing $\mathbf{D}$. The correctness of $\mathsf{tag}_{\mathbf{M}}$ depends on the correctness of $\mathbf{D}$. Thus it is essential that the correctness of the tag generated via Π_{OptMAC} must be verified by checking the correctness of the reconstructed $\mathbf{D}$. This verification process is described next.

Procedure Π_{OptMAC}

INPUT: $[\mathbf{M}] \in \mathbb{Z}_{2^{k+s}}^{\mathsf{T}_1 \times \mathsf{T}_2}$ for $\mathbf{M} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$.

OUTPUT: $[\mathsf{tag}_{\mathbf{M}}], [\mathsf{tag}_{\mathbf{R}}]_i, \tilde{\mathbf{D}}$.

Preprocessing phase: Generate $[\![\cdot]\!]$-shares of a random matrix $\mathbf{R} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$ using $\mathcal{F}_{\mathsf{MAC}}$ (see Section 2.5), i.e., $[\![\mathbf{R}]\!] = ([\mathbf{R}], [\mathsf{tag}_{\mathbf{R}}], [\Delta])$.

Online phase: Parties do the following.

1. Compute $[\mathbf{D}] = [\mathbf{R}] - [\mathbf{M}]$.
2. P_i broadcasts its $[\cdot]$-share of $\mathbf{D}$, followed by all parties computing $\mathbf{D} =_k \sum_{i=1}^{n} [\mathbf{D}]_i$ and $\tilde{\mathbf{D}} =_{k+s} \sum_{i=1}^{n} [\mathbf{D}]_i$.
3. Compute $[\mathsf{tag}_{\mathbf{M}}] =_{k+s} [\mathsf{tag}_{\mathbf{R}}] - [\Delta] \cdot \mathbf{D}$.
4. Output $[\mathsf{tag}_{\mathbf{M}}]$, $[\mathsf{tag}_{\mathbf{R}}]$, and $\tilde{\mathbf{D}}$.

Figure 5: Procedure for optimistically generating $[\cdot]$-shares of tag on matrix $\mathbf{M}$.

Observe that Π_{OptMAC} requires generating a $[\![\cdot]\!]$-shared matrix $\mathbf{R}$ in the preprocessing phase via $\mathcal{F}_{\mathsf{MAC}}$. In the online phase, it requires broadcasting $[\cdot]$-shares of $\mathbf{D}$ and incurs a communication cost of $\mathsf{T}_1 \cdot \mathsf{T}_2$ elements. One should not be misled to think this is an additional overhead incurred by CompactTag when using Π_{OptMAC}. In ML workloads (which is the focus of this work) that operate on fixed-point arithmetic (FPA), every multiplication is followed by a truncation operation, which involves. Section 3.4 demonstrates how Π_{OptMAC} can be merged with the $\Pi_{\mathsf{truncation}}$, thereby nullifying those overhead.

3.3 Matrix multiplication with Π_{OptMAC}

We next discuss how to reduce the computational overhead due to tag generation using Π_{Compress} and Π_{OptMAC} when performing matrix multiplication. Consider the online computation of $\mathbf{Z} = \mathbf{X} \odot \mathbf{Y}$ where $\mathbf{X} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$ and $\mathbf{Y} \in \mathbb{Z}_{2^k}^{\mathsf{T}_2 \times \mathsf{T}_3}$. As in the standard approach discussed in Section 2.5 (Figure 2), parties begin by reconstructing $\mathbf{E} = \mathbf{X} - \mathbf{A}$ and $\mathbf{U} = \mathbf{Y} - \mathbf{B}$, followed by computing $[\mathbf{Z}]$. Next, instead of computing $[\mathsf{tag}_{\mathbf{Z}}]$ as per Eq. 3, they invoke Π_{OptMAC} to optimistically generate $[\mathsf{tag}_{\mathbf{Z}}]$, which internally involves an optimistic reconstruction of $\mathbf{D} = \mathbf{R} - \mathbf{Z}$ where $\mathbf{R} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_3}$. To verify the correctness of $[\mathsf{tag}_{\mathbf{Z}}]$, one must ensure the correct reconstruction of $\mathbf{D}$. Moreover, it is also required to ensure that we do not inflate the computation cost in the process. To reduce the computation while facilitating the verification of $\mathbf{D}$, parties generate a *compact* tag on $\mathbf{Z}$ via a modified version of Eq. 3 where the compressed versions of $\mathsf{tag}_{\mathbf{C}}, \mathsf{tag}_{\mathbf{B}}$ and $\mathbf{U}$ are used. Elaborately, parties invoke Π_{Compress} (Figure 4) on $\mathsf{tag}_{\mathbf{C}}, \mathsf{tag}_{\mathbf{B}}$ and $\mathbf{U}$ to generate their compressed versions $(\mathsf{tag}_{\mathbf{C}})^{\mathsf{c}}$, $(\mathsf{tag}_{\mathbf{B}})^{\mathsf{c}}$ and $(\mathbf{U})^{\mathsf{c}}$, respectively, under the common set of linear combiners, say $\chi \in \mathbb{Z}_{2^s}^{\mathsf{T}_3 \times 1}$. Parties use those compact operands to compute $(\mathsf{tag}_{\mathbf{Z}})^{\mathsf{c}}$ as

$$[\mathsf{tag}_{(\mathbf{Z})^{\mathsf{c}}}] = [\mathsf{tag}_{(\mathbf{C})^{\mathsf{c}}}] + \mathbf{E} \odot [\mathsf{tag}_{(\mathbf{B})^{\mathsf{c}}}] + [\mathsf{tag}_{\mathbf{A}}] \odot (\mathbf{U})^{\mathsf{c}} + \mathbf{E} \odot (\mathbf{U})^{\mathsf{c}} \quad (5)$$

Observe that this computation of $[(\mathsf{tag}_{\mathbf{Z}})^{\mathsf{c}}]$ only requires $O(\mathsf{T}_1 \cdot \mathsf{T}_2)$ multiplications which is much less than $O(\mathsf{T}_1 \cdot \mathsf{T}_2 \cdot \mathsf{T}_3)$. Having generated $[(\mathsf{tag}_{\mathbf{Z}})^{\mathsf{c}}]$, the next step is to verify the correctness of the reconstructed $\mathbf{D}$. Parties first generate $[(\mathsf{tag}_{\mathbf{D}})^{\mathsf{c}}]$ as $[(\mathsf{tag}_{\mathbf{D}})^{\mathsf{c}}] = [(\mathsf{tag}_{\mathbf{R}})^{\mathsf{c}}] - [(\mathsf{tag}_{\mathbf{Z}})^{\mathsf{c}}]$ where $[(\mathsf{tag}_{\mathbf{R}})^{\mathsf{c}}]$ is also generated using Π_{Compress} under the same linear combiners. This is followed by steps similar to the ones described in *tag check* phase of Π_{BatchRec} (Figure 1) to verify the correctness of $(\mathbf{D})^{\mathsf{c}} = \mathbf{D} \odot \chi$ under the tag $(\mathsf{tag}_{\mathbf{D}})^{\mathsf{c}}$. This verification step is also computationally lightweight, involving $O(\mathsf{T}_1 \cdot \mathsf{T}_3)$ local multiplications. As shown in Appendix B, this verification has the same failure probability as *tag check* phase Π_{BatchRec}. Further, note that this verification can be performed in a single shot toward the end of the computation to verify the correctness of several matrix multiplications. This allows us to amortize the cost due to the verification.

To summarise the working of matrix multiplication, parties begin by computing $[\mathbf{Z}]$ using Beaver matrix triples. They then invoke Π_{OptMAC} to obtain an optimistic tag on $[\mathbf{Z}]$. Next, parties compress (via Π_{Compress}) the necessary matrices using the same public linear combiners and use the compressed operands to compute a compact tag for $[\mathbf{D}]$ via lightweight computations. This compact tag is then used to verify the correctness of the optimistically generated tag via a lightweight verification step.

The need for generating $\mathsf{tag}_{\mathbf{Z}}$ despite availability of $(\mathsf{tag}_{\mathbf{Z}})^{\mathsf{c}}$: Observe that when computing $\mathbf{Z} = \mathbf{X} \odot \mathbf{Y}$, the availability of the compact tag on $\mathbf{Z}$ ($\mathsf{tag}_{(\mathbf{Z})^{\mathsf{c}}}$) suffices to verify the correctness of $\mathbf{Z}$ while achieving the goal of reducing the online computation cost. A natural question that may arise then is about the need to generate $\mathsf{tag}_{\mathbf{Z}}$ as well. We would like to note that this is because while the compact tag on $\mathbf{Z}$ ($\mathsf{tag}_{(\mathbf{Z})^{\mathsf{c}}}$) suffices to verify the correctness of $\mathbf{Z}$, it limits the computation to only one layer of matrix multiplication. We explain this with the help of an example. Consider the scenario where two sequential matrix multiplications are required to be performed, where the output of the first matrix multiplication, say $\mathbf{X}$, is fed as input to the next matrix multiplication, say $\mathbf{Z} = \mathbf{X} \odot \mathbf{Y}$. Consider the case where only the compact tag on $\mathbf{X}$ ($(\mathsf{tag}_{\mathbf{X}})^{\mathsf{c}}$) is generated as part of the first matrix multiplication. Recall that during the computation of $[\mathbf{Z}]$, one needs to robustly open the value $\mathbf{E} = \mathbf{X} - \mathbf{A}$, where $\mathbf{A}$ is part of the matrix Beaver triple $(\mathbf{A}, \mathbf{B}, \mathbf{C} = \mathbf{A} \odot \mathbf{B})$. Verifying the correctness of each element in $\mathbf{E}$ requires the knowledge of a tag on each element in $\mathbf{X}$, i.e., $\mathsf{tag}_{\mathbf{X}}$, and the compact tag on $\mathbf{X}$ would not suffice. Moreover, since the random linear combiners used to generate the compact tag on $\mathbf{X}$ are publicly available at this point, using the compact tag on $\mathbf{X}$ to verify the correctness of the linear combination of elements in $\mathbf{E}$ (using the same linear combiners that were used when computing the compact tag on $\mathbf{X}$) will allow an adversary to introduce errors in $\mathbf{E}$. Thus, to enable performing matrix multiplications consecutively, where the output of one matrix multiplication is fed as input to another (which is a key requirement for DNN models), one needs to maintain the invariant that the inputs to the matrix multiplication are $[\![\cdot]\!]$-shared, i.e. the $[\cdot]$-shares of the tag on each element of the input (matrix) are available. Hence, to maintain the invariant, we need to also generate tags on each element of $\mathbf{Z}$. We generate this via an optimistic approach in Π_{OptMAC}. The correctness of this optimistically generated tag on $\mathbf{Z}$ is then verified using the compact tag on $\mathbf{Z}$.

3.4 Optimized matrix multiplication with CompactTag for DNN

In the above section, we presented our matrix multiplication protocol that uses the Π_{OptMAC}. However, this procedure requires generating $[\![\cdot]\!]$-shares of a random matrix $\mathbf{R} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$ in the preprocessing phase and broadcasting shares of $\mathbf{D}$ in the online phase. These steps require additional communication costs compared to SPD$\mathbb{Z}_{2^k}$. However, in ML algorithms, every multiplication (including matrix multiplication) is followed by a truncation operation when operating on fixed-point arithmetic. Since performing truncation incurs a communication cost equivalent to the cost of Π_{OptMAC}, we instead design an optimized protocol $\Pi_{\mathsf{OptMAC\&Trunc}}$ which can achieve truncation and optimistic tag generation in a single unified approach, while nullifying the communication cost due to Π_{OptMAC}.

Π_{OptMAC} requires generating $[\![\cdot]\!]$-shares of a random matrix $\mathbf{R} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$ in the preprocessing phase, which is also the case in $\Pi_{\mathsf{Truncation}}$ (see Section 2.6). Further, the online phase of Π_{OptMAC} involves broadcasting and reconstructing a matrix $\mathbf{D}$, similar to as done in $\Pi_{\mathsf{Truncation}}$. Finally, both Π_{OptMAC} and $\Pi_{\mathsf{Truncation}}$ involve computing a tag on $\mathbf{M}$ and $\mathbf{M}^{\mathsf{f}}$, respectively, as $[\mathsf{tag}_{\mathbf{M}}] = [\mathsf{tag}_{\mathbf{R}}] - [\Delta] \cdot \mathbf{D}$ and $[\mathsf{tag}_{\mathbf{M}^{\mathsf{f}}}] = [\mathsf{tag}_{\mathbf{R}^{\mathsf{f}}}] - [\Delta] \cdot \frac{\mathbf{D}}{2^{\mathsf{f}}}$. Given this similarity, we design $\Pi_{\mathsf{OptMAC\&Trunc}}$, which generates an optimistic tag and performs the steps required for truncation in a single shot. In do-

Procedure $\Pi_{\mathsf{OptMAC\&Trunc}}$

INPUT: $[\mathbf{M}] \in \mathbb{Z}_{2^{k+s}}^{\mathsf{T}_1 \times \mathsf{T}_2}$ for $\mathbf{M} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$.

OUTPUT: $[\mathbf{M}^{\mathsf{f}}]$, $[\mathsf{tag}_{\mathbf{M}^{\mathsf{f}}}]$, $[\mathsf{tag}_{\mathbf{R}}]$, and $\tilde{\mathbf{D}}$.

Preprocessing phase: Generate $[\![\cdot]\!]$-shares of a random matrix $\mathbf{R} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$ as well as $\mathbf{R}^{\mathsf{f}} = \frac{\mathbf{R}}{2^{\mathsf{f}}}$ [13].

ExpandTrunc:

1. P_i computes $[\mathbf{D}] = [\mathbf{R}] - [\mathbf{M}]$.
2. P_i broadcasts its $[\cdot]$-share of $\mathbf{D}$, followed by all parties computing $\mathbf{D} =_k \sum_{i=1}^{n} [\mathbf{D}]_i$ and $\tilde{\mathbf{D}} =_{k+s} \sum_{i=1}^{n} [\mathbf{D}]_i$
3. P_i computes $[\mathbf{M}^{\mathsf{f}}] = [\mathbf{R}^{\mathsf{f}}] - \frac{\mathbf{D}}{2^{\mathsf{f}}}$.
4. P_i computes $[\mathsf{tag}_{\mathbf{M}^{\mathsf{f}}}] = [\mathsf{tag}_{\mathbf{R}^{\mathsf{f}}}] - [\Delta] \cdot \frac{\mathbf{D}}{2^{\mathsf{f}}}$.

Figure 6: Procedure of optimistic tag generation coupled with truncation.

ing so, we are able to retain the computational improvements brought in by optimistic tag computation without inflating the communication cost.

3.4.1 $\Pi_{\mathsf{OptMAC\&Trunc}}$

$\Pi_{\mathsf{OptMAC\&Trunc}}$ takes a shared matrix $[\mathbf{M}]$ as input, and outputs $[\mathbf{M}^{\mathsf{f}}]$ (where $\mathbf{M}^{\mathsf{f}}$ denotes the truncated version of matrix $\mathbf{M}$, i.e., each element in $\mathbf{M}$ is truncated), as well as $[\cdot]$-shares of an optimistic tag on $\mathbf{M}^{\mathsf{f}}$. For this, in the preprocessing phase, it proceeds similarly to $\Pi_{\mathsf{Truncation}}$ where $[\![\cdot]\!]$-shares of a random matrix $\mathbf{R} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$ as well as $\mathbf{R}^{\mathsf{f}} = \frac{\mathbf{R}}{2^{\mathsf{f}}}$ are generated. In the online phase, it proceeds similarly to Π_{OptMAC} except that it computes $[\cdot]$-shares of the truncated version of $\mathbf{M}$ as $[\mathbf{M}^{\mathsf{f}}] = [\mathbf{R}^{\mathsf{f}}] - \frac{\mathbf{D}}{2^{\mathsf{f}}}$, and the optimistic tag as $[\mathsf{tag}_{\mathbf{M}^{\mathsf{f}}}] = [\mathsf{tag}_{\mathbf{R}^{\mathsf{f}}}] - [\Delta] \cdot \frac{\mathbf{D}}{2^{\mathsf{f}}}$ (similar to the steps of $\Pi_{\mathsf{Truncation}}$). Note as in the case of Π_{OptMAC}, the correctness of the reconstructed $\mathbf{D}$ is not verified. Thus, $\Pi_{\mathsf{OptMAC\&Trunc}}$ is also only passively secure, and it is the obligation of the matrix multiplication protocol to check the validity of the reconstructed $\mathbf{D}$. The formal protocol for $\Pi_{\mathsf{OptMAC\&Trunc}}$ appears in Figure 6. Note that $\Pi_{\mathsf{OptMAC\&Trunc}}$ has the same communication complexity as $\Pi_{\mathsf{Truncation}}$ while nullifying the added communication cost of Π_{OptMAC}.

3.4.2 Complete matrix multiplication protocol

This section will present the complete matrix multiplication procedure using CompactTag. Similar to the protocol outlined in Section 3.3, the initial steps of $\Pi_{\mathsf{CompactMatMul}}$ involve computing $[\mathbf{Z}]$ using the revealed matrices of $\mathbf{E}$ and $\mathbf{U}$. Subsequently, the parties invoke $\Pi_{\mathsf{OptMAC\&Trunc}}$ to truncate $[\mathbf{Z}]$, producing $[\mathbf{Z}^{\mathsf{f}}]$ and generate an optimistic $[\mathsf{tag}_{\mathbf{Z}^{\mathsf{f}}}]$. Following this, parties compress the pertinent operands to compute a compact tag, denoted as $[\mathsf{tag}_{(\mathbf{D})^{\mathsf{c}}}]$, for the optimistically reconstructing $\mathbf{D}$ within $\Pi_{\mathsf{OptMAC\&Trunc}}$. During the verification phase, the MPC parties use $[\mathsf{tag}_{(\mathbf{D})^{\mathsf{c}}}]$ to verify the correctness of $\mathbf{D}$. We provide the formal protocol details in Figure 7, and our security proof can be found in Appendix B.

3.4.3 Complexity analysis

Computation cost Without CompactTag, the resulting tag $[\mathsf{tag}_{\mathbf{Z}}]$ is computed using Equation 3 whose computational complexity (number of local multiplications) is at most $3(\mathsf{T}_1 \cdot \mathsf{T}_2 \cdot \mathsf{T}_3) + \mathsf{T}_1 \cdot \mathsf{T}_3$.

When using CompactTag, the resulting $[\mathsf{tag}_{\mathbf{Z}}]$ is generated using a computational lightweight $\Pi_{\mathsf{OptMAC\&Trunc}}$. The computational complexity of $\Pi_{\mathsf{OptMAC\&Trunc}}$ is $\mathsf{T}_1 \cdot \mathsf{T}_3$ because the size of matrix $[\mathbf{Z}]$ is $\mathsf{T}_1 \times \mathsf{T}_3$. However, because $\Pi_{\mathsf{OptMAC\&Trunc}}$ is only passive secure, parties need to compute and verify $[(\mathsf{tag}_{\mathbf{D}})^{\mathsf{c}}]$ (computed in $\Pi_{\mathsf{CompactMatMul}}$, Figure 7) against $\tilde{\mathbf{D}}$ (computed in $\Pi_{\mathsf{OptMAC\&Trunc}}$, Figure 6). This requires computing a compact tag on $\mathbf{Z}$ using Equation 5 and entails generating compressed versions of the matrices $[\mathsf{tag}_{\mathbf{B}}]$, $[\mathsf{tag}_{\mathbf{C}}]$, $\mathbf{U}$, $[\mathsf{tag}_{\mathbf{R}}]$ and $\tilde{\mathbf{D}}$. Compressing each of the matrices $[\mathsf{tag}_{\mathbf{B}}]$ and $\mathbf{U}$ entails performing $\mathsf{T}_2 \cdot \mathsf{T}_3$ local multiplications since these matrices are of dimension $\mathsf{T}_2 \times \mathsf{T}_3$. Similarly, compressing $[\mathsf{tag}_{\mathbf{C}}]$, $[\mathsf{tag}_{\mathbf{R}}]$ and $\tilde{\mathbf{D}}$, each requires $\mathsf{T}_1 \cdot \mathsf{T}_3$ local multiplication since these matrices are of dimension $\mathsf{T}_1 \times \mathsf{T}_3$. Finally, computing $[(\mathsf{tag}_{\mathbf{Z}})^{\mathsf{c}}]$ via Equation 5 requires $3(\mathsf{T}_1 \cdot \mathsf{T}_2) + \mathsf{T}_1$ local multiplications. In this way, the total number of local multiplications to be computed via CompactTag is $4(\mathsf{T}_1 \cdot \mathsf{T}_3) + 2(\mathsf{T}_2 \cdot \mathsf{T}_3) + 3(\mathsf{T}_1 \cdot \mathsf{T}_2) + \mathsf{T}_1$.

Note that a natural question that may arise is why compression of the matrices was not considered along both dimensions to further improve the computation complexity. While at first glance, this may appear to be true, this is not the case. This is because compressing along both the dimensions would additionally require compressing the $\mathsf{T}_1 \times \mathsf{T}_2$ dimension matrices $[(\mathsf{tag}_{\mathbf{C}})^{\mathsf{c}}]$, $\mathbf{E}$, $[\mathsf{tag}_{\mathbf{A}}]$ incurring additional $3(\mathsf{T}_1 \cdot \mathsf{T}_2)$ local multiplications. While the computation of Equation 5 now requires only $4\mathsf{T}_2$ local multiplications as opposed to the previous $3(\mathsf{T}_1 \cdot \mathsf{T}_2) + \mathsf{T}_1$ multiplications, observe that overall we still require $4\mathsf{T}_2 - \mathsf{T}_1$ additional multiplications. Thus, the computation cost of compressing along both dimensions turns out to be higher than when compressed along a single dimension. Hence, we compress only along a single dimension.

Communication cost When using SPD$\mathbb{Z}_{2^k}$ protocols, parties only need to evaluate Equation 3, and no tag-related communication is induced. On the other hand, when using CompactTag, Π_{OptMAC} induces one round of communication involving $\mathsf{T}_1 \cdot \mathsf{T}_3$ elements. However, we make the observation that we can merge Π_{OptMAC} and $\Pi_{\mathsf{Truncation}}$. Both these protocols require one round of communicating $\mathsf{T}_1 \cdot \mathsf{T}_3$ elements. By merging both these protocols, the resulting pro-

Procedure $\Pi_{\mathsf{CompactMatMul}}$

INPUT: $[\![\mathbf{X}]\!] = ([\mathbf{X}], [\mathsf{tag}_{\mathbf{X}}], [\Delta])$, $[\![\mathbf{Y}]\!] = ([\mathbf{Y}], [\mathsf{tag}_{\mathbf{Y}}], [\Delta])$ for $\mathbf{X} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$, and $\mathbf{Y} \in \mathbb{Z}_{2^k}^{\mathsf{T}_2 \times \mathsf{T}_3}$.

OUTPUT: $[\![\mathbf{Z}^{\mathsf{f}}]\!]$ where $\mathbf{Z} = \mathbf{X} \odot \mathbf{Y}$ and $\mathbf{Z}^{\mathsf{f}}$ denotes that each element in $\mathbf{Z}$ is truncated by f bits.

Preprocessing phase:

1. Generate $[\![\cdot]\!]$-shares of Beaver matrix triple $\mathbf{A} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$, $\mathbf{B} \in \mathbb{Z}_{2^k}^{\mathsf{T}_2 \times \mathsf{T}_3}$, and $\mathbf{C} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_3}$ such that $\mathbf{C} = \mathbf{A} \odot \mathbf{B}$ using the technique described in [12, 13].
2. Execute the preprocessing phase of $\Pi_{\mathsf{OptMAC\&Trunc}}$ to generate $[\![\cdot]\!]$-shares of a random matrix $\mathbf{R} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}$ as well as $\mathbf{R}^{\mathsf{f}} = \frac{\mathbf{R}}{2^{\mathsf{f}}}$.

Online phase:

1. Execute open phase of Π_{BatchRec} (Figure 1) to reconstruct $\mathbf{E} = \mathbf{X} - \mathbf{A}$ and $\mathbf{U} = \mathbf{Y} - \mathbf{B}$.
2. Locally compute $[\mathbf{Z}] = [\mathbf{C}] + \mathbf{E} \odot [\mathbf{B}] + [\mathbf{A}] \odot \mathbf{U}$.
3. Invoke $\Pi_{\mathsf{OptMAC\&Trunc}}$ (Figure 6) on $[\mathbf{Z}]$ to truncate $\mathbf{Z}$ to obtain $[\mathbf{Z}^{\mathsf{f}}]$ as well as optimistically generate the MAC $[\mathsf{tag}_{\mathbf{Z}^{\mathsf{f}}}]$. $\Pi_{\mathsf{OptMAC\&Trunc}}$ also outputs $[\mathsf{tag}_{\mathbf{R}}], \tilde{\mathbf{D}}$ generated in the process.
4. P_1 locally computes $[\mathbf{Z}^{\mathsf{f}}]_1 = [\mathbf{Z}^{\mathsf{f}}]_1 + \frac{\mathbf{E} \odot \mathbf{U}}{2^{\mathsf{f}}}$ and $[\mathsf{tag}_{\mathbf{Z}^{\mathsf{f}}}]_1 = [\mathsf{tag}_{\mathbf{Z}^{\mathsf{f}}}]_1 + [\Delta]_1 \cdot \frac{\mathbf{E} \odot \mathbf{U}}{2^{\mathsf{f}}}$.
5. Generate a public constant matrix $\chi \in \mathbb{Z}_{2^s}^{\mathsf{T}_3 \times 1}$ by invoking $\mathcal{F}_{\mathsf{Rand}}$.
6. Invoke Π_{Compress} (Figure 4) to compress $[\mathsf{tag}_{\mathbf{B}}]$, $[\mathsf{tag}_{\mathbf{C}}]$, $\mathbf{U}$, $[\mathsf{tag}_{\mathbf{R}}]$ and $\tilde{\mathbf{D}}$ under the combiners χ to generate $[(\mathsf{tag}_{\mathbf{B}})^{\mathsf{c}}]$, $[(\mathsf{tag}_{\mathbf{C}})^{\mathsf{c}}]$, $(\mathbf{U})^{\mathsf{c}}$, $[(\mathsf{tag}_{\mathbf{R}})^{\mathsf{c}}]$ and $(\tilde{\mathbf{D}})^{\mathsf{c}}$, respectively.
7. Parties locally compute $[(\mathsf{tag}_{\mathbf{Z}})^{\mathsf{c}}] = [(\mathsf{tag}_{\mathbf{C}})^{\mathsf{c}}] + \mathbf{E} \odot [(\mathsf{tag}_{\mathbf{B}})^{\mathsf{c}}] + [\mathsf{tag}_{\mathbf{A}}] \odot (\mathbf{U})^{\mathsf{c}}$ and $[(\mathsf{tag}_{\mathbf{D}})^{\mathsf{c}}] = [(\mathsf{tag}_{\mathbf{R}})^{\mathsf{c}}] - [(\mathsf{tag}_{\mathbf{Z}})^{\mathsf{c}}]$.

// Verification

1. Use the *tag check* phase from Π_{BatchRec} to verify the correctness of the reconstructed $\mathbf{E}, \mathbf{U}$.
2. All parties sample a public random matrix $\hat{\chi} \in \mathbb{Z}_{2^s}^{\mathsf{T}_1 \times 1}$ via $\mathcal{F}_{\mathsf{Rand}}$.
3. Compute $[\mathbf{CS}] =_{k+s} [\Delta] \cdot (\tilde{\mathbf{D}})^{\mathsf{c}}$.
4. Compute $[\mathbf{CS}] =_{k+2s} [\mathbf{CS}] * \hat{\chi} - [\mathsf{tag}_{(\mathbf{D})^{\mathsf{c}}}] * \hat{\chi}$.
5. Commit to $[\mathsf{cs}] =_{k+2s} \sum_{i=1}^{\mathsf{T}_1} [\mathbf{CS}_{i,1}]$ via $\mathcal{F}_{\mathsf{NICOM}}$. Broadcast commitments. Receive other commitments and decommit to $[\mathsf{cs}]$.
6. If reconstructed $\mathsf{cs} \neq 0$ modulo 2^{k+2s}, or the decommitments fail then abort.
7. If the previous steps do not abort, output $[\![\mathbf{Z}^{\mathsf{f}}]\!] = ([\mathbf{Z}^{\mathsf{f}}], [\mathsf{tag}_{\mathbf{Z}^{\mathsf{f}}}], [\Delta])$.

Figure 7: Matrix multiplication with CompactTag.

tocol $\Pi_{\mathsf{OptMAC\&Trunc}}$ allows the computation of optimistic tag generation as well as truncation to be performed with a single round of communication involving $\mathsf{T}_1 \cdot \mathsf{T}_3$ elements, thereby avoiding the additional communication due to the optimistic generation. Thus, the communication complexity when executing the state-of-art matrix multiplication and truncation protocols vs. when executing $\Pi_{\mathsf{CompactMatMul}}$ with CompactTag is the same.

3.5 Offline phase

When performing matrix multiplication with truncation, the preprocessing phase entails generating the following:

1. $[\![\cdot]\!]$-shares of Beaver matrix triple $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ such that $\mathbf{C} = \mathbf{A} \odot \mathbf{B}$ where $\mathbf{A} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_2}, \mathbf{B} \in \mathbb{Z}_{2^k}^{\mathsf{T}_2 \times \mathsf{T}_3}, \mathbf{C} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_3}$.
2. $[\![\cdot]\!]$-shares of matrix $\mathbf{R} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_3}$ and $\mathbf{R}^{\mathsf{f}} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1 \times \mathsf{T}_3}$ where $\mathbf{R}^{\mathsf{f}} = \frac{\mathbf{R}}{2^{\mathsf{f}}}$.

When using the SPD$\mathbb{Z}_{2^k}$ [12] for performing matrix multiplication with truncation, one needs to execute Π_{MatMul} (Figure 2) followed by $\Pi_{\mathsf{Truncation}}$ (Figure 3). On the other hand, when using CompactTag, we realize this operation via $\Pi_{\mathsf{CompactMatMul}}$ to perform both matrix multiplications and truncation in a single shot. The preprocessing materials needed are identical for CompactTag and SPD$\mathbb{Z}_{2^k}$.

Thus, our preprocessing phase has the same complexity as that of SPD$\mathbb{Z}_{2^k}$, and does not require any other material to be generated in the preprocessing phase to aid our computationally more efficient online phase.

4 Experimental evaluation

In this section, we report our experiments and demonstrate the benefits empirically.

4.1 Experimental setup

We have evaluated our design on servers with an AMD EPYC 7502 CPU and an Nvidia Quadro RTX 5000. Each server's network bandwidth is 14Gbps. We gather runtime from the average of **30** iterations of inference or training.

Models evaluated: We use ResNet50 [24], a Transformer encoder [15], and VGG16 [43] to evaluate the performance of CompactTag against SPD$\mathbb{Z}_{2^k}$. VGG16 and ResNet are popular vision models, and Transformer encoders are commonly used in Natural Language Processing models [15, 35, 32, 11]. Those models commonly consist of multiple Transformer encoders of the same configurations. The Transformer configuration used is the same as BERT [15] and XLM [32], which is H=1024 (hidden vector size), A=16 (attention head count).

MPC setup: We evaluated CompactTag using 2 sets of MPC parameters: 1) $(k=32,\sigma=26)$ and 2) $(k=64,\sigma=57)$. When using $(k=32,\sigma=26)$, the message space for a matrix $\mathbf{M}$ will be $\mathbb{Z}_{2^{32}}^{\mathsf{T}_1\times\mathsf{T}_2}$, where $\sigma=s-log(s+1)$. We also choose $s=32$ in such a case. Thus, σ is $26=32-log(32+1)$, according to Theorem 1. Consequently, with $(k=32,\sigma=26)$, $([\mathbf{M}],[\mathsf{tag}_{\mathbf{M}}],[\Delta])\in(\mathbb{Z}_{2^{64}}^{\mathsf{T}_1\times\mathsf{T}_2}\times\mathbb{Z}_{2^{64}}^{\mathsf{T}_1\times\mathsf{T}_2}\times\mathbb{Z}_{2^{32}})$. Similarly, when using $(k=64,\sigma=57)$, $([\mathbf{M}],[\mathsf{tag}_{\mathbf{M}}],[\Delta])\in(\mathbb{Z}_{2^{128}}^{\mathsf{T}_1\times\mathsf{T}_2}\times\mathbb{Z}_{2^{128}}^{\mathsf{T}_1\times\mathsf{T}_2}\times\mathbb{Z}_{2^{64}})$. The choice of parameters is dictated by the native 32-bit and 64-bit operations on a native CPU architecture, and previous works [13, 12] also consider the same set of security parameters. We let $\kappa=128$ be the computational security parameter.

4.2 Implementation details

We implement the online phase of SPD$\mathbb{Z}_{2^k}$ and CompactTag for DNNs on the basis of CrypTen [29], a PyTorch-based [38] MPC framework. CrypTen provides an easy-to-use API to build DNN models and supports fixed-point operations using the CUDA kernels. However, CrypTen was originally designed for a semi-honest majority MPC protocol and did not separate the online phase and the offline phase. We separated the offline phase and online phase and modified the CrypTen framework such that the MPC operations comply with those specified in [12, 13]. We implemented the entire protocol to execute on GPU parties, which exploit the parallelism in the hardware to accelerate MPC execution. We used the distributed communication backend from PyTorch to establish communication channels for our GPU-based implementation.

CUDA matrix operation implementation: For CUDA matrix multiplication and convolutions, we follow the same block multiplication technique as CrypTen [29] and CryptGPU [45] such that 64-bit or 128-bit integer multiplications are carried out using multiple floating point operations. For example, when using double-precision floating points (52 bits for the fraction) to carry out computations, we divide a 64-bit operand into four 16-bit sub-operands, and each sub-operation requires 32 bits to store the resulting multiplications. Given all operations for matrix multiplication and convolutions are multiply and accumulate operations when using double-precision floating points, 20 bits $(52-32)$ are left for accumulation, allowing $1048576=2^{20}$ accumulations, which is sufficient for ML workloads.

4.3 Performance analysis

The key enhancement brought forth by CompactTag is its optimized tag computation during the online phase. By incorporating optimistic tag expansion with truncation, CompactTag's offline phase remains identical to our baseline detailed in [12, 13]. Hence, to analyze the performance, we focus on the online phase runtime. The runtime decomposition is presented in Table 1. Furthermore, to provide a clearer understanding of the sources of CompactTag's speedups, we offer a visual representation of the runtime decomposition for the Transformer model decomposition in Figure 8 and 9, demonstrating a 22× speedup for tag computation. Figure 10 shows the online speedup of CompactTag on three different large ML models for both inference and training. Additionally, we also demonstrate CompactTag's scalability and CompactTag's impact on power consumption. We discuss as follows.

Inference: The first row in Figure 10 shows the inference performance improvements of CompactTag. For the $(k=32,\sigma=26)$ setting, CompactTag speed up the tag-related computation by 3.05×, 24.77× and 4.16×, for ResNet, Transformer, and VGG16, respectively. This accelerated tag computation translates to online speedups of 1.11×, 1.22×, and 1.21× for ResNet, Transformer, and VGG16, respectively. CompactTag's speedup for tag computation is determined by the model architectures, namely T_1, T_2, and T_3 in different models. T_1, T_2, and T_3 in Transformers allow us to have the best tag computation speedups among all models. For example, a layer in the Transformer, whose $\mathsf{T}_1=2048$, $\mathsf{T}_2=1024$, and $\mathsf{T}_3=1024$, allows us to reduce the number of computations by 384×. In this case, the Transformer model's tag generation for final results is almost free. This drastic reduction in tag generation is directly caused by the usage of smaller tags because smaller tags significantly reduce the computational complexity of tag generation.

For the $(k=64,\sigma=57)$ setting, all models see a higher performance improvement over the $(k=32,\sigma=26)$ setting. Table 1 presents the detailed results for this setting. Most noticeably, CompactTag reduces the tag-related computation by 23×, achieving an overall 1.31× speedup for Transformer. When using the $(k=64,\sigma=57)$ setting, the operand bit becomes $k+s=64+64=128$, instead of 64 bits for the $(k=32,\sigma=26)$ setting. With double operand bit width, MPC's computation costs for convolution and matrix multiplication quadrupled, whereas the communication cost only doubled (opening $\mathbf{E}$ and $\mathbf{U}$). For example, Figure 8 shows the runtime decomposition of Transformer inference with $(k=64,\sigma=57)$. In Figure 8, tag computation makes up 24% of the total runtime for SPD$\mathbb{Z}_{2^k}$. Compared to the $(k=32,\sigma=26)$'s 21%, this tag computation takes a larger proportion of the total runtime. Since CompactTag drastically reduces tag computation costs, CompactTag will have better speedups when tag computation accounts for a larger proportion of the total runtime.

Training: The second row in Figure 10 shows the training performance improvements. CompactTag speeds up tag-related computation by 4.4×, 22×, and 17.5× for Resnet, Transformer and VGG16 models, respectively. For all models and all MPC settings, CompactTag has higher performance

Table 1: Online run time (seconds/input) analysis for inference and training corresponding to $(k = 64, \sigma = 57)$. The "Tag" and "Output" columns represent the time spent on computing the tag ($[\mathsf{tag}_{\mathbf{Z}}]$) and output share ($[\mathbf{Z}]$) for Linear layers (Convolution and Matrix Multiplication).

Model	Protocol	Inference					Training				
		Computation			Comm.	Total	Computation			Comm.	Total
		Tag	Output	Others			Tag	Output	Others		
ResNet	SPD$\mathbb{Z}_{2^k}$	4.34	6.57	2.17	5.69	18.76	11.64	18.88	6.31	9.15	45.99
	CompactTag	1.26	6.54	2.13	5.67	15.59	2.61	18.72	6.47	9.12	36.92
	Speedup	**3.4**$\times$	-	-	-	**1.2**$\times$	**4.45**$\times$	-	-	-	**1.25**$\times$
Transformer	SPD$\mathbb{Z}_{2^k}$	4.16	6.31	1.69	4.88	17.04	12.56	20.35	0.87	6.96	40.74
	CompactTag	0.18	6.30	1.68	5.00	13.15	0.57	20.27	1.21	6.87	28.92
	Speedup	**23**$\times$	-	-	-	**1.3**$\times$	**22**$\times$	-	-	-	**1.4**$\times$
VGG16	SPD$\mathbb{Z}_{2^k}$	5.94	8.96	1.35	4.95	21.20	44.27	73.57	3.67	12.22	133.73
	CompactTag	1.36	8.92	1.63	4.54	16.46	2.52	75.23	4.53	11.51	93.80
	Speedup	**4.4**$\times$	-	-	-	**1.3**$\times$	**17**$\times$	-	-	-	**1.47**$\times$

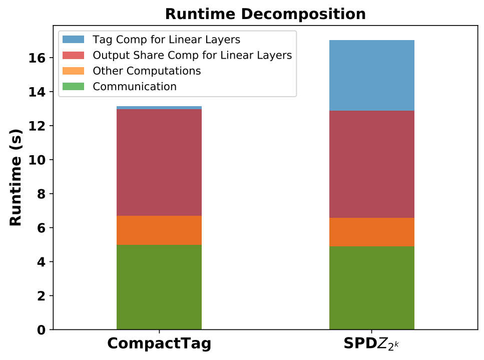


Figure 8: Transformer's Inference Online Phase Runtime Analysis $(k = 64, \sigma = 57)$.

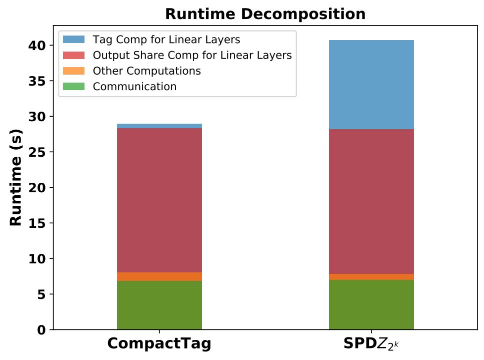


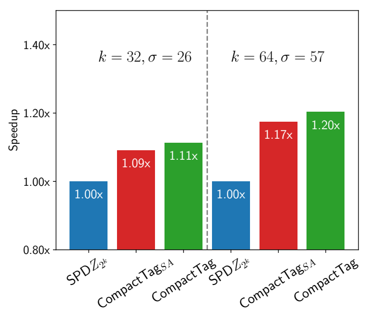
Figure 9: Transformer's Training Online Phase Runtime Analysis $(k = 64, \sigma = 57)$.

benefits for training than for inference. This result is due to the fact that training has increased computations without a significant increased in communication. For non-linear layers such as ReLU and MaxPooling, the backward function involves an element-wise multiplication between the output gradient and a secret shared input mask. This element-wise multiplication in the backward pass induces much less communication cost than those in the forward pass. Moreover, linear layers will need to perform additional large matrix multiplications. For fully connected layers, during the backward pass, MPC parties need to multiply the output gradients with both the input and the weight matrices. Those operations incur more computations than communications and will also incur more tag computations. Thus, the tag computation will take a larger proportion of the total training runtime than inference. Compared with tag computation in Figure 8, tag computation takes a larger proportion in Figure 9. Consequently, CompactTag showcases more speedups in training than inference.

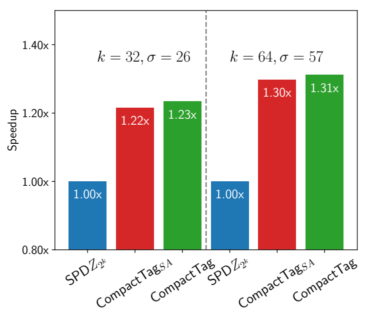
CPU Execution: While we focused all our results using GPUs, given their popularity in ML, it is useful to note that our protocol runs equally well on CPUs. In particular, we have implemented our protocol on CPUs and ran the Transformer training. CompactTag reduces the tag computation by 20$\times$, resulting in 1.54$\times$ online speedup, which is similar to the GPU results shown above.

Scaling with more parties & faster networks: Speedups showcased in Figure 10 are in the 2PC setting. Table 2 shows the CompactTag's speedup w.r.t. SPD$\mathbb{Z}_{2^k}$ with more numbers of participating parties. When scaling to more parties, we assume a point-to-point connection between every pair of parties. The speedups in Table 2 are averaged over all three models. See Appendix C for details about specific models.

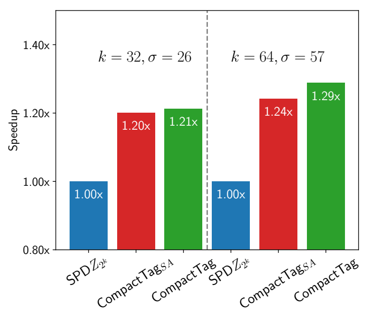
With more participating parties, computation costs for tag computation stay the same. And the tag computation reduction factors are also unchanged. But with more parties the communication costs will start to grow, reducing the tag com-



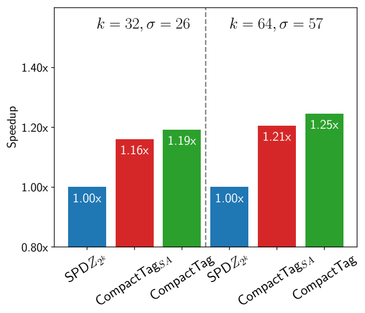
(a) ResNet Inference



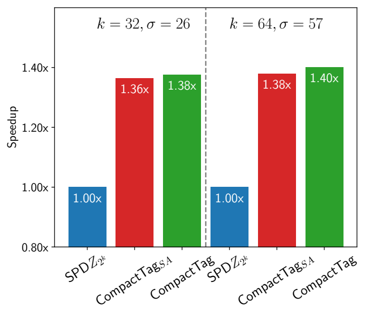
(b) Transformer Inference



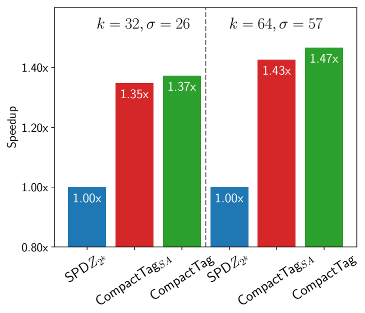
(c) VGG16 Inference



(d) ResNet Training



(e) Transformer Training



(f) VGG16 Training

Note: CompactTag$_{\mathsf{SA}}$ is CompactTag where Π_{OptMAC} generates optimistic tags. CompactTag$_{\mathsf{SA}}$ introduces additional communication costs resulting in a performance gap.

Figure 10: Average Online-Phase Speedup of CompactTag over SPD$\mathbb{Z}_{2^k}$ for 2PC.

		2PC	3PC	4PC	5PC
Inference	Total	1.26×	1.20×	1.17×	1.14×
	Tag	**10.31×**	**10.31×**	**10.31×**	**10.31×**
Training	Total	1.38×	1.32×	1.28×	1.25×
	Tag	**14.69×**	**14.69×**	**14.69×**	**14.69×**

Table 2: CompactTag's average speedup over SPD$\mathbb{Z}_{2^k}$ with varying number of parties."Tag" denote CompactTag's speedup on tag computation for linear layers.

putation portion in the overall performance. As such, the end-to-end speedups are reduced slightly as we move from 2-5 parties.

Faster network: On the other hand, CompactTag benefits from faster network connections. Figure 11 shows CompactTag's speedups with faster interconnections. The speedup is averaged across 3 models. From Figure 11, we can see that as the interconnection speed increases, the performance improvements of CompactTag protocols increase. A faster interconnection speed does not reduce computation runtime but reduces the time spent on data transmission. Reduced communication runtime and constant computation runtime amplify the proportion of tag-related computation in the total runtime. As a result CompactTag achieves better performance improvement with faster interconnections.

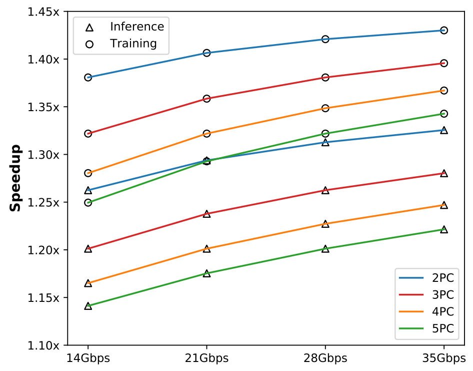


Figure 11: CompactTag's Average Online Performance Improvements over SPD$\mathbb{Z}_{2^k}$ on Different Interconnect Speed.

Power usage: Tables 3 and 4 show the reduction in GPU power usage by CompactTag in comparison to SPD$\mathbb{Z}_{2^k}$. Data presented in this section are collected using PyJoules [3].

Configuration	ResNet	Transformer	VGG16
$k = 32, \sigma = 26$	21.87%	31.51%	30.55%
$k = 64, \sigma = 57$	23.78%	35.34%	34.54%

Table 3: CompactTag's power usage reduction over GPUs during inference; the reduction is measure over SPD$\mathbb{Z}_{2^k}$.

Configuration	ResNet	Transformer	VGG16
$k = 32, \sigma = 26$	24.31%	30.74%	32.98%
$k = 64, \sigma = 57$	28.24%	34.55%	36.42%

Table 4: CompactTag's power usage reduction over GPUs during training; the reduction is measure over SPD$\mathbb{Z}_{2^k}$.

Generally, CompactTag achieves 21% to 36% power usage reduction. This power usage reduction is primarily due to the decreased computational complexity associated with tag-related calculations. For the SPD$\mathbb{Z}_{2^k}$ protocol, large uncompressed tags are used to compute sizeable tags. However, when using CompactTag, smaller compressed tags are used to compute small tags. Consequently, a significant reduction in power usage is realized due to this decrease in computational complexity.

5 Conclusion

We present CompactTag, an optimization in the SPD$\mathbb{Z}_{2^k}$ framework that efficiently generates tags for linear layers in DNN. CompactTag drastically reduces the tag-related computation in the SPD$\mathbb{Z}_{2^k}$ and SPDZ framework for large matrices. Specifically, when multiplying two matrices with dimensions $\mathsf{T}_1 \times \mathsf{T}_2$ and $\mathsf{T}_2 \times \mathsf{T}_3$, the tag computation in CompactTag only requires quadratic computation, i.e., $O(\mathsf{T}_1 \times \mathsf{T}_2 + \mathsf{T}_1 \times \mathsf{T}_3 + \mathsf{T}_2 \times \mathsf{T}_3)$ local multiplications, as opposed to previous protocols that demand cubic computation. This leads to concrete asymptotic enhancements in the online phase. The offline phase of CompactTag remains unchanged from SPD$\mathbb{Z}_{2^k}$ without any additional adjustments. Our experiments show that the asymptotic improvements enable CompactTag to reduce the online tag computation in the SPD$\mathbb{Z}_{2^k}$ framework by a $4.4\times - 22\times$, resulting in $1.11\times - 1.47\times$ inference and training online phase speedups.

Acknowledgment

This material is based upon work supported by Defense Advanced Research Projects Agency (DARPA) under Contract Nos. HR001120C0088, NSF award number 2224319, REAL@USC-Meta center, and VMware gift. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Arpita Patra would like to acknowledge financial support from the Sony Faculty Innovation Award 2021, the J.P Morgan Chase Faculty Award 2022 and the National Security Council, India. Nishat Koti would like to acknowledge financial support from the National Security Council, India.

References

[1] Cs231n: Convolutional neural networks for visual recognition. https://cs231n.github.io/convolutional-networks/. Accessed: 2023-10-12.

[2] Brian Bailey. AI Power Consumption Exploding — semiengineering.com. https://semiengineering.com/ai-power-consumption-exploding. [Accessed 17-10-2023].

[3] Mohammed chakib Belgaid, Romain Rouvoy, and Lionel Seinturier. Pyjoules: Python library that measures python code snippets, November 2019.

[4] Aner Ben-Efraim, Michael Nielsen, and Eran Omri. Turbospeedz: Double your online SPDZ! Improving SPDZ using function dependent preprocessing. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19*, volume 11464 of *LNCS*, pages 530–549. Springer, Heidelberg, June 2019.

[5] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, January 2000.

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

[7] Ran Canetti, Pratik Sarkar, and Xiao Wang. Efficient and round-optimal oblivious transfer and commitment with adaptive security. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 277–308. Springer, Heidelberg, December 2020.

[8] Ran Canetti, Pratik Sarkar, and Xiao Wang. Triply adaptive UC NIZK. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part II*, volume 13792 of *LNCS*, pages 466–495. Springer, Heidelberg, December 2022.

[9] Nishanth Chandran, Divya Gupta, Sai Lakshmi Bhavana Obbattu, and Akash Shah. SIMC: ML inference secure against malicious clients at semi-honest cost. In Kevin

R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 1361–1378. USENIX Association, August 2022.

[10] Minsu Cho, Zahra Ghodsi, Brandon Reagen, Siddharth Garg, and Chinmay Hegde. Sphynx: A deep neural network design for private inference. volume 20, pages 22–34, 2022.

[11] Alexis Conneau and Kartikay Khandelwal. Unsupervised cross-lingual representation learning at scale. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020.

[12] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. Spd$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 769–798. Springer, 2018.

[13] Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1102–1120. IEEE, 2019.

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

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of NAACL-HLT 2019*, 2018.

[16] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 823–852. Springer, Heidelberg, August 2020.

[17] Daniel Escudero, Vipul Goyal, Antigoni Polychroniadou, Yifan Song, and Chenkai Weng. Superpack: Dishonest majority MPC with constant online communication. In *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part II*, volume 14005 of *Lecture Notes in Computer Science*, pages 220–250. Springer, 2023.

[18] Zahra Ghodsi, Nandan Kumar Jha, Brandon Reagen, and Siddharth Garg. Circa: Stochastic relus for private deep learning. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 2241–2252, 2021.

[19] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.

[20] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Sharing transformation and dishonest majority MPC with packed secret sharing. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 3–32. Springer, Heidelberg, August 2022.

[21] Kanav Gupta, Neha Jawalkar, Ananta Mukherjee, Nishanth Chandran, Divya Gupta, Ashish Panwar, and Rahul Sharma. SIGMA: secure GPT inference with function secret sharing. *IACR Cryptol. ePrint Arch.*, page 1269, 2023.

[22] Kanav Gupta, Deepak Kumaraswamy, Nishanth Chandran, and Divya Gupta. LLAMA: A low latency math library for secure inference. *Proc. Priv. Enhancing Technol.*, 2022(4):274–294, 2022.

[23] Hanieh Hashemi, Yongqin Wang, and Murali Annavaram. Darknight: A data privacy scheme for training and inference of deep neural networks. *Proceedings on the 54th International Symposium on Microarchitecture*, 2021.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[25] Neha Jawalkar, Kanav Gupta, Arkaprava Basu, Nishanth Chandran, Divya Gupta, and Rahul Sharma. Orca: Fss-based secure training with gpus. *IACR Cryptol. ePrint Arch.*, page 206, 2023.

[26] Nandan Kumar Jha, Zahra Ghodsi, Siddharth Garg, and Brandon Reagen. Deepreduce: Relu reduction for fast private inference. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 4839–4849. PMLR, 2021.

[27] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016*

ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016, pages 830–842. ACM, 2016.

[28] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 158–189. Springer, 2018.

[29] Brian Knott, Shobha Venkataraman, Awni Y. Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. Crypten: Secure multi-party computation meets machine learning. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 4961–4973, 2021.

[30] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. SWIFT: Super-fast and robust privacy-preserving machine learning. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 2651–2668. USENIX Association, August 2021.

[31] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. Tetrad: Actively secure 4pc for secure training and inference. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022.

[32] Guillaume Lample and Alexis Conneau. Cross-lingual language model pretraining. *33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.*, 2019.

[33] Ryan Lehmkuhl, Pratyush Mishra, Akshayaram Srinivasan, and Raluca Ada Popa. Muse: Secure inference resilient to malicious clients. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 2201–2218. USENIX Association, August 2021.

[34] Dacheng Li, Rulin Shao, Hongyi Wang, Han Guo, Eric P. Xing, and Hao Zhang. Mpcformer: fast, performant and private transformer inference with mpc. arXiv, 2022. *(Accepted in ICLR'23)*.

[35] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[36] Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 35–52, 2018.

[37] Krishna Giri Narra, Zhifeng Lin, Yongqin Wang, Keshav Balasubramaniam, and Murali Annavaram. Privacy-preserving inference in machine learning services using trusted execution environments. *IEEE International Conference on Cloud Computing*, 2021.

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[39] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: improved mixed-protocol secure two-party computation. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2165–2182. USENIX Association, 2021.

[40] Arpita Patra and Ajith Suresh. BLAZE: blazing fast privacy-preserving machine learning. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[41] Rachit Rajat, Yongqin Wang, and Murali Annavaram. Pageoram: An efficient dram page aware oram strategy. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[42] Rachit Rajat, Yongqin Wang, and Murali Annavaram. Laoram: A look ahead oram architecture for training large embedding tables. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[43] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[44] Emil Stefanov, Marten Van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious ram protocol. *J. ACM*, 65(4), April 2018.

[45] Sijun Tan, Brian Knott, Yuan Tian, and David J. Wu. Cryptgpu: Fast privacy-preserving machine learning on the GPU. 2021.

[46] Florian Tramèr and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287*, 2019.

[47] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 39–56. ACM Press, October / November 2017.

[48] Yongqin Wang, Rachit Rajat, and Murali Annavaram. Mpc-pipe: an efficient pipeline scheme for secure multiparty machine learning inference, 2022.

[49] Yongqin Wang, G. Edward Suh, Wenjie Xiong, Benjamin Lefaudeux, Brian Knott, Murali Annavaram, and Hsein-Hsin S. Lee. Characterization of mpc-based private inference for transformer-based models. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 187–197, Los Alamitos, CA, USA, may 2022. IEEE Computer Society.

[50] Kang Yang, Xiao Wang, and Jiang Zhang. More efficient MPC from improved triple generation and authenticated garbling. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1627–1646. ACM Press, November 2020.

A Additional Functionalities

We present the additional functionalities for randomness and MAC generation in this section.

A.1 $\mathcal{F}_{\mathsf{Rand}}$

This section shows the functionality $\mathcal{F}_{\mathsf{Rand}}$ (Fig. 12) each party relies on to generate random numbers.

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

INPUT: Each party P_i invoke this functionality.

ALGORITHM: Sample a random number r from $\mathcal{R}$.

OUTPUT: r to each P_i.

Figure 12: Functionality for generating a random number.

A.2 $\mathcal{F}_{\mathsf{MAC}}$

This section will show the authentication procedure CompactTag relies on. This functionality $\mathcal{F}_{\mathsf{MAC}}$ (Fig. 13) is taken from [12].

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

INITIALIZE: After each party P_i invoke **Initialize**, sample random values $[\Delta]_i \leftarrow \mathbb{Z}_{2^s}$ for $i \notin \mathcal{A}$ and receive $[\Delta]_i \leftarrow \mathbb{Z}_{2^s}$ for $i \in \mathcal{A}$. Store the $\Delta = \sum_{i=1}^{n} [\Delta]_i$ and output $[\Delta]_i$ to P_i.

AUTH: After all P_i invoke **Auth** with $[\mathsf{v}]_i \in \mathbb{Z}_{2^{k+s}}$.

- Compute $\mathsf{v} =_{k+s} \sum_{i=1}^{n} [\mathsf{v}]_i$ and $\mathsf{tag}_\mathsf{v} =_{k+s} \Delta \cdot \mathsf{v}$.
- Wait for $[\mathsf{tag}_\mathsf{v}]_i$ for $i \in \mathcal{A}$ and sample $[\mathsf{tag}_\mathsf{v}]_i$ randomly for $i \notin \mathcal{A}$ such that $\mathsf{tag}_\mathsf{v} =_{k+s} \sum_{i=1}^{n} [\mathsf{tag}_\mathsf{v}]_i$.

AUTHENTICATION: After all P_i invoke **MAC** with $[\mathsf{v}]_i \in \mathbb{Z}_{2^{k+s}}$.

- Wait for the adversary to send messages (**guess**, i, S_i) for all $i \in \mathcal{A}$, where S_i efficiently describes a subset of $\{0,1\}^s$. If $[\Delta]_i \in S_i$ for all i then send (**success**) to $\mathcal{A}$. Otherwise, send (**abort**) to all parties and abort.
- P_i executes **Auth** with $[\mathsf{v}]_i$, then wait for the adversary to send either **success** or **Abort**. If the adversary sends **success** then send the $[\mathsf{tag}_\mathsf{v}]_i$ to P_i, otherwise abort.

Figure 13: Functionality for generating shares of global key and distributing shares of inputs and tags.

B Security Analysis

In this section, we will analyze the privacy and provide a simulation-based proof for the in Figure 7.

Privacy argument: For procedure $\Pi_{\mathsf{CompactMatMul}}$, the inputs are $\mathbf{X}$ and $\mathbf{Y}$. All those matrices are authenticated and shared among a set of MPC parties $\mathcal{P}$. Due to the use of secret sharing, the secrecy of input $\mathbf{X}$ and $\mathbf{Y}$ are provided. During computation of $[\mathbf{Z}]$ and $[\mathsf{tag}_\mathbf{Z}]$ matrices $\mathbf{E} = \mathbf{X} - \mathbf{A}$, $\mathbf{U} = \mathbf{Y} - \mathbf{B}$, and $\mathbf{D} = \mathbf{R} - \mathbf{Z}$ are revealed to all MPC parties. Given matrix $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{R}$ are all random matrices that are unknown to MPC parties, revealed matrices $\mathbf{E}$, $\mathbf{U}$, and $\mathbf{D}$ will not compromise the privacy of $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$.

Proof sketch: We provide a simulation proof sketch of our protocol from Figure 7 by showing that our protocol implements the ideal functionality $\mathcal{F}_{\mathsf{MatMul}}$ (Figure 14) for authenticated matrix multiplication. The preprocessing phase simulation is performed by running the simulator from the SPD$\mathbb{Z}_{2^k}$ protocol [12] since our protocol is the same as theirs. The simulator Sim knows the values $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{R}$ from the simulator of the preprocessing phase. In the online phase, we assume that the inputs are shared in an authenticated way where Sim knows the global MAC key Δ. We describe the simulator for

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

NOTATIONS: List of honest parties is $\mathcal{H}$, and list of corrupt parties is $\mathcal{C}$.

INPUT: Each party P_i (among n parties) invokes the functionality with inputs $[\![\mathbf{X}]\!]_i = ([\mathbf{X}]_i, [\mathsf{tag}_{\mathbf{X}}]_i, [\Delta]_i)$ for $\mathbf{X} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1\times\mathsf{T}_2}$, and $[\![\mathbf{Y}]\!]_i = ([\mathbf{Y}]_i, [\mathsf{tag}_{\mathbf{Y}}]_i, [\Delta]_i)$ for $\mathbf{Y} \in \mathbb{Z}_{2^k}^{\mathsf{T}_2\times\mathsf{T}_3}$.

CORRUPTION: Each corrupt party $\mathsf{P}_j \in \mathcal{C}$ provides its input shares for $([\mathbf{Z}]_j, [\mathsf{tag}_{\mathbf{Z}}]_j)$.

ALGORITHM:

1. Reconstruct $\Delta = \sum_{i=1}^{n}[\Delta]_i$.
2. Reconstruct $\mathbf{X} = \Sigma_{i=1}^{n}[\mathbf{X}]_i$, and $\mathsf{tag}_{\mathbf{X}} \sum_{i=1}^{n}[\mathsf{tag}_{\mathbf{X}}]_i$. Abort if $\mathsf{tag}_{\mathbf{X}} \neq \mathbf{X}\cdot\Delta$.
3. Reconstruct $\mathbf{Y} = \Sigma_{i=1}^{n}[\mathbf{Y}]_i$, and $\mathsf{tag}_{\mathbf{Y}} \sum_{i=1}^{n}[\mathsf{tag}_{\mathbf{Y}}]_i$. Abort if $\mathsf{tag}_{\mathbf{Y}} \neq \mathbf{Y}\cdot\Delta$.
4. Compute $\mathbf{Z} = \mathbf{X}\odot\mathbf{Y}$ and $\mathsf{tag}_{\mathbf{Z}} = \mathbf{Z}\cdot\Delta$.
5. Compute $\widetilde{\mathbf{Z}} = \mathbf{Z} - \sum_{j\in\mathcal{C}}[\mathbf{Z}]_j$ and $\widetilde{\mathsf{tag}_{\mathbf{Z}}} = \mathsf{tag}_{\mathbf{Z}} - \sum_{j\in\mathcal{C}}[\mathsf{tag}_{\mathbf{Z}}]_j$.
6. For each $i \in \mathcal{H}$, sample random $[\mathbf{Z}]_i$ and $[\mathsf{tag}_{\mathbf{Z}}]_i$ values s.t. $\widetilde{\mathbf{Z}} == \sum_{i\in\mathcal{H}}[\mathbf{Z}]_i$ and $\widetilde{\mathsf{tag}_{\mathbf{Z}}} == \sum_{i\in\mathcal{H}}[\mathsf{tag}_{\mathbf{Z}}]_i$.

OUTPUT: $[\![\mathbf{Z}]\!]_i = ([\mathbf{Z}]_i, [\mathsf{tag}_{\mathbf{Z}}]_i, [\Delta])$ for $\mathbf{Z} \in \mathbb{Z}_{2^k}^{\mathsf{T}_1\times\mathsf{T}_3}$ to each party P_i.

Figure 14: Ideal Functionality $\mathcal{F}_{\mathsf{MatMul}}$ for computing matrix multiplication over authenticated shares of matrices.

the online phase step-by-step as follows:

1. Sim executes the open phase of Π_{BatchRec} (Figure 1) to reconstruct $\mathbf{E} = -\mathbf{A}$ and $\mathbf{U} = -\mathbf{B}$ by setting its share values as 0.
2. Same as the protocol steps. Sim locally computes $[\mathbf{Z}] = [\mathbf{C}] + \mathbf{E}\odot[\mathbf{B}] + [\mathbf{A}]\odot\mathbf{U}$.
3. Same as the protocol steps. Sim invokes $\Pi_{\mathsf{OptMAC\&Trunc}}$ (Figure 6) on $[\mathbf{Z}]$ to truncate $\mathbf{Z}$ to obtain $[\mathbf{Z}^{\mathsf{f}}]$ as well as optimistically generate the MAC $[\mathsf{tag}_{\mathbf{Z}^{\mathsf{f}}}]$. $\Pi_{\mathsf{OptMAC\&Trunc}}$ also outputs $[\mathsf{tag}_{\mathbf{R}}], \tilde{\mathbf{D}}$ generated in the process.
4. Same as the protocol steps. If P_1 is corrupt, then do nothing else. Else, Sim simulates party P_1 by locally computing $[\mathbf{Z}^{\mathsf{f}}]_1 = [\mathbf{Z}^{\mathsf{f}}]_1 + \frac{\mathbf{E}\odot\mathbf{U}}{2^{\mathsf{f}}}$ and $[\mathsf{tag}_{\mathbf{Z}^{\mathsf{f}}}]_1 = [\mathsf{tag}_{\mathbf{Z}^{\mathsf{f}}}]_1 + [\Delta]_1\cdot\frac{\mathbf{E}\odot\mathbf{U}}{2^{\mathsf{f}}}$.
5. Same as the protocol steps. Sim generates a public constant matrix $\chi \in \mathbb{Z}_{2^s}^{\mathsf{T}_3\times 1}$ by invoking $\mathcal{F}_{\mathsf{Rand}}$.
6. Same as the protocol steps. Sim invokes Π_{Compress} (Figure 4) to compress $[\mathsf{tag}_{\mathbf{B}}]$, $[\mathsf{tag}_{\mathbf{C}}]$, $\mathbf{U}$, $[\mathsf{tag}_{\mathbf{R}}]$ and $\tilde{\mathbf{D}}$ under the combiners χ to generate $[(\mathsf{tag}_{\mathbf{B}})^{\mathsf{c}}]$, $[(\mathsf{tag}_{\mathbf{C}})^{\mathsf{c}}]$, $(\mathbf{U})^{\mathsf{c}}$, $[(\mathsf{tag}_{\mathbf{R}})^{\mathsf{c}}]$ and $(\tilde{\mathbf{D}})^{\mathsf{c}}$, respectively.
7. Same as the protocol steps. Sim locally computes $[(\mathsf{tag}_{\mathbf{Z}})^{\mathsf{c}}] = [(\mathsf{tag}_{\mathbf{C}})^{\mathsf{c}}] + \mathbf{E}\odot[(\mathsf{tag}_{\mathbf{B}})^{\mathsf{c}}] + [\mathsf{tag}_{\mathbf{A}}]\odot(\mathbf{U})^{\mathsf{c}}$ and $[(\mathsf{tag}_{\mathbf{D}})^{\mathsf{c}}] = [(\mathsf{tag}_{\mathbf{R}})^{\mathsf{c}}] - [(\mathsf{tag}_{\mathbf{Z}})^{\mathsf{c}}]$.

// Verification

1. Sim simulates the *tag check* phase from Π_{BatchRec} (Figure 1) to extract the $\mathbf{X}$ and $\mathbf{Y}$ shares of the corrupt parties. In step 1, the parties invoke $\mathcal{F}_{\mathsf{Rand}}$ to generate a random value. In step 3, Sim commits to random values on behalf of honest parties and broadcasts it. Upon receiving the commitments from the corrupt parties, the simulator uses the knowledge of $\mathbf{A}$, $\mathbf{B}$ and Δ to extract the $[\mathbf{X}]$, $[\mathbf{Y}]$, $[\mathsf{tag}_{\mathbf{X}}]$ and $[\mathsf{tag}_{\mathbf{Y}}]$ shares of the corrupt parties. Sim proceeds with the simulation where it passes the checks in steps 4 and 5 of the verification phase in Figure 1 by programming the commitments to open to simulated values (computed using the knowledge of Δ) which pass the checks.
2. All parties sample a public random matrix $\hat{\chi} \in \mathbb{Z}_{2^s}^{\mathsf{T}_1\times 1}$ via $\mathcal{F}_{\mathsf{Rand}}$.
3. Sim performs nothing.
4. Sim performs nothing.
5. Sim commits to random values on behalf of the honest parties. Sim extracts the corrupt parties' shares of $[\mathbf{Z}]$ using the knowledge of $[\mathbf{X}]$ and $[\mathbf{Y}]$ of the corrupt parties. Upon receiving the corrupt parties' commitments, Sim extracts the committed corrupt $[\mathsf{cs}]$ values. Sim computes random $[\mathsf{cs}]_i$ values for $i \in \mathcal{H}$ s.t. $\sum_{i\in\mathcal{H}}[\mathsf{cs}]_i + \sum_{j\in\mathcal{C}}[\mathsf{cs}]_j == 0 \mod 2^{k+2s}$. Sim programs the commitments s.t. the simulated honest party P_i decommits to $[\mathsf{cs}]_i$.
6. Sim extracts $[\mathsf{tag}_{\mathbf{Z}}]_j$ of the corrupt parties using the knowledge of $[\mathbf{Z}]_j$ and $[\Delta]_j$. Sim invokes $\mathcal{F}_{\mathsf{MatMul}}$ with corrupt parties' inputs $([\mathbf{Z}]_j, [\mathsf{tag}_{\mathbf{Z}}]_j)$. If $\mathcal{F}_{\mathsf{MatMul}}$ aborts, then Sim also aborts, else it continues.
7. Sim ends the simulation.

The adversary distinguishes between the real and ideal world if it breaks the consistency checks in Step 1 and Step 5 of the verification phase, where the tags of $\mathbf{X}$ (via verification of tag of $\mathbf{E}$), $\mathbf{Y}$ (via verification of tag of $\mathbf{U}$), and $\mathbf{Z}$ (via verification of tag of $\mathbf{D}$) are verified. However, this occurs with negligible probability, as we discuss next.

Correctness of consistency checks: In this section, we will discuss the correctness of $[\mathsf{tag}_{\mathbf{Z}^{\mathsf{f}}}]$ obtained from procedure $\Pi_{\mathsf{OptMAC\&Trunc}}$. In $\Pi_{\mathsf{CompactMatMul}}$, an optimistic tag is generated by invoking $\Pi_{\mathsf{OptMAC\&Trunc}}$. In $\Pi_{\mathsf{OptMAC\&Trunc}}$, $[\mathsf{tag}_{\mathbf{Z}^{\mathsf{f}}}]$ is computed using revealed matrix $\mathbf{D}$ and local share of $[\mathsf{tag}_{\mathbf{R}}]$ and $[\Delta]$. Both $[\mathsf{tag}_{\mathbf{R}}]$ and $[\Delta]$ are obtained and verified during the offline phase, so active adversaries can only introduce errors to the honest parties' $\mathbf{D}$. Thus, to verify the correctness of $[\mathsf{tag}_{\mathbf{Z}^{\mathsf{f}}}]$, MPC parties only need to check the correctness of $\mathbf{D}$. In $\Pi_{\mathsf{CompactMatMul}}$, during the verification stage, MPC parties will compute and broadcast the checksum $[\mathsf{cs}]$. This checksum is computed using committed operands ($[\Delta]$, $\hat{\chi}$, and $[\mathsf{tag}_{(\mathbf{D})^{\mathsf{c}}}]$) and a broadcasted matrix $\mathbf{D}$. MPC parties will abort if the broadcasted checksum is not zero modulo 2^{k+2s}. The broadcasted checksum can be parsed as:

$$\mathsf{cs} = \sum_{i=1}^{\mathsf{T}_1} \Delta \cdot \hat{\chi}_i \cdot (\tilde{\mathbf{D}})_i^{\mathsf{c}} - \hat{\chi}_i \cdot \mathsf{tag}_{(\mathbf{D})_i^{\mathsf{c}}} \tag{6}$$

$$= \sum_{i=1}^{\mathsf{T}_1} \hat{\chi}_i \cdot (\Delta \cdot (\tilde{\mathbf{D}})_i^{\mathsf{c}} - \mathsf{tag}_{(\mathbf{D})_i^{\mathsf{c}}}) \tag{7}$$

$$= \sum_{i=1}^{\mathsf{T}_1} \hat{\chi}_i \cdot (\Delta \cdot \sum_{j=1}^{\mathsf{T}_3} \chi_j \cdot \tilde{\mathbf{D}}_{i,j} - \sum_{j=1}^{\mathsf{T}_3} \chi_j \cdot \mathsf{tag}_{\mathbf{D}_{i,j}}) \tag{8}$$

$$= \sum_{i=1}^{\mathsf{T}_1} \hat{\chi}_i \cdot \sum_{j=1}^{\mathsf{T}_3} (\Delta \cdot \chi_j \cdot \tilde{\mathbf{D}}_{i,j} - \chi_j \cdot \mathsf{tag}_{\mathbf{D}_{i,j}}) \tag{9}$$

After adversaries add errors to the matrix **D**, the checksum is rewritten as

$$\sum_{i=1}^{\mathsf{T}_1} \hat{\chi}_i \cdot \sum_{j=1}^{\mathsf{T}_3} (\Delta \cdot \chi_j \cdot (\tilde{\mathbf{D}}_{i,j} + e_{i,j}) - \chi_j \cdot \mathsf{tag}_{\mathbf{D}_{i,j}}) \tag{10}$$

$$= \sum_{i=1}^{\mathsf{T}_1} \hat{\chi}_i \cdot \sum_{j=1}^{\mathsf{T}_3} \Delta \cdot \chi_j \cdot e_{i,j} + (\Delta \cdot \chi_j \cdot \tilde{\mathbf{D}}_{i,j} - \chi_j \cdot \mathsf{tag}_{\mathbf{D}_{i,j}}) \tag{11}$$

$$= \sum_{i=1}^{\mathsf{T}_1} \hat{\chi}_i \cdot \sum_{j=1}^{\mathsf{T}_3} \Delta \cdot e_{i,j} \cdot \chi_j \tag{12}$$

where $e_{i,j}$ is the error added to the $\mathbf{D}_{i,j}$. With the simplified checksum, we will show the probability of adversaries introducing errors to **D** while the checksum equals 0 mod 2^{k+2s}.

Theorem 1. *Suppose that $\chi_1, \chi_2, ..., \chi_{\mathsf{T}_1}$ and $\hat{\chi}_1, \hat{\chi}_2, ..., \hat{\chi}_{\mathsf{T}_3}$ are sampled randomly from $\mathbb{Z}_{2^s}$. Adversaries can choose **any** combination of $e_{i,j}$ where not all $e_{i,j}$ are zero modulo 2^k, and $\sum_{j=1}^{\mathsf{T}_3} \Delta \cdot e_{i,j} \cdot \chi_j$ is in modulo of 2^{k+s}, such that*

$$\Pr[\sum_{i=1}^{\mathsf{T}_1} \hat{\chi}_i \cdot \sum_{j=1}^{\mathsf{T}_3} \Delta \cdot e_{i,j} \cdot \chi_j =_{k+2s} 0] \leq 2^{-s+log(s+1)} \tag{13}$$

Two lemmas are used to prove this theorem.

Lemma 1. *Suppose that $\hat{\chi}_1, \hat{\chi}_2, ..., \hat{\chi}_{\mathsf{T}_1}$ are sampled randomly from $\mathbb{Z}_{2^s}$. Adversaries can choose **any** combination of $y_1, y_2, ..., y_{\mathsf{T}_1}$ from $\mathbb{Z}_{2^{k+s}}$, where not all y_is are zero modulo 2^{k+s}. Then,*

$$\Pr[\sum_{i=1}^{\mathsf{T}_1} \hat{\chi}_i \cdot y_i =_{k+2s} 0] \leq 2^{-s} \tag{14}$$

Proof. Let's assume 2^v is the largest power of two dividing y_i. We know that $y_i < 2^{k+s}$ by assumption. Thus, we have $v < k+s$. Therefore,

$$\Pr[\sum_{i=1}^{\mathsf{T}_1} \hat{\chi}_i \cdot y_i =_{k+2s} 0] \tag{15}$$

$$= \Pr[\hat{\chi}_1 \cdot y_1 =_{k+2s} \sum_{i=2}^{\mathsf{T}_1} \hat{\chi}_i \cdot y_i] \tag{16}$$

$$= \Pr[\hat{\chi}_1 \cdot \frac{y_1}{2^v} =_s \frac{\sum_{i=2}^{\mathsf{T}_1} \hat{\chi}_i \cdot y_i}{2^v}] \tag{17}$$

$$= \Pr[\hat{\chi}_1 =_s \frac{\sum_{i=2}^{\mathsf{T}_1} \hat{\chi}_i \cdot y_i}{2^v} \cdot (\frac{y_1}{2^v})^{-1}] \tag{18}$$

$$\leq 2^{-s} \tag{19}$$

□

Lemma 2. *Suppose that $\chi_1, \chi_2, ..., \chi_{\mathsf{T}_3}$ are sampled randomly from $\mathbb{Z}_{2^s}$, and Δ are uniformly sampled from $\mathbb{Z}_{2^s}$. Adversaries can choose **any** combination of $e_1, e_2, ..., e_{\mathsf{T}_3}$ from $\mathbb{Z}_{2^k}$, where not all e_is are zero modulo 2^k. Then,*

$$\Pr[\sum_{i=1}^{\mathsf{T}_3} \Delta \cdot e_i \cdot \chi_i =_{k+s} y] \leq 2^{-s+log(s+1)} \tag{20}$$

where y can be any value in $\mathbb{Z}_{k+s}$.

Lemma 2 are proven in [12]. With Lemma 1 and Lemma 2, we will prove Theorem 1. Let's denote $\sum_{j=1}^{\mathsf{T}_3} \Delta \cdot e_{i,j} \cdot \chi_j$ as G_i. Then, the probability in Theorem 1 can be re-written as:

$$\Pr[\sum_{i=1}^{\mathsf{T}_1} \hat{\chi}_i \cdot G_i =_{k+2s} 0] \tag{21}$$

$$= \sum_{\forall \{y_i\}_{i=1}^{\mathsf{T}_1}} \Pr[\sum_{i=1}^{\mathsf{T}_1} \hat{\chi}_i \cdot G_i =_{k+2s} 0 | \cap_{i=1}^{\mathsf{T}_1} G_i =_{k+s} y_i]$$
$$\cdot \Pr[\cap_{i=1}^{\mathsf{T}_1} G_i =_{k+s} y_i] \tag{22}$$

$$= \sum_{\forall \{y_i\}_{i=1}^{\mathsf{T}_1}} \Pr[\sum_{i=1}^{\mathsf{T}_1} \hat{\chi}_i \cdot y_i =_{k+2s} 0] \cdot \Pr[\cap_{i=1}^{\mathsf{T}_1} G_i =_{k+s} y_i] \tag{23}$$

Equation 22 holds due to total probability. Equation 23 holds because of the definition of the conditional probability. Now, adversaries can choose from two attack schemes: 1) make all y_is zeros, and 2) not all y_is are zero.

In the first scenario, Equation 23 can be written as:

$$\Pr[\sum_{i=1}^{\mathsf{T}_1} \hat{\chi}_i \cdot 0 =_{k+2s} 0] \cdot \Pr[\cap_{i=1}^{\mathsf{T}_1} G_i =_{k+s} 0] \tag{24}$$

$$= 1 \cdot \Pr[\cap_{i=1}^{\mathsf{T}_1} G_i =_{k+s} 0] \tag{25}$$

$$\leq min\{\Pr[G_i =_{k+s} 0]\}_{i=1}^{\mathsf{T}_1} \tag{26}$$

$$\leq min\{\Pr[\sum_{j=1}^{\mathsf{T}_3} \Delta \cdot e_{i,j} \cdot \chi_j =_{k+s} 0]\}_{i=1}^{\mathsf{T}_1} \tag{27}$$

$$\leq 2^{-s+log(s+1)} \tag{28}$$

Inequality 26 holds because the probability of the intersection of all events is less or equal to the minimum probability of all intersected events. Inequality 27 holds due to the definition of G_i. Inequality 28 holds due to Lemma 2.

In the second scenario, Equation 23 can be written as:

$$\sum_{\forall\{y_i\}_{i=1}^{\mathsf{T}_1}\neq\{0\}_{i=1}^{\mathsf{T}_1}} \Pr[\sum_{i=1}^{\mathsf{T}_1}\hat{\chi}_i\cdot y_i =_{k+2s} 0]\cdot\Pr[\cap_{i=1}^{\mathsf{T}_1} G_i =_{k+s} y_i] \tag{29}$$

$$\leq \sum_{\forall\{y_i\}_{i=1}^{\mathsf{T}_1}\neq\{0\}_{i=1}^{\mathsf{T}_1}} 2^{-s}\cdot\Pr[\cap_{i=1}^{\mathsf{T}_1} G_i =_{k+s} y_i] \tag{30}$$

$$\leq 2^{-s}\cdot \sum_{\forall\{y_i\}_{i=1}^{\mathsf{T}_1}\neq\{0\}_{i=1}^{\mathsf{T}_1}} \Pr[\cap_{i=1}^{\mathsf{T}_1} G_i =_{k+s} y_i] \tag{31}$$

$$\leq 2^{-s} \tag{32}$$

Inequality 30 holds due to Lemma 1. Inequality 32 holds because the summation of the probability of mutually exclusive events will be less or equal to one. Comparing scenario 1 and scenario 2, adversaries will have a better probability in the first scenario. Consequently,

$$\Pr[\sum_{i=1}^{\mathsf{T}_1}\hat{\chi}_i\cdot\sum_{j=1}^{\mathsf{T}_3}\Delta\cdot e_{i,j}\cdot\chi_j =_{k+2s} 0]\leq 2^{-s+log(s+1)} \tag{33}$$

C CompactTag scales with parties

This section shows the CompactTag's speedup over SPD$\mathbb{Z}_{2^k}$ with more number of parties for VGG16, ResNet, and Transformer. The speedup for tag computation for all models stays the same as we scale to more parties.

		2PC	3PC	4PC	5PC
Inference	Total	1.29×	1.25×	1.22×	1.20×
	Tag	**4.37**×	**4.37**×	**4.37**×	**4.37**×
Training	Total	1.47×	1.42×	1.38×	1.35×
	Tag	**17.57**×	**17.57**×	**17.57**×	**17.57**×

Table 5: CompactTag's speedup for VGG16 over SPD$\mathbb{Z}_{2^k}$ with varying number of parties."Tag" denote CompactTag's speedup on tag computation for linear layers.

		2PC	3PC	4PC	5PC
Inference	Total	1.30×	1.21×	1.16×	1.13×
	Tag	**23.11**×	**23.11**×	**23.11**×	**23.11**×
Training	Total	1.43×	1.35×	1.29×	1.25×
	Tag	**22.04**×	**22.04**×	**22.04**×	**22.04**×

Table 6: CompactTag's speedup for Transformer over SPD$\mathbb{Z}_{2^k}$ with varying number of parties."Tag" denote CompactTag's speedup on tag computation for linear layers.

		2PC	3PC	4PC	5PC
Inference	Total	1.20×	1.15×	1.12×	1.10×
	Tag	**3.44**×	**3.44**×	**3.44**×	**3.44**×
Training	Total	1.25×	1.20×	1.16×	1.14×
	Tag	**4.50**×	**4.50**×	**4.50**×	**4.50**×

Table 7: CompactTag's speedup for ResNet over SPD$\mathbb{Z}_{2^k}$ with varying number of parties."Tag" denote CompactTag's speedup on tag computation for linear layers.