SPRINT: Scalable Secure & Differentially Private Inference for Transformers

Francesco Capano
SAP SE
Karlsruhe, Germany
francesco.capano@sap.com

Jonas Böhler
SAP SE
Karlsruhe, Germany
jonas.boehler@sap.com

Benjamin Weggenmann*
Technische Hochschule
Würzburg-Schweinfurt, Germany
benjamin.weggenmann@thws.de

Abstract

Machine learning as a service (MLaaS) enables scalable model deployment and inference on cloud servers. However, MLaaS exposes user queries and model parameters to servers. To guarantee confidentiality of queries and model parameters, multi-party computation (MPC) enables secure inference by distributing data and computations across multiple service providers. MPC eliminates single points of failure, mitigates provider breaches and ensures confidentiality beyond legal agreements. Beyond confidentiality of queries and parameters, the model itself can memorize and leak training data during inference. To mitigate privacy concerns, differential privacy (DP) provides a formal privacy guarantee for training data, which can be satisfied by injecting carefully calibrated noise into gradients during training. However, naive combinations of DP and MPC amplify accuracy loss due to DP noise and MPC approximations, and incur high computational and communication overhead due to cryptographic operations.

We present SPRINT, the first scalable solution for efficient MPC inference on DP fine-tuned models with high accuracy. SPRINT fine-tunes public pre-trained models on private data using DP. It integrates DP-specific optimizations, e.g., parameter-efficient fine-tuning and noise-aware optimizers, with MPC optimizations, e.g., cleartext public parameters and efficient approximations of non-linear functions. We evaluate SPRINT on GLUE benchmark with RoBERTa, achieving up to 1.6× faster MPC inference than the state-of-the-art non-DP solution SHAFT, reducing communication by 1.6×. Notably, SPRINT maintains high accuracy during MPC inference, with < 1 percentage point gap compared to cleartext accuracy.

Keywords

Differential privacy, Multiparty computation, secure inference, transformers fine-tuning

1 Introduction

LLMs can be pretrained on large-scale public datasets and fine-tuned on smaller, private datasets to improve task-specific performance, e.g., sentiment analysis. Model owners can deploy fine-tuned models via *machine learning as a service* (MLaaS), outsourcing inference to cloud servers. MLaaS relieves model owners of resource management and infrastructural burdens of running inference themselves while providing users seamless access to predictions. However, model parameters, user queries, and predictions are exposed to MLaaS servers [39], potentially leaking sensitive information. Even servers adhering to legal agreements remain vulnerable to data breaches. To guarantee confidentiality of model parameters, user queries and predictions against cloud servers, *multi-party computation* (MPC) enables secure inference. MPC assumes multiple non-colluding servers that jointly compute a function on secret-shared data, i.e, random split of data, where each share reveals nothing on its own. By distributing trust among multiple servers from different providers, MPC eliminates single points of failure, mitigating risks such as provider breaches and providing cryptographic protections beyond contractual and legal agreements. Besides confidentiality, LLMs face privacy issues since they can memorize fine-tuning data and leak sensitive information in predictions [9, 29, 40, 46]. To mitigate memorization, *differential privacy* (DP) provides a formal privacy guarantee for fine-tuning data satisfied by injecting carefully calibrated noise into gradients [16, 41]. Specifically, DP bounds the maximum difference of the model's behavior when including or excluding a record from fine-tuning data.

While offering strong guarantees, DP and MPC introduce practical challenges when deploying transformer models, i.e., trade-offs between privacy, accuracy, and computational efficiency. Combining DP fine-tuning and MPC inference introduces two error sources that degrade final accuracy. (I) DP injects noise during fine-tuning, which scales with the model size and can hinder convergence [41]. (II) MPC relies on fixed-point arithmetic [36] for efficiency and approximates costly non-linear functions, e.g., GeLU, which can lead to numerical errors. Balancing these two error sources while optimizing performance is crucial to ensure scalability and accuracy of MPC inference, as larger models and datasets have wider parameter ranges and require more precise approximations trading accuracy for performance. Additionally, MPC inference introduces significant performance overhead as computation on secret shares requires interaction, e.g., servers exchanging shares for matrix multiplications. This is especially challenging for large transformer models (e.g., BERT with 110M parameters) as communication overhead grows with model size. Consequently, naively combining DP fine-tuning and MPC inference is impractical due to DP's noise-induced accuracy loss combined with MPC's cryptographic overhead and approximation errors. To overcome these challenges, we propose SPRINT, the first scalable approach co-designing DP fine-tuning and efficient MPC inference for high accuracy. SPRINT enables private and secure inference by ensuring privacy for fine-tuning data (via DP) and confidentiality for model parameters, user queries, and predictions during inference (via MPC). Unlike prior works focusing solely on DP [41, 70] or MPC [54], SPRINT integrates both

*Work done while he was at SAP SE.

Proceedings on Privacy Enhancing Technologies 2026(1), 134–153

https://doi.org/10.56553/popets-2026-0008

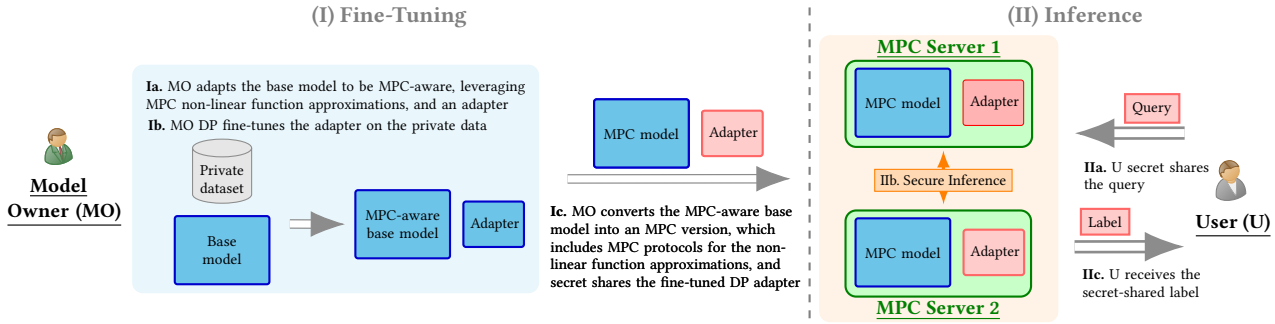

Figure 1: High level SPRINT overview: during fine-tuning (I) the model owner (*MO*) DP fine-tunes a large base model adapted to be MPC-aware, on a private dataset using a small adapter. Before inference (Ic), the *MO* converts the fine-tuned model into an MPC protocol and secret shares only the small adapter. During inference (II), the user secret shares its query to MPC servers which run secure inference, returning the secret-shared labels. Red blocks are secret-shared, while blue blocks are in cleartext.

techniques. Fig. 1 outlines SPRINT's approach. Initially (Ia), model owner (MO) adapts a large public base model to be MPC-aware (i.e., integrating suitable non-linear function approximations used in MPC inference), and adds a small adapter. Then, only the small adapter is DP fine-tuned on private data, while the large base model remains frozen (Ib). For secure inference (Ic), MO converts the fine-tuned model to an MPC version (i.e., translating all model functions to MPC protocols) and secret-shares only the fine-tuned adapter. Consequently, during inference (II), user (U) secret-shares its query (IIa) and each MPC server performs the bulk of large matrix operations locally on the cleartext base model, with no interaction. Only the small adapter matrix multiplications and activation functions in the model uses secure protocols (IIb). Finally, U reconstructs the secret-shared prediction (IIc). We evaluate SPRINT by fine-tuning RoBERTa [44] on the GLUE benchmark [66], achieving high MPC inference accuracy, i.e., within 1 percentage point (pp) of cleartext accuracy. Notably, SPRINT maintains high accuracy while scaling to large datasets (e.g., QQP with more than 40k validation samples), while many prior MPC works [31, 45, 54] tested only on smaller datasets (e.g., SST2 with < 1 k validation samples). Additionally, SPRINT delivers up to 1.6× faster MPC inference than the state-of-the-art non-DP solution SHAFT [31], reducing communication by 1.6×. We evaluate SPRINT for practical deployments using Poisson subsampling for rigorous privacy guarantees, which some prior DP works [41, 70] neglected, impacting their privacy guarantees.

1.1 Contributions

We present SPRINT, a scalable solution for DP fine-tuning and MPC inference for transformer models. SPRINT integrates DP-specific and MPC-aware techniques, to improve accuracy, computation and communication efficiency. Our contributions include:

1. Unified solution for secure and private MPC inference on DP fine-tuned transformer-based models. We present the first solution to integrate DP fine-tuning with MPC inference for transformer-based models, ensuring: (I) privacy for the fine-tuning dataset, (II) model confidentiality during inference, and (III) confidentiality for inference query and results. SPRINT bridges the gap between prior works which solely focus on either DP fine-tuning [7, 41, 70] or MPC inference [31, 45, 54]. We address computational overhead and numerical errors in MPC, and accuracy degradation due to DP noise (Sec. 3.1) to provide a scalable solution with high-accuracy for private and secure inference for transformer-based models (Sec. 4). While some of SPRINT's components build on existing work, their integration under the combined constraints of DP fine-tuning and MPC inference presents non-trivial challenges.

2. MPC-aware DP fine-tuning with high accuracy MPC inference. To achieve high accuracy during MPC inference, SPRINT balances DP noise with MPC approximation errors through: (I) parameter-efficient fine-tuning to mitigate the impact of DP noise, and (II) making DP fine-tuning MPC-aware by using the same non-linear function approximations as in inference. Specifically, (I) DP fine-tuning adds carefully calibrated noise to clipped gradients, with noise scaling with model size, hindering convergence for large models [41]. To mitigate noise impact, we use a variant of parameter-efficient fine-tuning, i.e., FALoRA [61], that reduces trainable parameters by $\approx 200\times$ (Sec. 4.1). While FALoRA is effective in non-DP settings and DP federated learning [61], it has not been previously applied to centralized DP fine-tuning. We find that centralized DP fine-tuning with FALoRA causes an accuracy drop of about 2 pp (App. A), which we mitigate using the DP-specific optimizer DPAdamBC [62] that accounts for noise-induced bias (Sec. 4.1). (II) Finally, our DP fine-tuning is made MPC-aware by using the same non-linear function approximations (e.g., GeLU) as in MPC inference. This ensures consistency in the non-linear functions between fine-tuning and inference improving final MPC accuracy (Sec. 5). However, integrating MPC approximations during fine-tuning introduces additional error sources. While previous works apply this approach only for MPC inference, we investigate the effect of combining it with DP fine-tuning and how to balance the two error sources. We find that while MPC inference-only works can use more aggressive approximations (as in distillation, App. D), our integrated approach requires more conservative approximations, trading off performance to achieve high accuracy.

3. Scalable and efficient secure inference via MPC. To improve efficiency and scalability in MPC inference, we introduce targeted optimizations: (I) for Linear layers, we minimize MPC matrix multiplications (i.e., which require interactions) by secret-sharing only fine-tuned parameters; and (II) for non-linear functions, we prioritize numerical stability and accuracy by adapting and improving state-of-the-art approximations. Specifically, (I) Linear layers require high-dimensional matrix multiplications, introducing

significant overhead in MPC since each multiplication between secret-shared matrices requires interaction. While ML LoRA implementations merge low-rank matrices into pre-trained weights [28], SPRINT keeps them separated during inference and secret-shares only fine-tuned parameters. Despite increasing the total parameter count, this approach enhances scalability. Only secret-shared parameters require costly MPC multiplication with server interaction, whereas multiplication with cleartext, public weights can happen locally. With FALoRA, SPRINT keeps ≈ 99% of the parameters in cleartext (Sec. 4.2). Regarding non-linear functions (II), they can account for up to 75% of MPC inference runtime [24]. MPC relies on fixed-point representations for efficiency, introducing numerical errors, e.g., overflows, particularly for non-linear function approximations which trade off accuracy for efficiency. While existing MPC works [31, 45, 54] typically train/test on small datasets (e.g., SST2 <1k validation samples), SPRINT scales to large datasets (e.g., QQP >40k validation samples) with broader input and parameter ranges, which normally, increase numerical instability. To mitigate this, we propose specific enhancements: (I) for Softmax, we introduce SoftCap-based approximation that bounds maximum input values and eliminates expensive max operations; (II) for LayerNorm, we develop a piecewise approximation for inverse square root improving accuracy with minimal overhead; and (III) for GeLU, we evaluate multiple approximations (App. D), and select and adapt BOLT's GeLU [54] which offers the best accuracy-performance trade-off.

4. Comprehensive evaluation with various network settings. We evaluate SPRINT on GLUE benchmark datasets [66], i.e., SST2, MNLI, QQP and QNLI, in multiple network conditions, i.e., LAN and WANs, to measure MPC inference performance (Sec. 5). We compare with state-of-the-art DP fine-tuning [41, 70] and MPC inference [24, 31, 45, 54]. SPRINT improves MPC inference, achieving up to 1.6× faster MPC inference than the state-of-the-art non-DP solution SHAFT [31], reducing communication by 1.6× (Tab. 2, Sec. 5.2). Despite approximations of non-linear function required for MPC in fine-tuning, SPRINT maintains accuracy comparable to DP-LoRA fine-tuning [70] (Tab. 1, Sec. 5.1), and MPC inference accuracy within 1 pp of cleartext (Tab. 6, Sec. 5.2). We also compare and discuss different approximations for non-linear functions from related works, e.g., from SHAFT [31] and BOLT [54] (Tab. 3, Sec. 5.2), and we evaluate the advantage of GPU acceleration and batched inference (Tab. 5, Sec. 5.2). Additionally, we perform extensive hyperparameter search, including different initialization strategies (for FALoRA matrices) to guide hyperparameter choice for faster convergence with high accuracy (Tab. 10, Sec. 5.1), and evaluate the impact of MPC-aware DP fine-tuning on model accuracy (Sec. 5.2).

5. Practical and modular framework with DP deployment considerations. SPRINT is a modular framework for DP fine-tuning and MPC inference, which builds on top of existing libraries, i.e., Opacus [69] for DP fine-tuning and CrypTen [36] for MPC inference.[1] SPRINT is designed to be extendible and include additional models via PyTorch-like modeling files. While this work focuses on BERT-like models for sequence classification due to computational constraints, SPRINT can be easily adapted to other architectures and tasks, such as GPT-like models for text generation. By building on top of CrypTen, SPRINT supports integration of any non-linear function approximation (App. D), to assess how each approximation impacts performance. This modularity allows SPRINT to quickly adapt to new models and approximations. SPRINT integrates Poisson subsampling for DP fine-tuning, contrarily to a common practice in the literature of replacing Poisson subsampling with shuffling for computational efficiency [73], which degrades effective privacy guarantees by, e.g., over 4× for the MNLI dataset [3].

[1]Code available at: https://github.com/SAP/sprint

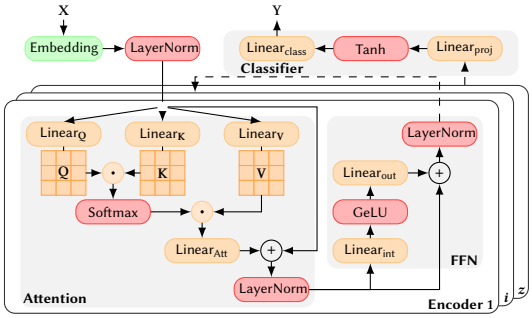

Figure 2: RoBERTa architecture with z encoders, highlighting non-linear layers and layers with matrix multiplications.

The rest of this work introduces preliminaries (Sec. 2), details SPRINT's design and challenges (Sec. 3), and presents our optimizations (Sec. 4). We then evaluate our approach (Sec. 5), review related work (Sec. 6) and discuss limitations and future work (Sec. 7).

2 Preliminaries

Next, we define our notation, overview the transformer architecture and fine-tuning, and introduce MPC and DP.

Notation. Upper case bold letters, e.g., $\mathbf{X} \in \mathbb{R}^{a\times b}$, denote a matrix of $a \times b$ real values. For indices (i, j), $\mathbf{X}_i = \mathbf{x} \in \mathbb{R}^b$ represents the i-th row vector, and $\mathbf{X}_{i,j} = \mathbf{x}_j = x \in \mathbb{R}$ is a scalar. An ML model $\mathcal{M}$ is defined by its parameters θ and architecture $\mathcal{M}^A$, where $\mathcal{M}^A$ encapsulates the algorithmic description of how parameters θ and inputs $\mathbf{X}$ are processed to output classifier logits $\mathbf{Y}$. Model parameters θ include weights $\mathbf{W}$ and biases $\mathbf{b}$ for each layer in $\mathcal{M}^A$. Dataset D contains N samples $(\mathbf{X}_i, \mathbf{Y}_i)$, and batch $\mathcal{B}$ has size B.

2.1 Transformers

Transformers [65] are widely used in NLP tasks, e.g., text classification, question answering. We focus on RoBERTa, a variant of BERT [13], for sequence classification, as DP related works [41, 54, 70]. RoBERTa consists of a stack of $z = 12$ Encoder that process an input sequence and output hidden states. Each Encoder contains a multi-head self-attention mechanism (Attention) and a feed-forward neural network (FFN). The Classifier processes the last encoder's output to output logits for the labels. Next, we detail on the main components of the RoBERTa architecture in Fig. 2.

Embedding. The input text is tokenized and mapped to a continuous space via the Embedding layer implemented as lookup tables of size $N_V \times d$, where $N_V = 50625$ (vocabulary size) and $d = 768$ (model dimension). Token embeddings $\mathbf{X}^{\text{tok}} \in \mathbb{R}^{m\times d}$ are summed with positional embeddings $\mathbf{X}^{\text{pos}} \in \mathbb{R}^{m\times d}$ to encode position in the sequence ($m = 128$). Then, the output is normalized via LayerNorm.

LayerNorm. LayerNorm ensures numerical stability by normalizing across all the features in the hidden states (or input sequence).

For a row vector $\mathbf{x} \in \mathbb{R}^d$, weights $\mathbf{W}$ and bias $\mathbf{b}$:

$$\text{LayerNorm}(\mathbf{x}) = \frac{\mathbf{x} - \mu(\mathbf{x})}{\sqrt{\nu(\mathbf{x})}}\mathbf{W} + \mathbf{b}, \tag{1}$$

where $\mu(\mathbf{x})$ and $\nu(\mathbf{x})$ are the mean and variance of the input.

Attention. Attention captures the relationships between words in a sequence by weighting their importance. Each input is projected into *query* ($\mathbf{Q} \in \mathbb{R}^{m\times d'}$), *key* ($\mathbf{K} \in \mathbb{R}^{m\times d'}$), and *value* ($\mathbf{V} \in \mathbb{R}^{m\times d'}$) matrices. Here $d' = d/H$, where $H = 12$ is the number of attentions heads. Each head processes a distinct subset of $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ in parallel. The intermediate attention scores are computed as:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}(\frac{\mathbf{Q} \cdot \mathbf{K}^T}{\sqrt{d'}}) \cdot \mathbf{V},$$

where Softmax is applied row-wise, i.e., for each row $\mathbf{x} \in \mathbb{R}^m$:

$$\text{Softmax}(\mathbf{x}) = \frac{e^{\mathbf{x}}}{\sum_{j=1}^{m} e^{\mathbf{x}_j}}. \tag{2}$$

Outputs from all heads are concatenated, passed through a linear layer ($\text{Linear}_{\text{Att}}$), and normalized.

FFN. The FFN applies two Linear layers with a GeLU activation to the Attention output $\mathbf{X} \in \mathbb{R}^{m\times d}$. GeLU introduces non-linearity and is applied element-wise to $x \in \mathbb{R}$:

$$\text{GeLU}(x) = \frac{1}{2}x(1 + \text{erf}(\frac{x}{\sqrt{2}})), \tag{3}$$

where $\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$ is the error function [27].

Classifier. For sequence classification, the first output row $\mathbf{X}_0^{cl} \in \mathbb{R}^d$ from the last encoder is processed by two linear layers ($\text{Linear}_{\text{proj}}$, $\text{Linear}_{\text{class}}$) with Tanh activation to output logits for l labels.

2.2 Fine-Tuning

LLMs are pre-trained on large amounts of public data to learn general language representations. However, pre-trained models may not perform well on specific tasks. Fine-tuning is a standard approach to adapt a large, pre-trained model (e.g., RoBERTa) to a downstream, specific task (e.g., sentiment analysis) using a smaller, task-specific dataset. This approach is particularly relevant in privacy-sensitive scenarios, where data is scarce, and the public model needs to be adapted to a specific task on private data.

Low-Rank Adaptation (LoRA). LoRA [28] is a parameter-efficient fine-tuning technique that reduces the number of trainable parameters and computational cost. LoRA decomposes the weight matrix $\mathbf{W}_0 \in \mathbb{R}^{d\times k}$ of a linear layer into two low-rank matrices $\mathbf{A} \in \mathbb{R}^{r\times k}$ and $\mathbf{B} \in \mathbb{R}^{d\times r}$, where r is the rank. For an input $\mathbf{X} \in \mathbb{R}^{m\times d}$, the output $\mathbf{Y} \in \mathbb{R}^{m\times k}$ of the linear layer is computed as follows:

$$\mathbf{Y} = \mathbf{X} \cdot \mathbf{W}_0 + \mathbf{X} \cdot (\mathbf{B} \cdot \mathbf{A})\frac{\alpha}{r}, \tag{4}$$

where α is a scaling factor. During fine-tuning only $\mathbf{A}$ and $\mathbf{B}$ are updated, where $r \ll \min(d, k)$ reduces trainable parameters to e.g., $\approx 1\%$ of RoBERTa parameter with $r = 16$.

2.3 Multi-Party Computation

Multi-Party Computation (MPC) is a cryptographic technique enabling multiple servers to jointly compute a function on private inputs while keeping the inputs confidential; only the output is revealed. We focus on the *2-out-of-2 secret sharing* scheme (SS) of CrypTen [36], where 2 servers S_1, S_2 perform the computation.[2] A *2-out-of-2* scheme splits a secret into 2 shares via Shr and reconstructs it from the shares via Rec. Shares are of two types: *arithmetic* suitable for operations like additions and multiplications (e.g., in linear layers), and *binary* suitable for comparisons (e.g., in GeLU) [36].

Arithmetic Secret Sharing. Arithmetic secret sharing maps a scalar value x to shares $\text{Shr}(x) = (\langle x\rangle_1^A, \langle x\rangle_2^A)$ such that the sum of all shares is x, i.e., $\text{Rec}(\langle x\rangle_1^A, \langle x\rangle_2^A) = \langle x\rangle_1^A + \langle x\rangle_2^A = x$. Notably, SPRINT relies on the fact that additions and multiplications with public values can be executed on local shares, whereas multiplication of shares requires interaction between the servers. Specifically, CrypTen leverages Beaver triples [5] requiring one round of interaction. Before Shr, floating-point values (x_{fl}) are converted to fixed-point (x_{fx}), where $x_{\text{fx}} = x_{\text{fl}} \cdot 2^f$ rounded to integers, and f is the precision. Fixed-point arithmetic uses integers, reducing communication overhead in MPC, whereas floating-point requires handling multiple shares (e.g., base and exponent) [19]. In SPRINT we set $f = 16$ and the shares are mapped into a 64-bit field, i.e., $\mathbb{Z}_{2^{64}}$ [36].

Binary Secret Sharing & Conversion. Binary secret sharing maps each bit x_b of x to shares $\text{Shr}(x_b) = (\langle x_b\rangle_1^B, \langle x_b\rangle_2^B)$ in a binary field $\mathbb{Z}_2$, such that $\text{Rec}(\langle x_b\rangle^B) = \langle x_b\rangle_1^B \oplus \langle x_b\rangle_2^B$ [36]. Transformers require arithmetic and binary SS for linear and non-linear operations, respectively. For non-linear operations, arithmetic shares are converted to binary shares (A2B), which after the computations, are converted back to arithmetic shares (B2A) [36]. In A2B, each S_i converts $\langle x\rangle_i^A$ to binary shares $\langle x\rangle_i^B$ via bit-wise decomposition. In B2A, $\langle x\rangle_i^A = \sum_{b=1}^{n} 2^b \langle x_b\rangle_i^B$, where n is the bit-width [36].

2.4 Differential Privacy

DP is a formal privacy guarantee that the presence or absence of a record in a dataset D does not significantly affect the outcome of any function over D. Formally, a randomized mechanism $\mathcal{M}$ satisfies (ϵ, δ)-DP if for any neighboring dataset D_1, D_2, for $\epsilon > 0, \delta \geq 0$, and for any subset $\mathcal{S}$ of possible outputs:

$$Pr[\mathcal{M}(D_1) \in \mathcal{S}] \leq \exp(\epsilon) \times Pr[\mathcal{M}(D_2) \in \mathcal{S}] + \delta$$

Here, ϵ is the privacy budget and bounds the output variation of $\mathcal{M}$ on D_1, D_2. Smaller ϵ indicates stronger privacy. Parameter δ represents the probability of breaking the privacy guarantee, typically set to $\delta \ll 1/N$, where N is the dataset size. The neighboring datasets D_1 and D_2 differ by a single record, i.e., D_2 can be obtained from D_1 by adding or removing a record [17]. Notably, DP is immune to *post-processing*, i.e., any data-independent transformation of the output of an (ϵ, δ)-DP mechanism does not reduce the privacy guarantees [17]. Furthermore, DP mechanisms are *composable*, enabling tracking of privacy loss across training iterations. The privacy budget accumulates as multiple DP operations are performed, requiring careful accounting to bound the overall privacy guarantee. For tighter accounting, we employ the Rényi Differential Privacy (RDP) [50] definition. RDP is parameterized by (α, ρ) where $\alpha > 1$ is the Rényi order and $\rho \geq 0$ is the privacy parameter. RDP can be converted to (ε, δ)-DP using $\varepsilon = \rho + \frac{\log((\alpha-1)/\alpha) - (\log\delta + \log\alpha)}{\alpha - 1}$ [4].

[2] All SS operation are performed in a finite field, i.e., modulo a prime number, and we omit the modulo for notational simplicity.

DP in Machine Learning. To achieve DP guarantees in ML training, stochastic gradient descent (SGD) is adapted into DP-SGD [1] through two steps: (I) per-example *gradient clipping* Clip, which bounds the sensitivity of the gradient $\mathbf{g}$ by clipping the l_2-norm to a fixed threshold C:

$$\text{Clip}_C(\mathbf{g}) = \mathbf{g} \cdot \min(1, C/||\mathbf{g}||_2) = \mathbf{g} \cdot \min(1, C/\sqrt{\sum_i g_i \in \mathbf{g} g_i^2}),$$

and (II) *noise addition*, where Gaussian noise $\Psi \sim \mathcal{N}(0, \sigma^2 C^2)$ (with σ derived from ϵ, δ via [69]) is added to clipped gradients:

$$\theta^{(t+1)} = \theta^{(t)} - \eta(\text{Clip}_C(\mathbf{g}) + \Psi).$$

The DP guarantee can be amplified by randomly *subsampling* a batch of size B for each iteration. Subsampling introduces uncertainty for an adversary to infer the presence of a specific record in the dataset. By applying Poisson subsampling (Alg. 6, App. C), each sample is included with a probability $q = B/N$, yielding a variable batch size that follows a Poisson distribution with mean B. Subsampling improves the privacy-accuracy trade-off amplifying the DP guarantees for a given ϵ to $(O(q(\exp(\epsilon) - 1)), O(q\delta))$-DP [1, 57].

DP for Transformers. Naively computing per-example gradient norms is memory-intensive and can be impractical for large models like RoBERTa. To enable batched gradient computation, we employ GhostClipping (Alg. 5), reducing memory and computational overhead [41]. GhostClipping indirectly estimates per-example gradient norms by computing the squared Frobenius norm and decomposing the batch-level operations into matrix multiplications, avoiding the need to compute or store individual gradients explicitly. GhostClipping achieves substantial memory savings (up to 22× for GPT-2) without affecting accuracy or privacy [41]. For more details, see Alg. 5 and App. B. Additionally, transformers training and fine-tuning often use adaptive optimizers, e.g., Adam [35], which maintain moving averages of the gradient's first and second moments, i.e., its mean m_t and variance v_t (Alg. 4), to enable adaptive learning rates and faster convergence. Adam is adapted into DP-Adam by adding noise to the gradients as in DP-SGD [70].

3 SPRINT Design

Next, we introduce our problem setting and threat model, followed by an overview of the challenges (Sec. 3.1) and high-level design of SPRINT, which integrates DP fine-tuning with MPC inference for LLMs (Sec. 3.2). Sec. 4 details the optimizations.

Problem Setting. In our scenario (Fig. 3) model owner *MO* seeks to deploy a secure inference service for an LLM fine-tuned on its sensitive data D_{FT}. Our goal is threefold: (I) to guarantee the privacy of D_{FT} from the user U of the service; (II) to ensure model confidentiality from the MLaaS servers (MPC servers S_1 and S_2); and (III) to protect the confidentiality of the user's query from the servers. To combine these privacy guarantees, *MO* DP-fine-tunes a public model, $\mathcal{M}_0^A$ with parameters θ_0, on D_{FT}. Then, *MO* secret-shares DP fine-tuned parameters θ_{DP} across S_1, S_2. Finally, U secret-shares the embeddings of its sample $\langle \mathbf{X}_U^{\text{emb}} \rangle$ to securely query the model and reconstruct the output $\langle \mathbf{Y}_U \rangle$.

Threat Model. We consider two *semi-honest*, non-colluding servers S_1, S_2, i.e., they follow the protocol but may try to infer private information from their view of the protocol execution.

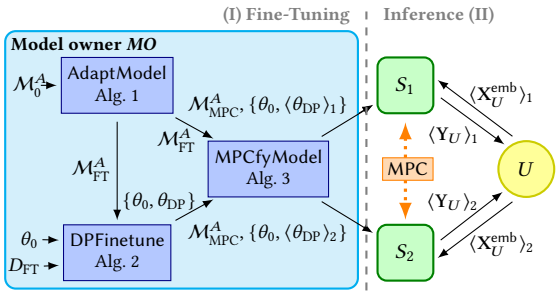

Figure 3: Overview of MPC-aware DP fine-tuning and MPC inference. Model owner *MO* prepares the model (AdaptModel) and fine-tunes it (DPFinetune). Before inference, *MO* converts the model to an MPC version (MPCfyModel) and secret-shares the DP fine-tuned parameters θ_{DP} to servers S_1, S_2. The user U secret-shares the embeddings of its data D_{INF} to the servers, which execute MPC inference and return the result.

The semi-honest assumption is standard in private inference literature [2, 15, 23, 38, 47, 54] as it balances security and performance [54], and is used in real-world applications. For instance, Firefox [18] privately collects telemetry data via two independent, semi-honest servers. Some protocols require precomputed cryptographic material, i.e., multiplication triples [5]. We follow [31, 36] and assume a trusted dealer provides necessary cryptographic materials during an offline preprocessing phase.

3.1 Design Challenges

Next, we outline the challenges arising from combining DP fine-tuning and MPC inference.

Accuracy degradation from DP noise and MPC limited precision. Key challenges when integrating DP fine-tuning and MPC inference for transformer-based models include: (I) mitigating accuracy degradation from both DP-induced noise and the limited precision inherent in MPC approximations; and (II) minimizing the accuracy gap between cleartext training and MPC inference while improving performance. DP noise degrades model accuracy, especially in large models like RoBERTa (125M parameters), as noise typically scales with number of parameters [41]. Furthermore, the limited precision of fixed-point arithmetic impacts accuracy, introducing approximation errors not only during the conversion of weights from floating-point to fixed-point but also in the approximations of non-linear functions (e.g., GeLU). Consequently, naively performing MPC inference on DP fine-tuned models can lead to accuracy degradation (Sec. 4.2). To address these issues, SPRINT integrates targeted strategies. First, to minimize the impact of DP noise on accuracy SPRINT employs parameter-efficient fine-tuning (PEFT) such as FALoRA (Sec. 4.1). Second, SPRINT implements an MPC-aware fine-tuning process (Sec. 3.2); this involves using the same non-linear function approximations during DP fine-tuning and MPC inference. This approach allows the model to learn and adapt to specific approximations, thereby reducing the accuracy gap between cleartext fine-tuning and MPC inference (Sec. 5). Furthermore, we carefully select non-linear approximations balancing their inherent approximation error and cryptographic overhead (Sec. 4.2).

Scalability in MPC Inference. Achieving scalable MPC inference for transformers presents two primary challenges: (I) managing the high computation and communication overhead from high-dimensional matrix multiplications in Linear layers; and (II) ensuring the numerical stability of MPC inference across diverse and wide input ranges. Transformer-based models can be very large and require high-dimensional matrix multiplications. For example, in RoBERTa, Linear layers in Attention have size 768×768. For non-optimized inference, a single sample in MPC requires 1.4 GB communication for Linear layers (Tab. 4, Sec. 5.2). Thus, optimizing Linear layers is crucial to reduce the communication overhead. Notably, MPC protocols for Linear layer can be optimized indirectly by keeping public, non-trainable parameters in cleartext, allowing local, i.e., interaction free, matrix multiplications between secret shared inputs and cleartext parameters (Sec. 4.2). Furthermore, scaling to larger datasets and models requires the MPC protocol, especially non-linear function approximations, to be numerically stable for a wider input range. Fixed-point arithmetic is prone to numerical errors, e.g., underflow/overflow, modulo wrap-around, and truncation errors. These errors are difficult to detect in multi-party settings as no single server has full visibility of intermediate computations [36]. Defining a stable and accurate approximation for a wide input range requires more communication and computation overhead, which can be a bottleneck in MPC inference. For example, the inverse square root for LayerNorm in CrypTen [36] is stable only for a limited input range, i.e., it causes overflow for input with absolute values larger than 330, and by getting closer to this threshold the mean squared error increases significantly (Sec. 4.2). While more stable approximations negatively impact performance, they also reduce the gap between the MPC and cleartext inference accuracy. We not only propose stable approximations for Softmax, LayerNorm, and GeLU [54], but also target measures to ensure numerical stability across the model, such as logits capping which bounds the input to the Softmax reducing the risk of overflow during MPC inference, and modifying the attention masking for padding to values which do not cause numerical issues (Sec. 4.2).

3.2 System Design

Our solution addresses the dual challenge of minimizing the accuracy loss due to DP noise and MPC approximations while enabling efficient MPC inference for transformer-based models. Fig. 3 shows SPRINT's system architecture in its two phases: (I) DP fine-tuning performed by *MO*, and (II) MPC inference involving S_1, S_2, and U. In the fine-tuning phase, we address the challenges of MPC inference and DP fine-tuning (Sec. 3.1) in three steps, summarized here and detailed in the following paragraphs. Specifically, we: (a) adapt the pre-trained model for MPC-aware fine-tuning (AdaptModel, Alg. 1), (b) perform DP fine-tuning on the adapted model (DPFinetune, Alg. 2), and (c) convert the fine-tuned model functions (i.e., matrix multiplications and non-linear functions) into MPC-compatible protocols for inference (MPCfyModel, Alg. 3).

AdaptModel for MPC-aware fine-tuning. The pretrained model $\mathcal{M}_0^A$ is not optimized for MPC inference. Specifically, it relies on non-linear functions, i.e., Softmax, GeLU, and LayerNorm, which lead to accuracy degradation and numerical issues. Furthermore, the high dimensionality of the model parameters makes DP fine-tuning and MPC inference computationally expensive. To address these challenges, AdaptModel (Alg. 1) adapts the pretrained model architecture $\mathcal{M}_0^A$ (Fig. 2) to be MPC-aware. Specifically, *MO* first applies LoRA to the Linear layers in the Encoder to reduce the number of trainable parameters, and minimize the impact of DP noise during fine-tuning (Sec. 4.1). The LoRA $\mathbf{A}$ matrices are initialized to maximize accuracy during fine-tuning (App. F.2). Second, *MO* applies SoftCap to limit the input range of Softmax, and the embeddings output, to prevent numerical issues in inference (Sec. 4.2). Third, *MO* replaces GeLU with a cleartext version of the MPC approximation GeLU_{MPC} [54] to be MPC-compatible and improve the MPC inference accuracy (Sec. 4.2). Finally, *MO* adds the Classifier to create the MPC-aware model for fine-tuning $\mathcal{M}_{\text{FT}}^A$.

Input: Public pretrained model $\mathcal{M}_0^A$, capping threshold K, LoRA params r, α.
Output: MPC-aware cleartext model $\mathcal{M}_{\text{FT}}^A$ for DPFinetune.
Function AdaptModel($\mathcal{M}_0^A, r, \alpha$, K):
  foreach Encoder $\in \mathcal{M}_0^A$ **do**
    /* Apply FALoRA */
    $\mathbf{A}, \mathbf{B} \leftarrow$ Initialize Lora Matrices // Sec. 5.1
    $\text{Linear}(x) \leftarrow \text{Linear}(x) + x \cdot (\mathbf{B} \cdot \mathbf{A}) \frac{\alpha}{r}$ // Sec. 4.1
    /* Apply MPC-Aware approximations Sec. 4.2 */
    $\text{Softmax}(x) \leftarrow \text{Softmax}(\text{SoftCap}(x, \text{K}))$ // Cap logits
    $\text{GeLU}(x) \leftarrow \text{GeLU}_{\text{MPC}}(x)$ // Replace GeLU Eq. (3)
  $\text{Embedding}(x) \leftarrow \text{SoftCap}(\text{Embedding}(x))$// Sec. 4.2
  $\mathcal{M}_{\text{FT}}^A \leftarrow \mathcal{M}_0^A$ with added Classifier // Add classifier
  return $\mathcal{M}_{FT}^A$

Algorithm 1: *MO* adapts cleartext model $\mathcal{M}_0^A$ for the MPC-aware DP fine-tuning.

Input: MPC-aware cleartext model $\mathcal{M}_{\text{FT}}^A$, with parameters θ_0, fine-tuning data D_{FT} of size N, batch size B, epochs E.
Output: Finetuned parameters θ_{DP}.
Function DPFinetune($\mathcal{M}_{FT}^A, \theta_0, D_{FT}, B, N, E$):
  /* Freeze Embedding, non-LoRA parameters in Attention, and LoRA $\mathbf{A}$ matrices (Sec. 4.1) */
  $\theta_{\text{DP}}^{(0)} \leftarrow \theta_0$ // Initialize trainable parameters
  for $t = 1, \ldots, E$ **do**
    /* Sample random batch with probability P */
    $\mathcal{B} \leftarrow \text{PoissonSampling}(P = B/N)$ // Alg. 6, App. C
    $\mathcal{L}(\theta_{\text{DP}}^{(t-1)}, \mathcal{B}) \leftarrow \text{Loss}(\mathcal{M}_{\text{FT}}^A(\theta_{\text{DP}}^{(t-1)}, \theta_0; \mathbf{X}_{\mathcal{B}}), \mathbf{Y}_{\mathcal{B}})$
    /* Compute per-example clipped gradients */
    $\bar{G} = \text{GhostClipping}(\mathcal{L}(\theta_{\text{DP}}^{(t-1)}, \mathcal{B}), \mathcal{B})$// Alg. 5, App. B
    /* Add noise and update model parameters */
    $\theta_{\text{DP}}^{(t)} = \text{DPAdamBC}(\bar{G}, t, \theta_{\text{DP}}^{(t-1)})$ // Alg. 4
  return $\theta_{\text{DP}}^{(t)}$

Algorithm 2: *MO* runs MPC-aware DP fine-tuning on cleartext model $\mathcal{M}_{\text{FT}}^A$ for a given loss function $\text{Loss}(\cdot)$.

DPFinetune. DP fine-tuning introduces noise into model updates, which can degrade model accuracy. Additionally, DP fine-tuning typically requires per-example clipping and noise sampling, which increases memory and computation costs. To address these challenges, DPFinetune (Alg. 2) minimizes the impact of DP noise via DP-specific techniques (Sec. 4.1), and optimizes fine-tuning performance with batched gradient clipping (Sec. 4.1). Specifically, before fine-tuning, *MO* freezes Embedding, non-LoRA parameters, and LoRA $\mathbf{A}$ matrices (θ_0 in Alg. 2), and trains from scratch LoRA $\mathbf{B}$ matrices as well as the two linear layers in the Classifier (with parameters θ_{DP}). For each fine-tuning iteration, *MO* first sub-samples a set of training examples $\mathcal{B}$ with probability $P = B/N$ via Poisson

Input: MPC-aware DP fine-tuned model $\mathcal{M}^A_{\text{FT}}$, with fine-tuned and frozen parameters θ_{DP} and θ_0, respectively.
Output: MPC model $\mathcal{M}^A_{\text{MPC}}$, shares of parameters $\langle\theta_{\text{DP}}\rangle_1, \langle\theta_{\text{DP}}\rangle_2$.
Function MPCfyModel($\mathcal{M}^A_{FT}, \theta_{\text{DP}}, \theta_0$):
  /* Apply MPC approximations from Sec. 4.2 */
  foreach Encoder $\in \mathcal{M}^A_{FT}$ **do** // For each Encoder
    /* Replace Softmax and add logits capping */
    $\text{Softmax}(x) \leftarrow \text{SoftmaxK}(\text{SoftCap}(x, \text{K}))$// Alg. 8-9, App. E
    $\text{GeLU}(x) \leftarrow \text{GeLU}_{\text{MPC}}(x)$ // Alg. 10, App. E
    $\text{LayerNorm}(x) \leftarrow \text{LayerNorm}_{\text{MPC}}(x)$ // Alg. 11, App. E
  $\mathcal{M}^A_{\text{MPC}} \leftarrow$ Replace cleartext functions with MPC protocols in model $\mathcal{M}^A_{\text{FT}}$ with Embedding layer removed
  $\langle\theta_{\text{DP}}\rangle_1, \langle\theta_{\text{DP}}\rangle_2 \leftarrow \text{Shr}(\theta_{\text{DP}})$
  return $\mathcal{M}^A_{MPC}, \langle\theta_{\text{DP}}\rangle_1, \langle\theta_{\text{DP}}\rangle_2$

Algorithm 3: *MO* transforms the cleartext functions $\mathcal{M}^A_{\text{FT}}$ to MPC protocols $\mathcal{M}^A_{\text{MPC}}$, and shares parameters θ_{DP}.

subsampling (PoissonSampling in Alg. 6, App. C). Second, *MO* computes gradients and clips per-example gradients via GhostClipping (Alg. 5, App. B). GhostClipping enables batched computation of per-example gradient norms, reducing the overhead of DP fine-tuning computing the Frobenius norm $|\cdot|_F$ from activations and gradients. Third, *MO* updates the model parameters θ_{DP} using the DP-noise aware optimizer DPAdamBC (Alg. 4, Sec. 4.1), which corrects the gradient updates to account for the DP noise.

MPCfyModel for MPC Inference. To enable efficient MPC inference, MPCfyModel (Alg. 3) replaces cleartext functions with optimized MPC protocols (Sec. 4.2) to balance accuracy and cryptographic overhead. Specifically, in MPCfyModel, *MO* first applies SoftCap to the Softmax input, and replaces non-linear layers Softmax, GeLU, and LayerNorm with MPC approximations SoftmaxK, GeLU_{MPC}, and $\text{LayerNorm}_{\text{MPC}}$ (Sec. 4.2). Second, *MO* builds an MPC model $\mathcal{M}^A_{\text{MPC}}$ from clear model $\mathcal{M}^A_{\text{FT}}$, replacing cleartext functions with MPC protocols. *MO* also removes the Embedding layer, since they are computed locally by U to reduce MPC overhead. Third, *MO* secret-shares the fine-tuned parameters θ_{DP} between servers S_1, S_2, ensuring θ_{DP} confidentiality.

MPC Inference. $\mathcal{M}^A_{\text{MPC}}$ defines the MPC inference protocol, i.e., how servers S_1, S_2 interact to securely process the secret-shared parameters and user input to compute the model output. During inference (Alg. 3), user U locally computes the embeddings of their private input $\mathbf{X}_U$ via the publicly available and unchanged Embedding layer. U secret-shares the computed embedding with the servers to query the model. The servers perform MPC inference applying $\mathcal{M}^A_{\text{MPC}}$ on $\langle\theta_{\text{DP}}\rangle$ and θ_0, and return the secret-shared result to U.

Next, we formalize our optimizations (Sec. 4), and present a comprehensive evaluation of SPRINT (Sec. 5).

4 SPRINT Optimizations

Naively applying DP fine-tuning and MPC inference to transformer-based models results in significant accuracy and performance degradation. DP fine-tuning suffers due to DP noise. MPC inference incurs accuracy loss and high computational and communication costs due to non-linear function approximations (e.g., GeLU, Softmax), numerical errors from fixed-point arithmetic, and large secret shared matrix multiplications. To reduce performance overhead and preserve high accuracy we integrate MPC, DP, and ML techniques.

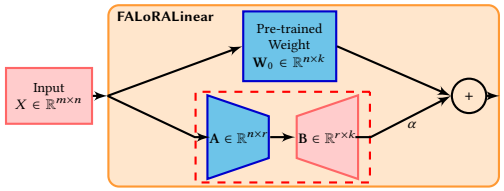

Figure 4: FALoRA in Linear layer. **Blue blocks are frozen during fine-tuning ($\{\mathbf{W}_0, \mathbf{A}\} \in \theta_0$, in clear during inference), while red blocks are secret-shared during inference ($\mathbf{B} \in \theta_{\text{DP}}$).**

4.1 MPC-aware DP Fine-tuning Optimizations

Next, we detail two optimizations for DP fine-tuning: FALoRA to reduce trainable parameters and DP-Adam Bias-Correction to improve optimizer performance under DP.

Reducing trainable parameters with FALoRA. To reduce the number of trainable parameters, improve performance during MPC inference (Sec. 4.2) and reduce the impact of DP noise during fine-tuning, we freeze the Embedding layer and perform parameter-efficient fine-tuning with FALoRA which freezes LoRA's $\mathbf{A}$ matrix. Previous works in non-DP context [72] and DP federated learning [61] have shown that freezing the $\mathbf{A}$ matrix (thus further reducing the number of trainable parameters compared to LoRA) improves fine-tuning performance while maintaining high accuracy. From a theoretical perspective, the DP noise Ψ_A, Ψ_B in LoRA updates the weights as follows (omitting gradient notation):

$$\mathbf{W}_0 + (\mathbf{B} + \Psi_B)(\mathbf{A} + \Psi_A) = \mathbf{W}_0 + \mathbf{BA} + \Psi_B\mathbf{A} + \mathbf{B}\Psi_A + \Psi_B\Psi_A.$$

Here, the quadratic noise term $\Psi_B\Psi_A$ no longer follows a Gaussian distribution, which can degrade model performance. By freezing $\mathbf{A}$, the quadratic noise term is removed obtaining the desired per-layer noise variance from Ψ_B [61]. While FALoRA offers theoretical advantages in reducing the noise amplification, and improves the accuracy in previous works, our preliminary experiments revealed that FALoRA alone performs worse than state-of-the-art LoRA [70] (Tab. 7, App. A). This could be due to the MPC-aware fine-tuning, where the model may struggle to adapt to the non-linear approximations with such a small number of trainable parameters. However, FALoRA is still beneficial for reducing the number of trainable parameters and improving performance during MPC inference (Sec. 4.2). To overcome this accuracy drop, we propose to combine FALoRA with a DP-specific optimizer, DPAdamBC [62], which corrects the bias introduced in the second moment v_t of Adam, and improves convergence, which is discussed below in detail. Notably, since $\mathbf{A}$ is not updated, its initialization directly affects accuracy; we therefore evaluate several initialization techniques in App. F.2.

DP-Adam Bias-Correction. Naively adapting Adam to DP by adding noise to the gradients leads to a biased second moment v_t due to the variance of the DP Gaussian noise σ^2C^2 [62]. This bias makes DP-Adam behave more like DP-SGD (Sec. 2.4), losing convergence benefits while consuming 3× the memory of DP-SGD [62]. DPAdamBC [62] corrects the bias in v_t by a factor σ^2C^2/B^2, retaining convergence advantages of Adam [62]. The first moment m_t is unbiased as the noise is zero-centered (App. A). In our experiments on SST2, DPAdamBC shows, on average, about 2 pp higher accuracy than DPAdamW, confirming its effectiveness (Tab. 7, App. A).

Input: Clipped gradients $\bar{\mathbf{G}}$, iteration t, parameters $\theta_{\text{DP}}^{(t-1)}$, clipping threshold C, noise variance σ^2 computed from ϵ, δ via [69], stability constant γ', decay rates β_1, β_2, batch size B.
Output: Updated DP parameters $\theta_{\text{DP}}^{(t)}$
Function DPAdamBC($\bar{\mathbf{G}}$, t, $\theta_{\text{DP}}^{(t-1)}$):
$\tilde{\mathbf{g}}_t = \frac{1}{B}\left(\sum_{\bar{\mathbf{g}}\in\bar{\mathbf{G}}} \bar{\mathbf{g}} + \psi_t\right)$ with $\psi_t \sim \mathcal{N}(0, \sigma^2 C^2 \mathbb{I}^d)$
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1)\cdot \tilde{\mathbf{g}}_t$
$\hat{m}_t \leftarrow m_t/(1-\beta_1^t)$
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2)\cdot \tilde{\mathbf{g}}_t^2$
$\hat{v}_t \leftarrow v_t/(1-\beta_2^t)$
$\theta_{\text{DP}}^{(t)} \leftarrow \theta_{\text{DP}}^{(t-1)} - \eta \cdot \hat{m}_t/\sqrt{\max(\hat{v}_t - (\sigma C/B)^2, \gamma')}$
return $\theta_{\text{DP}}^{(t)}$

Algorithm 4: DPAdamBC step t from [62], with bias-correction term $\sigma C/B^2$ (m_0, v_0 are initialized to zero).

4.2 MPC Inference Optimizations

Next, we detail optimizations to achieve scalable MPC inference while maintaining accuracy comparable to cleartext. We introduce enhancements for linear layers, mitigations for numerical instability in MPC fixed-point arithmetic (e.g., SoftCap), and widely applicable non-linear function approximations (e.g., for LayerNorm, Softmax).

Local Cleartext Computations. Typically, in non-private inference, matrices $\mathbf{A}$ and $\mathbf{B}$ are merged into the layer weights $\mathbf{W}_0$, otherwise the number of parameters for inference would increase compared to the original model. However, in our setting where LoRA matrices contain sensitive information from fine-tuning, the weights matrices, merged with $\mathbf{A}$ and $\mathbf{B}$, need to be secret-shared during MPC inference, resulting in a significant performance overhead. The optimization for Linear layer proposed by SPRINT is to not merge the matrices $\mathbf{A}$ and $\mathbf{B}$ into the layer weights $\mathbf{W}_0$. Keeping LoRA matrices separated allows SPRINT to keep the large, frozen parameters of the public pretrained model in cleartext. Fine-tuning with FALoRA further enhances performance by freezing the matrix $\mathbf{A}$, i.e., no need for secret sharing as $\mathbf{A}$ is not data-dependent, halving the number of secret-shared parameters compared to LoRA. Consequently, after fine-tuning, the model owner *MO* secret shares only the matrix $\mathbf{B}$ and the Classifier weights to the servers S_1,S_2. As shown in Fig. 4 only the matrix $\mathbf{B}$ and the input to the Linear layer are secret shared. This enables S_1,S_2 to perform local matrix multiplications between secret-shared inputs and cleartext parameters (as formalized in Alg. 7, App. E). More formally:

$$\langle \mathbf{Y} \rangle = (\langle \mathbf{X} \rangle \cdot \mathbf{A}) \cdot \langle \mathbf{B} \rangle + \langle \mathbf{X} \rangle \cdot \mathbf{W}_0.$$

Overall, FALoRA reduces the communication costs for Linear layers by 70× compared to full fine-tuning and 10× compared to LoRA. Also, FALoRA improves LAN runtime by 5× and 2× compared to full fine-tuning and LoRA, respectively (Tab. 4, Sec. 5). Additionally, since the Embedding layer is frozen, users U can locally precompute embeddings and secret-share them. This approach offloads the computation of the embeddings from the MPC servers S_1, S_2.

Logits Capping. To improve numerical stability during inference, we apply *logits capping* to Softmax inputs in Attention layers. Given input vector $\mathbf{x}$ and threshold K, the output is:

$$\text{SoftCap}(\mathbf{x}, \mathsf{K}) = \mathsf{K} \cdot \text{Tanh}(\mathbf{x}/\mathsf{K}).$$

This method, employed in Gemma 2 models [63], helps the model maintain more stable and effective learning dynamics by bounding the Softmax inputs, while preserving the proportional relationships between them via Tanh [63], as formalized in Alg. 8 (App. E). Additionally, we cap embeddings to bound the input range of the first Encoder and minimize truncation errors from large matrix multiplications in Attention. Although capping enhances numerical stability, it requires Tanh computation in MPC and a new hyperparameter to tune, i.e., K. In terms of runtime, SoftCap accounts for 20% of the total MPC runtime, and 25% of the total communication cost (Tab. 3, Sec. 5.2). However, SoftCap enables our Softmax optimization SoftmaxK, improving the overall runtime and communication cost compared to non-optimized $\text{Softmax}_{\text{MPC}}$. Regarding accuracy, a capping threshold $\mathsf{K} = 50$ shows comparable accuracy to non-capped fine-tuning (reducing at most by 0.1 pp) when applied during inference only or also during fine-tuning (Tab. 11, App. F.4).

SoftCap-based Softmax Approximation. Computing a highly accurate Softmax approximation in MPC is costly requiring exponentiation and division, which are also prone to numerical errors. Existing MPC approximations, such as fully connected neural networks [49], quadratic function [38], and ReLU-based approximations [2, 51], do not achieve high accuracy in our evaluation (App. D.1). For example, quadratic function approximation [38] achieves accuracy close to random guessing. Most of these approximations are designed either for model distillation or training neural networks, rather than fine-tuning LLMs, which may explain the poor performance. To achieve accuracy comparable to the state of the art while stabilizing Softmax, we first adopt a common trick from MPC literature ($\text{Softmax}_{\text{MPC}}$) [15, 23, 36, 47, 54]: normalizing the input of the exponential function by subtracting the maximum value for each of the input vectors $\mathbf{x} \in \mathbb{R}^d$. The normalization ensures that the exponentiation is computed only for non-positive values, preventing overflows. Additionally, we avoid computation of the maximum since we already apply logits capping (SoftCap) to the Softmax input, which constrains the input range to $[-\mathsf{K}, \mathsf{K}]$. We set the maximum to K (as formalized in Alg. 8, App. E), resulting in:

$$\text{SoftmaxK}(\mathbf{x}) = \frac{e^{\mathbf{x}-\mathsf{K}}}{\sum_{j=1}^{d} e^{\mathbf{x}_j-\mathsf{K}}}.$$

SoftmaxK provides a reliable alternative, reducing runtime and communication while maintaining accuracy. Specifically, compared to $\text{Softmax}_{\text{MPC}}$, SoftmaxK reduces communication by 10×, and is up to 5× faster. Even when SoftCap is not strictly necessary for numerical stability, it enables SoftmaxK, improving MPC performance. In our evaluation, SoftCap with SoftmaxK reduces total runtime and communication by $\approx 2\times$ compared to $\text{Softmax}_{\text{MPC}}$.

Modified Attention Masking. In Attention, padding tokens are added to handle variable-length sequences in batched inputs. To prevent padding tokens from affecting the model output, their logits are typically masked with the smallest representable value. After Softmax, the padding positions receive negligible scores, ensuring only relevant input tokens contribute to the output. However, this approach causes numerical errors during the computation of Softmax, and Tanh (in SoftCap) in MPC. To address this issue, we modify the padding mask by setting the minimum value to -200 which does not cause errors in the Softmax inverse computation.

When SoftCap is applied, we further adjust the padding value by scaling it with K to ensure stability for Tanh which works correctly within $[-250, 250]$. This adjustment minimizes the influence of padding tokens and ensures numerical stability (Sec. 5.2).

LayerNorm Approximation. LayerNorm computes the inverse square root of the variance (Eq.(1)). Rather than sequentially computing the square root and reciprocal, we adapt the inverse square root protocol InvSqrt [36], which enhances numerical stability (maximum absolute value of ≈ 330 vs. 220) and improves performance, achieving a 3× speedup, and reducing communication cost by 3×.

$$\mathsf{LayerNorm}_{\mathsf{MPC}}(\mathbf{x}) = (\mathbf{x} - \mu(\mathbf{x})) \cdot \mathsf{InvSqrt}(\nu(\mathbf{x}, \mu(\mathbf{x}))) \cdot \mathbf{W}_0 + \mathbf{b}_0$$

Here, $\mu(\mathbf{x})$ and $\nu(\mathbf{x})$ are the mean and variance of the input tensor. The variance is not computed from scratch but reusing the mean result. However, large datasets, e.g., MNLI and QQP, present samples that when processed have variances exceeding the maximum input range of InvSqrt. To address this, we scale the variance before applying InvSqrt. Yet, the scaled InvSqrt performs poorly for inputs below 1, resulting in ≈ 10 pp accuracy drop in MPC inference. To minimize the approximation error, we implemented a piecewise approximation of InvSqrt. Specifically, for inputs $x \in \mathbf{x}$ lower than a threshold T, i.e., $x \leq T$, we use the unscaled InvSqrt, while for larger inputs, we apply the scaled variant with a scaling factor β:

$$\mathsf{InvSqrt}_{\mathsf{MPC}}(\mathbf{x}) = \begin{cases} \mathsf{InvSqrt}(x) & \text{if } x \leq T \\ \mathsf{InvSqrt}(x/\beta) \cdot 1/\sqrt{\beta} & \text{else} \end{cases} \tag{5}$$

While the piecewise $\mathsf{InvSqrt}_{\mathsf{MPC}}$ increases InvSqrt communication by 2%, rounds by 1.5×, and runtime by 1.5× in WAN settings due to A2B and B2A conversions, it effectively extends the input range. With a scaling factor $\beta = 10^3$ and a threshold $T = 100$, $\mathsf{InvSqrt}_{\mathsf{MPC}}$ achieves an MSE of 1.9×10^{-9} compared to 2×10^{-4} for scaled InvSqrt in the range $(0, 10000]$ (evaluated over 10^6 evenly spaced points), and 3.9×10^{-6} compared to 2 in $(0, 1]$. We formalize the $\mathsf{LayerNorm}_{\mathsf{MPC}}$ MPC protocol in Alg. 11 (App. E).

GeLU Approximation. Computing the GeLU (Eq. (3)) in MPC is challenging due to numerical instability of the erf function [36]. Specifically, CrypTen's approximation for erf [36] becomes unstable for inputs $|x| > 5$. We tested different alternatives, such as GeLU that uses Tanh instead of erf [27], but it is numerically unstable for $|x| > 20$. We evaluated with ReLU [49], a quadratic function [38], and HardTanh [53] in place of Tanh to improve stability. However, these methods showed poor performance, likely because they have been primarily used for model distillation rather than direct training. We find that GeLU approximation from BOLT [54] ($\mathsf{GeLU}_{\mathsf{MPC}}$) offers the best accuracy-performance trade-off among the tested alternatives. Despite being slower than the Tanh approximation, it provides a more stable solution across any input range. $\mathsf{GeLU}_{\mathsf{MPC}}$ is a piecewise approximation that combines a polynomial (polyGeLU) around 0 with ReLU elsewhere, formalized as:

$$\mathsf{GeLU}_{\mathsf{MPC}}(x) = \begin{cases} \mathsf{ReLU}(x) & \text{if } |x| > 2.7 \\ \mathsf{polyGeLU} & \text{else} \end{cases} \tag{6}$$

Here, $\mathsf{polyGeLU}(x) = (P_0(x) + g_0|x| + g_3) \cdot P_0(x) + g_4 + 0.5x$, with $P_0(x) = (g_0|x| + g_1) \cdot |x| + g_2$. In the range $[-2.7, 2.7]$ where $\mathsf{GeLU}_{\mathsf{MPC}}$ approximates the non-linearity of the GeLU compared to ReLU, polyGeLU achieves a floating-point error of 9.77×10^{-4} for inputs in $[-5, 5]$ [54]. To optimize $\mathsf{GeLU}_{\mathsf{MPC}}$ implementation in CrypTen, we addressed the inefficiency of computing $|x|$ which requires costly A2B and subsequent B2A conversions. We pre-compute $|x|$ and use its value in both polyGeLU and ReLU. Specifically, we modify the CrypTen implementation by computing ReLU as $\mathsf{ReLU}(x) = (x + |x|)/2$, instead of $\mathsf{ReLU}(x) = x \cdot \mathsf{sign}(x)$. This saves one A2B and B2A conversion, as $|x|$ is computed only once. We formalize the $\mathsf{GeLU}_{\mathsf{MPC}}$ MPC protocol in Alg. 10 (App. E).

Security of SPRINT. SPRINT inherits its security guarantees from the underlying CrypTen framework. Specifically, CrypTen protocols are secure against information leakage from any static passive adversary corrupting up to $|P| - 1$ of the $|P|$ parties involved in the computation [36, Lemma 1]. The primitives used by CrypTen are detailed in [36, Tab. 2, App. D], with security proofs for arithmetic multiplication [6, Thm. 1], binary AND operations [12, Sec. III.B], A2B conversions [11, Sec. 3], B2A conversions and truncation protocols [36, App. 1.2, App. A.1.1]. We prove the security of SPRINT in the semi-honest model using the simulation paradigm [21] (App. E). We provide formal security proofs for each novel protocol introduced in SPRINT (Tab. 8) in App. E. The security of SPRINT follows from the sequential composition theorem [8].

5 Experiments

This section provides a comprehensive evaluation of SPRINT, structured in two phases (as in Fig. 3): (I) MPC-aware DP fine-tuning (Sec. 5.1) to evaluate the cleartext accuracy (on $\mathcal{M}^A_{\mathrm{FT}}$), and (II) MPC inference (Sec. 5.2) to evaluate inference runtime, communication cost and the accuracy of the MPC model ($\mathcal{M}^A_{\mathrm{MPC}}$). We split the evaluation to isolate the impact of DP fine-tuning and MPC inference on final accuracy (Tab. 6). Notably, SPRINT is the first end-to-end framework for *both* DP fine-tuning and MPC inference. We compare the DP fine-tuning accuracy with works covering only DP fine-tuning [41, 70], and the MPC inference performance with works performing only MPC inference (with non-DP fine-tuning) [31, 45, 54]. First, we introduce model and datasets and then describe the frameworks used for DP fine-tuning and MPC inference.

Model and Datasets. We fine-tune the RoBERTa-base [44] model on GLUE benchmark tasks [66] with over 10k training examples, i.e., SST2 (67.3k examples), QNLI (105k), MNLI (393k), and QQP (364k), following DP fine-tuning baselines [41, 70].

Frameworks. We implement DP fine-tuning with PyTorch [56] due to its flexibility and large availability of pre-trained models. We use Opacus [69] for Poisson subsampling, Gaussian noise sampling and DP accounting, and integrate it with the PrivateTransformers library [41] to support GhostClipping [41]. For MPC inference, we use CrypTen [36] since it has PyTorch-like APIs and data structures.

5.1 MPC-Aware DP Fine-Tuning

Next, we evaluate the cleartext accuracy of MPC-aware DP fine-tuning (on $\mathcal{M}^A_{\mathrm{FT}}$) and provide details on our hyperparameter search.

State-of-the-Art Baselines. Several works [7, 41, 68, 70] explore DP fine-tuning for RoBERTa or BERT models. Among these, we use as baselines the two works achieving the highest accuracy: Yu et al. [70] fine-tune the full RoBERTa model, while Li et al. [41] apply various parameter efficient fine-tuning techniques, including

LoRA. However, none uses DP-specific optimizers. Furthermore, they do not employ theoretically sound Poisson subsampling, using shuffling instead, which can impact formal DP guarantees [3].

Privacy Accounting. We use Gaussian noise $\Psi \sim \mathcal{N}(0, \sigma^2 C^2)$ and set $\epsilon = 8$ and $\delta \approx 0.1/|D_{\text{FT}}|$ as typical in related works [41, 70]. Specifically, $\delta = 1 \times 10^{-5}$ for SST2 and QNLI, and $\delta = 1 \times 10^{-6}$ for MNLI and QQP. We track the privacy budget across epochs using the Rényi Differential Privacy (RDP) [50] accounting tool provided by Opacus [69]. The noise variance σ^2 for each training step is numerically computed by Opacus from ϵ, δ, the sampling rate q, the batch size B, and the number of training epochs E.

Hyperparameters Discussion. To achieve the best accuracy, we explore a comprehensive set of hyperparameters, including not only ML parameters, but also DP parameters (e.g., C), SoftCap K, and FALoRA **A** initialization. We report the set of hyperparameters tested for each dataset in Tab. 9 (App. F.1). Due to the large number of hyperparameters, and the high resources needed to test different MPC approximations, we set a compute-bound of 10 epochs. In DP fine-tuning, small batch sizes (e.g., $B = 128$) with small learning rates (e.g., $\eta = 5 \times 10^{-5}$) perform considerably worse than in non-DP fine-tuning, since the DP noise scales with $1/B$ [41]. Hence, we test large *batch sizes* (i.e., $B \in \{1024, 2048\}$) for Poisson subsampling to reduce the effective noise in gradient updates [41, 57]. Similar to [41], we found that $B = 2048$ consistently performs better across all learning rates by reducing noise variance, despite less frequent parameter updates. We test different *learning rates* η and *weigh decay* from related works [41, 70]. SPRINT performs best with higher $\eta = 5 \times 10^{-4}$ for smaller datasets (SST2, QNLI) to compensate for fewer samples per epoch, and smaller $\eta = 5 \times 10^{-5}$ for larger datasets (MNLI, QQP). A small weight decay of 0.001 performs best across all dataset. The *clipping threshold* C is critical in balancing gradient magnitude and noise variance. From our experiments, $C = 1$ performs best across all tested learning rates and datasets, providing an optimal balance between noise variance and gradient updates magnitude. A larger threshold $C = 10$ (as in [70]) performs better with low $\eta = 5 \times 10^{-5}$ but does not achieve the best accuracy, likely due to excessive noise variance preventing optimal convergence [41]. Instead, a smaller $C = 0.1$ (as in [41]), which clips almost all gradients, performs well with higher learning rates ($\eta > 1 \times 10^{-4}$), achieving 3 pp lower accuracy than the best values (Tab. 1). This may stem from FALoRA updating only $\approx 0.5\%$ of parameters, requiring larger updates to converge. For FALoRA, the optimal setup varies by dataset. We evaluate applying LoRA to different set of Linear layers, with different ranks ($r = \{8, 16, 32\}$) setting $\alpha = \{r, 2r\}$ following LoRA literature [28, 61, 70]. For all datasets, applying FALoRA to all Linear layers yields the best results, except for QNLI, where restricting FALoRA to Linear_{Q}, Linear_{V}, and $\text{Linear}_{\text{Att}}$ is more effective. The best rank is $r = 16$, with $\alpha = 16$ for SST2, QNLI, and $\alpha = 32$ for MNLI, QQP, indicating that larger datasets benefit from a higher α. Since matrix **A** is not updated during fine-tuning, its initialization impacts final accuracy. We evaluate different initialization methods (detailed in App. F.2) expanding on the research direction proposed by FFA-LoRA [61]. We find that the best initialization is dataset-dependent. For example, Kaiming performs best on SST2, while Orthogonal is better on QNLI. We test also the effect of different *capping thresholds* K_{FT} in fine-tuning: $\text{K}_{\text{FT}} = 50$ following [63], and $\text{K}_{\text{FT}} = 10$ to reduce the probability of numerical error in inference. Consistent with Gemma [63], $\text{K}_{\text{FT}} = 50$ generally preserves model accuracy. Contrarily, a smaller $\text{K}_{\text{FT}} = 10$ leads to an accuracy drop of, e.g., up to 3.7 pp compared to non-capped training on QNLI.

DP fine-tuning type ($\epsilon = 8$)	SST2	MNLI	QNLI	QQP
Full [41]	92.1	83.2	87.9	86.1
LoRA [70]	92.2	83.5	87.3	85.7
SPRINT	92.2	78.6	85.2	82.7
SPRINT ($\text{K}_{\text{FT}} = 50$)	92.3	78.6	84.8	82.7
SPRINT ($\text{K}_{\text{FT}} = 10$)	90.7	77.2	81.5	81.3

Table 1: Cleartext accuracy (in %) on GLUE with RoBERTa.

Fine-Tuning Accuracy. We evaluate the fine-tuning accuracy using DPAdamBC and DPAdamW on the SST2 dataset across different capping thresholds (Tab. 7, App. A). Our results show that DPAdamBC consistently outperforms DPAdamW in accuracy for any capping threshold, with an average of $\approx$ 2 pp accuracy improvement confirming its effectiveness in DP fine-tuning. Tab. 1 compares the cleartext accuracy (on $\mathcal{M}^A_{\text{FT}}$) with related works [7, 41], and applying different *capping thresholds* in fine-tuning (K_{FT}). Compared to DP baselines, i.e., full fine-tuning [41] and LoRA [70], SPRINT is more parameter-efficient, using only 50% of LoRA parameters and 0.52% of the parameters for full fine-tuning. Despite training fewer parameters and leveraging MPC-aware approximations, SPRINT fine-tuning achieves comparable accuracy on the SST2 dataset. On average, SPRINT achieves accuracy within 2.5 pp compared to LoRA [70], with at most 5 pp accuracy drop on MNLI (Tab. 1). The accuracy drop may stem from the compute-bound constraint of 10 epochs, since Tab. 1 shows that the accuracy gap widens with increasing dataset size and larger datasets generally need more iterations for convergence. Notably, the accuracy drop is not due to Poisson subsampling, as it performs similarly to shuffling; e.g., on MNLI, Poisson achieves 0.2 pp higher accuracy. Similarly, the MPC approximation integrated during fine-tuning do not significantly impact accuracy due to their integration in fine-tuning.

Finally, to assess practical deployment we evaluate the impact of random seeds on accuracy across different LoRA **A** initializations (App. F.3). Tab. 10 (App. F.3) shows that fixed-seed accuracy can be misleading, with up to 1.1 pp variation between fixed and random seeds on SST2. However, with random seeds, most initialization techniques achieve similar performance within a 0.4 pp range.

5.2 MPC Inference

Here, we evaluate and compare SPRINT's inference runtime and communication with state-of-the-art MPC inference works. We also evaluate GPU acceleration and batched inference. Finally, we evaluate the accuracy drop from the cleartext evaluation (Tab. 1, Sec. 5.1) due to MPC approximations, and perform ablation studies on the impact of MPC-aware fine-tuning on model accuracy.

Evaluation Setup. We evaluate latency and communication cost with 2 AWS c6.xlarge machines (as BOLT [54]). We simulate network conditions via Linux traffic control in 3 scenarios: LAN (3Gbps bandwidth, 0.8ms delay), regional WAN_{R} (200Mbps, 40ms), and global WAN_{G} (100Mbps, 80ms). Communication is measured in bytes exchanged and number of rounds. All reported numbers are averaged over 10 runs. Following DP fine-tuning works (Sec. 5.1),

Framework	Runtime (s)			Communication	
	LAN	WAN_R	WAN_G	GB	Rounds
Iron [24]	475.0	6453.0	12996.0	280.99	13663
BOLT [54]	91.0	913.0	1744.0	25.74	10901
BumbleBee [45]	135.8	650.7	1161.8	6.81	-
SHAFT [31]	51.2	385.6	736.1	10.46	1495
SPRINT	32.6	310.8	613.4	6.55	1813

Table 2: End-to-end runtime and communication for RoBERTa MPC inference.

we fine-tune and perform MPC inference on RoBERTa [44] instead of BERT used in MPC inference works [24, 31, 45, 54]. However, the two models share the same architecture and number of parameters, the important aspects to evaluate MPC inference.[3] For the GPU evaluations (Tab. 5), we use 2 AWS g5.xlarge instances with 1 NVIDIA A10G GPU. Finally, we compare the accuracy drop from cleartext (i.e., on $\mathcal{M}^A_{\text{FT}}$ with cleartext θ_{DP}, Tab. 1) to MPC inference (i.e., on $\mathcal{M}^A_{\text{MPC}}$ with secret-shared $\langle\theta_{\text{DP}}\rangle$).

State-of-the-art MPC Inference Baselines. We compare with BOLT [54], SHAFT [31], BumbleBee [45], and Iron [24], which perform MPC inference without DP fine-tuning. These solutions can be categorized based on used cryptographic primitives: BOLT, BumbleBee, and Iron are hybrid solutions combining homomorphic encryption (HE) for matrix multiplications and secret sharing (SS) for non-linear layers, while SHAFT relies only on SS and it is built on CrypTen[36] like SPRINT. The goal of HE matrix multiplications is to reduce the communication cost compared to SS, however, this requires additional communication to convert shares to HE ciphertexts. Hybrid solutions BOLT, BumbleBee, and Iron propose optimizations for Linear layers leveraging HE. Specifically, Iron and BOLT use compact packing of HE ciphertexts to reduce communication, while BumbleBee uses ciphertext interleaving and matrix oblivious linear evaluation. For GeLU, BumbleBee leverages a 4-segment polynomial approximation, while BOLT and SHAFT propose different approximations for the function $\zeta(x) = \text{GeLU}(x) - \text{ReLU}(x)$, and use ReLU elsewhere. Specifically, BOLT uses a polynomial approximation, while SHAFT uses a sinusoidal function with Fourier approximation. Iron uses the Tanh approximation (App. D.2) with an optimized protocol for squaring and Tanh. For Softmax, BOLT uses the integer-only exponential function from I-BERT [34]. BumbleBee and Iron normalize the argument of the exponential function (Sec. 4.2). SHAFT combines input clipping and an iterative approximation (i.e., ordinary differential equation) to avoid computing the maximum and reciprocal. Notably, BOLT integrates word elimination to reduce the input size of non-linear layers by discarding words with low attention scores. From BOLT's evaluation, this technique allows to reduce the end-to-end communication by 2.3× and the runtime by up to 2×. However, this requires an oblivious protocol to rank and discard tokens with low attention scores. Next, we empirically compare the end-to-end performance of SPRINT with these baselines.

Performance: End-to-End. Tab. 2 compares the end-to-end MPC inference performance of SPRINT with BOLT [54], SHAFT [31], BumbleBee [45], and Iron [24] in three network conditions. SPRINT is the fastest in all networking settings, followed by SHAFT. SPRINT is 1.6× faster than SHAFT in LAN, and 1.2× faster in WAN_R and WAN_G. The best hybrid solution in LAN is BOLT, yet, SPRINT is 3× faster than BOLT on average across all network settings. While BumbleBee is the best hybrid solution in WAN_R and WAN_G, it is still 2× slower on average than SPRINT in those settings. In terms of total communication, SPRINT is the best solution, closely followed by BumbleBee. SHAFT requires 1.6× more communication than SPRINT, while BOLT 3× and Iron 43× more. Regarding number of communication rounds, SHAFT is the most efficient solution, followed by SPRINT which requires 1.2× more rounds. This explains why SHAFT performance are closer to SPRINT in WAN_R and WAN_G, despite requiring more communication. BOLT and Iron require 10× more rounds than SPRINT and SHAFT, likely due to HE-to-secret sharing conversions.[4] In App. F.5, we show how SPRINT scales to larger models, specifically to RoBERTa-large (with 2× more Encoder layers than RoBERTa-base) while remaining 1.4× faster than SHAFT. CrypTen is optimized for 2-party setting [36], as most MPC inference works [24, 31, 45, 54]. Still, we also evaluate SPRINT with 3 and 4 parties. Overall, runtime increases by 2.6× and 3.6× respectively (details in App. F.5). Next, we evaluate per-layer runtime and communication costs to assess the impact of different optimizations, comparing with BOLT [54] and SHAFT [31], whose implementations provide such a breakdown.

Framework	Layer	Runtime (s)			Communication	
		LAN	WAN_R	WAN_G	GB	Rounds
BOLT [54]	Linear	63.8	75.0	89.0	2.75	122
	Softmax	6.0	164.0	315.0	5.28	2748
	GeLU	3.6	205.0	423.0	9.1	1056
	LayerNorm	11.5	312.0	626.0	3.4	2640
SHAFT [31]	Linear	10.84	54.21	100.3	1.52	98
	Softmax	16.97	137.1	266.1	4.28	492
	GeLU	17.4	123.5	233.9	4.18	228
	LayerNorm	2.7	56.21	109.92	0.11	650
SPRINT	Linear	2.67	10.84	22.2	0.03	97
	SoftmaxK	3.35	54.09	106.02	0.33	564
	SoftCap	6.94	58.56	113.6	1.65	276
	GeLU_{MPC}	16.1	118.6	242.0	4.36	240
	LayerNorm	2.85	58.9	121.8	0.07	696

Table 3: Per-layer runtime and communication for BOLT, SHAFT**, and SPRINT for RoBERTa MPC inference.**

Linear optimization	None	LoRA ($r = 16$)	FALoRA ($r = 16$)
Runtime	10.4 s	4.0 s	2.3 s
Rounds	74	144	72
GB	1.43	0.2	0.02

Table 4: Linear layers runtime and communication for MPC inference per fine-tuning method (excluding Classifier).

Performance: Linear Layers. SPRINT outperforms BOLT [54] and SHAFT [31] in terms of runtime and communication cost for Linear layers (Tab. 3). Specifically, SPRINT is 5× faster than SHAFT across all network settings, 2.5× faster than BOLT in LAN and 1.3× in WAN_G. BOLT is less sensitive to networking conditions for Linear layers, since it leverages homomorphic encryption to minimize interactions. Despite this, BOLT is the most communication-intensive solution, requiring 90× more communication than SPRINT and 1.2× more rounds. This is mostly due to the HE ciphertext conversion. SPRINT, unlike related works [24, 31, 45, 54], does not

[3] RoBERTa and BERT differs for the pre-training strategy and have different vocabulary sizes in the embeddings, which are not computed in MPC during inference.

[4] BumbleBee is implemented with SPU [48] which does not compute rounds.

Framework	Batch Size	Runtime (s)			Communication	
		LAN	WAN_R	WAN_G	GB	Rounds
SPRINT	1	20.4	298.6	606.6	6.55	1909
	8	13.0	180.8	368.82	6.53	238.6
	16	12.5	175.1	341.12	6.53	119.3
	32	12.3	168.50	327.64	6.52	59.6
SHAFT [31]	1	29.59	358.70	723.89	10.46	1495

Table 5: E2E per-sample (amortized) cost for RoBERTa MPC inference on GPUs and batched inference.

Dataset	K_{FT} /K_{INF}				
	None/50	**None/10**	**50/50**	**50/10**	**10/10**
SST2	**91.4** (0.8)	89.8 **(0.2)**	**91.4** (0.9)	89.5 (0.5)	90.0 (0.3)
MNLI	78.2 (0.4)	70.6 (0.5)	**78.3 (0.3)**	71.3 (0.4)	76.6 (0.6)
QNLI	**85.0** (0.2)	73.6 (0.6)	84.8 **(0)**	70.1 (0.5)	80.7 (0.8)
QQP	**82.6** (0.1)	75.1 **(0)**	**82.6** (0.1)	75.9 **(0)**	79.4 (0.2)

Table 6: MPC inference accuracy ($\mathcal{M}^A_{\mathbf{MPC}}$ with $\langle\theta_{DP}\rangle$) with different K_{FT}/K_{INF}. Parentheses show accuracy gap (in pp) vs. cleartext ($\mathcal{M}^A_{\mathbf{FT}}$ with θ_{DP}) with the same K_{FT}/K_{INF}.

secret share all parameters but only a small subset (i.e., FALoRA **B**). Next, we also investigate SPRINT's performance without FALoRA. Specifically, we compare the MPC runtime of Linear layers in LAN across three configurations based on the fine-tuning strategy (Tab. 4). With full fine-tuning the entire model is secret shared. With LoRA, only the LoRA **A** and **B** matrices are secret shared, whereas FALoRA secret shares only **B** matrices. During inference, FALoRA reduces runtime by 5× compared to full fine-tuning and 1.7× compared to LoRA. Also, FALoRA fine-tuning significantly reduces communication costs of Linear layers, i.e., by 70× and 10× compared to full fine-tuning and LoRA (Tab. 4), respectively. Unlike FALoRA and full fine-tuning, which require one secret-shared matrix multiplication per layer, LoRA needs two rounds for its sequential **A** and **B** multiplications. Notably, even without FALoRA, SPRINT is faster than SHAFT and BOLT, since encrypting all linear layer increases runtime by 1.3× (Tab 4, Tab. 2).

Performance: Non-Linear Layers. SoftmaxK is the most efficient solution across the proposed approximations (Tab. 3). Combined with SoftCap (required to optimize SoftmaxK), SPRINT reduces Softmax runtime by up to 1.4× compared to BOLT in WAN_R and 1.6× compared to SHAFT in LAN. While BOLT's Softmax is the fastest in LAN, its performance drops in WAN_R and WAN_G, since it requires 2.7× more communication and 3.3× more rounds than SPRINT. Instead, SHAFT's Softmax runtime gets closer to SPRINT in WAN_R and WAN_G, since it requires 1.7× less communication rounds. For GeLU, SPRINT adapts BOLT's solution to CrypTen. Despite this, BOLT is the fastest implementation in LAN, while SPRINT is 1.7× faster in WAN_R and WAN_G, since BOLT's implementation requires 2.1× more communication and 7.3× more rounds. SHAFT's approximation has comparable performance to SPRINT in terms of runtime and communication, i.e., SHAFT GeLU is 1.1× slower than SPRINT in LAN, but 1.03× faster in WAN_G. LayerNorm performance are comparable for SPRINT and SHAFT, since they both use CrypTen. However, SPRINT uses a piecewise approximation for InvSqrt in LayerNorm to reduce the approximation error (Sec. 4.2). Compared to BOLT, our LayerNorm is faster across all settings with up to a 10× speedup in WAN_G. Across all layers in SPRINT, GeLU makes up the largest part of the runtime (up to 50% in LAN) and communication (over 20%). SoftCap has the second-highest cost, i.e., 17% of the total.

GPU and Batched Inference. We evaluate the performance of SPRINT with GPUs and batching, comparing against SHAFT where possible (Tab. 5).[5] While we tested batched inference on CPU and GPU, we found it only beneficial on GPUs. Batching on CPU leads up to 1.4× slower per-sample runtime. For a single inference, using a GPU is most effective in a LAN, providing a 1.6× speedup over CPU (Tab. 2). This advantage diminishes in high-latency networks, reducing to a 1.04× speedup in WAN_R, and comparable runtimes in WAN_G. However, enabling batched inference provides a significant speedup in all scenarios. As the batch size increases to 32, the amortized, per-sample communication rounds decrease by 32×, and the runtime improves significantly. Specifically, in LAN, $B = 32$ leads to a 1.7× speedup over the non-batched GPU baseline and a 2.6× speedup over the non-batched CPU baseline. More importantly, batching effectively mitigates the network bottleneck, with speedups reaching 1.8× in WAN_R and 1.9× in WAN_G. Compared to SHAFT, SPRINT is up to 1.4× faster in LAN. We expect similar trends with batched inference, since both leverage CrypTen.

MPC Inference Accuracy. Tab. 6 reports the MPC inference accuracy of SPRINT on SST2, QNLI, MNLI, and QQP datasets using inference capping thresholds $K_{INF} \in \{10, 50\}$, where $K_{INF} \leq K_{FT}$ from fine-tuning. We compare the accuracy gap between cleartext ($\mathcal{M}^A_{FT}$) and MPC inference ($\mathcal{M}^A_{MPC}$) for matching K_{FT} and K_{INF} combinations. SPRINT achieves high MPC inference accuracy with gaps under 0.9 pp compared to cleartext. Across all datasets and settings, the average accuracy gap is at most 0.5 pp. The optimal setting is $K_{FT} = K_{INF} = 50$ with only 0.3 pp gap, demonstrating that $K = 50$ provides the best trade-off between cleartext and MPC accuracy. All MPC related works [31, 45, 54] show accuracy drops < 1.0 pp. However, they evaluate on small datasets (e.g., SHAFT validates on QNLI with 5k samples), whereas we include larger datasets like QQP (over 40k validation samples) for a comprehensive assessment of SPRINT's accuracy and stability. Fine-tuning on large datasets (e.g., QQP and MNLI) produces model weights with broader value ranges due to more training updates, increasing risk of numerical error in inference. The InvSqrt protocol in LayerNorm causes overflows with these datasets, leading to near-random accuracy. We mitigate this with the piecewise approximation InvSqrt_{MPC} (Eq. (5)), ensuring numerical stability while prioritizing accuracy over performance (Sec. 4.2). Notably, SPRINT achieves MPC accuracy within 0.1 pp of its best cleartext accuracy on QQP, highlighting its scalability and robustness with large datasets. Next, we evaluate how our MPC-aware fine-tuning strategy impacts MPC inference accuracy.

Impact of MPC-aware Fine-tuning. To quantify the benefit of MPC-aware fine-tuning, we conducted ablation studies on the impact of SoftCap, which enables SoftmaxK, and GeLU approximation. For SoftCap, Tab. 6 shows that applying it during fine-tuning can reduce the cleartext-to-MPC accuracy gap (in parentheses) but does not always yield the best final MPC accuracy. For instance, on QNLI, the highest MPC accuracy is achieved without SoftCap during fine-tuning, while the lowest accuracy gap is with $K_{FT} = K_{INF} = 50$ (0 pp). This suggests fine-tuning with SoftCap is

[5] SHAFT [31] does not report results for batched inference, and their implementations leads to out-of-memory errors with batch size of 8.

not always necessary, though SoftCap enables SoftmaxK by avoiding the max computation. For GeLU approximation, we evaluate: (I) using non MPC-aware GeLU (Eq. (3)) during fine-tuning and inference, (II) performing a naive post-hoc conversion from GeLU to MPC-aware GeLU_{MPC} (Eq. (6)) for inference. We observe that using the non MPC-aware GeLU also during inference (I) leads to random guessing accuracy and causes overflows. The failure occurs since CrypTen's MPC approximation for the erf function in GeLU is only numerically stable for a small input range (i.e., [-5, 5]). Additionally, not performing MPC-aware fine-tuning with GeLU_{MPC} (II) widens the accuracy gap between cleartext and MPC inference, especially with low capping thresholds. Specifically, with $K_{\text{INF}} = 10$, the MPC accuracy drops from 0.2 pp to 3 pp from the cleartext baseline. With $K_{\text{INF}} = 50$, the accuracy drops from 0.8 pp to 1.7 pp. Overall, these results show that MPC-aware fine-tuning on cleartext provides clear benefits for MPC inference on secret shares, alleviating overflow and accuracy issues.

6 Related Works

SPRINT is the first solution to integrate DP fine-tuning and MPC inference for transformer-based models. Existing works focus on either DP fine-tuning [41, 70] (Sec. 5.1) or MPC inference [24, 31, 45, 54] (Sec. 5.2). Next, we discuss additional MPC inference works.

MPC Inference for Transformers. Several approaches have been proposed for MPC inference on transformers [2, 15, 23, 38, 47, 49, 54, 58, 59]. Distillation-based approaches [15, 38, 47, 49] reduce model size and use aggressive approximations for non-linear layers, such as quadratic Softmax and GeLU [38] or replacing Softmax with a neural network [49]. Akimoto et al. [2] replace Softmax with ReLU using a kernelized version of Attention. PUMA [15] uses piecewise polynomial approximations for GeLU and normalization in Softmax. Gupta et al. [23] use function secret sharing (FSS) for efficient comparisons and LUTs for non-linear functions. However, FSS differs from standard MPC, relying on large evaluation keys with large generation and transfer overheads (e.g., $\approx$ 16 GB key size and transfer runtime 10× larger than inference for BERT). Concurrent work [58] applies LoRA to GPT-based models, fine-tuning only a fraction of the decoder layers, without relying on any non-linear function approximations, but on heads merging. Recently, Curl [59] considers LUTs for non-linear functions in CrypTen. Curl inference is 10s faster than SPRINT in LAN. However, its accuracy evaluation is limited to LAN and small models (BERT-tiny, $z = 4$). Furthermore, LUTs have inherent limitations, requiring predefined input ranges and larger table sizes for higher accuracy and larger models.

7 Discussion & Future Works

Next, we discuss limitations and future research directions.

Expand Evaluation. SPRINT can be extended to more complex models, such as Llama [64], and tasks like table-to-text generation [52] and summarization. Despite our computational constraints limiting fine-tuning to 10 epochs, SPRINT achieves strong accuracy: matching state-of-the-art on SST2 and an average accuracy within 2.5 pp of LoRA-based fine-tuning [70]. Extended training would likely improve accuracy and close this gap. We focus on encoder-only models since, due to resource constraints, we cannot perform a full hyperparameter search also for encoder-decoder architectures. However, SPRINT can be easily extended to encoder-decoder models (e.g., GPT-2) thanks to its modularity and scalability. We expect SPRINT to perform similarly to encoder-only models by keeping the encoder in cleartext and applying FALoRA in the decoder.

Consumption of privacy budget from hyperparameter search. Like prior works [41, 70], we do not account for hyperparameter search costs in the privacy budget. Existing approaches [55] for tracking the privacy budget during hyperparameter search assume a fixed set of public hyperparameters and a non-adaptive search. These assumptions do not align with SPRINT's adaptive hyperparameter search. And, more importantly, since SPRINT guarantees model parameters confidentiality, also hyperparameters and potentially the hyperparameter search space itself are not assumed to be public in a deployment scenario, e.g., to guarantee a competitive advantage by keeping proprietary information on the model.

Malicious Security. SPRINT assumes two semi-honest servers, a common setting in private inference [15, 23, 31, 45, 47, 54]. To detect malicious behavior with a certain probability (covert security), SPRINT can use the Mix-and-Match technique [14] without server-side changes. Users mix N private samples with T public, labelled samples. Malicious servers succeed only if they correctly identify and tamper with private samples only. Cheating probability decreases with increasing T [14, Sec. IV.B] at the cost of T additional samples per query. Alternatively, CrypTen can be enhanced with cryptographic protocols, such as MASCOT [33] or zero-knowledge proofs [43]. However, cryptographic protocols introduce significant overhead, e.g., up to 10× slowdown for logistic regression [32].

Advanced Privacy Accounting for DP-LoRA. Another direction is to improve the privacy-utility trade-off for the original DP-LoRA setting. The core challenge in DP-LoRA arises from the quadratic noise term created when adding noise to both $\mathbf{A}$ and $\mathbf{B}$ matrices (Sec. 4.1). FALoRA and recent works in DP federated learning [22, 67] sidestep this issue by freezing matrix $\mathbf{A}$ or modifying the training protocol. Therefore, developing advanced privacy accounting methods for DP-LoRA that can manage the quadratic noise term remains an open research question.

8 Conclusion

We introduced SPRINT, the first scalable solution integrating DP fine-tuning and MPC inference for transformers, ensuring privacy, scalability, and efficiency for secure LLM outsourcing. SPRINT guarantees privacy for fine-tuning data and confidentiality for model parameters, user queries, and predictions. SPRINT achieves high fine-tuning accuracy in cleartext, matching state-of-the-art cleartext DP performance on SST2 [70], and maintains MPC accuracy within 1 pp of its cleartext accuracy. Additionally, SPRINT achieves up to 1.6× faster MPC inference than the state-of-the-art non-DP solution (SHAFT) [31], reducing communication by 1.6×. SPRINT achieves these results through integrated optimizations for DP fine-tuning and MPC inference. For DP fine-tuning, we used noise-aware optimizers to minimize DP noise amplification. For MPC inference, SPRINT introduced accurate approximations for non-linear functions (e.g., GeLU, Softmax) and mitigates numerical instability with logits capping. A key optimization in SPRINT is FALoRA which reduces trainable parameters to minimize DP noise amplification and lowers inference overhead by reducing secret-shared computations.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable feedback. The authors used generative AI-based tools to revise the text, correct any typos, and grammatical errors. This work has received funding from the European Union's Horizon Europe research and innovation program under grant agreement No 101070141 (GLACIATION).

References

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. CCS. ACM, 2016.
[2] Yoshimasa Akimoto, Kazuto Fukuchi, Youhei Akimoto, and Jun Sakuma. Privformer: Privacy-preserving transformer with mpc. EUROS&P . IEEE, 2023.
[3] Meenatchi Sundaram Muthu Selva Annamalai, Borja Balle, Emiliano De Cristofaro, and Jamie Hayes. To shuffle or not to shuffle: Auditing dp-sgd with shuffling. *arXiv preprint arXiv:2411.10614*, 2024.
[4] Borja Balle, Gilles Barthe, Marco Gaboardi, Justin Hsu, and Tetsuya Sato. Hypothesis testing interpretations and renyi differential privacy. AISTATS , 2020.
[5] Donald Beaver. Efficient multiparty protocols using circuit randomization. CRYPTO. Springer, 1992.
[6] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security*, 2008.
[7] Zhiqi Bu, Yu-Xiang Wang, Sheng Zha, and George Karypis. Differentially private bias-term only fine-tuning of foundation models. 2022.
[8] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of CRYPTOLOGY*, 2000.
[9] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al. Extracting training data from large language models. USENIXSec, 2021.
[10] Edward Chou, Josh Beal, Daniel Levy, Serena Yeung, Albert Haque, and Li Fei-Fei. Faster cryptonets: Leveraging sparsity for real-world encrypted inference. *arXiv preprint arXiv:1811.09953*, 2018.
[11] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography Conference*, 2006.
[12] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.
[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina N. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. 2018. URL https://arxiv.org/abs/1810.04805.
[14] Caiqin Dong, Jian Weng, Jia-Nan Liu, Yue Zhang, Yao Tong, Anjia Yang, Yudan Cheng, and Shun Hu. Fusion: Efficient and secure inference resilient to malicious servers. NDSS, 2023.
[15] Ye Dong, Wen-jie Lu, Yancheng Zheng, Haoqi Wu, Derun Zhao, Jin Tan, Zhicong Huang, Cheng Hong, Tao Wei, and Wenguang Cheng. Puma: Secure inference of llama-7b in five minutes. *arXiv preprint arXiv:2307.12533*, 2023.
[16] Cynthia Dwork. Differential privacy. ICALP. Springer, 2006.
[17] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *FTTCS* , 2014.
[18] Steven Englehardt. Next steps in privacy-preserving telemetry with prio. urlhttps://blog.mozilla.org/security/2019/06/06/next-steps-in-privacy-preserving-telemetry-with-prio/, 2010.
[19] David Evans, Vladimir Kolesnikov, Mike Rosulek, et al. A pragmatic introduction to secure multi-party computation. 2018.
[20] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. AISTATS , 2010.
[21] Oded Goldreich. *The Foundations of Cryptography - Volume 1: Basic Techniques*. Cambridge University Press, 2001.
[22] Pengxin Guo, Shuang Zeng, Yanran Wang, Huijie Fan, Feifei Wang, and Liangqiong Qu. Selective aggregation for low-rank adaptation in federated learning. *arXiv preprint arXiv:2410.01463*, 2024.
[23] Kanav Gupta, Neha Jawalkar, Ananta Mukherjee, Nishanth Chandran, Divya Gupta, Ashish Panwar, and Rahul Sharma. Sigma: secure gpt inference with function secret sharing. *Cryptology ePrint Archive*, 2023.
[24] Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, Guowen Xu, and Tianwei Zhang. Iron: Private inference on transformers. *NeurIPS*, 2022.
[25] Christopher Harth-Kitzerow, Ajith Suresh, Yongqin Wang, Hossein Yalame, Georg Carle, and Murali Annavaram. High-throughput secure multiparty computation with an honest majority in various network settings. *arXiv preprint arXiv:2206.03776*, 2022.
[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. ICCV, 2015.
[27] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.
[28] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. ICLR , 2022.
[29] Jie Huang, Hanyin Shao, and Kevin Chen-Chuan Chang. Are large pre-trained language models leaking your personal information? *arXiv preprint arXiv:2205.12628*, 2022.
[30] Stephanie L Hyland and Shruti Tople. An empirical study on the intrinsic privacy of sgd. *arXiv preprint arXiv:1912.02919*, 2019.
[31] Andes YL Kei and Sherman SM Chow. Shaft: Secure, handy, accurate, and fast transformer inference. NDSS, 2025.
[32] Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. CCS, 2020.
[33] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: faster malicious arithmetic secure computation with oblivious transfer. CCS, 2016.
[34] Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. I-bert: Integer-only bert quantization. ICML, 2021.
[35] Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
[36] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. Crypten: Secure multi-party computation meets machine learning. *NeurIPS*, 2021.
[37] Yann LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. *Neural networks: Tricks of the trade*, 1998.
[38] Dacheng Li and Rulin Shao. Mpcformer: Fast, performant and private transformer inference with mpc. *ICLR* , 2023.
[39] Haoran Li, Yulin Chen, Jinglong Luo, Jiecong Wang, Hao Peng, Yan Kang, Xiaojin Zhang, Qi Hu, Chunkit Chan, Zenglin Xu, et al. Privacy in large language models: Attacks, defenses and future directions. *arXiv preprint arXiv:2310.10383*, 2023.
[40] Haoran Li, Dadi Guo, Wei Fan, Mingshi Xu, Jie Huang, Fanpu Meng, and Yangqiu Song. Multi-step jailbreaking privacy attacks on chatgpt. *arXiv preprint arXiv:2304.05197*, 2023.
[41] Xuechen Li, Florian Tramer, Percy Liang, and Tatsunori Hashimoto. Large language models can be strong differentially private learners. *arXiv preprint arXiv:2110.05679*, 2021.
[42] Yehuda Lindell. How to simulate it–a tutorial on the simulation proof technique. *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, 2017.
[43] Tianyi Liu, Xiang Xie, and Yupeng Zhang. Zkcnn: Zero knowledge proofs for convolutional neural network predictions and accuracy. CCS, 2021.
[44] Yinhan Liu. Roberta: A robustly optimized bertpretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.
[45] Wen-jie Lu, Zhicong Huang, Zhen Gu, Jingyu Li, Jian Liu, Cheng Hong, Kui Ren, Tao Wei, and WenGuang Chen. Bumblebee: Secure two-party inference framework for large transformers. NDSS, 2025.
[46] Nils Lukas, Ahmed Salem, Robert Sim, Shruti Tople, Lukas Wutschitz, and Santiago Zanella-Béguelin. Analyzing leakage of personally identifiable information in language models. SP. IEEE, 2023.
[47] Jinglong Luo, Yehong Zhang, Jiaqi Zhang, Xin Mu, Hui Wang, Yue Yu, and Zenglin Xu. Secformer: Towards fast and accurate privacy-preserving inference for large language models. *arXiv preprint arXiv:2401.00793*, 2024.
[48] Junming Ma, Yancheng Zheng, Jun Feng, Derun Zhao, Haoqi Wu, Wenjing Fang, Jin Tan, Chaofan Yu, Benyu Zhang, and Lei Wang. {SecretFlow-SPU}: A performant and {User-Friendly} framework for {Privacy-Preserving} machine learning. USENIXSec, 2023.
[49] Neo Wei Ming, Zhehui Wang, Cheng Liu, Rick Siow Mong Goh, and Tao Luo. Ma-bert: Towards matrix arithmetic-only bertinference by eliminating complex non-linear functions. ICLR , 2022.
[50] Ilya Mironov. Rényi differential privacy. CSF. IEEE, 2017.
[51] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. SP. IEEE, 2017.
[52] Jekaterina Novikova, Ondřej Dušek, and Verena Rieser. The e2e dataset: New challenges for end-to-end generation. *arXiv preprint arXiv:1706.09254*, 2017.
[53] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378*, 2018.
[54] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. Bolt: Privacy-preserving, accurate and efficient inference for transformers. *Cryptology ePrint Archive*, 2023.
[55] Nicolas Papernot and Thomas Steinke. Hyperparameter tuning with renyi differential privacy. ICLR , 2022.
[56] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *NeurIPS*, 2019.
[57] Natalia Ponomareva, Hussein Hazimeh, Alex Kurakin, Zheng Xu, Carson Denison, H Brendan McMahan, Sergei Vassilvitskii, Steve Chien, and Abhradeep Thakurta.

How to dp-fy ml: A practical guide to machine learning with differential privacy. *arXiv preprint arXiv:2303.00654*, 2023.
[58] Deevashwer Rathee, Dacheng Li, Ion Stoica, Hao Zhang, and Raluca Popa. Mpc-minimized secure llm inference. *arXiv preprint arXiv:2408.03561*, 2024.
[59] Manuel B Santos, Dimitris Mouris, Mehmet Ugurbil, Stanislaw Jarecki, José Reis, Shubho Sengupta, and Miguel de Vega. Curl: Private llms through wavelet-encoded look-up tables. *Cryptology ePrint Archive*, 2024.
[60] Andrew M Saxe, James L McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. ICLR , 2014.
[61] Youbang Sun, Zitao Li, Yaliang Li, and Bolin Ding. Improving lora in privacy-preserving federated learning. *arXiv preprint arXiv:2403.12313*, 2024.
[62] Qiaoyue Tang, Frederick Shpilevskiy, and Mathias Lécuyer. Dp-adambc: Your dp-adam is actually dp-sgd (unless you apply bias correction). volume 38, 2024.
[63] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.
[64] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.
[65] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *NeurIPS*, 30, 2017.
[66] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.
[67] Ming Wen, Jiaqi Zhu, Yuedong Xu, Yipeng Zhou, and Dingding Han. Differentially private federated low rank adaptation beyond fixed-matrix. *arXiv preprint arXiv:2507.09990*, 2025.
[68] Lukas Wutschitz, Huseyin A. Inan, and Andre Manoel. dp-transformers: Training transformer models with differential privacy. https://www.microsoft.com/en-us/research/project/dp-transformers, August 2022.
[69] Ashkan Yousefpour, Igor Shilov, Alexandre Sablayrolles, Davide Testuggine, Karthik Prasad, Mani Malek, John Nguyen, Sayan Ghosh, Akash Bharadwaj, Jessica Zhao, Graham Cormode, and Ilya Mironov. Opacus: User-friendly differential privacy library in PyTorch. *arXiv preprint arXiv:2109.12298*, 2021.
[70] Da Yu, Saurabh Naik, Arturs Backurs, Sivakanth Gopi, Huseyin A Inan, Gautam Kamath, Janardhan Kulkarni, Yin Tat Lee, Andre Manoel, Lukas Wutschitz, et al. Differentially private fine-tuning of language models. ICLR , 2021.
[71] Da Yu, Huishuai Zhang, Wei Chen, Jian Yin, and Tie-Yan Liu. Large scale private learning via low-rank reparametrization. ICML. PMLR, 2021.
[72] Longteng Zhang, Lin Zhang, Shaohuai Shi, Xiaowen Chu, and Bo Li. Lora-fa: Memory-efficient low-rank adaptation for large language models fine-tuning. *arXiv preprint arXiv:2308.03303*, 2023.
[73] Yuqing Zhu and Yu-Xiang Wang. Poission subsampled rényi differential privacy. ICML. PMLR, 2019.

A Adam BC

In Adam, computing the second moment on noisy gradients introduces a statistical bias, since the DP noise has a variance greater than 0. Following Alg. 4, the first moment m_t depends on the noise gradient $\tilde{g_\tau}$, whereas the second moment v_t depends on the square of the noisy gradient.

Computation of the bias term. Next, we compute the expectation of $\tilde{g_\tau}$ and $\tilde{g_\tau}^2$ showing why only v_t is biased. We recall that the noise $\psi_\tau \sim \mathcal{N}(0, \sigma^2 C^2)$ is statistically independent of the gradient g_τ, and for each training iteration is averaged over the batch size B.

For the first moment m_t:

$$E\left[\left(g_\tau + \frac{\psi_\tau}{B}\right)\right] = E\left[g_\tau\right] + \frac{1}{B}E\left[\psi_\tau\right] = E\left[g_\tau\right].$$

Instead, for the second moment v_t:

$$E\left[\left(g_\tau + \frac{\psi_\tau}{B}\right)^2\right] = E\left[g_\tau^2\right] + 2E\left[g_\tau \frac{\psi_\tau}{B}\right] + \frac{1}{B^2}E\left[\psi_\tau^2\right]$$

$$= E\left[g_\tau^2\right] + \frac{2}{B}E\left[g_\tau\right]E\left[\psi_\tau\right] + \frac{1}{B^2}E\left[\psi_\tau^2\right]$$

	No K_{FT}	$K_{FT} = 50$	$K_{FT} = 10$
FA-LoRA & AdamW	90.5	90.4	88.5
FA-LoRA & AdamBC	92.3	92.3	90.7

Table 7: Comparison of DPAdamBC **and** DPAdamW **accuracies on SST2 dataset with different capping thresholds.**

Input: Batch of samples $\mathcal{B} \in \mathbb{R}^{B\times m\times d}$, weight matrix $\mathbf{W} \in \mathbb{R}^{p\times d}$, gradient of the batched loss $\mathcal{L}$ w.r.t output $\mathbf{G} \in \mathbb{R}^{B\times m\times p}$.
Output: Clipped gradient $\bar{G}$.
Function GhostClipping$(\mathcal{B}, \mathbf{W}, \mathbf{G})$:
  foreach $\mathbf{X}_i \in \mathcal{B}$ **do**
    $\|\nabla_{\mathbf{W}} \mathcal{L}_i\|_F^2 = \mathrm{vec}(\mathbf{X}_i\mathbf{X}_i^\top)^\top \mathrm{vec}(\mathbf{G}_i\mathbf{G}_i^\top)$
    /* Compute the per-example scaling factor Γ_i */
    $\Gamma_i = \max(1, C/\|\nabla_{\mathbf{W}} \mathcal{L}_i\|^2)$
  /* Scale each loss and compute clipped gradient */
  $\bar{G} \leftarrow \nabla_{\mathbf{W}} \sum_{\mathbf{X}_i\in\mathcal{B}} \Gamma_i \mathcal{L}_i$
  return $\bar{G}$

Algorithm 5: Ghost clipping for Linear layers [41].

Input: Fine-tuning dataset D_{FT}, total dataset size N, batch size B.
Output: Random subsampled batch $\mathcal{B}$.
Function PoissonSampling $(D_{FT}, P = B/N)$:
  /* Initialize the subsampled batch */
  $\mathcal{B} \leftarrow \{\}$
  /* Sample each data point independently with probability P using a Uniform distribution */
  foreach $x_i \in D_{FT}$ **do**
    if $\mathrm{Uniform}(0, 1) < P$ **then**
      $\mathcal{B} \leftarrow \mathcal{B} \cup \{x_i\}$
  return $\mathcal{B}$

Algorithm 6: Poisson Subsampling for DP Fine-tuning

$$= E\left[g_\tau^2\right] + \frac{1}{B^2}\left(E\left[\psi_\tau\right]^2 + Var\left[\psi_\tau\right]\right) = E\left[g_\tau^2\right] + \frac{\sigma^2 C^2}{B^2}.$$

which results in the bias term $\frac{\sigma^2 C^2}{B^2}$ corrected in Alg. 4.

Evaluation. We compare the cleartext accuracy of fine-tuning with DPAdamBC and DPAdamW on the SST2 dataset with different capping thresholds. Tab. 7 shows that DPAdamBC consistently outperforms DPAdamW across all thresholds, achieving an average improvement of 2 pp in accuracy. This demonstrates the effectiveness of bias correction in DPAdamW's second moment estimation.

B Ghost Clipping

GhostClipping (Alg. 5) enables batched gradient computation and reduces memory overhead by avoiding the need to explicitly compute or store individual gradients. For a Linear layer with $\mathbf{W} \in \mathbb{R}^{p\times d}$, given an input $\mathbf{X} \in \mathbb{R}^{B\times m\times d}$, GhostClipping computes the squared Frobenius norm of the per-example gradient as $\|\nabla_W \mathcal{L}_i\|_F^2 = \mathrm{vec}(\mathbf{X}_i\mathbf{X}_i^\top)^\top \mathrm{vec}(\mathbf{G}_i\mathbf{G}_i^\top)$, where $\mathbf{G}_i \in \mathbb{R}^{B\times m\times p}$ is the gradient of the loss with respect to the output, and $\mathrm{vec}(\cdot)$ denotes the vectorization of a matrix. This approach reduces memory complexity to $O(Bm^2)$, compared to the naive method $O(Bpd)$ [41].

C Poisson Subsampling

Alg. 6 outlines the Poisson subsampling algorithm used in our experiments. Each data point in the fine-tuning dataset D_{FT} (size N) is independently sampled into the batch $\mathcal{B}$ with probability

$P = B/N$. The resulting batch size follows a Poisson distribution with mean $P \cdot N = B$.

D Non-Linear functions

Next, we discuss the MPC approximations for Softmax and GeLU evaluated on our preliminary experiments on SST2 to select the best-performing approximations balancing accuracy and efficiency.

D.1 Softmax Approximations

We evaluated various Softmax approximations to improve computational efficiency and numerical stability in MPC inference, testing them alongside GeLU from PyTorch [56] to isolate the impact of the Softmax approximation on the overall model accuracy.

Neural Network Approximation. Ming et al. [49] propose a neural network-based approximation for Softmax, which consists of two Linear layers with a ReLU activation in between. This method is computationally efficient, requiring only two matrix multiplications between secret shared inputs and cleartext weights of the network, along with a ReLU activation. However, this approximation yielded random accuracy results in our fine-tuning, likely because the neural network weights were tailored for model distillation and not fine-tuned for our specific use case.

Quadratic Approximation. MPCFormer [38] introduces a quadratic transformation of the Attention logits, followed by a normalization. Formally, for an input vector $\mathbf{x} \in \mathcal{R}^n$:

$$f(x) = \frac{(x+c)^2}{\sum_i (x_i + c)^2},$$

where c is a constant, set to 5 in the original work. Even though this approximation is computationally efficient, it differs significantly from the original Softmax function in terms of numerical values. This discrepancy leads to random accuracy results in our fine-tuning experiments, as the approximation was designed for model distillation and not fine-tuning.

Double ReLU Approximation. Proposed in [38], this method uses the ReLU activation to retain positive logits while discarding negative values, with normalization to ensure a valid probability distribution:

$$f(x) = \frac{\mathsf{ReLU}(x) + \iota/d}{\sum_i \mathsf{ReLU}(x_i) + \iota},$$

where ι is a small constant, and d is the input dimension. This approximation is more accurate than the quadratic approximation, but still not suitable for fine-tuning, since it was designed to be used in a model distillation scenario for inference.

D.2 GeLU Approximations

We tested several approximations of GeLU to balance accuracy and efficiency in MPC inference. First, we report the coefficients of the polynomial approximation in $\mathsf{GeLU}_{\mathsf{MPC}}$, and then we discuss the other approximations tested. We tested all the approximations alongside Softmax from PyTorch [56], to isolate the impact of the GeLU approximation on the overall model accuracy.

Coefficients for polyGeLU in $\mathsf{GeLU}_{\mathsf{MPC}}$. SPRINT leverages an optimized version of $\mathsf{GeLU}_{\mathsf{MPC}}$ [54] for GeLU approximation, which is the best performing approximation in our experiments, not only in terms of computational efficiency, but also in terms of accuracy and numerical stability. Here, we provide the polynomial coefficients used in $\mathsf{GeLU}_{\mathsf{MPC}}$:

$$\begin{aligned} g_0 &= 0.1444, \\ g_1 &= -0.7077, \\ g_2 &= 4.5703, \\ g_3 &= -8.1544, \\ g_4 &= 16.3823. \end{aligned}$$

Quadratic Approximation. MPCFormer [38] leverages a quadratic approximation (as for Softmax), also for GeLU, which was proposed to approximate the ReLU function by Chou et al. [10]. The approximation is computed as:

$$f(x) = 0.125x^2 + 0.5x + 0.25.$$

However, despite being computationally efficient, this approximation is too aggressive for fine-tuning, leading to random accuracy.

Tanh Approximation. Proposed by Google [27], this approximation is computed as:

$$f(x) = 0.5x\left(1 + \mathsf{Tanh}\left(\pi\left(x + 0.044715x^3\right)\right)\right),$$

where Tanh is defined as:

$$\mathsf{Tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

This approximation of GeLU provides reliable accuracy results. However, during MPC inference, it causes numerical instability due to the Tanh approximation implemented in CrypTen, which is numerically stable only for values in the range $[-250, 250]$.

HardTanh Approximation. To enhance numerical stability of the GeLU approximation based on Tanh [27], we replaced Tanh with HardTanh, resulting in the following approximation:

$$f(x) = 0.5x\left(1 + \mathsf{HardTanh}\left(\pi\left(x + 0.044715x^3\right)\right)\right).$$

where HardTanh is defined as:

$$\mathsf{HardTanh}(x) = \begin{cases} -1, & \text{if } x \le -1, \\ x, & \text{if } -1 < x < 1, \\ 1, & \text{if } x \ge 1. \end{cases}$$

HardTanh, implemented as a piecewise function, ensures numerical stability for all inputs and is more efficient than the Tanh-based approximation. However, it results in a 10 pp accuracy drop compared to the $\mathsf{GeLU}_{\mathsf{MPC}}$ approximation adopted in SPRINT.

E Security Analysis

We prove the security of SPRINT using the simulation paradigm [21], which ensures that each server learns nothing from the protocol execution beyond its own inputs and outputs. Specifically, for any adversary corrupting a server, there exists a simulator that can generate a view indistinguishable from the real protocol execution, given only the server's input and output. Thus, a corrupted server cannot distinguish between a real execution and a simulated one, guaranteeing security of the computation. We start by introducing notation for the simulation paradigm. Let $f = (f_1, f_2)$ be a probabilistic time functionality and let Π be a two-party protocol to realize f. The view of server S_i during an execution of π on inputs $x = (x_1, x_2)$ is denoted $\mathsf{view}_i^{\pi}(x_1, x_2)$. This view contains the party's input, its internal random tape, and all

Ideal Functionality	Input/Output Description	Proof
Multiplication $f_\times$	$\langle \mathbf{X}\rangle, \langle \mathbf{Y}\rangle \to \langle \mathbf{XY}\rangle$	[36, App. A]
Less-than $f_<$	$\langle x\rangle, \langle y\rangle$ or $y \to \langle 1_{x<y}(x,y)\rangle$	[36, App. A]
InvSqrt f_{InvSqrt}	$\langle x\rangle \to \langle 1/\sqrt{x}\rangle$	[36, App. A]
Tanh f_{Tanh}	$\langle x\rangle \to \langle \mathsf{Tanh}(x)\rangle$	[36, App. A]
Exponential $f_{\exp}$	$\langle x\rangle \to \langle \exp(x)\rangle$	[36, App. A]
Reciprocal $f_{\mathsf{Reciprocal}}$	$\langle x\rangle \to \langle 1/x\rangle$	[36, App. A]
MUX f_{MUX}	$\langle c\rangle, \langle x_0\rangle, \langle x_1\rangle \to \langle c(x_1 - x_0) + x_0\rangle$	[36, App. A]
FALoRA f_{FALoRA}	$\langle \mathbf{x}\rangle, \langle \mathbf{B}\rangle \to \langle \mathsf{FALoRA}(\mathbf{x})\rangle$	Sec. E.1
SoftmaxK f_{SoftmaxK}	$\langle \mathbf{x}\rangle \to \langle \mathsf{SoftmaxK}(\mathbf{x})\rangle$	Sec. E.3
$\mathsf{GeLU}_{\mathsf{MPC}}$ $f_{\mathsf{GeLU}_{\mathsf{MPC}}}$	$\langle x\rangle \to \langle \mathsf{GeLU}_{\mathsf{MPC}}(x)\rangle$	Sec. E.4
SoftCap f_{SoftCap}	$\langle x\rangle, \mathsf{K} \to \langle \mathsf{K}\cdot\mathsf{Tanh}(x/\mathsf{K})\rangle$	Sec. E.2
$\mathsf{LayerNorm}_{\mathsf{MPC}}$ $f_{\mathsf{LayerNorm}_{\mathsf{MPC}}}$	$\langle \mathbf{x}\rangle \to \langle \mathsf{LayerNorm}_{\mathsf{MPC}}(\mathbf{x})\rangle$	Sec. E.5

Table 8: Definitions of ideal functionalities from CrypTen (first half) and newly introduced by SPRINT.

Input: Servers hold shares of input $\langle \mathbf{X}\rangle$. Public parameters $\mathbf{W}_0, \mathbf{A}$. Servers hold shares of fine-tuned matrix $\langle \mathbf{B}\rangle$.
Output: Servers hold shares of output $\langle \mathbf{Y}\rangle$.
Protocol $\Pi_{\mathsf{FALoRA}}(\langle \mathbf{X}\rangle, \mathbf{W}_0, \mathbf{A}, \langle \mathbf{B}\rangle)$:
- /* Local, non-interactive computations */
- $\langle \mathbf{Y}_0\rangle_i \leftarrow \langle \mathbf{X}\rangle_i \cdot \mathbf{W}_0$
- $\langle \mathbf{X}_A\rangle_i \leftarrow \langle \mathbf{X}\rangle_i \cdot \mathbf{A}$
- /* Interactive MPC multiplication */
- $\langle \mathbf{Y}_{DP}\rangle \leftarrow \Pi_\times(\langle \mathbf{X}_A\rangle, \langle \mathbf{B}\rangle)$
- /* Final local addition */
- $\langle \mathbf{Y}\rangle_i \leftarrow \langle \mathbf{Y}_0\rangle_i + \langle \mathbf{Y}_{DP}\rangle_i$
- **return** $\langle \mathbf{Y}\rangle$

Algorithm 7: FALoRA Linear Layer Protocol, Π_{FALoRA}

messages it received. The joint output of both parties is denoted by $\mathrm{out}^\pi(x_1, x_2) = (\mathrm{out}_1^\pi(x_1, x_2), \mathrm{out}_2^\pi(x_1, x_2))$. Tab. 8 provides the ideal functionality for existing building blocks in CrypTen [36] and SPRINT protocols in the second half of the table. Note that all functionalities in Tab. 8 are deterministic, so we can use the following definition of security [42].

Definition E.1 ([42]). Let $f = (f_1, f_2)$ be functionality. We say that Π securely realizes f in the presence of a static semi-honest adversary if there exist probabilistic polynomial time simulators $\mathcal{S}_1, \mathcal{S}_2$ such that for any input (x, y):

$$\{(\mathcal{S}_1(x_1, f_1(x)))\}_{x_1} \equiv \{(\mathrm{view}_1^\pi(x))\}_{x_1}, \text{and}$$
$$\{(\mathcal{S}_2(x_2, f_2(x)))\}_x \equiv \{(\mathrm{view}_2^\pi(x))\}_x,$$

where $x_1, x_2 \in \{0,1\}^*$ such that $|x_1| = |x_2|$.

All functionalities in Tab. 8 introduced by SPRINT (i.e., FALoRA, SoftmaxK, $\mathsf{GeLU}_{\mathsf{MPC}}$, SoftCap, $\mathsf{LayerNorm}_{\mathsf{MPC}}$) are defined to compute the approximated version of the functions. Next, we provide the security proofs for the protocols introduced in SPRINT.

E.1 Security of FALoRA Linear Layer

Theorem E.2. *Suppose $\Pi_\times$ securely realizes $f_\times$. Then Π_{FALoRA} (Algorithm 7) securely realizes the functionality f_{FALoRA} against static semi-honest adversaries.*

Proof. Let x be an arbitrary input and $\Pi_\times$ securely realize the multiplication functionality $f_\times$. By definition $f_{\mathsf{FALoRA}}(x) \overset{c}{\equiv} \mathrm{out}^{\Pi_{\mathsf{FALoRA}}(x)}$. Since Π_{FALoRA} (Alg. 7) sequentially invokes local computations and calls $\Pi_\times$, by the sequential composition theorem [8], the security of the protocol follows from the security of the underlying multiplication protocol $\Pi_\times$. Hence, $\{(\mathcal{S}_i(x_i, f_i(x)))\}_x \overset{c}{\equiv} \{(\mathrm{view}_i^{\Pi_{\mathsf{FALoRA}}}(x))\}_x$ for $i \in \{1,2\}$. □

Input: Servers hold shares of input $\langle x\rangle$. Public capping threshold is K.
Output: Servers hold shares of capped output $\langle y\rangle$.
Protocol $\Pi_{\mathsf{SoftCap}}(\langle x\rangle, \mathsf{K})$:
- /* Local, non-interactive scaling */
- $\langle x'\rangle_i \leftarrow \langle x\rangle_i / \mathsf{K}$
- /* Interactive MPC Tanh computation */
- $\langle y'\rangle \leftarrow \Pi_{\mathsf{Tanh}}(\langle x'\rangle)$
- /* Final local, non-interactive scaling */
- $\langle y\rangle_i \leftarrow \mathsf{K} \cdot \langle y'\rangle_i$
- **return** $\langle y\rangle$

Algorithm 8: SoftCap Protocol, Π_{SoftCap}

Input: Servers hold shares of input vector $\langle \mathbf{x}\rangle$, guaranteed to be in $[-\mathsf{K}, \mathsf{K}]$. Public capping constant K.
Output: Servers hold shares of softmax output $\langle \mathbf{y}\rangle$.
Protocol $\Pi_{\mathsf{SoftmaxK}}(\langle \mathbf{x}\rangle, \mathsf{K})$:
- /* Local, non-interactive normalization */
- $\langle \mathbf{x}_{\mathrm{norm}}\rangle_i \leftarrow \langle \mathbf{x}\rangle_i - \mathsf{K}$
- /* Interactive MPC exponentiation */
- $\langle \mathbf{y}_{\mathrm{num}}\rangle \leftarrow \Pi_{\exp}(\langle \mathbf{x}_{\mathrm{norm}}\rangle)$
- /* Summation and interactive MPC reciprocal */
- $\langle \mathrm{sum}\rangle_i \leftarrow \sum_j \langle \mathbf{y}_{\mathrm{num}}\rangle_{i,j}$
- $\langle \mathbf{y}_{\mathrm{den}}\rangle \leftarrow \Pi_{\mathsf{Reciprocal}}(\langle \mathrm{sum}\rangle)$
- /* Final local multiplication */
- $\langle \mathbf{y}\rangle_i \leftarrow \langle \mathbf{y}_{\mathrm{num}}\rangle_i \cdot \langle \mathbf{y}_{\mathrm{den}}\rangle_i$
- **return** $\langle \mathbf{y}\rangle$

Algorithm 9: SoftmaxK Protocol, Π_{SoftmaxK}

E.2 Security of SoftCap

Theorem E.3. *Suppose Π_{Tanh} securely realizes f_{Tanh}. Then Π_{SoftCap} (Algorithm 8) securely realizes the functionality f_{SoftCap} against static semi-honest adversaries.*

Proof. Let x be an arbitrary input and Π_{Tanh} securely realize the hyperbolic tangent functionality f_{Tanh}. By definition $f_{\mathsf{SoftCap}}(x) \overset{c}{\equiv} \mathrm{out}^{\Pi_{\mathsf{SoftCap}}(x)}$. The protocol Π_{SoftCap} (Alg. 8) consists of local computations and a single call to the secure sub-protocol Π_{Tanh}, which computes the hyperbolic tangent. Since the local computations are secure, the security of the overall protocol follows from the security of the underlying Π_{Tanh} protocol by the sequential composition theorem [8]. Hence, $\{(\mathcal{S}_i(x_i, f_i(x)))\}_x \overset{c}{\equiv} \{(\mathrm{view}_i^{\Pi_{\mathsf{SoftCap}}}(x))\}_x$ for $i \in \{1,2\}$. □

E.3 Security of SoftmaxK

Theorem E.4. *Suppose $\Pi_{\exp}$, and $\Pi_{Reciprocal}$ securely realize $f_{\exp}$ and $f_{Reciprocal}$, respectively. Then $\Pi_{SoftmaxK}$ (Algorithm 9) securely realizes the functionality $f_{SoftmaxK}$ against static semi-honest adversaries.*

Proof. Let $\mathbf{x}$ be an arbitrary input vector and the protocols $\{\Pi_{\exp}, \Pi_{\mathsf{Reciprocal}}\}$ securely realizes $\{f_{\exp}, f_{\mathsf{Reciprocal}}\}$. By definition, $f_{\mathsf{SoftmaxK}}(\mathbf{x}) \overset{c}{\equiv} \mathrm{out}^{\Pi_{\mathsf{SoftmaxK}}(\mathbf{x})}$. Since the protocol Π_{SoftmaxK} (Alg. 9) consists of local computations and calls to secure sub-protocols $\Pi_{\exp}$ and $\Pi_{\mathsf{Reciprocal}}$, by sequential composition theorem [8], the security of the protocol follows from the security of the underlying protocols. Hence, $\{(\mathcal{S}_i(x_i, f_i(x)))\}_x \overset{c}{\equiv} \{(\mathrm{view}_i^{\Pi_{\mathsf{SoftmaxK}}}(x))\}_x$ for $i \in \{1,2\}$. □

Input: Servers hold shares of input $\langle x \rangle$. Public threshold is 2.7.
Output: Servers hold shares of GeLU_{MPC} output $\langle y \rangle$.
Protocol $\Pi_{\text{GeLU}_{\text{MPC}}}(\langle x \rangle)$:

```
/* Pre-compute absolute value                                  */
⟨|x|⟩ ← Π×(⟨x⟩, 1 − 2 · Π<(⟨x⟩, 0))
// Requires A2B/B2A conversions
/* Compute polynomial approximation and ReLU                   */
⟨y_poly⟩ ← Π_polyGeLU(⟨x⟩, ⟨|x|⟩)
Each server S_i locally computes ⟨y_relu⟩_i ← (⟨x⟩_i + ⟨|x|⟩_i)/2
/* Select the correct output based on input range              */
⟨b_comp⟩ ← Π>(⟨|x|⟩, 2.7)
⟨y⟩ ← Π_MUX(⟨b_comp⟩, ⟨y_poly⟩, ⟨y_relu⟩);
return ⟨y⟩
```

Algorithm 10: GeLU_{MPC} Protocol, $\Pi_{\text{GeLU}_{\text{MPC}}}$

Input: Servers hold shares of input $\langle \mathbf{x} \rangle$, weights $\langle W \rangle$, bias $\langle b \rangle$. Public threshold T, scaling factor β.
Output: Servers hold shares of normalized output $\langle y \rangle$.
Protocol $\Pi_{\text{LayerNorm}_{\text{MPC}}}(\langle \mathbf{x} \rangle, \langle W \rangle, \langle b \rangle, T, \beta)$:

```
/* Compute Mean and Variance                                   */
⟨μ⟩_i ← 1/|⟨x⟩_i| Σ_{⟨x⟩_i ∈ ⟨x⟩_i} ⟨x⟩_i
⟨x_c⟩_i ← ⟨x⟩_i − ⟨μ⟩_i
⟨x_c^2⟩ ← Π×(⟨x_c⟩, ⟨x_c⟩)
⟨ν⟩_i ← 1/|⟨x⟩_i| Σ_{⟨x_c^2⟩_i ∈ ⟨x_c^2⟩_i} ⟨x_c^2⟩_i
/* Piecewise Inverse Square Root                               */
⟨b_comp⟩ ← Π≤(⟨ν⟩, T)
⟨ν_unscaled_inv⟩ ← Π_InvSqrt(⟨ν⟩)
⟨ν_scaled_in⟩ ← ⟨ν⟩/β
⟨ν_scaled_out⟩ ← Π_InvSqrt(⟨ν_scaled_in⟩)
⟨ν'_scaled_out⟩ ← ⟨ν_scaled_out⟩/√β
⟨ν'⟩ ← Π_MUX(⟨b_comp⟩, ⟨ν'_scaled_out⟩, ⟨ν_unscaled_inv⟩)
/* Final Normalization                                         */
⟨x_norm⟩ ← Π×(⟨x_c⟩, ⟨ν'⟩)
⟨y_temp⟩ ← Π×(⟨x_norm⟩, ⟨W⟩)
⟨y⟩_i ← ⟨y_temp⟩_i + ⟨b⟩_i
return ⟨y⟩
```

Algorithm 11: $\text{LayerNorm}_{\text{MPC}}$ Protocol, $\Pi_{\text{LayerNorm}_{\text{MPC}}}$

E.4 Security of GeLU

Theorem E.5. *Suppose the protocols for absolute value ($\Pi_{|\cdot|}$), polynomial evaluation (Π_{polyGeLU}), greater-than ($\Pi_{>}$), and MUX (Π_{MUX}) securely realize their respective ideal functionalities, i.e., $f_{|\cdot|}$, f_{polyGeLU}, $f_{>}$, and f_{MUX}. Then $\Pi_{\text{GeLU}_{\text{MPC}}}$ (Algorithm 10) securely realizes $f_{\text{GeLU}_{\text{MPC}}}$ against static semi-honest adversaries.*

Proof. Let x be an arbitrary input, and $\{\Pi_{\text{polyGeLU}}, \Pi_{>}, \Pi_{\text{MUX}}\}$ securely realize $\{f_{\text{polyGeLU}}, f_{>}, f_{\text{MUX}}\}$. By definition, $f_{\text{GeLU}_{\text{MPC}}}(x) \overset{c}{\equiv} \text{out}^{\Pi_{\text{GeLU}_{\text{MPC}}}(x)}$. Since the protocol $\Pi_{\text{GeLU}_{\text{MPC}}}$ (Alg. 10) consists of local computations and sequential calls to secure sub-protocols $\{\Pi_{\text{polyGeLU}}, \Pi_{>}, \Pi_{\text{MUX}}\}$, the security of $\Pi_{\text{GeLU}_{\text{MPC}}}$ follows from the security of the underlying protocols by the sequential composition theorem [8]. Thus, $\{(\mathcal{S}_i(x_i, f_i(x)))\}_x \overset{c}{\equiv} \{(\text{view}_i^{\Pi_{\text{GeLU}_{\text{MPC}}}}(x))\}_x$ for $i \in \{1, 2\}$. □

E.5 Security of LayerNorm

Theorem E.6. *Suppose $\Pi_{\times}$, $\Pi_{\leq}$, Π_{InvSqrt}, and Π_{MUX} securely realize $\{f_{\times}, f_{\leq}, f_{\text{InvSqrt}}, f_{MUX}\}$. Then $\Pi_{\text{LayerNorm}_{\text{MPC}}}$ (Algorithm 11) securely realizes $f_{\text{LayerNorm}_{\text{MPC}}}$ against static semi-honest adversaries.*

Proof. Let x be an arbitrary input, and $\{\Pi_{\times}, \Pi_{\leq}, \Pi_{\text{InvSqrt}}, \Pi_{\text{MUX}}\}$ securely realize $\{f_{\times}, f_{\leq}, f_{\text{InvSqrt}}, f_{\text{MUX}}\}$. By definition, $f_{\text{LayerNorm}_{\text{MPC}}}(x) \overset{c}{\equiv} \text{out}^{\Pi_{\text{LayerNorm}_{\text{MPC}}}(x)}$. The protocol $\Pi_{\text{LayerNorm}_{\text{MPC}}}$ (Alg. 11) consists of local computations and calls to secure sub-protocols $\Pi_{\times}$, $\Pi_{\leq}$, Π_{InvSqrt}, and Π_{MUX}. By the sequential composition theorem [8], the security of the protocol follows from the security of the underlying protocols. Hence, $\{(\mathcal{S}_i(x_i, f_i(x)))\}_x \overset{c}{\equiv} \{(\text{view}_i^{\Pi_{\text{LayerNorm}_{\text{MPC}}}}(x))\}_x$ for $i \in \{1, 2\}$. □

F Evaluation

This section expands our evaluation (Sec. 5) by providing the complete list of hyperparameters, an evaluating the impact of different LoRA **A** initialization techniques on the accuracy, the effect of random seeds and SoftCap on cleartext inference accuracy, and a detailed comparison of SPRINT and BOLT in terms of runtime and communication for decomposed Linear layers. Furhtrmore, we evaluate how SPRINT scales to larger models (i.e., RoBERTa-large) and more than two parties, a setting not explored by related works.

F.1 Hyperparameters Search

Tab. 9 reports the hyperparameters values tested for each dataset for DP fine-tuning.

F.2 LoRA **A** initialization techniques

Next, we detail the initialization techniques tested for LoRA **A**.

Initialization techniques. The Kaiming initialization is designed for layers with ReLU activations. It initializes weights from a distribution with zero mean and a variance of $2/n$ (either Uniform or Normal), where n is the number of input units [26]. This approach compensates for the ReLU behavior of outputting zero for negative inputs, which effectively reduces the number of active neurons. By using a higher variance, Kaiming initialization ensures that the variance of activations remains consistent across layers, preventing vanishing activations. In our case, the GeLU activation function is used, which behaves similarly to ReLU since it also outputs values close to zero for sufficiently negative inputs (approximately those less than -2.7 based on our approximation). The Xavier initialization aims to maintain the scale of gradients across layers by initializing weights from a distribution with zero mean and a variance of $2/(n_{\text{in}} + n_{\text{out}})$ (either Uniform or Normal), where n_{in} and n_{out} are the number of input and output units, respectively [20]. The Normal initialization is more suited for layers with Sigmoid or Tanh activations and initializes weights from a normal distribution with zero mean and a variance of $1/n$, where n is the number of input units [37]. The Orthogonal initialization, on the other hand, initializes weight matrices as orthogonal matrices, ensuring uncorrelated weights with unit variance. This property can help preserve the variance of activations across layers, potentially improving stability during training [60].

Impact on accuracy. We found that the initialization of the frozen **A** matrix directly affects SPRINT accuracy, with variations of up to 2.1 pp. To evaluate the effect of **A** initialization on accuracy, we tested several techniques: Kaiming [26], Xavier [20], Normal [37], and Orthogonal [60], using uniform and normal distributions for Kaiming and Xavier. Tab. 10 reports our evaluation on SST2. Here,

Parameter	Component / Phase	Description	Tested Values
Batch size (B)	DP fine-tuning	Number of samples per training step out of N samples	1024, 2048
Subsampling rate (q)	DP PoissonSampling	Probability of a sample being included in a batch.	B/N
Learning rate (η)	DP fine-tuning	Step size for the optimizer.	$5\times10^{-3}, 1\times10^{-3}, 5\times10^{-4}, 1\times10^{-4}, 5\times10^{-5}$
Epochs (E)	DP fine-tuning	Total number of steps over the training dataset size.	10
Clipping threshold (C)	DP fine-tuning	L2 norm limit for per-sample gradients.	0.1, 1, 10
Weight decay	DP fine-tuning	L2 regularization coefficient to prevent overfitting.	0.1, 0.01, 0.001
Privacy budget (ϵ)	DP Definition	Target epsilon (ϵ) for the privacy guarantee.	8
Failure probability (δ)	DP Definition	Target delta (δ) for approximate DP.	10^{-5} (SST,QNLI), 10^{-6} (QQP,MNLI)
Noise variance (σ^2)	DP Mechanism	DP noise variance for Gaussian noise	Analytically computed from ϵ, δ, E, q by Opacus [69]
FALoRA rank (r)	FALoRA	Rank r of the low-rank adapter matrices.	8, 16, 32
FALoRA alpha (α)	FALoRA	Scaling factor α for the FALoRA update (α/r).	$r, 2r$
FALoRA layers	FALoRA	Specific linear layers where FA-LoRA adapters are applied.	All Linear, all Linear except $\text{Linear}_{\mathbf{K}}$, $\{\text{Linear}_{\mathbf{Q}}, \text{Linear}_{\mathbf{V}}, \text{Linear}_{\text{Att}}\}$
A Initializations	FALoRA	Initialization method for the FALoRA A matrix.	Kaiming Uniform/Normal, Xavier Uniform/Normal, Orthogonal, Normal
Capping (K_{FT})	SoftCap	Capping threshold for Attention logits.	None, 50, 10

Table 9: Set of hyperparameters tested in fine-tuning for each dataset.

Seed	Kaiming Uniform	Kaiming Normal	Xavier Uniform	Xavier Normal	Normal	Orthogonal
0	92.2	92.1	90.7	90.2	92.1	91.9
Rnd	91.1 ± 0.3	91.3 ± 0.2	90.1 ± 0.2	90.5 ± 0.2	91.2 ± 0.2	91.5 ± 0.2

Table 10: Cleartext accuracy (in %) on SST2 for different FALoRA A initialization, comparing fine-tuning with fixed seed (0) and the average over 10 random seeds (rnd).

Dataset	K_{FT} /K_{INF} None/None	None/50	None/10	50/50	50/10	10/10
SST2	92.2	92.2 (0)	90.0 (2.2)	92.3	90.0 (2.3)	90.3
MNLI	78.6	78.5 (0.1)	71.1 (7.5)	78.6	71.7 (6.9)	77.2
QNLI	85.2	85.1 (0.1)	74.2 (11)	84.8	70.6 (14.2)	81.5
QQP	82.7	82.8 (+0.1)	75.1 (7.6)	82.7	75.9 (6.8)	79.6

Table 11: Cleartext accuracy ($\mathcal{M}^A_{\text{FT}}$ with θ_{DP}) with different combinations of K_{FT}/K_{INF} for fine-tuning and inference. In parentheses, the accuracy gap (in pp) compared to $\text{K}_{\text{FT}} = \text{K}_{\text{INF}}$ (Tab. 1, Sec. 5.1).

we discuss the results with fixed seed (seed = 0), and consider random seeds later. Kaiming Uniform performs best, followed closely by Kaiming Normal and Normal (both within 0.2 pp). In contrast, Xavier shows the worst performance, with accuracy drops of up to 2.1 pp. We extended this evaluation (with seed=0) also on QQP, QNLI, MNLI, finding that the best initializations are either Normal or Orthogonal. For example, on QNLI, Orthogonal performs best with 0.5 pp accuracy improvement over Kaiming Uniform. These results confirm the impact of FALoRA **A** initialization, and shows the best initialization is dataset-dependent.

F.3 Practical Deployment Considerations

A common practice in literature [41, 71] is to replace Poisson subsampling with shuffling for computational efficiency. However, shuffling processes all samples and lacks Poisson's theoretical guarantees [3]. Specifically, shuffling ensures each sample in the dataset is processed exactly once per epoch. In contrast, with Poisson subsampling, a sample might be selected multiple times or not at all within a single epoch which is essential for DP guarantees. Additionally, existing works [41, 71] fix the PRNG seed for reproducibility. However, the seed influences parameter initialization, DP noise, and subsampling, which affect model convergence leading to different local minima [30]. To address these limitations, we perform fine-tuning with Poisson subsampling (Sec. 5.1) and evaluate the impact of different random seeds on model accuracy.

Evaluation. We assess the impact of Poisson subsampling with random seeds on model accuracy using the SST2 dataset. Poisson subsampling offers stricter privacy guarantees than shuffling, but random seeds may introduce accuracy variability, since the seed alters the data sampled during fine-tuning. Furthermore, the seed is also responsible for the initialization of the **A** matrix and the DP noise. To evaluate SPRINT under these conditions, we fine-tuned RoBERTa with 10 random seeds, and tested different LoRA **A** initializations. Tab. 10 shows an accuracy drop between fixed and random seeds ranging from 0.4 pp to 2.1 pp across initializations. While Kaiming Uniform achieves the highest fixed-seed accuracy (92.2%), its accuracy drops by 1.1 pp under random seeds. Instead, Orthogonal performs best with random seeds, showing only a 0.4 pp drop compared to its fixed-seed counterpart. These results suggest that relying on the best-performing initialization with fixed seeds may not yield optimal performance under random seeds and the best initialization for practical deployment should be selected based on its resistance to seed variability.

F.4 Cleartext Inference Capping

Tab. 11 presents the cleartext accuracy of $\mathcal{M}^A_{\text{FT}}$ under various SoftCap configurations during fine-tuning (K_{FT}) and inference (K_{INF}). Applying $\text{K}_{\text{INF}} = 50$ during inference only results in, at most, a 0.1 pp accuracy drop compared to non-capped fine-tuning. Additionally, fine-tuning with $\text{K}_{\text{FT}} = 50$ achieves similar accuracy to non-capped fine-tuning when combined with $\text{K}_{\text{INF}} = 50$ during inference. In contrast, using $\text{K}_{\text{INF}} = 10$ during inference only results in a significant accuracy degradation (up to 14.2 pp on QNLI) when combined with $\text{K}_{\text{FT}} = 50$ or non-capped fine-tuning. However, fine-tuning with $\text{K}_{\text{FT}} = 10$ reduces the accuracy gap to a smaller margin (up to 3.8 pp on QNLI) compared to non-capped fine-tuning.

Framework	Runtime (s)			Communication	
	LAN	WAN_R	WAN_G	GB	Rounds
SHAFT [31]	140.88	970.15	1836.88	28.46	2935
SPRINT	95.90	735.22	1438.22	17.4	3649

Table 12: E2E cost of private inference on RoBERTa-large in different network settings.

N parties	Runtime (s)			Communication	
	LAN	WAN_R	WAN_G	GB	Rounds
3	85.79	875.50	1710.94	14.54	5728
4	118.91	1285.96	2563.94	20.8	14320

Table 13: SPRINT E2E per-sample cost of private inference on RoBERTa-base for more than 2 parties.

F.5 Scaling SPRINT

To evaluate how SPRINT scales to larger models and more than two parties, we conduct two additional experiments. First, we evaluate SPRINT on RoBERTa-large, which has 2× more Encoder layers than RoBERTa-base. Second, we evaluate SPRINT with more than two parties, a setting not explored by related works [24, 31, 45, 54].

Scaling SPRINT to RoBERTa-large. For the first experiment, we evaluate SPRINT's performance on RoBERTa-large and compare it with SHAFT. As shown in Tab. 12, scaling from RoBERTa-base to RoBERTa-large increases communication by 2.7× and rounds by 1.6×. SPRINT's runtime increases by 2.9× in LAN, 2.4× in WAN_R, and 2.3× in WAN_G compared to the RoBERTa-base baseline. Despite this expected overhead, SPRINT remains faster than SHAFT, i.e., from 1.3× in WAN_G to 1.4× in LAN, and reduces communication by 1.6× despite requiring 1.2× more rounds. This shows that SPRINT's optimizations scale effectively to larger models, maintaining a performance advantage over SHAFT.

Scaling SPRINT to more than 2 parties. For the second experiment, we evaluate SPRINT's performance with 3 and 4 parties on RoBERTa-base. As shown in Tab. 13, the runtime increases significantly, i.e., from 2.6× in LAN to 2.8× in WAN_G for 3 parties, and from 3.6× in LAN to 4.2× in WAN_G for 4 parties compared to the 2-party baseline. The communication also increases, i.e., from 2.2× for 3 parties to 3.2× for 4 parties. Notably, also the number of communication rounds increases significantly, i.e., from 3.1× for 3 parties to 7.9× for 4 parties. These results confirm the effectiveness of CrypTen in 2-party settings, but also highlight the scalability challenges of MPC protocols in multi-party settings. The focus of this work is on the two-party setting, but we note that custom protocols for 3 and 4 parties also exist, e.g., [15, 25].