Private and Verifiable Outsourcing of Open-Weight LLM Inference

Kanav Gupta[1,2], Jonathan Katz[2], and Ian Miers[1]

[1] University of Maryland {kanav,imiers}@umd.edu
[2] Google jkatz2@gmail.com

Abstract. Open-weight models allow clients to run LLMs locally, thus keeping their data private from untrusted providers. However, running large models requires massive hardware and storage resources (especially challenging on resource-constrained devices like smartphones), limiting local execution to smaller models. This leaves clients with a frustrating compromise: settle for a less-capable model that can be run locally, or sacrifice privacy by sending queries to an external server.

We present an efficient protocol that allows a client to *privately* and *verifiably* outsource LLM inference of an open-weight model to a pair of malicious (but non-colluding) servers. Privacy implies that neither server learns anything about the client's queries. At the same time, the client can verify the claimed result using information posted by the model owner along with the model weights.

Compared to prior state-of-the-art for private LLM inference (SIGMA, PETS' 24)—which does not provide verifiability—our protocol is ≈11–14× faster while imposing no overhead at the servers (beyond the cost of inference in the original model). Our protocol also scales to larger models not supported by prior work: for example, with our protocol a client can run the Llama 2-70B model using just 179 MB of local storage (instead of the 140 GB required to run the model locally).

1 Introduction

Users continually want access to the latest and most powerful AI models. Typically, those models are private, and the only way for users to access such a model is to upload their input (query) to a service provider hosting the model and then receive the result. This is detrimental for user privacy, as the entire history of users' queries is revealed to the service provider. It is also not verifiable, in the sense that users cannot be assured of getting results from the model they intended to query. Indeed, an unethical provider could use a less performant model than the one the user requested [24], thereby overcharging the user for a substandard service.

Open-weight LLMs like Llama [26] and Gemma [20] can be downloaded by users and executed locally, thus addressing the privacy and verifiability concerns raised above. Due to users' resource constraints, however, local execution is often limited to smaller, less-capable models. Users' constraints are related to both

computational power (i.e., local inference would take too long or use too much battery power) and *storage*, (i.e., lacking sufficient memory to run large models or even store their weights). In particular, although open-weight models with up to 1 trillion parameters are available [22], mobile devices are only capable of running models with ≈3 billion parameters, and then only when using aggressive quantization that degrades output quality [18].

1.1 Our Contributions

We show how to give users the best of both worlds by designing a protocol for *private* and *verifiable* outsourcing of LLM inference. It allows a lightweight client to delegate the costs (both storage and computation) of LLM inference to two untrusted (but non-colluding) servers while ensuring both the privacy of the client's queries and verifiability of the results. Our protocol ensures not only that the necessary client resources are small, but that there is essentially *no* additional overhead for the servers.

Our protocol relies on two observations: (1) most of the computational and storage costs of inference are due to the linear layers of neural networks. (E.g., for models with 6.7B parameters, 95% of the storage cost and 85% of the computational cost are due to linear layers [5].) At the same time, (2) the overhead of privacy-preserving cryptography for linear operations is small—indeed, in the setting we focus on, the overhead is negligible. This motivates the high-level structure of our protocol, in which the (expensive) computation of the linear layers is delegated to the servers, while the (cheap) computation of the non-linear layers is done locally by the client. In further detail:

1. The core of our protocol is a technique for private delegation of *matrix-vector* products. Here, a client holds a (private) vector $\mathbf{x}$ and wants to compute $\mathbf{Mx}$ for a public matrix $\mathbf{M}$ that is too large for the client to store locally. This is exactly what is needed for the client to outsource computation of the linear layers to the servers.
2. We also show how the client can efficiently verify matrix-vector products. For a matrix of dimension $m \times n$, the client needs to store $O(m+n)$-size verification information; it can then verify the result using $O(m+n)$ work. The client can also use *batch verification* to verify multiple executions of the matrix-vector product protocol, in some cases at a cost essentially independent of the number of executions and in the general case with cost sublinear in the number of executions.

   Our verification procedure is of independent interest, and could be used in a non-private version of our protocol involving only a single server.
3. We implement our protocol, apply multiple optimizations, and run it on two GPU-enabled servers. We achieve a speed of 5.2 sec/token for the Llama 2-70B model on a client with just 2 vCPU and 4GB of RAM, while requiring only 179 MB of client-side storage; local execution of this model would require 140GB of storage and industrial GPUs to get any usable performance.

Compared to the state-of-the-art protocol SIGMA [10], our protocol is 11–14× faster, while additionally providing verifiability. Our protocol also requires no server-side overhead, whereas SIGMA would require servers to store roughly 2GB of cryptographic material per token for the Llama 2-7B model and perform significantly more computation.

1.2 Related Work

Generic solutions. It is possible to construct a protocol for private outsourcing using generic cryptographic tools. For example, one could use fully homomorphic encryption [3,28] even with only a single untrusted server; verifiability could be added using SNARKs. Alternately, one could have the client share its input between two (or more) untrusted servers, and then have the servers run a generic protocol for secure two-party computation (2PC); verifiability in this case is more difficult to achieve. Such solutions would be efficient for the client but prohibitively inefficient in terms of both computation and memory overhead—to the point of being unimplementable for large models—for the server(s).

Tailored solutions. Many prior works have designed tailored 2PC protocols for private (but not verifiable) inference [21,27,25,17,2,16,13,8,15,23,11,10,9]. All assume a client-server setting where the goal is to preserve privacy of both the client's query and the model held by the server, and the client's work is greater than the work required for the client to evaluate the model locally; nevertheless, they could be adapted to our setting by having two servers emulate the roles of the parties in the 2PC protocol, and having a client share its query between those servers. All these solutions incur a large number of rounds and high communication complexity. For instance, one state-of-the-art protocol [25] uses 0.4 KB of communication *per ReLU activation function*; a single token requires 2 million activation-function calls in larger 70B models.

1.3 Organization

We give a brief overview of our protocol in Section 2. In Section 3 we review some notation and preliminaries, and then in Section 4 we provide a detailed description of our protocol. In Section 5, we discuss the implementation of our protocol (including several optimizations), its experimental evaluation, and a comparison with prior state-of-the-art.

2 Protocol Overview

Fix some open-weight neural network that the client wants to use for inference. As mentioned earlier, the overall structure of our protocol has the client compute non-linear layers of the network locally, and delegate the computation of the linear layers to the two untrusted servers. (We also handle delegation of the initial embedding layer but omit a discussion from this overview; see Section 4.3.)

Since computation at the linear layers is dominated by computing the product of a public matrix $\mathbf{M}$ with a private input $\mathbf{x}$, we describe how to privately and verifiably outsource such a computation.

In the base network, the entries of $\mathbf{M}$ and $\mathbf{x}$ are floating-point numbers. Before running our protocol, the client and servers scale and truncate the entries of $\mathbf{M}$ (and the client does so for $\mathbf{x}$) so the entries are all integers; we then compute the matrix-vector product modulo $N = 2^{64}$ which is essentially a fixed-point computation. (We choose a scaling factor so that overflow modulo N does not occur.) We observe little accuracy loss from the truncation (which corresponds to a small loss of precision); see Table 1.

In the description that follows, we simply assume that $\mathbf{M} \in \mathbb{Z}_N^{m \times n}$ and $\mathbf{x} \in \mathbb{Z}_N^n$, and omit writing "modulo N" for the arithmetic operations when clear.

Privacy. Private outsourcing of a matrix-vector product can be done efficiently using secret sharing. The client begins by generating a 2-of-2 secret sharing of $\mathbf{x}$, i.e., it chooses uniform $\mathbf{x}^{(0)} \in \mathbb{Z}_N^n$ and sets $\mathbf{x}^{(1)} = \mathbf{x} - \mathbf{x}^{(0)} \bmod N$, so that $\mathbf{x}^{(0)} + \mathbf{x}^{(1)} = \mathbf{x} \bmod N$ but both $\mathbf{x}^{(0)}$ and $\mathbf{x}^{(1)}$ on their own leak no information about $\mathbf{x}$. The client then sends $\mathbf{x}^{(0)}$ to the first server and $\mathbf{x}^{(1)}$ to the second server; each server responds with $\mathbf{y}^{(b)} = \mathbf{M}\mathbf{x}^{(b)}$. The client computes

$$\mathbf{y} = \mathbf{y}^{(0)} + \mathbf{y}^{(1)} = \mathbf{M}\mathbf{x}^{(0)} + \mathbf{M}\mathbf{x}^{(1)} = \mathbf{M}\mathbf{x}.$$

It can then re-scale $\mathbf{y}$ and proceed to locally compute the next (non-linear) layer.

Note that the client's work and memory overhead is negligible, and each server's work/memory is the same as for performing inference using the original model.

Adding verifiability. The above ensures privacy of the client's input, but provides no protection against a server that returns an incorrect result. We thus provide a mechanism for the client to verify a claim that $\mathbf{y} = \mathbf{M}\mathbf{x}$. (This could be applied individually to each server's response, or once to the final result. In the latter case the client would learn that an error occurred but would not know which server caused the error unless it performed a second check.) Our mechanism extends the classic algorithm of Freivalds [6], which we recall here.

Freivalds's algorithm allows a client to verify a claim that $\mathbf{Y} = \mathbf{M}\mathbf{X} \bmod p$ (where p is a large prime and $\mathbf{X}, \mathbf{Y}$ are matrices) more cheaply than computing $\mathbf{M}\mathbf{X}$ on its own. Specifically, the client chooses a uniform vector $\mathbf{r}$ and checks that $\mathbf{r}^T\mathbf{Y} = (\mathbf{r}^T\mathbf{M})\mathbf{X}$. This check involves three matrix-vector products, which are cheaper than a single matrix-matrix multiplication. A correct result is always accepted, while an incorrect result is rejected except with probability $1/p$.

There are several issues that must be overcome in applying this idea to our setting: First, Freivalds's algorithm requires working modulo a prime p, whereas we work modulo a composite number N. (Although we could modify our protocol to work modulo a prime $p > N$, doing so would be less efficient especially when using GPUs. Note that we cannot work purely over the integers because of the secret sharing done by the client.) However, since we ensure no overflow occurs anyway, the result $\mathbf{M}\mathbf{x}$ is the same regardless of whether it is computed modulo N or modulo $p > N$.

More problematic is that Freivalds's algorithm does not save computation when applied to verifying the matrix-vector multiplication $\mathbf{y} = \mathbf{M}\mathbf{x}$. Moreover, it requires the client to know $\mathbf{M}$, which would partly defeat the point of delegation in our setting where we are trying to reduce the client's memory requirements. A natural attempt that resolves both problems is for the model owner to choose $\mathbf{r}$, compute $\mathbf{s}^T = \mathbf{r}^T \cdot \mathbf{M}$, and publish $\mathbf{s}$ along with $\mathbf{M}$; the client could then store $\mathbf{r}, \mathbf{s}$ and check that $\mathbf{r}^T\mathbf{y} = \mathbf{s}^T\mathbf{x}$. Unfortunately, this is not secure since Freivalds's algorithm requires that $\mathbf{r}$ be chosen *after* the claimed result $\mathbf{y}$ is provided.

We resolve this issue by relying on cryptographic techniques. Let $\mathbb{G}$ be a group of prime order $p > N$ in which the discrete-logarithm assumption holds (concretely, we use an elliptic-curve group), and let $g \in \mathbb{G}$ be a fixed generator. The model owner samples a uniform vector $\mathbf{r}$, computes $\mathbf{s}^T = \mathbf{r}^T\mathbf{M} \bmod p$, and publishes $g^{\mathbf{r}} = (g^{r_1}, \ldots, g^{r_m})$ and $g^{\mathbf{s}} = (g^{s_1}, \ldots, g^{s_n})$. The client can then run the verification algorithm *in the exponent*. Specifically, to verify that $\mathbf{y} = \mathbf{M}\mathbf{x}$ the client checks that

$$\prod (g^{r_i})^{y_i} \stackrel{?}{=} \prod (g^{s_i})^{x_i}.$$

This can be rewritten as a single multi-exponentiation of size $m + n$, which can be computed at roughly the cost of $(m + n)/\log(m + n)$ exponentiations. The client stores only $m + n$ group elements, which require significantly less memory than the mn entries of $\mathbf{M}$. The servers do not perform any additional work, and do not even need to be aware of the verification information $g^{\mathbf{r}}, g^{\mathbf{s}}$. (Though security holds even if they know those values.)

We can further optimize the client's work in our protocol by using *batch verification*, so that instead of checking the result of each linear layer the client performs only a single check at the very end. We consider two cases. To verify k matrix-vector multiplications $\{\mathbf{y}_i = \mathbf{M}\mathbf{x}_i\}_{i=1}^k$ using the same matrix $\mathbf{M}$ (as in text-generation applications where the same matrices are used to generate each token), the client chooses uniform $\alpha_1, \ldots, \alpha_k \in \mathbb{Z}_p$ and uses the idea from above to verify the *single* matrix-vector multiplication $\sum_{i=1}^k \alpha_i \cdot \mathbf{y}_i = \mathbf{M} \cdot \sum_{i=1}^k \alpha_i \cdot \mathbf{x}_i \bmod p$. Since group operations dominate operations in $\mathbb{Z}_p$, the cost is essentially equivalent to verifying a single matrix-vector multiplication. In the general case when we want to verify $\{\mathbf{y}_i = \mathbf{M}_i\mathbf{x}_i\}_{i=1}^k$ for different matrices $\{\mathbf{M}_i\}$, where the model owner has published $(g^{\mathbf{r}_i}, g^{\mathbf{s}_i})$ per matrix, the client samples uniform $\alpha_1, \ldots, \alpha_k$ and checks

$$\begin{aligned}&\left(\prod (g^{r_{1,i}})^{y_{1,i}}\right)^{\alpha_1} \cdots \left(\prod (g^{r_{k,i}})^{y_{k,i}}\right)^{\alpha_k} \\ &\stackrel{?}{=} \left(\prod (g^{s_{1,i}})^{x_{1,i}}\right)^{\alpha_1} \cdots \left(\prod (g^{s_{k,i}})^{x_{k,i}}\right)^{\alpha_k}.\end{aligned}$$

This can again be expressed as a single multi-exponentiation (of larger size), so scales sublinearly in k.

3 Preliminaries

Notation. For a finite set D, we use $x \leftarrow D$ to denote uniform sampling of x from D. We denote vectors with boldface lowercase letters and matrices with boldface uppercase letters. We use x_i to denote the ith element of the vector $\mathbf{x}$. For a real number x, we use $\lfloor x \rceil$ to denote the rounding function which maps x to the closest integer. For a positive integer n, we let $[n]$ denote $\{1, 2, \ldots, n\}$. We use $\oplus$ to denote element-wise exclusive-or between two boolean vectors. For a boolean a and a boolean vector $\mathbf{x}$, we use $a \cdot \mathbf{x}$ to denote the element-wise AND between a and each element of $\mathbf{x}$.

Threat model. We assume an honest model owner, two servers S_0, S_1, and a client C. Either (or both) servers can be malicious, but for privacy we assume the servers do not collude with each other. (Verifiability holds even when both servers are malicious and colluding with each other.) The client may be malicious but in our setting this does not matter since the security requirements of the servers are vacuous.

4 Detailed Protocol Description

We now provide a detailed description of our protocol. We proceed bottom-up: we describe a sub-protocol for private delegation of matrix-vector products in Section 4.1, show how to add verifiability to that sub-protocol in Section 4.2, discuss delegation of the initial embedding layer in Section 4.3, and put everything together in Section 4.4.

Fixed-point encoding. Matrices at linear layers have floating-point weights and inputs have real (floating-point) entries. Our matrix-vector multiplication protocol, however, operates on integers modulo N. We map a floating-point value $\tilde{x}$ into an associated fixed-point value $x = \lfloor \tilde{f} \cdot x \rceil$ where f is a scaling factor. We choose $f \ll N$ so no overflow occurs during the computation; in our experiments, we find $N = 2^{64}$, $f = 2048$ to be adequate. Since $\tilde{\mathbf{M}}\tilde{\mathbf{x}} \approx f^{-2}\mathbf{M}\mathbf{x}$, given a protocol for computing the (fixed-point) value $\mathbf{M}\mathbf{x}$ the client can compute an approximation to the (floating-point) value $\tilde{\mathbf{M}}\tilde{\mathbf{x}}$. In what follows, we assume the conversion from floating point to fixed point has already been done.

4.1 Delegation of Matrix-Vector Products

Recall the client holds an input $\mathbf{x} \in \mathbb{Z}_N^n$ and wants to learn $\mathbf{y} = \mathbf{M}\mathbf{x}$ for a public matrix $\mathbf{M} \in \mathbb{Z}_N^{m \times n}$. Our protocol proceeds as follows: the client splits $\mathbf{x}$ into $\mathbf{x}^{(0)}, \mathbf{x}^{(1)} \in \mathbb{Z}_N^n$ such that $\mathbf{x}^{(0)}$ is uniform and $\mathbf{x}^{(1)} = \mathbf{x} - \mathbf{x}^{(0)} \bmod N$, and sends $\mathbf{x}^{(b)}$ to S_b. Each server S_b computes $\mathbf{y}^{(b)} = \mathbf{M} \cdot \mathbf{x}^{(b)} \bmod N$ and sends back $\mathbf{y}^{(b)}$ to the client. Then, the client adds the two output shares modulo N to obtain the desired result. Note

$$\begin{aligned} \mathbf{y} = \mathbf{y}^{(0)} + \mathbf{y}^{(1)} \bmod N &= \mathbf{M} \cdot (\mathbf{x}^{(0)} + \mathbf{x}^{(1)}) \bmod N \\ &= \mathbf{M} \cdot \mathbf{x} \bmod N \end{aligned}$$

For large enough N (relative to the magnitude of the entries of $\mathbf{M}$ and $\mathbf{x}$, which in turn depends on the scaling factor f) there is no overflow in the final result and hence

$$\mathbf{y} = \mathbf{Mx} \bmod N = \mathbf{Mx},$$

which is the desired result. Privacy holds as each server S_b only learns a uniform value $\mathbf{x}^{(b)}$. We provide the formal protocol in Figure 1.

The work of each server is the same as in the computation of the base model, and the client's work consists merely of two vector additions.

Π_{MVP}

Inputs: Client C has $\mathbf{x} \in \mathbb{Z}_N^n$; servers S_0, S_1 have $\mathbf{M} \in \mathbb{Z}_N^{m \times n}$.
Outputs: C outputs $\mathbf{y} = \mathbf{M} \cdot \mathbf{x}$

1. C picks $\mathbf{x}^{(0)} \leftarrow \mathbb{Z}_N^n$ and sets $\mathbf{x}^{(1)} = \mathbf{x} - \mathbf{x}^{(0)} \bmod N$
2. C sends $\mathbf{x}^{(0)}$ to S_0 and $\mathbf{x}^{(1)}$ to S_1
3. $\forall b \in \{0, 1\}$, S_b sends back $\mathbf{y}^{(b)} = \mathbf{M} \cdot \mathbf{x}^{(b)} \bmod N$
4. C outputs $\mathbf{y} = \mathbf{y}^{(0)} + \mathbf{y}^{(1)} \bmod N$

Fig. 1. Private delegation of a matrix-vector product.

4.2 Verification of Matrix-Vector Products

We now present a mechanism that allows the client to verify that $\mathbf{y} = \mathbf{M} \cdot \mathbf{x}$. (See Figure 2.) It consists of algorithms Setup, Publish, and Verify, where the first two are run by the model owner and the third is run by the client. We assume a fixed group $\mathbb{G}$ of prime order $p \gg N$ with generator g in which the discrete-logarithm problem is hard.

Setup generates global setup material that can be used to verify multiple executions of the matrix-vector product protocol. It first generates uniform $r_1, r_2, \ldots, r_{m'} \leftarrow \mathbb{Z}_p$, where m' is the maximum row dimension across all matrices to be verified. Then it publicly releases $\{R_i = g^{r_i}\}_{i \in [m']}$, keeping the $\{r_i\}$ private.

Publish is used to provide per-matrix verification information. For a matrix $\mathbf{M} \in \mathbb{Z}^{m \times n}$ with $m \leq m'$, this algorithm computes $\mathbf{s} = \mathbf{M}^T \cdot \mathbf{r}$ for $\mathbf{r} = (r_1, r_2, \ldots, r_m)$. Then, it publishes $\{S_i = g^{s_i}\}_{i=1}^n$. Once verification information for all matrices has been published, the $\{r_i\}_{i=1}^{m'}$ can be deleted.

To verify a claim that $\mathbf{y} = \mathbf{Mx}$, the client uses the global setup information $\{R_i\}$ and the specific verification information $\{S_i\}$ corresponding to $\mathbf{M}$ to check whether $\prod_{i=1}^n S_i^{x_i} \stackrel{?}{=} \prod_{i=1}^m R_i^{y_i}$. Note that although we have written this as involving the computation of two group elements, it is more efficient to check that $\prod S_i^{x_i} \prod R_i^{-y_i} \stackrel{?}{=} 1$, which requires just a single multi-exponentiation.

The verification procedure does not impose any additional overhead on the servers. The client's work is dominated by a single multi-exponentiation, which incurs a cost of $(m+n)/\log(m+n)$ group exponentiations. For large enough m, n, this is significantly less than the cost of the $O(mn)$ arithmetic operations needed to compute the matrix-vector product. More importantly, the memory requirements of the client are reduced from $O(mn)$ to $O(m+n)$.

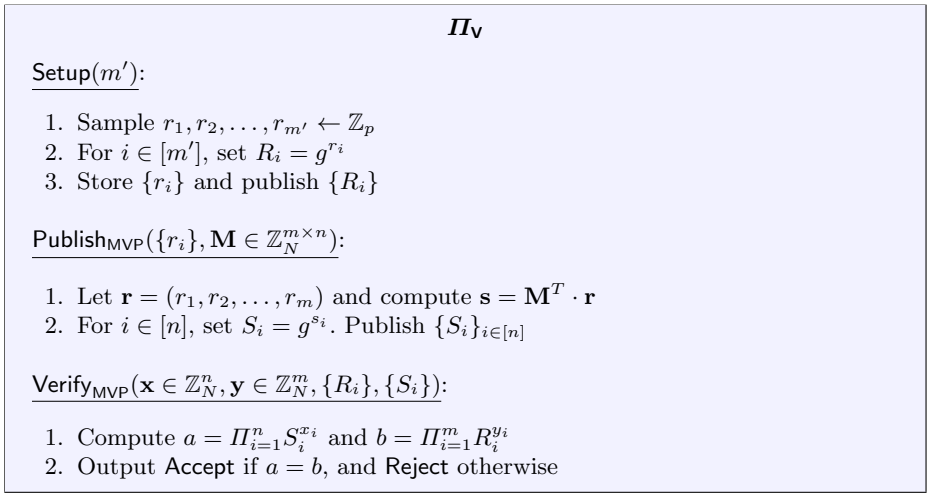

Fig. 2. Verification of a matrix-vector product.

Soundness. As in the analysis of Freivalds's algorithm, finding $\mathbf{M}, \mathbf{x}, \mathbf{y} \neq \mathbf{Mx}$ for which verification succeeds implies finding a nonzero vector $\mathbf{d} = \mathbf{y} - \mathbf{Mx}$ for which

$$\prod_{i=1}^{m} R_i^{d_i} = \prod_{i=1}^{m} (g^{r_i})^{d_i} = 1.$$

(Note that the $\{S_i\}$ can be written in terms of the $\{R_i\}$ given $\mathbf{M}$.) In the original setting of Freivalds's algorithm finding such a $\mathbf{d}$ is (information-theoretically) hard to do when $\mathbf{r} = (r_1, \ldots, r_{m'})$ is chosen *after* $\mathbf{M}, \mathbf{x}, \mathbf{y}$ are fixed, but easy to do if $\mathbf{y}$ can be selected adaptively based on $\mathbf{r}$. In our setting, it involves finding a non-trivial representation of 1 relative to the basis $\{R_i\}$, which is known to be equivalent to solving the discrete-logarithm problem. Thus, it is *computationally* infeasible for the servers to cause the client to output an incorrect result (except with negligible probability) even when the servers may choose $\mathbf{y}$ adaptively based on the $\{R_i\}$ (and the $\{S_i\}$).

Batching. Instead of verifying each matrix-vector product, we perform batch verification of multiple products. To verify k query-response pairs $\{\mathbf{y}_i = \mathbf{Mx}_i\}_{i=1}^k$ using the same matrix $\mathbf{M}$, the client picks uniform $\alpha_1, \ldots, \alpha_k \in \mathbb{Z}_p$ and verifies

the single matrix-vector multiplication $\sum_{i=1}^{k} \alpha_i \cdot \mathbf{y}_i = \mathbf{M} \cdot \sum_{i=1}^{k} \alpha_i \cdot \mathbf{x}_i \bmod p$ as above. The number of group operations here is independent of k.

To verify query-response pairs $\{\mathbf{y}_i = \mathbf{M}_i \mathbf{x}_i\}_{i=1}^{k}$ for different matrices, the client samples uniform $\alpha_1, \ldots, \alpha_k \in \mathbb{Z}_p$ and verifies the (larger) matrix-vector product

$$\alpha_1 \mathbf{y}_1 + \cdots + \alpha_k \mathbf{y}_k = [\mathbf{M}_1 \mid \cdots \mid \mathbf{M}_k] \begin{bmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_k \end{bmatrix}.$$

This again involves only a single multi-exponentiation, albeit of larger size. This saves runtime as the computational cost of multi-exponentiations scale sublinearly in the input size.

Note that these techniques can be combined, namely, the client can reduce all equations involving the same matrix to a single matrix-vector product without increasing the dimension, and then combine equations involving different matrices to obtain a matrix-vector product of larger dimension.

4.3 Handling the Embedding Layer

LLMs also use an embedding layer, which converts an input token w into its corresponding embedding $T[w]$ using a lookup in a (public) table T. For vocabulary size V and embedding dimension d, the table T consists of V entries in $\mathbb{R}^d$. For, e.g., Llama 70B, $V = 128000$ and $d = 8192$ which results in a table of size 2GB. In principle, then, the embedding operation could be done locally by the client as long as they have 2GB of memory. Nevertheless, we provide a private, verifiable outsourcing protocol for the embedding layer to support devices that do not have this much memory, or to support future models with larger tables.

$\Pi_{\mathsf{Embedding}}$

Inputs: C has $w \in [V]$; the servers S_0, S_1 have a public table $T \in \mathbb{R}^{V \times d}$
Outputs: C outputs $\mathbf{y} = T[w]$

1. Let $\mathbf{x} \in \{0,1\}^V$ be such that $x_w = 1$ and $x_j = 0$ for all $j \neq w$
2. C samples $\mathbf{x}^{(0)} \leftarrow \{0,1\}^V$ and sets $\mathbf{x}^{(1)} = \mathbf{x} \oplus \mathbf{x}^{(1)}$
3. C sends $\mathbf{x}^{(0)}$ to S_0 and $\mathbf{x}^{(1)}$ to S_1
4. For $b \in \{0,1\}$, S_b returns $\mathbf{y}^{(b)} = \bigoplus_{j=1}^{V} T[j] \cdot \mathbf{x}_j^{(b)}$
5. C outputs $\mathbf{y}^{(0)} \oplus \mathbf{y}^{(1)}$

Fig. 3. Delegation of the embedding layer.

To perform this lookup, we use a simple private-information retrieval (PIR) [4] protocol. The client converts its token w into a one-hot vector $\mathbf{x} \in \{0,1\}^V$,

i.e., a vector for which $x_w = 1$ and $\forall j \neq w, x_j = 0$. It then secret shares this vector bitwise, i.e., it samples uniform $\mathbf{x}^{(0)} \leftarrow \{0,1\}^V$, sets $\mathbf{x}^{(1)} = \mathbf{x} \oplus \mathbf{x}^{(0)}$, and sends $\mathbf{x}^{(b)}$ to S_b. Each server S_b computes $\mathbf{y}^{(b)} = \bigoplus_{j=1}^{V} x_j^{(b)} \cdot T[j]$ (viewing $T[j]$ as a fixed-length bit-string) and sends this to the client. The client outputs $\mathbf{y}^{(0)} \oplus \mathbf{y}^{(1)}$. Correctness holds since:

$$\begin{aligned}\mathbf{y}^{(0)} \oplus \mathbf{y}^{(1)} &= \bigoplus_{j=1}^{V} \left(x_j^{(0)} \oplus x_j^{(1)}\right) \cdot T[j] \\ &= T[w].\end{aligned}$$

See Figure 3 for details. Note that the total communication is V bits from the client to each server, and a (bit-string encoding a) single table entry in return.

Note that here we do not need a fixed-point encoding, but can work directly with the floating-point values in the table.

Verification. To verify the output of delegated embedding, we use hashing. The model owner publishes a (second preimage-resistant) hash of each entry in the table T. To verify a claimed result y, a client can simply hash y and compare it with the corresponding hash value. If we instantiate the hash function with SHA256 and take $V = 128000$, the size of the verification information stored by the client is 4MB; verification requires only one hash evaluation. We provide a formal description of these procedures in Figure 4.

$\boldsymbol{\Pi_{\mathsf{V}}}$ (Continued.)

Let H be a hash function

$\underline{\mathsf{Publish}_{\mathsf{Embedding}}(T \in \mathbb{R}^{V \times d})}$:

1. Output $\mathbf{c}$, where $c_j = H(T[j])$ for $j \in [V]$

$\underline{\mathsf{Verify}_{\mathsf{Embedding}}(w \in [V], \mathbf{y} \in \mathbb{R}^d, \mathbf{c})}$:

1. Output Accept if $H(\mathbf{y}) = c_w$, and Reject otherwise

Fig. 4. Verification for embedding operations.

4.4 Overall Protocol for Outsourced Inference

We provide a description of our overall protocol in Figure 5. We model the inference computation as a sequence of layers. Each layer is associated with an operation g and optionally with some weights W. Specifically, each linear layer is associated with a matrix $\mathbf{M}$ and outputs $\mathbf{M} \cdot \mathbf{x}$ on input $\mathbf{x}$. Similarly, the embedding layer is associated with a table T and outputs $T[i]$ on input i. We

omit a description of the non-linear layers since they are computed locally by the client.

In our protocol the client delegates linear layers using Π_{MVP} and (batch) verifies them using Π_{V}. Similarly, we use $\Pi_{\mathsf{Embedding}}$ to delegate the embedding, and use corresponding methods from Π_{V} to verify correctness.

$\Pi_{\mathsf{Inference}}$

Inputs: Client C has input x and the verification information; servers S_0, S_1 have the model weights

Outputs: C outputs the correct result y or aborts if malicious behavior is detected

---

Setup Phase (Run by model owner):

1. Let m' be the largest row dimension of all linear-layer weight matrices
2. $\{R_i\} \leftarrow \Pi_{\mathsf{V}}.\mathsf{Setup}(m')$
3. Initiate empty list $\mathsf{cmt} = \{\}$
4. Iterate over all layers:
   (a) For embedding layer with table T, push $\Pi_V.\mathsf{Publish}_{\mathsf{Embedding}}(T)$ to cmt
   (b) For linear layer with matrix $\mathbf{M}$, push $\Pi_V.\mathsf{Publish}_{\mathsf{MVP}}(\mathbf{M})$ to cmt
5. Publicly release the verification information $\{R_i\}, \mathsf{cmt}$ along with the model weights

Execution Phase (Jointly run by C, S_0, S_1):

1. For each linear layer ℓ, input dimension n, and output dimension m, C initializes $\mathbf{x}_{\ell,\mathsf{acc}} = 0^n, \mathbf{y}_{\ell,\mathsf{acc}} = 0^m$
2. C sequentially processes layers:
   (a) For embedding layer with input (w, T):
      i. S_0, S_1 and C jointly invoke $\Pi_{\mathsf{Embedding}}$ with inputs w, T, T, respectively, and C learns $\mathbf{y}$
      ii. C fetches verification information $\mathbf{c}$ from cmt and aborts if $\Pi_{\mathsf{V}}.\mathsf{Verify}_{\mathsf{Embedding}}(w, \mathbf{y}, \mathbf{c}) = \mathsf{Reject}$
   (b) For linear layer ℓ with input $(\mathbf{x}, \mathbf{M})$:
      i. S_0, S_1 and C jointly invoke Π_{MVP} with inputs $\mathbf{x}, \mathbf{M}, \mathbf{M}$, respectively, and C learns $\mathbf{y}$
      ii. C samples $\alpha \leftarrow \mathbb{F}$, and updates $\mathbf{x}_{\ell,\mathsf{acc}} = \mathbf{x}_{\ell,\mathsf{acc}} + \alpha \cdot \mathbf{x}$ and $\mathbf{y}_{\ell,\mathsf{acc}} = \mathbf{y}_{\ell,\mathsf{acc}} + \alpha \cdot \mathbf{y}$
   (c) For other layers, C does the computation locally
3. C learns y as the output
4. C concatenates $\mathbf{x}_{\ell,\mathsf{acc}}, \mathbf{y}_{\ell,\mathsf{acc}}, \{S_i\}, \{R_i\}$ accross all layer identifiers ℓ to get $\hat{\mathbf{x}}$ or each linear layer ℓ and verification information $\{S_i\}$, C checks if $\Pi_{\mathsf{V}}.\mathsf{Verify}_{\mathsf{MVP}}(\mathbf{x}_{\ell,\mathsf{acc}}, \mathbf{y}_{\ell,\mathsf{acc}}, \{S_i\}, \{R_i\}) = \mathsf{Accept}$, aborts otherwise.
5. C outputs y

Fig. 5. Overall protocol for outsourced inference

5 Implementation and Evaluation

We implement the client in C++. We start with the open-source LLM inference software `llama2.c` [14] (and its fork `llama3.2.c` [12]) and replace matrix-vector products and embedding calls with remote server calls. We utilizes AES-NI for as a pseudorandom number generator (PRNG), SHA256 as the hash function, and secp256k1 as the underlying cryptographic group. We use arkworks (https://arkworks.rs) to implement multi-exponentiations. We reduce communication for secret sharing by observing that since one of the two shares is uniform, it suffices to send the seed used by the PRNG to generate the share instead of sending the share itself.

We implement the servers in C++ using CUDA to accelerate matrix multiplications. We use 16-bit integers to encode weight matrices as fixed-point numbers with a scaling factor of $f = 2048$. We implement a custom CUDA kernel for computing the product of matrices with 16-bit integer weights and vectors with 64-bit integer entries, using optimizations from prior work [1].

Experimental setup. We deploy the two servers on GCP VMs each equipped with 2 A100-80GB GPUs. While we utilize both GPUs for storing the weights, we do not use multi-GPU parallelization; we only use a single GPU at a time to speed up matrix-vector multiplications. We deploy server S_0 in Ashburn, Virginia (us-east4) and server S_1 in Columbus, Ohio (us-east5). The client C is an E2 instance with 2 vCPUs and 4GB of RAM located in South Carolina (us-east1). Ping latency (RTT) and bandwidth between these VMs are: (1) C and S_0: 12ms, 4Gbits/s (2) C and S_1: 22ms, 3Gbits/s, and (3) S_0 and S_1: 7ms, 25Gbits/s. Note that lower latency and higher bandwidth between S_0 and S_1 is more favorable to the baseline protocol SIGMA that we compare to, as SIGMA requires communication between the two servers while our protocol does not but instead uses more client-server communication.

5.1 Model Performance

To validate whether our fixed-point encoding in the linear layers causes the utility of the model to drop, we evaluate the perplexity and next-token prediction accuracy on the Wikitext dataset [19] for both the base model and our fixed-point approximation; see Table 1. We find that the degradation of model quality, if at all, is nominal.

5.2 Efficiency

To test performance of our protocol, we generate 100 tokens for different models using our protocol and report runtime, communication, and round trips per token (see Figure 2). Note that the number of round trips is exactly the number of linear layers plus one (for the embedding layer). For completeness, we also benchmark small models, like Llama 3.2 1B, which can be run on edge devices.

We observe that our protocols are latency-bound. That is, most of the runtime is caused by the round trips between the client and servers. For instance, for

Model Name	Perplexity (Lower is better)		Accuracy (Larger is better)	
	Float	Fixed	Float	Fixed
Llama-3.2-1B	8.62	8.63	53.36%	53.33%
Llama-3.2-3B	6.94	6.95	56.39%	56.41%
Llama-2-7B	4.86	4.87	62.77%	62.75%
Llama-2-13B	4.35	4.36	64.65%	64.64%
Gemma-3-1B	20.32	20.40	44.37%	44.37%
Gemma-3-4B	10.62	10.73	52.68%	52.64%
Gemma-3-12B	7.67	7.69	55.47%	55.46%

Table 1. Comparing floating-point and fixed-point models on the Wikitext dataset

Model Name	Time (s)	Comm. (MB)	Rounds	Verification (s)
Llama-3.2-1B	1.00	9.50	66	3.8
Llama-3.2-3B	1.77	17.55	114	6.7
Llama-2-7B	2.01	26.99	130	10.4
Llama-3.1-8B	2.04	29.53	130	10.6
Llama-2-13B	2.74	41.98	162	15.1
Llama-2-70B	5.21	135.62	322	56.5
Gemma-3-1B	1.65	13.09	106	5.1
Gemma-3-4B	2.21	23.99	138	9.3
Gemma-3-12B	3.21	48.15	194	18.9
Gemma-3-27B	4.29	79.92	250	30.7

Table 2. Latency, communication, round complexity, and verification time across different models for generating 1 token, amortized over 100 tokens. The reported verification time is for verifying matrix-vector products accross all 100 tokens.

Llama 2-70B we observe that 4.9 out of 5.2 seconds are spent in 322 round trips due to its 80-layer architecture. For this reason, our protocol performs better on Llama 2-13B— a larger model than Gemma 3-12B—since that model has fewer layers. We also note that even for the biggest model available, Kimi K3 [22], the number of layers is 96, which is close to Llama 70B's 80 layers. Therefore, we expect the performance of our protocol for Kimi K3 to be close to its performance for Llama 70B.

5.3 Comparison with Prior Work

SIGMA. We compare our work with SIGMA [10], a state-of-the-art protocol for private (but not verifiable) inference. SIGMA is designed to keep model weights private, but for the purposes of our comparison we instrument it to use public weights.

SIGMA assumes a trusted third party that generates and sends cryptographic material to the two servers in an offline phase before the client's input is determined. The offline phase of SIGMA is not reusable, in contrast to the setup done by the model owner in our protocol. Moreover, SIGMA only supports the prefill stage of inference. In our experimental setup, SIGMA runs out memory for Llama 2-7B and 13B because of the large amount of cryptographic material the servers need to store. Therefore, we only benchmark prefill tasks of token length 32 and 64 on Llama 2-7B. Comparing the online phase of SIGMA with our protocol, we see a speedup of 4–6× with 3.3× lower communication. Taking into account both the online and offline phases of SIGMA, we are 11–15× faster and use 182–237× less communication.

Prompt Length	**Tool**	**Time (s)**		**Comm. (GB)**	
		Offline	Online	Offline	Online
32	SIGMA	47	32	194	2.77
	Our work	-	5.2	-	0.83
64	SIGMA	69	41	297	5.70
	Our work	-	10.0	-	1.66

Table 3. Comparison with SIGMA on Llama 2 7B. We run prefill tasks on prompts consisting of 32 and 64 tokens.

NEXUS. We also compare our protocol with the state-of-the-art FHE-based protocol NEXUS [28]. (Note that a follow-up paper [7], was recently shown to be insecure [29].) NEXUS's implementation does not support full transformer inference, so we directly compare with the numbers reported in the paper. For Llama 3-8B using NEXUS, a prefill task of 8 tokens requires 51.84 seconds on a dual A100 setup. on the same task our protocol requires 4 seconds, achieving 13× lower latency.

6 Discussion

Protocol-friendly models. As noted earlier, the running time of our protocol is dominated by the round-trip communication between the client and servers. The number of round trips required is directly proportional to the number of (linear) layers in the network. Thus, outsourcing inference using our protocol is cheaper for wide but shallow networks rather than tall but thin networks. (For example, our protocol performs better on the larger Llama 2-13B model than the Gemma 3 12B model, precisely because it has fewer layers.)

This suggests that it could be worthwhile for model developers to explicitly aim for wide-but-shallow models more compatible with delegated inference.

One such class of models are diffusion models, like DiffusionGemma. Instead of generating one token at a time, these models generate 256 tokens in one shot. Although they require a dynamic number of diffusion repetitions (typically 15–20), the trade-off is favorable for our system, as it reduces the number of round trips required for generation compared to a non diffusion variant. For e.g., we need $256/20\times$ fewer rounds for 20 iterations.

Supporting mixture-of-expert models. In Mixture of Expert (MoE) models, instead of having a plain matrix multiplication, a two-stage matrix multiplication is performed. The input $\mathbf{x} \in \mathbb{F}^n$ is first multiplied with a small router matrix $\mathsf{R} \in \mathbb{F}^{k \times n}$ (for small k) to get a score vector $\mathbf{s} \in \mathbb{F}^k$. Based on the top t elements of $\mathbf{s}$, indices $(i_1, i_2, \ldots i_t)$ are chosen. Based on these indices, matrices with which the input could be multiplied to determine the final activation are picked. That is, for a set of matrices (or experts) $\mathbf{M}_1, \mathbf{M}_2, \ldots \mathbf{M}_k$, the final activation is $y = \sum_{j=1}^{t} \mathbf{M}_{i_j} \mathbf{x}$.

For such a model, the choice of matrix to multiply with leaks some information about $\mathbf{x}$. To reduce leakage, we can use the random server selection approach from the previous paragraph. It reduces the leakage from "the client chose these t matrices" to "the client chose this one matrix as a part of some t matrices." To prevent any leakage, the client needs to query all matrices and ignore unused results.

References

1. Simon Boehm. How to optimize a CUDA matmul kernel for cuBLAS-like performance: A worklog, 2026. https://siboehm.com/articles/22/CUDA-MMM.
2. Nishanth Chandran, Divya Gupta, Sai Lakshmi Bhavana Obbattu, and Akash Shah. SIMC: ML inference secure against malicious clients at semi-honest cost. In *USENIX Security Symposium*, 2022.
3. Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology—Asiacrypt 2017, Part I*, pages 409–437. Springer, 2017.
4. Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, November 1998.
5. Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. LLM.int8(): 8-bit matrix multiplication for transformers at scale, 2022. https://arxiv.org/abs/2208.07339.
6. Rusins Freivalds. Probabilistic machines can use less running time. In *IFIP Congress*, 1977.
7. Xinwen Gao, Shaojing Fu, Lin Liu, Zhuotao Liu, Yuchuan Luo, and Yongjun Wang. Euston: Efficient and user-friendly secure transformer inference with non-interactivity. *IEEE S&P*, 2026.
8. Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML*, 2016.
9. Kanav Gupta, Nishanth Chandran, Divya Gupta, Jonathan Katz, and Rahul Sharma. Shark: Actively secure inference using function secret sharing. In *IEEE S&P*, 2025.

10. Kanav Gupta, Neha Jawalkar, Ananta Mukherjee, Nishanth Chandran, Divya Gupta, Ashish Panwar, and Rahul Sharma. SIGMA: Secure GPT inference with function secret sharing. In *PETS*, 2024.
11. Kanav Gupta, Deepak Kumaraswamy, Nishanth Chandran, and Divya Gupta. LLAMA: A low latency math library for secure inference. In *PETS*, 2022.
12. Dylan Harden. Inference Llama3.2 1B/3b base/instruct models in 1 file of pure C, 2026. https://github.com/Dylan-Harden3/llama3.2.c.
13. Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. Gazelle: A low latency framework for secure neural network inference. In *USENIX Security Symposium*, 2018.
14. Andrej Karpathy. Inference Llama 2 in one file of pure C, 2026. https://github.com/karpathy/llama2.c.
15. Brian Knott, Shobha Venkataraman, Awni Hannun, Shubhabrata Sengupta, Mark Ibrahim, and Laurens van der Maaten. CrypTen: Secure multi-party computation meets machine learning. In *NeurIPS*, 2021.
16. Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. CrypTFlow: Secure tensorflow inference. In *IEEE S&P*, 2020.
17. Ryan Lehmkuhl, Pratyush Mishra, Akshayaram Srinivasan, and Raluca Ada Popa. Muse: Secure inference resilient to malicious clients. In *USENIX Security Symposium*, 2021.
18. Anmol Mekala, Anirudh Atmakuru, Yixiao Song, Marzena Karpinska, and Mohit Iyyer. Does quantization affect models' performance on long-context tasks?, 2025. https://arxiv.org/abs/2505.20276.
19. Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016. https://arxiv.org/abs/1609.07843.
20. Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, Léonard Hussenot, Pier Giuseppe Sessa, Aakanksha Chowdhery, Adam Roberts, Aditya Barua, Alex Botev, Alex Castro-Ros, Ambrose Slone, Amélie Héliou, Andrea Tacchetti, Anna Bulanova, Antonia Paterson, Beth Tsai, Bobak Shahriari, Charline Le Lan, Christopher A. Choquette-Choo, Clément Crepy, Daniel Cer, Daphne Ippolito, David Reid, Elena Buchatskaya, Eric Ni, Eric Noland, Geng Yan, George Tucker, George-Christian Muraru, Grigory Rozhdestvenskiy, Henryk Michalewski, Ian Tenney, Ivan Grishchenko, Jacob Austin, James Keeling, Jane Labanowski, Jean-Baptiste Lespiau, Jeff Stanway, Jenny Brennan, Jeremy Chen, Johan Ferret, Justin Chiu, Justin Mao-Jones, Katherine Lee, Kathy Yu, Katie Millican, Lars Lowe Sjoesund, Lisa Lee, Lucas Dixon, Machel Reid, Maciej Mikuła, Mateo Wirth, Michael Sharman, Nikolai Chinaev, Nithum Thain, Olivier Bachem, Oscar Chang, Oscar Wahltinez, Paige Bailey, Paul Michel, Petko Yotov, Rahma Chaabouni, Ramona Comanescu, Reena Jana, Rohan Anil, Ross McIlroy, Ruibo Liu, Ryan Mullins, Samuel L Smith, Sebastian Borgeaud, Sertan Girgin, Sholto Douglas, Shree Pandya, Siamak Shakeri, Soham De, Ted Klimenko, Tom Hennigan, Vlad Feinberg, Wojciech Stokowiec, Yu hui Chen, Zafarali Ahmed, Zhitao Gong, Tris Warkentin, Ludovic Peran, Minh Giang, Clément Farabet, Oriol Vinyals, Jeff Dean, Koray Kavukcuoglu, Demis Hassabis, Zoubin Ghahramani, Douglas Eck, Joelle Barral, Fernando Pereira, Eli Collins, Armand Joulin, Noah Fiedel, Evan Senter, Alek Andreev, and Kathleen Kenealy. Gemma: Open models based on Gemini research and technology, 2024. https://arxiv.org/abs/2403.08295.
21. Payman Mohassel and Yupeng Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*, 2017.

22. Moonshot AI. Kimi k3, 2026. https://www.kimi.com/blog/kimi-k3.
23. Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. BOLT: Privacy-preserving, accurate and efficient inference for transformers. In *IEEE S&P*, 2024.
24. Zilan Qian. How to buy cheap Claude tokens in China, 2026. https://www.chinatalk.media/i/196186465/.
25. Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. CrypTFlow2: Practical 2-Party Secure Inference. In *CCS*, 2020.
26. Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023. https://arxiv.org/abs/2302.13971.
27. Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-party secure computation for neural network training. *PoPETs*, 2019.
28. Jiawen Zhang, Xinpeng Yang, Lipeng He, Kejia Chen, Wen jie Lu, Yinghao Wang, Xiaoyang Hou, Jian Liu, Kui Ren, and Xiaohu Yang. Secure transformer inference made non-interactive. In *NDSS*, 2025.
29. Jiaqi Zhao and Fengwei Wang. Breaking Euston: Recovering private inputs from secure inference by exploiting subspace leakage, 2026. https://arxiv.org/abs/2604.17238.