Maverick: Private and Verifiable LLM Inference Made Practical via Matrix–Vector Multiplication Delegation

Ben Merbaum* *Yale University*
Mohammad Amin Raeisi* *Yale University*
Wenhao Wang* *Yale University, IC3*
Charalampos Papamanthou *Yale University*
Katerina Sotiraki *Yale University*
Fan Zhang *Yale University, IC3*

Abstract

Open-source large language models (LLMs) are increasingly competitive with closed-source models while offering transparency and the ability to run inference without exposing user inputs to a service provider. However, running large-scale models locally requires substantial computational resources. In practice, users may still resort to a third-party provider, giving rise to privacy and correctness concerns. Existing solutions that address these problems often impose substantial server overhead or introduce additional trust assumptions.

In this paper, we present Maverick, a novel approach to private and verifiable LLM inference based on a protocol for delegating matrix–vector multiplication, a dominant operation in LLMs. At its core, Maverick provides, to our knowledge, the first information-theoretically sound verification protocol for matrix–vector multiplication delegation with *transparent preprocessing*, *efficient (batch) verification*, and *virtually no server overhead*. We combine this verification primitive with LPN-based pseudorandom masking to provide input privacy.

We implement Maverick and evaluate both of its modes on large open-source models. When only verification is required, Maverick lets commodity clients use models far beyond their own capacity: a laptop and a desktop with a 16 GB GPU generate tokens 81–270× faster on `Qwen3-32B` than with local inference, and reach 7–33 tokens per second on `Llama 4`, a 400B-parameter model that neither can run at all. When input privacy is also required, Maverick generates tokens 10–12× faster on `Qwen2.5-72B` than the concurrent system MOSAIC, using precomputed masks.

1 Introduction

Open-source models such as DeepSeek, Qwen, and Kimi are increasingly matching the performance of closed-source models [33, 59, 84]. For example, as of July 2026, Kimi K3 outperforms Claude models on frontend design tasks [34]. In addition, open-source models can be run locally, allowing users with sufficient hardware to keep their inputs private from service providers. Their publicly available weights also offer better transparency [28] and reproducibility [30], while reducing dependence on provider-controlled changes to pricing, model availability, and access policies. Despite various advantages, their substantial computational requirements make local deployment impractical or undesirable for most users (e.g., running Kimi K3 requires a cluster of high-end GPUs estimated to cost $0.55–$1 million to set up [36]). In practice, users resort to specialized service providers, reintroducing privacy and correctness concerns.

The problem of private and verifiable outsourcing of AI and LLM inference has been extensively studied. However, aside from recent systems based on trusted execution environments (TEEs) [5, 7, 80], existing cryptographic approaches typically incur significant overhead on the server side. For instance, proof systems [16, 37, 42, 44, 64, 65, 73, 74, 83, 92, 93, 95] and homomorphic encryption [45, 49–51, 54, 55, 62, 68, 70, 94, 100] have been used, either individually or in combination, to provide verifiability, input privacy, or both. However, generating proofs and performing fully homomorphic encryption (FHE) computations can be orders of magnitude more expensive than the original workload. Solutions based on TEEs offer practical performance, though their security guarantees may not always hold in practice [32].

In this paper, we present a lightweight protocol for private and verifiable outsourced LLM inference. Our protocol allows a *computationally constrained client*, such as a mobile device, to use an open-source model run by a powerful, untrusted server while hiding the client's private input and supporting verification, all with *negligible* server overhead.

In particular, our technical contributions are as follows. **1)** A protocol for fast verifiable matrix–vector multiplication delegation (vMVMD). To our knowledge, it is the first information-theoretically sound construction with transparent preprocessing and efficient verification runtime. **2)** We com-

*These authors contributed equally to this work and are ordered alphabetically.

†Emails: {`ben.merbaum`, `amin.raeisi`, `wenhao.wang`, `charalampos.papamanthou`, `katerina.sotiraki`, `f.zhang`}`@yale.edu`.

bine vMVMD with pseudorandom masks using coding theory to hide client inputs without introducing server overhead, providing private and verifiable matrix–vector multiplication delegation (pvMVMD). **3)** Building on pvMVMD, we present Maverick, a system for private and verifiable LLM inference. Maverick outsources the linear operations of LLM inference to an untrusted server. Nonlinear operations [89] are inexpensive by comparison and are performed locally. We implement Maverick with a GPU server and evaluate it on open-source models of up to 400B parameters.

1.1 Verifiable Matrix–Vector Multiplication Delegation (vMVMD)

In a vMVMD protocol, a client with input $\mathbf{x} \in \mathbb{F}^n$ verifies that a server's output $\mathbf{y} \in \mathbb{F}^m$ satisfies $\mathbf{y} = \mathbf{M}\mathbf{x}$ for a public matrix $\mathbf{M} \in \mathbb{F}^{m \times n}$. Since the client can always compute $\mathbf{y}$ locally in $O(mn)$ time, verification time must be less than $O(mn)$ for delegation to be meaningful. One idea is to apply Freivalds' algorithm [41]. Given matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$, Freivalds' algorithm checks whether $\mathbf{AB} \stackrel{?}{=} \mathbf{C}$ by sampling a random vector $\mathbf{r}$ and checking $(\mathbf{r}^\top \mathbf{A})\mathbf{B} \stackrel{?}{=} \mathbf{r}^\top \mathbf{C}$. However, when applied to the matrix–vector multiplication (MVM) $\mathbf{Mx} \stackrel{?}{=} \mathbf{y}$, it requires the client to compute $\mathbf{r}^\top \mathbf{M}$, which is as expensive as computing $\mathbf{y}$ locally.

Our idea is to use a *sparse* vector in Freivalds' algorithm. Rather than sampling a uniform challenge $\mathbf{r}$, we sample a sparse vector $\mathbf{e}$ from $\mathbb{F}^m$ with t nonzero elements. To check $\mathbf{Mx} \stackrel{?}{=} \mathbf{y}$, the client computes $(\mathbf{e}^\top \mathbf{M})\mathbf{x}$ and $\mathbf{e}^\top \mathbf{y}$ in time $O(tn)$ and accepts if and only if they match. When $t = O(\kappa)$, verification takes $O_\kappa(m+n)$ time instead of $O(mn)$ time.

However, this strawman protocol is not sound. Consider an invalid claim $\mathbf{Mx} \neq \mathbf{y}$ for which $\mathbf{Mx}$ and $\mathbf{y}$ differ at only one index, say k. The above protocol only rejects if the k-th element of $\mathbf{e}$ happens to be nonzero, which happens with probability t/m, i.e., the soundness error is at least $1 - t/m$.

To reduce the soundness error, we use *error-correcting codes* to perform checks on encoded values. Let $\mathbf{G} \in \mathbb{F}^{m \times N}$ be the generator matrix of a linear code. The client samples a sparse challenge (of length N) and checks $(\mathbf{e}^\top \mathbf{G}^\top \mathbf{M})\mathbf{x} = \mathbf{e}^\top \mathbf{G}^\top \mathbf{y}$. Intuitively, the encoding spreads every nonzero error vector $\mathbf{\Delta} := \mathbf{Mx} - \mathbf{y}$ into a codeword $\mathbf{G}^\top \mathbf{\Delta} \in \mathbb{F}^N$ with enough nonzero entries that its support is likely to intersect the support of $\mathbf{e}$. This requires a code with high minimum Hamming weight (equivalently, high relative distance) and also fast encoding for fast client online time, properties satisfied by, for example, RAA codes [3].

Computing $\mathbf{G}^\top \mathbf{M}$ is relatively heavy, but $\mathbf{Q} = \mathbf{G}^\top \mathbf{M}$ is independent of client inputs and thus can be prepared in a *preprocessing phase*. The preprocessing adds no extra trust assumption because both $\mathbf{G}$ and $\mathbf{M}$ are public and anyone can verify the correctness of $\mathbf{Q}$. With $\mathbf{Q}$ preprocessed, verification is efficient since $\mathbf{e}$ is sparse and $\mathbf{G}$ has linear-time encoding.

To further optimize client verification time, we develop a protocol for efficiently verifying a batch of MVMs involving different matrices and vectors. For each matrix $\mathbf{M}_b$, preprocessing produces $\mathbf{Q}_b = \mathbf{G}^\top \mathbf{M}_b$. The same sparse challenge $\mathbf{e}$ can be shared across all claims in a repetition, but the client would still need to compute the expensive vectors $\mathbf{Q}_b^\top \mathbf{e}$ separately for every matrix. We instead stack the computation of $\mathbf{Q}_b^\top \mathbf{e}$ into a single MVM and delegate it to the server. We then recursively apply vMVMD to verify this computation. Crucially, its input is now the sparse vector $\mathbf{e}$ itself, so the second check can be evaluated efficiently by the client.

1.2 Private and Verifiable Matrix–Vector Multiplication Delegation (pvMVMD)

We now show how to add privacy (i.e., hiding $\mathbf{x}$ from the server) on top of vMVMD without introducing server overhead. A naive construction is to have the client sample a uniformly random mask $\mathbf{x}'$ and send $\widehat{\mathbf{x}} = \mathbf{x} + \mathbf{x}'$ to the server; upon receiving $\widehat{\mathbf{y}}$, the client performs vMVMD to check $\mathbf{M}\widehat{\mathbf{x}} \stackrel{?}{=} \widehat{\mathbf{y}}$. To remove the mask from $\widehat{\mathbf{y}} = \mathbf{M}(\mathbf{x} + \mathbf{x}')$, however, the client needs to compute $\mathbf{Mx}'$ and subtract it from $\widehat{\mathbf{y}}$; computing $\mathbf{Mx}'$ takes $O(mn)$ work.

To avoid this overhead, we use another code $\mathbf{G}_\mathrm{x}$ to generate pseudorandom privacy masks $\mathbf{x}' = \mathbf{G}_\mathrm{x}\mathbf{e}_\mathrm{x}$ from sparse random vectors $\mathbf{e}_\mathrm{x}$ under standard cryptographic assumptions. To distinguish this code from the one used for verification, we denote it by $\mathbf{G}_\mathrm{x}$, using the subscript x because privacy masks are added to the input $\mathbf{x}$. The resulting mask can be removed efficiently as follows. Because $\mathbf{MG}_\mathrm{x}$ is independent of the client's input, it can be computed during preprocessing. With $\mathbf{P} := \mathbf{MG}_\mathrm{x} \in \mathbb{F}^{m \times N_\mathrm{x}}$, we can write $\mathbf{Mx}' = (\mathbf{MG}_\mathrm{x})\mathbf{e}_\mathrm{x} = \mathbf{Pe}_\mathrm{x}$. Since $\mathbf{e}_\mathrm{x}$ is sparse, computing $\mathbf{Pe}_\mathrm{x}$ requires combining only the columns of $\mathbf{P}$ corresponding to the nonzero entries of $\mathbf{e}_\mathrm{x}$. In our construction, we use a code that satisfies the above requirement under the dual-LPN assumption [21].

Overall, the client time is $O_{\lambda,\kappa}(m+n)$, while the server performs the ordinary multiplication without any overhead.

1.3 Maverick: Private and Verifiable Outsourced LLM Inference

Building on pvMVMD, we present Maverick, a system for private and verifiable outsourced LLM inference. An LLM produces its output by processing the prompt in a prefill phase and then generating tokens one by one in a decode phase. Both phases involve computing a *forward pass* for each token. The key design of Maverick is to delegate the linear operations of the forward passes (mainly MVMs), which account for the bulk of the computation, to the server, and to have the client compute the nonlinear operations locally, such as normalization, attention, and activation functions. The design is simple, yet it relieves the client of most of the work of inference. We highlight several design choices below.

Real-time vs. optimistic verification. The client can verify the server's results in real time, i.e., after each generated token, or in one batch at the end, a strategy we call *optimistic verification*. The former has the benefit of showing the user only verified tokens, while the latter yields a throughput gain due to batch verification.

Verification-only vs. privacy. Maverick can operate in two modes. The *verification-only mode* provides integrity alone: the client learns that the response is the output of the claimed model, while the server sees the prompt, as it does in today's inference services. This is a useful setting in its own right, and the target of recent systems such as Anthropic's FlashProve [6]. We optimize this mode by making it completely non-interactive: once the prompt is sent, the server evaluates the entire model and streams the results of the MVMs to the client, which replays the nonlinear operations and verifies the results. A request thus takes a single round trip, so its performance does not depend on the latency of the network.

In the *privacy mode*, the client additionally masks the inputs of the MVMs and unmasks their results, which hides the prompt and the response from the server. Each MVM then takes a round of interaction, so this mode requires a low-latency link between the client and the server. Generating the masks is a significant part of the client's work, so the client can generate them ahead of time, similar to how Beaver triples are pre-generated in multi-party computation (MPC) protocols [13]; we call this the *boost mode*.

Quantization. pvMVMD operates over a finite field, whereas a model computes in floating point. Maverick therefore quantizes the weights and the inputs of the MVMs into integers; we use an adaptive quantization strategy designed to ensure that no result can wrap around the modulus. Because the client returns to floating point between MVMs, inputs are quantized afresh and the integers do not grow from one MVM to the next, allowing us to use a small field of 31 bits. A small field has two benefits. First, field arithmetic can be performed efficiently with the native integer operations of GPUs. Second, it reduces the traffic between the client and the server (vectors of field elements).

Other optimizations. Our implementation includes further optimizations that reduce the number of rounds in the privacy mode, compress the server's outputs in the verification-only mode, and let a client with a small GPU evaluate part of the model itself to save communication.

1.4 Implementation and Evaluation

We implement our pvMVMD protocol and Maverick in about 38,000 lines of Rust and 2,000 lines of CUDA. We now highlight evaluation results.

Delegation primitive. We evaluate pvMVMD on square matrices of dimension up to $n = 2^{14}$. The client time grows approximately linearly with n, while the preprocessing and native MVM grow quadratically. At $n = 2^{14}$, the client completes its online work in 7.27 ms, whereas performing the native matrix–vector multiplication locally would take 135.12 ms, yielding a 18.60× client-side speedup. We compare the pvMVMD protocol against Dumas–Zucca [35], which is specialized for verifying matrix–vector products, and against Sum-Check instantiated with BaseFold [85,99], which represents the general proof-based approach used by recent verifiable-ML systems, including DeepProve [42]. Across the evaluated dimensions up to $n = 2^{12}$, our implementation is up to 34.8× faster than Dumas–Zucca and up to 194.8× faster than Sum-Check+BaseFold, and our reported client time additionally includes input privacy.

Verification-only mode. The verification-only mode targets computationally weak clients that cannot run large models on their own. We use two widely used open-source models: `Qwen3-32B` [75] and `Llama 4` [79], a model of 400B parameters. We evaluate it on two such clients: a laptop with an 8-core CPU, 30 GB of memory and no discrete GPU, and a desktop with a 16 GB consumer GPU and 16 GB of memory. The clients reside in a university network and the servers in data centers; they communicate over a WAN with a round-trip time of about 10 ms and a bandwidth of 1 Gb/s.

As the baseline, each client runs the model locally with llama.cpp [43], a widely used engine for local inference, in its 8-bit format. We report two quantities that a user experiences: the *time to first token* (TTFT), from sending the prompt until the first token is available and verified, and the *generation throughput*, the number of tokens generated per second.

For `Qwen3-32B`, which doesn't fit in the memory of either client, local inference is impractical: a 128-token response takes close to an hour on the laptop. With Maverick, the same clients achieve a generation throughput 81–270× higher, and a TTFT about 2× shorter on the laptop and 7–11× shorter on the desktop, which processes the prompt on its own GPU. For `Llama 4`, which neither client can run at all, the laptop and the desktop generate 7–15 and 24–33 tokens per second. By contrast, FlashProve [6], the state-of-the-art zero-knowledge proof system for LLM inference, reports 58.8 s on an H100 GPU to prove the computation of a single next token of the same model after a 1,024-token prompt, and about 21 s to verify the proof.

Privacy mode. Because the privacy mode takes a round of interaction for each MVM, it is practical where the client and the server are connected over a low-latency link. Such a setting arises in data centers: a trusted GPU, e.g., one running in a TEE, seeks to increase its throughput by outsourcing the bulk of its computation to untrusted accelerators. This is the setting of the concurrent system MOSAIC [98] We compare Maverick with MOSAIC on `Qwen2.5-72B`, with H200 GPUs connected by NVLink in the boost mode.

MOSAIC is the natural point of comparison: it targets the same deployment and, like Maverick, outsources the linear operations of the model to untrusted GPUs. The two sys-

tems make different trade-offs in what they guarantee: Maverick verifies the server's results and keeps the outputs exact, whereas MOSAIC also hides the model from the server, but it does not verify the results and adds noise to the outputs. The comparison therefore shows how efficiently each design delivers its guarantees in the same setting. With one untrusted GPU instead of MOSAIC's three, Maverick processes prompts 2.6–4.8× faster and generates tokens 10–12× faster, and its trusted side spends 19–27 ms per generated token against 43 ms. These numbers exclude the preparation of Maverick's masks, which the boost mode performs ahead of time, whereas MOSAIC generates its masks within the request.

Accuracy. Because Maverick computes over a finite field, it must quantize the model, and we measure what this costs in accuracy. We run each model on the WikiText-2 test set [67], the standard corpus for evaluating quantized models, once with the weights released by its developers and once with Maverick's quantization, and compare the perplexity of the two runs. For the two Qwen models, the perplexity differs by at most 0.04%, and for `Llama 4` by 0.2%. The quantization of Maverick thus incurs only a minor loss in accuracy.

2 Related Work

HE- and MPC-based private-inference systems protect client inputs and, in some settings, model parameters, but replace native linear algebra with encrypted or secure computation [45, 46, 49–51, 54–56, 62, 68–70, 77, 100]. Specialized protocols exploit the structure of linear computation to reduce this overhead. EMVP [14], for instance, uses structured codes to efficiently delegate encrypted matrix–vector products while additionally hiding the stored matrix, but assumes a semi-honest server and cannot provide verifiability. Braverman and Newman [24] use trapdoored matrices for delegated linear algebra, including matrix multiplication and batched MVMs with efficiency in the amortized setting. Relatedly, Abbaszadeh et al. [1] use dual-LPN-friendly codes and transparent preprocessing to privately delegate multi-scalar multiplications. However, these works do not provide efficient verifiable delegation of a single MVM in our setting.

A large body of work verifies neural-network and LLM inference using interactive proofs and SNARKs [11, 37, 42, 44, 64, 65, 73, 74, 83]. These systems can certify substantially more of the inference computation, but require additional proof-generation work at the server. Since linear layers account for a substantial fraction of inference computation, efficient verification of matrix products is a central subproblem. A direct Freivalds-style check provides information-theoretic soundness, but the verification requires quadratic time [41]. Prior work has also combined Freivalds-style verification with coding theory to reduce its randomness complexity [15]; however, the resulting verification time remains quadratic. Slalom [87] performs quadratic verification work in a private preprocessing by computing a secret reusable Freivalds state; however, this state encounters leakage over a bounded number of queries. Further, Slalom uses fresh one-time masks to provide privacy for MVMs, but the corresponding demasking term is computed in preprocessing via an independent MVM, leaving quadratic preprocessing work per query. Slalom at the Carnival [26] reduces this per-query privacy-preprocessing cost by deriving fresh masks from reusable public structure. However, we show in Appendix E that its vector-space instantiation can leak linear information about the client's input, violating its claimed input-privacy guarantee.

Publicly verifiable schemes such as Fiore–Gennaro [38], Dumas–Zucca [35], and general sum-check or SNARK-based approaches [85] instead require authentication or proof generation in addition to the underlying computation, introducing considerable additional server-side work beyond the target MVM. LAMP [61] also combines Freivalds-style verification with error-correcting codes, but uses them inside a commit-and-prove SNARK for matrix multiplication.

Other systems that provide both privacy and verifiability include verifiable computation over encrypted data through FHE+ZKP and verifiable-FHE constructions [9, 27, 39, 40, 71, 90, 102], SNARGs specialized to matrix multiplication over encrypted data [86], plaintext-authentication approaches such as VERITAS [29], whose main encodings were subsequently cryptanalyzed in the fully homomorphic setting [31], VOLE-based proof systems [91–93, 95], DataSeal's FHE-based redundancy checks [81], and schemes combining homomorphic matrix computation with Freivalds-style verification [63, 101]. These approaches require encrypted server computation, an auxiliary proof or authentication protocol, trusted hardware, or client-specific secret verification state. In contrast, Maverick achieves linear online client work with reusable transparent preprocessing and no secret verifier state; its base protocol requires neither trusted hardware nor heavy cryptographic machinery, and the server performs only one field MVM.

Concurrent work. We are aware of two concurrent papers, MOSAIC [98], and Gupta et al. [47], which have similarly employed matrix–vector multiplication delegation toward outsourcing LLM inference. MOSAIC provides privacy for the client input and model, but does not provide verifiability. Like Maverick, MOSAIC achieves asymptotically optimal online client time for an MVM; however, under the concrete parameterizations of the two protocols, Maverick has a substantially smaller constant, while its online client work additionally includes information-theoretic verification. Moreover, MOSAIC relies on both LWE and LPN and introduces Gaussian noise into the delegated computation, resulting in a nonzero protocol-error probability, whereas the MVM delegation of Maverick introduces no such error. Furthermore, Gupta et al. [47] provide both privacy and verifiability, but security relies on two non-colluding servers, whereas Maverick outsources inference to only a single server. Meanwhile, their verifiability mechanism requires a trusted setup party (as op-

posed to our transparent preprocessing) not to collude with the server performing outsourced computation; the verification protocol requires group rather than field operations, and soundness is computational rather than information-theoretic. Nevertheless, we note that both concurrent works require substantially less client-side storage for preprocessing state, which is a key bottleneck of our approach.

3 Preliminaries

Notation. We use $\mathbb{F}$ to denote a finite field, $\mathbb{F}^* = \mathbb{F} \setminus \{0\}$, and $\mathbb{F}_q$ for a prime power q to denote the field with q elements. Matrices are denoted by **bold** uppercase letters (e.g., $\mathbf{M}$) and vectors by **bold** lowercase letters (e.g., $\mathbf{x}$). For a vector $\mathbf{v}$, we write $\mathrm{wt}(\mathbf{v})$ to denote its Hamming weight, the number of nonzero entries of $\mathbf{v}$. For a vector $\mathbf{v}$ of length n, we use $\mathrm{supp}(\mathbf{v})$ to denote the support, the set of indices in $[n]$ where $\mathbf{v}$ is nonzero. We say that a vector $\mathbf{v}$ is t-sparse for a constant t if $\mathrm{wt}(\mathbf{v}) = t$. We denote the set of t-sparse vectors in $\mathbb{F}^n$ by $\mathcal{E}_t^n$.

We denote λ as a computational security parameter and κ as a statistical parameter. $\mathsf{negl}(\cdot)$ denotes a negligible function and $X \approx_c Y$ denotes computational indistinguishability.

3.1 Freivalds' Algorithm

Given matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$, Freivalds' algorithm [41] allows a verifier to efficiently check whether $\mathbf{AB} \stackrel{?}{=} \mathbf{C}$ without explicitly computing the matrix product $\mathbf{AB}$. The verifier samples a uniformly random vector $\mathbf{r}$ of the appropriate dimension and checks whether $(\mathbf{r}^\top \mathbf{A})\mathbf{B} \stackrel{?}{=} \mathbf{r}^\top \mathbf{C}$. For square $n \times n$ matrices, this requires only matrix–vector multiplications, costing $O(n^2)$ field operations, whereas explicitly computing $\mathbf{AB}$ costs $O(n^3)$ operations using the standard algorithm. If $\mathbf{AB} = \mathbf{C}$, the check always accepts; otherwise, it accepts with probability at most $|\mathbb{F}|^{-1}$.

For verifying a matrix–vector multiplication $\mathbf{Ab} \stackrel{?}{=} \mathbf{c}$, Freivalds' algorithm does not asymptotically improve the verifier's work. Computing $(\mathbf{r}^\top \mathbf{A})\mathbf{b} \stackrel{?}{=} \mathbf{r}^\top \mathbf{c}$ requires $O(n^2)$ field operations, equivalent to directly computing $\mathbf{Ab}$.

3.2 Linear Error-Correcting Codes

An error-correcting code encodes messages as fixed-length vectors, called codewords, by adding redundancy that enables error detection or correction; it is *linear* when its codewords form a linear subspace. Mathematically, let $n < N$ be integers and let $\mathbf{G} \in \mathbb{F}^{n \times N}$ have linearly independent rows. The corresponding linear code is $\mathcal{C} = \mathcal{C}(\mathbf{G}) := \{\mathbf{G}^\top \mathbf{v} : \mathbf{v} \in \mathbb{F}^n\} \subseteq \mathbb{F}^N$, and $\mathbf{G}$ is its generator matrix. The parameters n and N are the dimension and block length, respectively, and $R(\mathcal{C}) = n/N$ is the rate of $\mathcal{C}$. The minimum distance d of a linear code $\mathcal{C}$ is the minimum Hamming weight of a nonzero codeword. The *relative distance* of $\mathcal{C}$ is $\delta = d/N$.

We say that a family of codes $\{\mathcal{C}_n\}_{n=1}^{\infty}$ over a field $\mathbb{F}$ is *asymptotically good* if both the code rate R and the relative distance δ are constant for large enough n (see, e.g. [48]). Such codes provide efficiency in our constructions as they enable maximal error detection under minimal redundancy.

3.3 Learning Parity with Noise

Let $\mathcal{G}$ be a distribution over code generators $\mathbf{G} \in \mathbb{F}^{n \times N}$ and let $\mathcal{E}$ be a noise distribution over sparse vectors $\mathbf{e} \in \mathbb{F}^N$. Informally, we say the $(\mathcal{G}, \mathcal{E}, \mathbb{F})$-dual-LPN assumption holds for dimensions n and N if $(\mathbf{G}, \mathbf{Ge}) \approx_c (\mathbf{G}, \mathbf{u})$ where $\mathbf{G} \leftarrow \mathcal{G}, \mathbf{e} \leftarrow \mathcal{E}$, and $\mathbf{u} \leftarrow \mathbb{F}^n$. We say $\mathcal{G}$ is *dual-LPN-friendly* if this assumption is believed to hold for the exact noise distributions $\mathcal{E} = \mathcal{E}_t^N$ for a suitable sparsity parameter t. A formal definition of the LPN assumption and a discussion of plausibly dual-LPN-friendly code families is provided in Appendix A.

4 Model and Security Definitions

We study cryptographic primitives for private and verifiable delegation of matrix–vector products, and their applications for LLM inference. The core primitives, vMVMD and pvMVMD, involve a *client* holding private inputs $\mathbf{x}$, and an untrusted *server*. In both primitives, the client wishes to obtain $\mathbf{y} = \mathbf{M} \cdot \mathbf{x}$ from the server for a given public matrix $\mathbf{M}$. vMVMD allows the client to verify the correctness of $\mathbf{y}$, and pvMVMD additionally hides $\mathbf{x}$ from the server.

Before defining their security guarantees, we explain what it means for these primitives to be efficient and field-agnostic. For a matrix $\mathbf{M} \in \mathbb{F}^{m \times n}$ and vectors $\mathbf{x} \in \mathbb{F}^n$ and $\mathbf{y} \in \mathbb{F}^m$, computing $\mathbf{y} = \mathbf{Mx}$ locally takes $O(mn)$ time. We call a protocol ***efficient*** if it runs in time $o_{\lambda,\kappa}(mn)$ and ***asymptotically time-optimal*** if it runs in time $O_{\lambda,\kappa}(m+n)$, matching the time required to process the input and output vectors $\mathbf{x}$ and $\mathbf{y}$.

We call a protocol *field-agnostic* if it treats $\mathbb{F}$ as a black box, allowing field elements to have arbitrary labels and accessing them only through an oracle for field operations. As in [14], we assume that the oracle provides access to addition, subtraction, multiplication, inversion, unit, zero testing, and uniform sampling of field elements.

4.1 Verifiable Matrix–Vector Multiplication Delegation (vMVMD)

In verifiable matrix–vector multiplication delegation (vMVMD), a client would like to efficiently verify if a server's claim $\mathbf{y} \stackrel{?}{=} \mathbf{Mx}$ is correct. We formalize this primitive in Definition 4.1.

A vMVMD protocol consists of two algorithms: PREPROCESS, which takes the matrix $\mathbf{M}$ and outputs public

parameters pp, and VERIFY, which uses pp to verify input-output pairs $(\mathbf{x},\mathbf{y})$.

Our soundness definition guarantees information-theoretic (statistical) security. In particular, it is not only computationally hard for an adversary to find a pair $(\mathbf{x},\mathbf{y})$ that fools VERIFY, but furthermore, there does not exist any such pair $(\mathbf{x},\mathbf{y})$ that fools VERIFY with non-negligible probability taken over the random coins within the VERIFY algorithm.

Definition 4.1 (Verifiable Matrix–Vector Multiplication Delegation)**.** *A vMVMD protocol* Π *is a pair of PPT algorithms* (PREPROCESS, VERIFY) *defined by:*

- $\text{PREPROCESS}(1^\kappa,\mathbf{M}) \to \mathsf{pp}$. *On input statistical parameter* κ *and matrix* $\mathbf{M}\in\mathbb{F}^{m\times n}$*, outputs public parameters* pp.
- $\text{VERIFY}(\mathsf{pp},\mathbf{x},\mathbf{y}) \to \{0,1\}$. *On input public parameters* pp*, input vector* $\mathbf{x}\in\mathbb{F}^n$*, and output vector* $\mathbf{y}\in\mathbb{F}^m$*, either accepts (outputs 1) or rejects (outputs 0).*

The protocol should satisfy the following properties:

- ***Completeness****:* Π *is complete if for all* $\mathbf{M}\in\mathbb{F}^{m\times n}$ *and* $\mathbf{x}\in\mathbb{F}^n$*,*

$$\Pr\left[b=1 \;\middle|\; \begin{array}{l}\mathsf{pp}\leftarrow\text{PREPROCESS}(1^\kappa,\mathbf{M})\\ \mathbf{y}=\mathbf{M}\mathbf{x}\\ b\leftarrow\text{VERIFY}(\mathsf{pp},\mathbf{x},\mathbf{y})\end{array}\right]=1.$$

- ***Soundness****:* Π *is sound if for all unbounded adversaries* $\mathcal{A}$ *there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\mathbf{M}\in\mathbb{F}^{m\times n}$ *and* $\mathbf{x}\in\mathbb{F}^n$[1]*,*

$$\Pr\left[\mathbf{y}\neq\mathbf{M}\mathbf{x}\wedge b=1 \;\middle|\; \begin{array}{l}\mathsf{pp}\leftarrow\text{PREPROCESS}(1^\kappa,\mathbf{M})\\ \mathbf{y}\leftarrow\mathcal{A}(\mathsf{pp},\mathbf{M},\mathbf{x})\\ b\leftarrow\text{VERIFY}(\mathsf{pp},\mathbf{x},\mathbf{y})\end{array}\right]<\mathsf{negl}(\kappa).$$

4.2 Private and Verifiable Matrix–Vector Multiplication Delegation (pvMVMD)

In private and verifiable matrix–vector multiplication delegation (pvMVMD), a client would like to outsource the computation of $\mathbf{M}\mathbf{x}$ to the server without revealing $\mathbf{x}$. The client encrypts $\mathbf{x}$ to obtain $\widehat{\mathbf{x}}$, the server evaluates $\widehat{\mathbf{y}}=\mathbf{M}\widehat{\mathbf{x}}$, and the client verifies and decrypts the result. The server may deviate from the prescribed protocol, and a public verification algorithm provides soundness for the client against such deviations. We formalize this primitive in Definition 4.2.

Definition 4.2 (Private and Verifiable Matrix–Vector Multiplication Delegation)**.** *A pvMVMD protocol* Π *is a tuple of PPT algorithms* (PREPROCESS,ENCRYPT,VERIFY,DECRYPT) *defined by:*

- $\text{PREPROCESS}(1^\lambda,1^\kappa,\mathbf{M}) \to \mathsf{pp}$. *On input security parameter* λ*, statistical parameter* κ*, and matrix* $\mathbf{M}\in\mathbb{F}^{m\times n}$*, outputs public parameters* pp.
- $\text{ENCRYPT}(\mathsf{pp},\mathbf{x}) \to (\widehat{\mathbf{x}},\mathsf{st})$. *On input public parameters* pp *and plaintext vector* $\mathbf{x}\in\mathbb{F}^n$*, outputs the ciphertext* $\widehat{\mathbf{x}}$ *and a private state* st.
- $\text{VERIFY}(\mathsf{pp},\widehat{\mathbf{x}},\widehat{\mathbf{y}}) \to \{0,1\}$. *On input public parameters* pp*, ciphertext* $\widehat{\mathbf{x}}\in\mathbb{F}^n$*, and output vector* $\widehat{\mathbf{y}}\in\mathbb{F}^m$*, either accepts (outputs 1) or rejects (outputs 0).*
- $\text{DECRYPT}(\mathsf{pp},\mathsf{st},\widehat{\mathbf{y}}) \to \mathbf{y}$. *On input public parameters* pp*, private state* st*, and ciphertext* $\widehat{\mathbf{y}}\in\mathbb{F}^m$*, outputs the plaintext vector* $\mathbf{y}$ *which is claimed to equal* $\mathbf{M}\mathbf{x}$.

The protocol should satisfy the following properties:

- ***Completeness****:* Π *is complete if, whenever* Π *is executed honestly by both the client and the server, then the client outputs* $\mathbf{y}=\mathbf{M}\mathbf{x}$ *on input* $\mathbf{x}$ *and public matrix* $\mathbf{M}$*. Formally, for any* $\mathbf{M}\in\mathbb{F}^{m\times n}$ *and* $\mathbf{x}\in\mathbb{F}^n$*,*

$$\Pr\left[\mathbf{y}=\mathbf{M}\mathbf{x}\wedge b=1 \;\middle|\; \begin{array}{l}\mathsf{pp}\leftarrow\text{PREPROCESS}(1^\lambda,1^\kappa,\mathbf{M})\\ (\widehat{\mathbf{x}},\mathsf{st})\leftarrow\text{ENCRYPT}(\mathsf{pp},\mathbf{x})\\ \widehat{\mathbf{y}}=\mathbf{M}\widehat{\mathbf{x}}\\ b\leftarrow\text{VERIFY}(\mathsf{pp},\widehat{\mathbf{x}},\widehat{\mathbf{y}})\\ \mathbf{y}\leftarrow\text{DECRYPT}(\mathsf{pp},\mathsf{st},\widehat{\mathbf{y}})\end{array}\right]=1.$$

- ***Soundness****:* Π *is sound if a computationally unbounded server* $\mathcal{A}$ *that returns a vector* $\widehat{\mathbf{y}}\neq\mathbf{M}\widehat{\mathbf{x}}$ *will be rejected by* VERIFY *with high probability. Formally, there exists some negligible function* $\mathsf{negl}(\cdot)$ *such that for any* $\mathbf{M}\in\mathbb{F}^{m\times n}$ *and* $\mathbf{x}\in\mathbb{F}^n$*,*

$$\Pr\left[\mathbf{y}\neq\mathbf{M}\mathbf{x}\wedge b=1 \;\middle|\; \begin{array}{l}\mathsf{pp}\leftarrow\text{PREPROCESS}(1^\lambda,1^\kappa,\mathbf{M})\\ (\widehat{\mathbf{x}},\mathsf{st})\leftarrow\text{ENCRYPT}(\mathsf{pp},\mathbf{x})\\ \widehat{\mathbf{y}}\leftarrow\mathcal{A}(\mathsf{pp},\mathbf{M},\widehat{\mathbf{x}})\\ b\leftarrow\text{VERIFY}(\mathsf{pp},\widehat{\mathbf{x}},\widehat{\mathbf{y}})\\ \mathbf{y}\leftarrow\text{DECRYPT}(\mathsf{pp},\mathsf{st},\widehat{\mathbf{y}})\end{array}\right]<\mathsf{negl}(\kappa).$$

- ***Client privacy****:* Π *is private for the client if executing* Π *reveals no information about* $\mathbf{x}$ *to the server. Formally, there exists a PPT simulator* Sim *such that for any matrix* $\mathbf{M}\in\mathbb{F}^{m\times n}$ *and* $\mathbf{x}\in\mathbb{F}^n$*, the following distributions are computationally indistinguishable with respect to* κ *and* λ*:*

$$\mathsf{Sim}(1^\lambda,1^\kappa,\mathbf{M}) \approx_c \mathsf{View}^{\Pi}_{\text{Server}}(\mathbf{M},\mathbf{x}),$$

where $\mathsf{View}^{\Pi}_{\text{Server}}(\mathbf{M},\mathbf{x}) := (\mathsf{pp},\mathbf{M},\widehat{\mathbf{x}})$ *denotes the server's view, given by* $\mathsf{pp}\leftarrow\text{PREPROCESS}(1^\lambda,1^\kappa,\mathbf{M})$ *and* $(\widehat{\mathbf{x}},\mathsf{st})\leftarrow\text{ENCRYPT}(\mathsf{pp},\mathbf{x})$.

Remark 4.3 (Multi-query privacy)**.** *Client privacy extends to multiple queries via a standard hybrid argument, and the multi-query distinguishing advantage for q queries is at most q times the single-query advantage.*

[1] The adversary is unbounded, providing information-theoretic security.

Remark 4.4 (Composition of vMVMD with pvMVMD)**.** *We remark that any possibly unsound pvMVMD protocol can be made sound by using the* VERIFY *algorithm of a sound vMVMD protocol. A formal proof appears in Appendix D.*

4.3 Transparent preprocessing

We call the preprocessing *transparent* if its output contains no secret trapdoor or client-specific verification key and can be validated against the public matrix $\mathbf{M}$ by any party. The server or a third party may therefore generate and publish the preprocessing state once for all clients delegating the same matrix. A client that does not trust the publisher can independently validate the state at a cost comparable to preprocessing and then reuse it for all future queries involving $\mathbf{M}$. Alternatively, the publisher may attach a proof of correct preprocessing to reduce this per-client validation cost.

5 Verifiable Matrix–Vector Multiplication Delegation (vMVMD)

In this section, we present a novel asymptotically time-optimal vMVMD protocol using tools from coding theory. We present our construction in Section 5.1, demonstrate its security in Section 5.2, analyze its costs in Section 5.3, and extend it with a batch verification protocol in Section 5.4, tailored to the batched MVM claims arising in our LLM application.

5.1 Our Construction

We begin by modifying Freivalds' algorithm to enable faster client verification of MVMs. Recall that $\mathcal{E}_t^m$ denotes the set of t-sparse vectors of dimension m. Rather than sampling a Freivalds' challenge uniformly at random, we sample a t-sparse vector $\mathbf{e} \in \mathcal{E}_t^m$. Given a verification claim $\mathbf{Mx} \stackrel{?}{=} \mathbf{y}$, the client can compute $(\mathbf{e}^\top \mathbf{M})\mathbf{x}$ and $\mathbf{e}^\top \mathbf{y}$ in time $O(tn)$ and accept if and only if they match. When $t = O(\kappa)$, we have a protocol that runs in $O_\kappa(m+n)$ to verify instead of $O(mn)$.

However, the soundness of this protocol is not ideal. Consider an invalid expression $\mathbf{Mx} \neq \mathbf{y}$, and define the *error vector* $\mathbf{\Delta} := \mathbf{Mx} - \mathbf{y} \in \mathbb{F}^m$. The previous protocol accepts whenever $\mathbf{e}^\top \mathbf{\Delta} = 0$. For instance, if there is exactly one nonzero entry in $\mathbf{\Delta}$ (i.e., $\mathrm{wt}(\mathbf{\Delta}) = 1$), then the protocol rejects only if $\mathbf{e}$ is nonzero in the same coordinate as $\mathbf{\Delta}$, which occurs with probability t/m. This yields a soundness error of at least $1 - t/m$.

To reduce the soundness error, we use *linear* codes (see Section 3.2) for dimension m and a block length N with the following two properties: (1) the code has *high relative distance* δ when the rate $R = m/N$ is kept constant for security, and (2) the code has *fast encoding time* for efficiency. Intuitively, the encoding spreads every nonzero error vector $\mathbf{\Delta}$ into a codeword $\mathbf{G}^\top \mathbf{\Delta} \in \mathbb{F}^N$ with at least δN nonzero entries.

Π_{vMVMD}

Input parameters:

- Distribution $\mathcal{G}$ over generator matrices $\mathbf{G} \in \mathbb{F}^{m \times N}$ of codes with rate $R = \Theta(1)$, relative distance $\delta = \Theta(1)$, and linear encoding time.
- Sparsity parameter $t = t(\kappa)$ and repetition parameter $\ell = \ell(\kappa)$.
- Block length $N = O(m)$.
- Field oracle $\mathbb{F}$.

PREPROCESS$(1^\kappa, \mathbf{M}) \to \mathsf{pp}$:

- Sample $\mathbf{G} \leftarrow \mathcal{G}$.
- Compute $\mathbf{Q} := \mathbf{G}^\top \mathbf{M} \in \mathbb{F}^{N \times n}$ and output $\mathsf{pp} = (\mathbf{G}, \mathbf{Q}, t, \ell)$.

VERIFY$(\mathsf{pp}, \mathbf{x}, \mathbf{y}) \to \{0,1\}$:

- Compute $\mathbf{G}^\top \mathbf{y} \in \mathbb{F}^N$
- Repeat ℓ times:
  - Sample $\mathbf{e} \leftarrow \mathcal{E}_t^N$.
  - Compute $v_L := (\mathbf{e}^\top \mathbf{Q})\mathbf{x}$ and $v_R := \mathbf{e}^\top (\mathbf{G}^\top \mathbf{y})$
  - If $v_L \neq v_R$, output 0 (reject) and halt.
- If all ℓ checks pass, output 1 (accept).

Protocol 1: Verifiable matrix–vector multiplication delegation protocol

Let $\mathbf{G} \in \mathbb{F}^{m \times N}$ be the generator matrix of such a linear code. Rather than verifying the original claim directly, we verify its encoding under $\mathbf{G}^\top$. That is, the client samples a sparse challenge $\mathbf{e} \in \mathcal{E}_t^N$ and accepts if $(\mathbf{e}^\top \mathbf{G}^\top \mathbf{M})\mathbf{x} = \mathbf{e}^\top \mathbf{G}^\top \mathbf{y}$.

If $\mathbf{\Delta} = \mathbf{Mx} - \mathbf{y} \neq 0$, the minimum distance of the code guarantees that $\mathbf{G}^\top \mathbf{\Delta} \in \mathbb{F}^N$ has at least δN nonzero entries. Thus, the support of a uniformly random t-sparse vector $\mathbf{e}$ is likely to intersect the support of $\mathbf{G}^\top \mathbf{\Delta}$.

Computing $\mathbf{G}^\top \mathbf{M}$ for every query incurs heavy work, but we recognize that the term $\mathbf{Q} = \mathbf{G}^\top \mathbf{M}$ is independent of client input, and thus can be prepared in a preprocessing phase. Anyone can verify the correctness of $\mathbf{Q}$ since both $\mathbf{G}$ and $\mathbf{M}$ are public; our preprocessing phase adds no extra trust assumption. The full protocol is formally presented in Protocol 1.

Meanwhile, $(\mathbf{e}^\top \mathbf{Q})\mathbf{x}$ and $\mathbf{e}^\top (\mathbf{G}^\top \mathbf{y})$ can be computed efficiently, since $\mathbf{e}^\top \mathbf{Q}$ can be computed in time proportional to the sparsity of $\mathbf{e}$ and the fast encoding time of $\mathbf{G}$ makes computing $\mathbf{G}^\top \mathbf{y}$ efficient.

5.2 Security

We show that Π_{vMVMD} satisfies soundness for suitable parameters t, ℓ, and δ. We first prove the following lemma.

Lemma 5.1. *Let $\mathbf{Q} := \mathbf{G}^\top \mathbf{M} \in \mathbb{F}^{N \times n}$, and let $\mathbf{e}$ be sampled uniformly from $\mathcal{E}_t^N$. If the code generated by $\mathbf{G}$ has relative distance δ, then for any $\mathbf{x} \in \mathbb{F}^n$ and $\mathbf{y} \in \mathbb{F}^m$ such that $\mathbf{Mx} \neq \mathbf{y}$, we have* $\Pr\left[\mathbf{e}^\top (\mathbf{G}^\top \mathbf{y}) = (\mathbf{e}^\top \mathbf{Q})\mathbf{x}\right] \leq (1-\delta)^t + \frac{1}{|\mathbb{F}|-1}$.

Proof. Let $\mathbf{u} := \mathbf{y} - \mathbf{Mx} \in \mathbb{F}^m$ be the nonzero error vector and

$\mathbf{z} := \mathbf{G}^\top \mathbf{u} \in \mathbb{F}^N$ its encoding. The event $\mathbf{e}^\top(\mathbf{G}^\top \mathbf{y}) = (\mathbf{e}^\top \mathbf{Q})\mathbf{x}$ is equivalent to $\mathbf{e}^\top \mathbf{z} = 0$. Since $\mathbf{G}$ generates a code of relative distance δ, we have $\mathrm{wt}(\mathbf{z}) \geq N\delta$.

Let $S_\mathbf{e} := \mathrm{supp}(\mathbf{e})$ and $S_\mathbf{z} := \mathrm{supp}(\mathbf{z})$. We bound $\Pr[\mathbf{e}^\top \mathbf{z} = 0]$ by splitting into two cases based on whether $S_\mathbf{e}$ and $S_\mathbf{z}$ intersect and applying the law of total probability.

$$\begin{aligned}
&\Pr[\mathbf{e}^\top \mathbf{z} = 0] \\
&= \Pr[\mathbf{e}^\top \mathbf{z} = 0 \mid S_\mathbf{e} \cap S_\mathbf{z} = \emptyset] \Pr[S_\mathbf{e} \cap S_\mathbf{z} = \emptyset] \\
&+ \Pr[\mathbf{e}^\top \mathbf{z} = 0 \mid S_\mathbf{e} \cap S_\mathbf{z} \neq \emptyset] \Pr[S_\mathbf{e} \cap S_\mathbf{z} \neq \emptyset] \\
&\leq \Pr[S_\mathbf{e} \cap S_\mathbf{z} = \emptyset] + \Pr[\mathbf{e}^\top \mathbf{z} = 0 \mid S_\mathbf{e} \cap S_\mathbf{z} \neq \emptyset]
\end{aligned}$$

Term 1. By assumption, $|S_\mathbf{z}| \geq N\delta$ and $|S_\mathbf{e}| = t$. Since $S_\mathbf{e}$ is a uniformly random size-t subset of $[N]$ and $S_\mathbf{z}$ is fixed, then,

$$\Pr[S_\mathbf{e} \cap S_\mathbf{z} = \emptyset] \leq \frac{\binom{N-N\delta}{t}}{\binom{N}{t}} = \prod_{i=0}^{t-1} \frac{N-N\delta-i}{N-i} \leq \left(\frac{N-N\delta}{N}\right)^t = (1-\delta)^t.$$

Term 2. Define the multivariate polynomial $f_\mathbf{z}(\mathbf{e}) = \mathbf{e}^\top \mathbf{z}$. Conditioned on $S_\mathbf{e} \cap S_\mathbf{z} \neq \emptyset$, $f_\mathbf{z}$ is nonzero and has total degree 1 in the variables $\{e_j\}_{j \in S_\mathbf{e}}$. These variables are sampled uniformly at random from the set $\mathbb{F} \setminus \{0\}$. By the Schwartz-Zippel lemma, the probability that this polynomial evaluates to 0 is at most $\Pr[f_\mathbf{z}(\mathbf{e}) = 0 \mid S_\mathbf{e} \cap S_\mathbf{z} \neq \emptyset] \leq \frac{1}{|\mathbb{F}|-1}$.

Combining both cases yields the final probability. □

The soundness bound of Lemma 5.1 can be amplified using independent repetitions. We choose t so that the two terms in the single-iteration bound are comparable, and repeat ℓ times to obtain soundness error at most $2^{-\kappa}$. We show that Π_{vMVMD} satisfies Definition 4.1 with the following theorem.

Theorem 5.2. *If the code generated by $\mathbf{G}$ has relative distance δ, then the protocol Π_{vMVMD} satisfies completeness and soundness as defined in Definition 4.1 for $|\mathbb{F}| \geq 4$, $\ell = \left\lceil \frac{\kappa}{\log_2(|\mathbb{F}|-1)-1} \right\rceil$, and $t = \left\lceil \frac{\max(\log_2(|\mathbb{F}|-1), \kappa+1)}{-\log_2(1-\delta)} \right\rceil$.*

Proof. Completeness is immediate. For any $\mathbf{M} \in \mathbb{F}^{m \times n}$ and $\mathbf{x} \in \mathbb{F}^n$, let $\mathbf{y} = \mathbf{M}\mathbf{x}$. Then, for every t-sparse vector $\mathbf{e} \in \mathbb{F}^N$,

$$(\mathbf{e}^\top \mathbf{Q})\mathbf{x} = \mathbf{e}^\top \mathbf{G}^\top \mathbf{M}\mathbf{x} = \mathbf{e}^\top \mathbf{G}^\top \mathbf{y}.$$

For soundness, fix any unbounded adversary $\mathcal{A}$, matrix $\mathbf{M} \in \mathbb{F}^{m \times n}$, and vector $\mathbf{x} \in \mathbb{F}^n$. Condition on any public parameters pp output by PREPROCESS for which the code generated by $\mathbf{G}$ has relative distance δ, and on any output $\mathbf{y} \leftarrow \mathcal{A}(\mathsf{pp}, \mathbf{M}, \mathbf{x})$ such that $\mathbf{M}\mathbf{x} \neq \mathbf{y}$. By Lemma 5.1, each iteration of VERIFY accepts with probability at most $(1-\delta)^t + \frac{1}{|\mathbb{F}|-1}$. The protocol accepts only if all ℓ iterations pass. Since the iterations sample their challenges $\mathbf{e}$ independently, the total acceptance probability is at most $\left((1-\delta)^t + \frac{1}{|\mathbb{F}|-1}\right)^\ell$.

We now consider the two parameter spaces.

Case 1: Suppose $\frac{1}{|\mathbb{F}|-1} \leq 2^{-(\kappa+1)}$. The choice of t gives $(1-\delta)^t \leq 2^{-(\kappa+1)}$. Since $\ell = 1$, the soundness error is at most

$$(1-\delta)^t + \frac{1}{|\mathbb{F}|-1} \leq 2^{-(\kappa+1)} + 2^{-(\kappa+1)} = 2^{-\kappa}.$$

Case 2: Suppose $\frac{1}{|\mathbb{F}|-1} > 2^{-(\kappa+1)}$. The choice of t gives $(1-\delta)^t \leq \frac{1}{|\mathbb{F}|-1}$. Therefore, the soundness error is at most $\left(\frac{2}{|\mathbb{F}|-1}\right)^\ell = 2^{-\ell(\log_2(|\mathbb{F}|-1)-1)}$. By the choice of ℓ, then $\ell\big(\log_2(|\mathbb{F}|-1)-1\big) \geq \kappa$, and hence $\left(\frac{2}{|\mathbb{F}|-1}\right)^\ell \leq 2^{-\kappa}$.

The bound holds for every public parameter pp satisfying the relative distance condition and every incorrect output chosen by $\mathcal{A}$. We therefore obtain

$$\Pr\left[\mathbf{y} \neq \mathbf{M}\mathbf{x} \wedge b = 1 \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \textsc{Preprocess}(1^\kappa, \mathbf{M}) \\ \mathbf{y} \leftarrow \mathcal{A}(\mathsf{pp}, \mathbf{M}, \mathbf{x}) \\ b \leftarrow \textsc{Verify}(\mathsf{pp}, \mathbf{x}, \mathbf{y}) \end{array}\right] \leq 2^{-\kappa}.$$

Thus, Π_{vMVMD} satisfies Definition 4.1. □

Remark 5.3 (Composing with code failure probability)**.** *Suppose $\mathbf{G} \leftarrow \mathcal{G}$ is sampled from a code distribution and let $p_{\mathrm{fail}} = p_{\mathrm{fail}}(\mathcal{G}, N, \delta)$ be the probability that the sampled code has relative distance smaller than δ, then the overall soundness error of Π_{vMVMD} is at most $p_{\mathrm{fail}} + 2^{-\kappa}$, which remains negligible when p_{fail} is negligible. Thus, for code families where p_{fail} can be made arbitrarily small under practical parameters, the protocol satisfies Definition 4.1.*

5.3 Efficiency

Operation	Cost
$\mathbf{G}^\top \mathbf{y}$	$O(N)$
$\mathbf{e}^\top(\mathbf{G}^\top \mathbf{y}) = \sum_{j \in \mathrm{supp}(\mathbf{e})} e_j (\mathbf{G}^\top \mathbf{y})_j$	$2t-1$
$\mathbf{e}^\top \mathbf{Q} = \sum_{j \in \mathrm{supp}(\mathbf{e})} e_j \mathbf{Q}_{j,*}$	$(2t-1)n$
$(\mathbf{e}^\top \mathbf{Q})\mathbf{x}$	$2n-1$
ℓ repetitions total	$O(N + \ell t n)$

Table 1: Cost tally for the client online phase of Π_{vMVMD}.

For optimal efficiency, we instantiate Π_{vMVMD} with a code with linear encoding time, constant rate, and constant distance. Under the parameters for ℓ and t specified in Theorem 5.2, $\ell \cdot t = O(\kappa)$, so Π_{vMVMD} has asymptotic time $O_\kappa(m+n)$. The verification cost is summarized in Table 1. The cost is asymptotically time-optimal, since reading the input $\mathbf{x}$ and output $\mathbf{y}$ takes $O(m+n)$ time.

Preprocessing time. The preprocessing phase computes $\mathbf{Q} = \mathbf{G}^\top \mathbf{M} \in \mathbb{F}^{N \times n}$, i.e., n encodings of the columns of $\mathbf{M}$. Since the encoding time is linear with linear-encodable codes, PREPROCESS has running time $O(mn)$.

5.4 Batch Verification

In our LLM application, a single inference produces many MVM claims that must eventually be verified. The same setting arises more generally whenever a client verifies computations under a fixed collection of matrices. Suppose the client wishes to verify $\mathbf{M}_b\mathbf{x}_b \stackrel{?}{=} \mathbf{y}_b$ for $b \in [B]$, where $\mathbf{M}_b \in \mathbb{F}^{m\times n}$; claims with different dimensions can be grouped by shape and verified in separate batches.

Recall that preprocessing computes $\mathbf{Q}_b := \mathbf{G}^\top \mathbf{M}_b$. Within each verification repetition, the same sparse challenge $\mathbf{e}$ can be shared across all B claims without increasing the soundness error. However, the client must still compute $\mathbf{Q}_b^\top \mathbf{e}$ for every claim and every repetition, costing $O(B\ell tn)$ field operations, which is usually the bottleneck. Our first observation is that these challenge-dependent projections can themselves be delegated to the server.

For one challenge $\mathbf{e}$, define $\mathbf{u}_b := \mathbf{Q}_b^\top \mathbf{e}$ and stack all the B computations into the single relation $\widetilde{\mathbf{u}} = \widetilde{\mathbf{Q}}\mathbf{e}$ where $\widetilde{\mathbf{Q}} := [\mathbf{Q}_1 \mid \mathbf{Q}_2 \mid \dots \mid \mathbf{Q}_B]^\top$. We ask the server to compute $\widetilde{\mathbf{u}}$ and verify this auxiliary relation using a second code $\mathbf{G}' \in \mathbb{F}^{Bn\times N'}$. Preprocessing computes $\widetilde{\mathbf{Q}}^{\mathsf{aux}} := \widetilde{\mathbf{Q}}^\top \mathbf{G}'$. After the server fixes its auxiliary response, the client samples a fresh t'-sparse challenge $\mathbf{e}'$, computes $\mathbf{r} := \mathbf{G}'\mathbf{e}'$, and checks $\mathbf{e}^\top \widetilde{\mathbf{Q}}^{\mathsf{aux}} \mathbf{e}' \stackrel{?}{=} \mathbf{r}^\top \widetilde{\mathbf{u}}$. The full protocol repeats this primary check ℓ times and uses ℓ' independent secondary repetitions.

Several observations make this second verification much more efficient. Unlike the original challenge projection, this auxiliary relation can be checked directly by the client: because both $\mathbf{e}$ and $\mathbf{e}'$ are sparse, the left-hand side accesses only a $t \times t'$ submatrix of $\widetilde{\mathbf{Q}}^{\mathsf{aux}}$, while the right-hand side is a single inner product of length Bn. Thus, no further delegation is needed. Moreover, the full protocol runs ℓ independent primary repetitions and ℓ' independent secondary repetitions. Within each secondary repetition, the same challenge $\mathbf{e}'$ can be used to check all ℓ auxiliary responses, since all of them are fixed before $\mathbf{e}'$ is sampled. Consequently, $\mathbf{G}'\mathbf{e}'$ is computed only once per secondary repetition, although the resulting scalar check is performed against each primary response.

The resulting protocol adds one challenge–response round and moves the expensive projection work from the client to the server, reducing the client's online time from $O(BN + B\ell tn)$ naively to $O(BN + B\ell(n+t) + \ell' t' Bn + \ell\ell'(Bn + tt'))$. We give the complete protocol in Protocol 2. In the following, we analyze this protocol.

We first prove the security of $\Pi_{\text{Batch-vMVMD}}$. The proof follows the ideas in the primary Π_{vMVMD} analysis.

Theorem 5.4 (Security of batch verification)**.** *Suppose* $\mathbf{G}$ *and* $\mathbf{G}'$ *generate codes of relative distances at least* δ *and* δ'*, respectively. Let* $\varepsilon := (1-\delta)^t + \frac{1}{|\mathbb{F}|-1}$ *and* $\varepsilon' := (1-\delta')^{t'} + \frac{1}{|\mathbb{F}|-1}$*. Then* $\Pi_{\text{Batch-vMVMD}}$ *has perfect completeness and soundness error at most*

$$\varepsilon^\ell + (\varepsilon')^{\ell'}.$$

In particular, the parameters can be chosen so that the soundness error is at most $2^{-\kappa}$*.*

Proof. Completeness follows directly from $\mathbf{u}_b^{(q)} = \mathbf{Q}_b^\top \mathbf{e}^{(q)}$ and $\widetilde{\mathbf{u}}^{(q)} = \widetilde{\mathbf{Q}}\mathbf{e}^{(q)}$.

For soundness, first suppose every auxiliary response is correct. Fix any incorrect original claim $b^\star$ and let $\boldsymbol{\Delta}_{b^\star} := \mathbf{y}_{b^\star} - \mathbf{M}_{b^\star}\mathbf{x}_{b^\star} \neq \mathbf{0}$. Acceptance in primary repetition q requires $(\mathbf{e}^{(q)})^\top \mathbf{G}^\top \boldsymbol{\Delta}_{b^\star} = 0$. By Lemma 5.1, this occurs with probability at most ε per repetition, and hence with probability at most ε^ℓ over all ℓ independent repetitions. Sharing $\mathbf{e}^{(q)}$ across the B claims introduces no factor B, since batch acceptance requires this fixed incorrect claim to accept.

Otherwise, fix any incorrect auxiliary response $q^\star \in [\ell]$ and define $\boldsymbol{\Gamma}^{(q^\star)} := \widetilde{\mathbf{u}}^{(q^\star)} - \widetilde{\mathbf{Q}}\mathbf{e}^{(q^\star)} \neq \mathbf{0}$. This vector is fixed before the secondary challenges are sampled. Acceptance in secondary repetition j requires $(\mathbf{e}'^{(j)})^\top \mathbf{G}'^\top \boldsymbol{\Gamma}^{(q^\star)} = 0$, which again occurs with probability at most ε' by Lemma 5.1. Thus all ℓ' secondary repetitions accept with probability at most $(\varepsilon')^{\ell'}$. Sharing each secondary challenge across the ℓ auxiliary responses introduces no factor ℓ, since batch acceptance requires the fixed incorrect response $q^\star$ to accept.

Combining the two cases gives the claimed bound. □

If either code is sampled from a distribution that may have relative distance below its target value, the corresponding code-distance failure probabilities are added to the above bound, as in Remark 5.3.

Client online time. The client first computes $\mathbf{z}_b := \mathbf{G}^\top \mathbf{y}_b$ for all $b \in [B]$, costing $O(BN)$ using the primary linear-time encoder. The original checks then cost $O(B\ell(n+t))$.

For each secondary repetition j, since $\mathbf{e}'^{(j)}$ is t'-sparse, $\mathbf{G}'\mathbf{e}'^{(j)} = \sum_{k\in \mathrm{supp}(\mathbf{e}'^{(j)})} e'_k \mathbf{G}'_{*,k}$ and can therefore be computed in $O(t'Bn)$ field operations. Because the same secondary challenge is shared across all ℓ primary responses, this cost is incurred only ℓ' times.

For every pair (q, j), computing $(\mathbf{e}^{(q)})^\top \widetilde{\mathbf{Q}}^{\mathsf{aux}} \mathbf{e}'^{(j)}$ costs $O(tt')$, while $(\mathbf{r}^{(j)})^\top \widetilde{\mathbf{u}}^{(q)}$ costs $O(Bn)$. Hence the total online client time is

$$O\big(BN + B\ell(n+t) + \ell' t' Bn + \ell\ell'(Bn + tt')\big).$$

Independent verification instead pays $O(B\ell tn)$ for computing the challenge-dependent projections $\mathbf{Q}_b^\top \mathbf{e}^{(q)}$. Thus, batch verification moves this work to the server. For constant-distance codes, t, ℓ, t', and ℓ' depend only on κ, so the client time is $O_\kappa(B(m+n))$.

The optimization adds one challenge–response round. The server performs $O(B\ell tn)$ additional field operations and returns $B\ell n$ field elements.

Preprocessing and state. Batch verification additionally preprocesses $\widetilde{\mathbf{Q}}^{\mathsf{aux}} := \widetilde{\mathbf{Q}}^\top \mathbf{G}' \in \mathbb{F}^{N\times N'}$. If applying the secondary encoder to a vector in $\mathbb{F}^{Bn}$ costs $T_{\mathbf{G}'}(Bn, N')$, this requires $O(NT_{\mathbf{G}'}(Bn, N'))$ preprocessing time and NN' field elements

of additional state. For a constant-rate, linear-time encodable secondary code, $N' = \Theta(Bn)$, so both quantities are $O(Bmn)$.

The storage can be reduced when $\mathbf{G}'$ is a systematic code. If $\mathbf{G}' = [I_{Bn} \mid \mathbf{P}]$, then $\widetilde{\mathbf{Q}}^{\mathsf{aux}} = [\widetilde{\mathbf{Q}}^\top \mid \widetilde{\mathbf{Q}}^\top \mathbf{P}]$. The first block is already represented by $\mathbf{Q}_1, \ldots, \mathbf{Q}_B$, so only $\widetilde{\mathbf{Q}}^\top \mathbf{P}$ needs additional storage. This optimization reduces storage but does not inherently reduce preprocessing time, which depends on the cost of applying the secondary encoder.

The auxiliary preprocessing depends only on the fixed matrices in the batch and can therefore be reused across future batches involving the same matrices.

6 Private and Verifiable Matrix–Vector Multiplication Delegation (pvMVMD)

In this section, we present a construction for pvMVMD by combining Π_{vMVMD} in Section 5 for verification with dual-LPN-friendly linear codes for efficient privacy. We introduce our construction in Section 6.1, analyze its security in Section 6.2, and its costs in Section 6.3.

6.1 Our Construction

To encrypt a vector $\mathbf{x} \in \mathbb{F}^n$, the client masks $\mathbf{x}$ with a vector $\mathbf{x}'$ to form the ciphertext $\widehat{\mathbf{x}} = \mathbf{x} + \mathbf{x}' \in \mathbb{F}^n$. The client sends $\widehat{\mathbf{x}}$ to the server, which returns $\widehat{\mathbf{y}} = \mathbf{M}\widehat{\mathbf{x}}$. The client verifies the correctness of $\widehat{\mathbf{y}}$ by running a vMVMD protocol over the ciphertexts. The client decrypts $\widehat{\mathbf{y}}$ by computing $\mathbf{y} = \widehat{\mathbf{y}} - \mathbf{M}\mathbf{x}' \in \mathbb{F}^m$. However, if $\mathbf{x}'$ is sampled uniformly at random, the time required for the client to compute $\mathbf{M}\mathbf{x}'$ matches the time for the client to locally compute $\mathbf{y} = \mathbf{M}\mathbf{x}$, rendering it useless to outsource this computation. To solve the efficiency problem, we prepare $\mathbf{x}'$ pseudorandomly such that $\mathbf{M}\mathbf{x}'$ can be computed in time $O(m+n)$. We make use of coding theory to construct such pseudorandom vectors.

Our construction for privacy follows the idea of prior works [1,14,24]. We use a separate distribution of linear codes $\mathcal{G}_{\mathrm{x}}$ for privacy, distinct from the code used by Π_{vMVMD} in Section 5. Let $\mathbf{G}_{\mathrm{x}} \in \mathbb{F}^{n \times N_{\mathrm{x}}}$ denote a sampled generator matrix, where N_{x} is the block length of the privacy code. Under the $(\mathcal{G}_{\mathrm{x}}, \mathcal{E}_{t_{\mathrm{x}}}^{N_{\mathrm{x}}}, \mathbb{F})$-dual-LPN assumption, if $\mathbf{e}_{\mathrm{x}} \in \mathbb{F}^{N_{\mathrm{x}}}$ is t_{x}-sparse then $\mathbf{x}' := \mathbf{G}_{\mathrm{x}}\mathbf{e}_{\mathrm{x}}$ is a pseudorandom vector. We mask $\mathbf{x}$ by computing $\widehat{\mathbf{x}} = \mathbf{x} + \mathbf{x}'$. By first computing $\mathbf{P} := \mathbf{M}\mathbf{G}_{\mathrm{x}} \in \mathbb{F}^{m \times N_{\mathrm{x}}}$ in a preprocessing phase, we can remove the mask by computing $\mathbf{M}\mathbf{x}' = \mathbf{M}(\mathbf{G}_{\mathrm{x}}\mathbf{e}_{\mathrm{x}}) = (\mathbf{M}\mathbf{G}_{\mathrm{x}})\mathbf{e}_{\mathrm{x}} = \mathbf{P}\mathbf{e}_{\mathrm{x}}$. Since $\mathbf{e}_{\mathrm{x}}$ is sparse, computing $\mathbf{P}\mathbf{e}_{\mathrm{x}}$ requires combining only the columns of $\mathbf{P}$ corresponding to the nonzero entries of $\mathbf{e}_{\mathrm{x}}$.

The full protocol Π_{pvMVMD} is presented in Protocol 3.

The client privacy property can be strengthened to additionally hide a private matrix $\mathbf{M}$ owned by the client. While such a protocol enables greater privacy, our construction requires a private client-run preprocessing, rendering this most useful when the preprocessing can be amortized across many queries. For further details on our construction, see Appendix C.

$\Pi_{\text{Batch-vMVMD}}$

Input parameters:

- Distribution $\mathcal{G}$ over generator matrices $\mathbf{G} \in \mathbb{F}^{m \times N}$ of codes with rate $R = \Theta(1)$, relative distance $\delta = \Theta(1)$, and efficient encoding time.
- Distribution $\mathcal{G}'$ over generator matrices $\mathbf{G}' \in \mathbb{F}^{Bn \times N'}$ of codes with rate $R' := Bn/N'$, relative distance δ', and efficient encoding time.
- Primary sparsity and repetition parameters t and ℓ.
- Secondary sparsity and repetition parameters t' and ℓ'.
- Primary block length $N = O(m)$ and secondary block length N'.
- Field oracle $\mathbb{F}$.

$\textsc{Preprocess}(1^\kappa, \{\mathbf{M}_b\}_{b\in[B]}) \to \mathsf{pp}$:

- Sample $\mathbf{G} \leftarrow \mathcal{G}$ and $\mathbf{G}' \leftarrow \mathcal{G}'$.
- For every $b \in [B]$, compute $\mathbf{Q}_b := \mathbf{G}^\top \mathbf{M}_b \in \mathbb{F}^{N \times n}$.
- Form $\widetilde{\mathbf{Q}} := (\mathbf{Q}_1, \ldots, \mathbf{Q}_B)^\top \in \mathbb{F}^{Bn \times N}$.
- Compute $\widetilde{\mathbf{Q}}^{\mathsf{aux}} := \widetilde{\mathbf{Q}}^\top \mathbf{G}' \in \mathbb{F}^{N \times N'}$.
- Output $\mathsf{pp} := \left(\mathbf{G}, \mathbf{G}', \{\mathbf{Q}_b\}_{b\in[B]}, \widetilde{\mathbf{Q}}^{\mathsf{aux}}, t, \ell, t', \ell'\right)$.
- Make the public matrices $\{\mathbf{Q}_b\}_{b\in[B]}$ available to the server.

$\textsc{Verify}(\mathsf{pp}, \{(\mathbf{x}_b, \mathbf{y}_b)\}_{b\in[B]}) \to \{0,1\}$:

- **Client:** Require that all alleged outputs $\mathbf{y}_1, \ldots, \mathbf{y}_B$ are fixed before sampling any verification challenge.
- **Client:** For every $b \in [B]$, compute $\mathbf{z}_b := \mathbf{G}^\top \mathbf{y}_b \in \mathbb{F}^N$.
- **Client:** Sample independent primary challenges $\mathbf{e}^{(1)}, \ldots, \mathbf{e}^{(\ell)} \leftarrow \mathcal{E}_t^N$ and send $\{\mathbf{e}^{(q)}\}_{q\in[\ell]}$ to the server. Within each repetition q, the same challenge $\mathbf{e}^{(q)}$ is used for all B claims.
- **Server:** For every $q \in [\ell]$, compute $\widetilde{\mathbf{u}}^{(q)} := \widetilde{\mathbf{Q}}\mathbf{e}^{(q)} \in \mathbb{F}^{Bn}$ and return $\{\widetilde{\mathbf{u}}^{(q)}\}_{q\in[\ell]}$ to the client.
- **Client:** After all server responses are fixed, sample independent secondary challenges $\mathbf{e}'^{(1)}, \ldots, \mathbf{e}'^{(\ell')} \leftarrow \mathcal{E}_{t'}^{N'}$.
- **Client:** For every $j \in [\ell']$, compute $\mathbf{r}^{(j)} := \mathbf{G}'\mathbf{e}'^{(j)} \in \mathbb{F}^{Bn}$.
- **Client:** For every $q \in [\ell]$ and $j \in [\ell']$, check $(\mathbf{e}^{(q)})^\top \widetilde{\mathbf{Q}}^{\mathsf{aux}} \mathbf{e}'^{(j)} \stackrel{?}{=} (\mathbf{r}^{(j)})^\top \widetilde{\mathbf{u}}^{(q)}$. If any secondary check fails, output 0 (reject) and halt.
- **Client:** For every $q \in [\ell]$, parse $\widetilde{\mathbf{u}}^{(q)} := ((\mathbf{u}_1^{(q)})^\top, \ldots, (\mathbf{u}_B^{(q)})^\top)^\top$, where $\mathbf{u}_b^{(q)} \in \mathbb{F}^n$.
- **Client:** For every $b \in [B]$ and $q \in [\ell]$, check $(\mathbf{u}_b^{(q)})^\top \mathbf{x}_b \stackrel{?}{=} (\mathbf{e}^{(q)})^\top \mathbf{z}_b$. If any original check fails, output 0 (reject) and halt. If all checks pass, output 1 (accept).

Protocol 2: Batch verification for multiple matrices and vectors. The client delegates the stacked matrix-dependent challenge projection to the server and verifies the returned vector using a second coded check.

Π_{pvMVMD}

Input parameters:

- Distribution $\mathcal{G}_{\mathrm{x}}$ over generator matrices $\mathbf{G}_{\mathrm{x}} \in \mathbb{F}^{n \times N_{\mathrm{x}}}$ of codes with rate $R_{\mathrm{x}} = \Theta(1)$ and relative distance $\delta_{\mathrm{x}} = \Theta(1)$, linear encoding time, and satisfying the dual-LPN assumption for error distribution $\mathcal{E}_{t_{\mathrm{x}}}^{N_{\mathrm{x}}}$.
- Distribution $\mathcal{G}_{\mathrm{y}}$ over generator matrices $\mathbf{G}_{\mathrm{y}} \in \mathbb{F}^{m \times N_{\mathrm{y}}}$ of codes with rate $R_{\mathrm{y}} = \Theta(1)$, relative distance $\delta_{\mathrm{y}} = \Theta(1)$, and linear encoding time.
- Sparsity parameters $t_{\mathrm{x}} = t_{\mathrm{x}}(\lambda)$ and $t_{\mathrm{y}} = t_{\mathrm{y}}(\kappa)$ and repetition parameter $\ell = \ell(\kappa)$.
- Block lengths $N_{\mathrm{x}} = O(n)$ and $N_{\mathrm{y}} = O(m)$.
- Field oracle $\mathbb{F}$.

$\textsc{Preprocess}(1^{\lambda}, 1^{\kappa}, \mathbf{M}) \to \mathsf{pp}$:

- Sample $\mathbf{G}_{\mathrm{x}} \leftarrow \mathcal{G}_{\mathrm{x}}$ and $\mathbf{G}_{\mathrm{y}} \leftarrow \mathcal{G}_{\mathrm{y}}$.
- Compute $\mathbf{P} := \mathbf{M}\mathbf{G}_{\mathrm{x}} \in \mathbb{F}^{m \times N_{\mathrm{x}}}$.
- Compute $\mathbf{Q} := \mathbf{G}_{\mathrm{y}}^{\top}\mathbf{M} \in \mathbb{F}^{N_{\mathrm{y}} \times n}$.
- Output $\mathsf{pp} = (\mathbf{G}_{\mathrm{x}}, \mathbf{G}_{\mathrm{y}}, \mathbf{P}, \mathbf{Q}, t_{\mathrm{x}}, t_{\mathrm{y}}, \ell)$.

$\textsc{Encrypt}(\mathsf{pp}, \mathbf{x}) \to (\widehat{\mathbf{x}}, \mathsf{st})$:

- Sample $\mathbf{e}_{\mathrm{x}} \leftarrow \mathcal{E}_{t_{\mathrm{x}}}^{N_{\mathrm{x}}}$ and compute $\mathbf{x}' := \mathbf{G}_{\mathrm{x}}\mathbf{e}_{\mathrm{x}} \in \mathbb{F}^{n}$.
- Output $\widehat{\mathbf{x}} := \mathbf{x} + \mathbf{x}'$ and $\mathsf{st} := \mathbf{e}_{\mathrm{x}}$.

$\textsc{Verify}(\mathsf{pp}, \widehat{\mathbf{x}}, \widehat{\mathbf{y}}) \to \{0,1\}$:

- Compute $\mathbf{G}_{\mathrm{y}}^{\top}\widehat{\mathbf{y}} \in \mathbb{F}^{N_{\mathrm{y}}}$
- Repeat ℓ times:
  - Sample $\mathbf{e}_{\mathrm{y}} \leftarrow \mathcal{E}_{t_{\mathrm{y}}}^{N_{\mathrm{y}}}$.
  - Compute $v_L := (\mathbf{e}_{\mathrm{y}}^{\top}\mathbf{Q})\widehat{\mathbf{x}}$ and $v_R := \mathbf{e}_{\mathrm{y}}^{\top}(\mathbf{G}_{\mathrm{y}}^{\top}\widehat{\mathbf{y}})$
  - If $v_L \neq v_R$, output 0 (reject) and halt.
- If all ℓ checks pass, output 1 (accept).

$\textsc{Decrypt}(\mathsf{pp}, \mathsf{st}, \widehat{\mathbf{y}}) \to \mathbf{y}$:

- Output $\mathbf{y} := \widehat{\mathbf{y}} - \mathbf{P}\mathbf{e}_{\mathrm{x}} \in \mathbb{F}^{m}$.

Protocol 3: Private and verifiable matrix–vector multiplication delegation protocol

6.2 Security

We now demonstrate that Π_{pvMVMD} satisfies completeness, soundness, and client privacy according to Definition 4.2. For verification, we use the same parameters as in Section 5. For privacy, the security analysis of prior works [21] demonstrates that a good minimum distance suffices for the dual-LPN assumption against known attacks under certain conditions (see Appendix A), and they provide the following heuristic for selecting t_{x} as a function of λ, N_{x} and δ_{x}:

$$t_{\mathrm{x}} \geq \frac{(\ln 2)(\lambda - \log_2(N_{\mathrm{x}}))}{\delta_{\mathrm{x}}}. \tag{1}$$

We therefore choose an asymptotically good code for $\mathbf{G}_{\mathrm{x}}$, with constant rate $R_{\mathrm{x}} = \Theta(1)$, $N_{\mathrm{x}} = n/R_{\mathrm{x}} = O(n)$, and good relative distance $\delta_{\mathrm{x}} = \Theta(1)$, guaranteeing $t_{\mathrm{x}} = O_{\lambda}(1)$ under this heuristic. We show that Π_{pvMVMD} satisfies Definition 4.2 in Theorem 6.1.

Theorem 6.1. *Suppose that $\mathcal{G}_x$ is a distribution over codes with relative distance at least δ_x satisfying the $(\mathcal{G}_x, \mathcal{E}_{t_x}^{N_x}, \mathbb{F})$-dual-LPN assumption for polynomially many independent samples, and that the code generated by $\mathbf{G}_y$ has relative distance δ_y. Then Π_{pvMVMD} satisfies completeness, soundness, and client privacy as defined in Definition 4.2 when instantiated with $|\mathbb{F}| \geq 4$, $\ell = \left\lceil \frac{\kappa}{\log_2(|\mathbb{F}|-1)-1} \right\rceil$, $t_y = \left\lceil \frac{\max(\log_2(|\mathbb{F}|-1), \kappa+1)}{-\log_2(1-\delta_y)} \right\rceil$.*

Proof. Completeness is immediate. For any $\mathbf{M} \in \mathbb{F}^{m \times n}$ and $\mathbf{x} \in \mathbb{F}^{n}$, let $\mathbf{e}_{\mathrm{x}} \leftarrow \mathcal{E}_{t_{\mathrm{x}}}^{N_{\mathrm{x}}}$, $\mathbf{x}' := \mathbf{G}_{\mathrm{x}}\mathbf{e}_{\mathrm{x}}$, and $\widehat{\mathbf{x}} := \mathbf{x} + \mathbf{x}'$. If the server evaluates honestly, then $\widehat{\mathbf{y}} = \mathbf{M}\widehat{\mathbf{x}}$. By completeness of the verification protocol, $\textsc{Verify}$ accepts with probability 1. Moreover, since $\mathbf{P} = \mathbf{M}\mathbf{G}_{\mathrm{x}}$, decryption gives

$$\begin{aligned}\mathbf{y} &= \widehat{\mathbf{y}} - \mathbf{P}\mathbf{e}_{\mathrm{x}} = \mathbf{M}\widehat{\mathbf{x}} - \mathbf{M}\mathbf{G}_{\mathrm{x}}\mathbf{e}_{\mathrm{x}} \\ &= \mathbf{M}(\mathbf{x} + \mathbf{G}_{\mathrm{x}}\mathbf{e}_{\mathrm{x}}) - \mathbf{M}\mathbf{G}_{\mathrm{x}}\mathbf{e}_{\mathrm{x}} = \mathbf{M}\mathbf{x}.\end{aligned}$$

For soundness, fix any unbounded adversary $\mathcal{A}$, matrix $\mathbf{M} \in \mathbb{F}^{m \times n}$, and input $\mathbf{x} \in \mathbb{F}^{n}$. Condition on any public parameters pp output by $\textsc{Preprocess}$ for which the code generated by $\mathbf{G}_{\mathrm{y}}$ has relative distance δ_{y}, and on any fresh privacy randomness $\mathbf{e}_{\mathrm{x}}$ used to form $\widehat{\mathbf{x}} = \mathbf{x} + \mathbf{G}_{\mathrm{x}}\mathbf{e}_{\mathrm{x}}$. Let $\widehat{\mathbf{y}} \leftarrow \mathcal{A}(\mathsf{pp}, \mathbf{M}, \widehat{\mathbf{x}})$ be the server's response, and suppose that verification accepts. The decrypted output satisfies $\mathbf{y} = \widehat{\mathbf{y}} - \mathbf{P}\mathbf{e}_{\mathrm{x}}$. Using $\mathbf{P} = \mathbf{M}\mathbf{G}_{\mathrm{x}}$, we have

$$\begin{aligned}\mathbf{y} - \mathbf{M}\mathbf{x} &= \widehat{\mathbf{y}} - \mathbf{M}\mathbf{G}_{\mathrm{x}}\mathbf{e}_{\mathrm{x}} - \mathbf{M}\mathbf{x} \\ &= \widehat{\mathbf{y}} - \mathbf{M}(\mathbf{x} + \mathbf{G}_{\mathrm{x}}\mathbf{e}_{\mathrm{x}}) \\ &= \widehat{\mathbf{y}} - \mathbf{M}\widehat{\mathbf{x}}.\end{aligned}$$

Therefore, if $\mathbf{y} \neq \mathbf{M}\mathbf{x}$, then $\widehat{\mathbf{y}} \neq \mathbf{M}\widehat{\mathbf{x}}$. By Theorem 5.2, under the stated choices of t_{y} and ℓ, $\textsc{Verify}$ accepts such an incorrect response with probability at most $2^{-\kappa}$, proving soundness.

For client privacy, the simulator Sim on input $(1^{\lambda}, 1^{\kappa}, \mathbf{M})$ runs $\mathsf{pp} \leftarrow \textsc{Preprocess}(1^{\lambda}, 1^{\kappa}, \mathbf{M})$, samples $\widetilde{\mathbf{x}} \leftarrow \mathbb{F}^{n}$ uniformly at random, and outputs $(\mathsf{pp}, \mathbf{M}, \widetilde{\mathbf{x}})$. By the $(\mathcal{G}_{\mathrm{x}}, \mathcal{E}_{t_{\mathrm{x}}}^{N_{\mathrm{x}}}, \mathbb{F})$-dual-LPN assumption, $(\mathbf{G}_{\mathrm{x}}, \mathbf{G}_{\mathrm{x}}\mathbf{e}_{\mathrm{x}}) \approx_c (\mathbf{G}_{\mathrm{x}}, \mathbf{u})$, where $\mathbf{G}_{\mathrm{x}} \leftarrow \mathcal{G}_{\mathrm{x}}$, $\mathbf{e}_{\mathrm{x}} \leftarrow \mathcal{E}_{t_{\mathrm{x}}}^{N_{\mathrm{x}}}$, and $\mathbf{u} \leftarrow \mathbb{F}^{n}$. Consequently, for any $\mathbf{x} \in \mathbb{F}^{n}$, $(\mathsf{pp}, \mathbf{M}, \mathbf{x} + \mathbf{G}_{\mathrm{x}}\mathbf{e}_{\mathrm{x}}) \approx_c (\mathsf{pp}, \mathbf{M}, \mathbf{x} + \mathbf{u})$. Because $\mathbf{u}$ is uniformly distributed over $\mathbb{F}^{n}$, the vector $\mathbf{x} + \mathbf{u}$ is also identically distributed to a uniform vector $\widetilde{\mathbf{x}} \leftarrow \mathbb{F}^{n}$. Hence,

$$\begin{aligned}&\mathsf{View}_{\mathrm{Server}}^{\Pi}(\mathbf{M}, \mathbf{x}) = (\mathsf{pp}, \mathbf{M}, \widehat{\mathbf{x}}) \\ &\quad \approx_c (\mathsf{pp}, \mathbf{M}, \widetilde{\mathbf{x}}) = \mathsf{Sim}(1^{\lambda}, 1^{\kappa}, \mathbf{M}).\end{aligned}$$

□

Remark 6.2 (Code failure probability in Π_{pvMVMD})**.** *As in Remark 5.3, when the verification code $\mathbf{G}_y$ is sampled from a distribution with failure probability p_{fail}, the overall soundness error is at most $p_{\mathrm{fail}} + 2^{-\kappa}$. Similarly, the privacy guarantee relies on the dual-LPN assumption, which requires that $\mathbf{G}_x$ generates a code with sufficient minimum distance. Both failure probabilities can be made negligible for suitable code families that we consider for instantiation.*

6.3 Efficiency

We now analyze the online runtime of the client for Π_{pvMVMD} under the specified parameters above.

In Table 2, we break down the runtime of each operation performed by the client in ENCRYPT and DECRYPT.

Phase	Operation	Cost
ENCRYPT	$\mathbf{x}' := \mathbf{G}_{\mathrm{x}}\mathbf{e}_{\mathrm{x}}$	$O(N_{\mathrm{x}})$
	$\widehat{\mathbf{x}} := \mathbf{x} + \mathbf{x}'$	n
DECRYPT	$\mathbf{Pe}_{\mathrm{x}} = \sum_{j \in \mathrm{supp}(\mathbf{e}_{\mathrm{x}})} e_{\mathbf{x},j}\mathbf{P}_{*,j}$	$(2t_{\mathrm{x}} - 1)m$
	$\mathbf{y} := \widehat{\mathbf{y}} - \mathbf{Pe}_{\mathrm{x}}$	m
VERIFY	See Table 1	$O\left(N_{\mathrm{y}} + \ell t_{\mathrm{y}} n\right)$
Total	–	$O\left(N_{\mathrm{x}} + N_{\mathrm{y}} + n\ell t_{\mathrm{y}} + mt_{\mathrm{x}}\right)$

Table 2: Cost tally for the client online phase of Π_{pvMVMD}.

Since we use the VERIFY algorithm of Π_{vMVMD} (Section 5.1), VERIFY runs in time $O_{\kappa}(m+n)$. ENCRYPT and DECRYPT have running time $O_{\lambda}(m+n)$, which guarantees that the total client online time of Π_{pvMVMD} is $O_{\lambda,\kappa}(m+n)$.

Preprocessing time. The preprocessed matrix $\mathbf{P} = \mathbf{MG}_{\mathrm{x}}$ can be computed by realizing $\mathbf{P}^{\top} = \mathbf{G}_{\mathrm{x}}^{\top}\mathbf{M}^{\top}$ as m encodings of the rows of $\mathbf{M}$. Similarly, $\mathbf{Q} = \mathbf{G}_{\mathrm{y}}^{\top}\mathbf{M}$ can be realized as n encodings of the columns of $\mathbf{M}$. When $\mathbf{G}_{\mathrm{x}}$ and $\mathbf{G}_{\mathrm{y}}$ have linear encoding times, PREPROCESS has running time $O(mn)$.

7 Implementation and Evaluations

We implement Π_{pvMVMD} and an end-to-end prototype of Maverick. In this section, we report on the implementation details and the evaluation results.

7.1 Evaluation of pvMVMD

Instantiation of codes. We implement Π_{pvMVMD} (Protocol 3) in Rust. We use the BabyBear field $\mathbb{F}_q$, where $q = 15 \cdot 2^{27} + 1$, and set the computational and statistical security parameters to $\lambda = 128$ and $\kappa = 40$, respectively.

We instantiate the privacy and verification codes using the RAA code construction of Akhiani et al. [3] (also see Appendix B), with rate $R_{\mathrm{x}} = R_{\mathrm{y}} = \frac{1}{8}$ and target relative distance $\delta_{\mathrm{x}} = \delta_{\mathrm{y}} = \frac{1}{2}$. We use RAA codes because they are linear-time encodable, field-agnostic, asymptotically good, and plausibly dual-LPN-friendly. For further discussion on these properties for RAA codes, see Appendix B.

We choose sparsity t_{x} according to (1): for $n \in \{2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}\}$ this gives $t_{\mathrm{x}} \in \{160, 159, 157, 156, 154\}$, respectively. We choose sparsity parameter $t_{\mathrm{y}} = 31$ and number of repetitions $\ell = 2$ per Theorem 6.1.

Experimental setup. We evaluate our pvMVMD implementation on a research cluster with Intel Xeon Platinum 8562Y+ CPUs, using a single thread. Unless otherwise noted, we measure computation time and exclude network communication. For simplicity, our experiments focus on square matrices where $m = n$.

n	Server	Client			Client speedup
		Total	Verification	Privacy	
2^8	0.02	0.14	0.08	0.06	0.14×
2^9	0.06	0.23	0.14	0.09	0.27×
2^{10}	0.25	0.41	0.25	0.15	0.62×
2^{11}	1.99	0.78	0.47	0.31	2.54×
2^{12}	8.24	1.56	0.91	0.65	5.29×
2^{13}	31.66	3.19	1.81	1.38	9.93×
2^{14}	135.12	7.27	3.95	3.31	18.60×

Table 3: Single-threaded online server and client runtime in Π_{pvMVMD}, in milliseconds. Values are means over three independent processes, each with ten measured online invocations after three warmups. Client speedup is the server MVM time divided by total client time, computed before rounding.

Online Time. The *online runtime* refers to the runtime after preprocessing. For clients, this includes ENCRYPT, VERIFY, and DECRYPT. **Client (total)** reports the total online time, broken down into **Client (verification)** for Π_{pvMVMD}.VERIFY, and **Client (privacy)** for Π_{pvMVMD}.ENCRYPT and Π_{pvMVMD}.DECRYPT. For servers, the online computation is the MVM on the masked input. The results are shown in Table 3.

For small matrices, the client online time is worse than the server time (i.e., outsourcing is not worthwhile). However, the advantage becomes significant as matrix dimension grows. At $n = 2^{13}$, the total client online time is 3.19 ms, consisting of 1.81 ms for verification and 1.38 ms for privacy, while the server multiplication takes 31.66 ms. Thus, the client's online work is approximately 9.93× smaller than the delegated multiplication. At $n = 2^{14}$, this gap increases to 18.60×, consistent with linear client and quadratic server costs.

Preprocessing time. Table 4 presents the preprocessing overhead. Recall that preprocessing involves computing $\mathbf{P} := \mathbf{MG}_{\mathrm{x}}$ and $\mathbf{Q} := \mathbf{G}_{\mathrm{y}}^{\top}\mathbf{M}$ where $\mathbf{M}$ is the public matrix and $\mathbf{G}_{\mathrm{x}}, \mathbf{G}_{\mathrm{y}}$ are sampled from code generator distributions. The computation overhead is essentially the same as encoding $\mathbf{M}$. As expected, preprocessing grows approximately quadratically with the matrix dimension because both $\mathbf{P}$ and $\mathbf{Q}$ require processing the complete matrix. At $n = 2^{14}$, preprocessing takes approximately 272.94 s, including 148.64 s for privacy-related preprocessing and 124.30 s for verification-related preprocessing.

Comparison with related work. We compare our protocol with Dumas–Zucca [35] and Sum-Check instantiated with the BaseFold polynomial commitment scheme [85, 99], where Dumas–Zucca is a specialized protocol for verifying matrix–vector products and Sum-Check+BaseFold represents the gen-

n	Preprocessing (total)	Preprocessing (verification)	Preprocessing (privacy)
2^8	13.58	7.42	6.16
2^9	51.53	28.65	22.88
2^{10}	280.98	159.34	121.63
2^{11}	1,493.87	874.19	619.67
2^{12}	7,938.89	4,727.21	3,211.68
2^{13}	44,473.28	27,722.33	16,750.95
2^{14}	272,935.60	148,636.17	124,299.42

Table 4: One-time preprocessing runtime for square matrices in milliseconds, averaged over three independent processes.

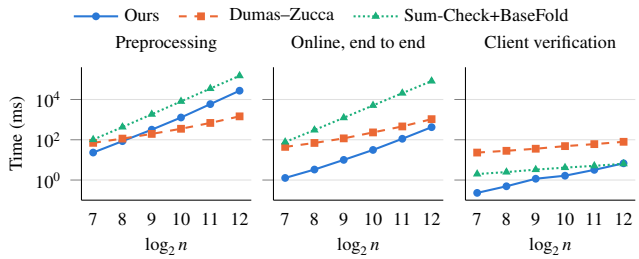


Figure 1: Comparison with verifiable matrix–vector multiplication baselines for $n \times n$ matrices (log scale).

eral proof-based approach used by recent verifiable-ML systems, including DeepProve [42]. Since both schemes provide verification only, we compare their runtime with our client verification time. To make a fair comparison, we benchmark over the scalar field of BLS12-381 and run the experiments on an AWS `c8i.4xlarge` instance.

Across the evaluated matrix dimensions, our end-to-end time is between 2.5× and 34.8× faster than Dumas–Zucca and between 60.7× and 194.8× faster than Sum-Check+BaseFold. At $n = 4{,}096$, for example, our protocol completes in 422 ms, compared with 1.056 s for Dumas–Zucca and 82.196 s for Sum-Check+BaseFold. Our client verification time is also comparable to theirs. The performance numbers are shown in Fig. 1.

7.2 Design and Implementation of Maverick

Maverick is a client-server LLM serving system where the client uses pvMVMD to outsource the MVMs in the linear operations while computing nonlinear operations locally. In this section, we present the implementation details and performance optimizations.

Architecture of an LLM (Qwen3-32B as an example). An LLM generates its response *autoregressively*, i.e., one token at a time, each conditioned on the prompt and on the tokens emitted so far. Each token is computed by one *forward pass* of the model. A forward pass alternates between linear operations, namely MVMs with the weight matrices of the model, and nonlinear operations such as normalization, attention, and activation functions. The MVMs involve large weight matrices and account for the bulk of the computation. We denote the MVMs of a forward pass by $\mathbf{y}_j = \mathbf{W}_j \mathbf{x}_j$ for $j \in [T]$, in the order in which they are evaluated. For Qwen3-32B [84], the main model of our evaluation, $T = 257$: each of the 64 transformer layers computes four products with $\mathbf{W}_{\text{qkv}}$, $\mathbf{W}_o$, $\mathbf{W}_{\text{gu}}$, and $\mathbf{W}_{\text{down}}$, plus a final product with $\mathbf{W}_{\text{lm}}$. The last MVM yields the logits, from which the next token is selected.

Inference begins by converting the natural language prompt into tokens, and then into embedding vectors. Next, in the *prefill* stage (prompt processing), a single forward pass processes embedding vectors of all prompt tokens and emits the first token of the response (each $\mathbf{W}_j$ multiplies a matrix with embedding vectors as columns). Then, in the *decode* stage (token generation), each further forward pass processes the last token, generating a new one. Through attention, all earlier tokens affect the probability of the new token. The decoding pass involves processing the *KV cache* of size proportional to the context (i.e., the number of prior tokens captured in attention), which becomes more expensive as the context grows.

7.2.1 Architecture of Maverick

Fig. 2 illustrates the architecture of Maverick on one forward pass of Qwen3-32B; the same flow applies to any model whose forward pass alternates MVMs with nonlinear operations. The client evaluates the nonlinear operations locally and delegates every MVM to the server using pvMVMD: it derives each input $\mathbf{x}_j$ from the outputs of the preceding MVMs, masks it, and unmasks the server's response to obtain $\mathbf{y}_j$. The preprocessing of pvMVMD is performed once for every weight matrix, independently of any prompt.

Processing a prompt gives rise to a large number of claims $\mathbf{W}_j \mathbf{x}_j = \mathbf{y}_j$. The client has two options. In *real-time verification*, it verifies the claims of each token as soon as the token is generated, so that the user is shown only verified tokens. In *optimistic verification*, it verifies the claims of all tokens in one batch at the end of the response: the user sees tokens before they are verified, but the batch verification yields a mild throughput gain, as we show in Section 7.3.1.

In a bit more detail, the prefill stage can always be verified in batch, as all prompt tokens are processed in a single forward pass. To do so, for each $j \in [T]$, the client maintains the running sums $\mathbf{X}_j = \sum_i \rho_{j,i} \widehat{\mathbf{x}}_{j,i}$ and $\mathbf{Y}_j = \sum_i \rho_{j,i} \widehat{\mathbf{y}}_{j,i}$ over the claims $\mathbf{W}_j \widehat{\mathbf{x}}_{j,i} = \widehat{\mathbf{y}}_{j,i}$ of all prompt tokens i, where $\rho_{j,i}$ is a secret random field element that the client draws afresh per matrix and token. Optimistic verification uses the same technique, with running sums over the claims of all tokens of the response. In both cases, the client verifies the single claim $\mathbf{W}_j \mathbf{X}_j = \mathbf{Y}_j$ for every j using the verification of Section 5.1. If any claim is false, the claim on the sums is also false except with probability $1/|\mathbb{F}|$. To boost soundness, one can keep, e.g., two pairs of running sums with independent coefficients with

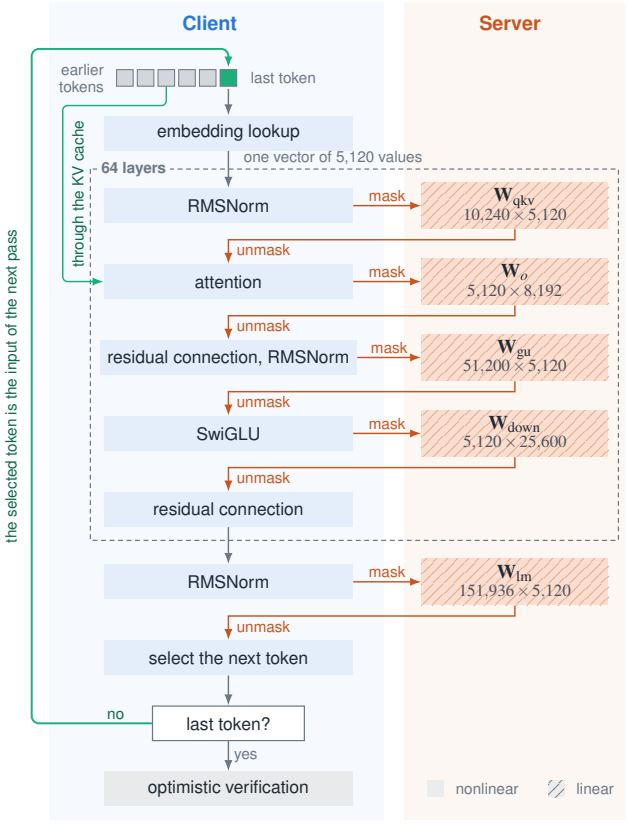


Figure 2: One decode pass of Maverick on Qwen3-32B in the privacy mode. For clarity, it omits the quantization and rescaling steps before masking and after unmasking.

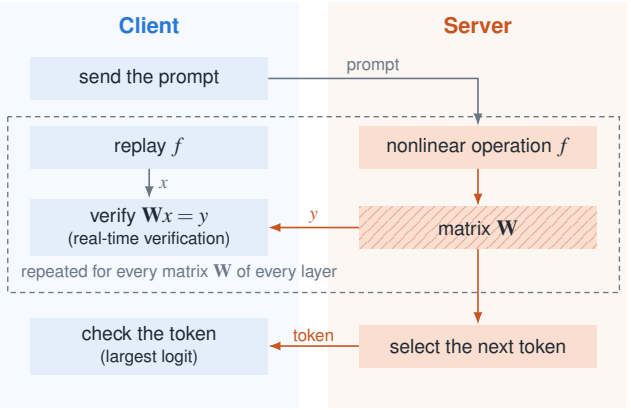


Figure 3: One forward pass of Maverick in the verification-only mode.

soundness of $|\mathbb{F}|^{-2} \approx 2^{-62}$.

Boost mode. Generating input masks and computing the corresponding output masks are a significant part of client overhead. Since these masks are independent of the client input, they can be generated beforehand (e.g., when the client's device is idle). This optimization is similar to how MPC protocols generate Beaver Triples ahead of time to accelerate the online phase. We call this the *boost mode* of Maverick. We refer to the original workflow where the client generates these masks during inference as the *standard mode*; we implement *pipelining* so the masks for step $j+1$ are generated while waiting for the server to complete step j.

Verification-only mode. Some users may not require input privacy, for example when their prompts contain no sensitive information or when they are willing to share them with the service provider, but may still want assurance that the provider performs the inference correctly. In this case, the masking is unnecessary and, importantly, so is the interaction for every MVM. The client sends the prompt to the server. The server evaluates the complete model, including the nonlinear operations, and streams the output $\mathbf{y}_j$ of every MVM, together with every generated token, to the client. The client replays the nonlinear operations on the received outputs, which yields the input $\mathbf{x}_j$ of every MVM, and verifies the claims $\mathbf{W}_j\mathbf{x}_j = \mathbf{y}_j$ as above, with the pairs $(\mathbf{x}_j, \mathbf{y}_j)$ in the running sums. Fig. 3 illustrates one decode pass with real-time verification. It also checks that every token output by the server is the one with the largest logit. This removes two costs of the privacy mode. First, the protocol is non-interactive after the prompt is sent: the server completes the inference without any input from the client, and the client replays and verifies concurrently, as the outputs arrive. The network's round-trip time therefore does not affect the throughput: a request takes a single round trip. Second, only the outputs $\mathbf{y}_j$ are transmitted, since the client recomputes the inputs $\mathbf{x}_j$ itself.

The replay must reproduce the server's nonlinear operations exactly, although the two parties run on different hardware (e.g., a laptop CPU and a datacenter GPU), where floating-point results generally differ. We therefore implement the nonlinear operations to be bitwise deterministic across platforms. The IEEE 754 standard [53] requires a small set of basic operations (addition, subtraction, multiplication, division, and square root) to return the same, correctly rounded result on every conforming platform. We build every nonlinear operation from these operations alone, and we fix the order in which they are evaluated. We avoid math libraries altogether, since their functions (e.g., exponentiation) carry no such guarantee.

7.2.2 Model quantization

The protocols of Sections 5 and 6 operate over a finite field, whereas the weights and activations of a model are floating-point numbers. We therefore need to quantize the weights and the inputs of every MVM into integers, which we embed in the BabyBear field $\mathbb{F}_q$, with $q = 2^{31} - 2^{27} + 1$. Quantizing a value means dividing it by a *step* and rounding the result to the nearest integer. The choice of the step matters. With a step that is too big, many different values round to the

same integer, and this loss of precision degrades the model's accuracy. With a step that is too small, the resulting integers are large in magnitude, and the sum computed by an MVM can exceed the modulus q and wrap around, in which case the result is wrong. We choose the steps such that a wrap-around cannot occur for any input, while the loss in accuracy remains negligible (see Section 7.3.3). We describe the quantization of a single MVM $\mathbf{y} = \mathbf{W}\mathbf{x}$.

Weights. Each row of a weight matrix is quantized with its own step. Since the magnitudes of the weights vary considerably across rows, a per-row step yields a finer quantization than a per-matrix step, at the moderate cost of storing one step per row. For row i of $\mathbf{W}$, we set the step to $s_i = \max_k |w_{ik}|/127$ and round the entries to the 8-bit integers $w_{ik}^{\text{int}} = \text{round}(w_{ik}/s_i)$. The weights are quantized once, in advance; we write $\mathbf{W}^{\text{int}}$ for the resulting integer matrix.

Input vectors. An input vector is quantized with a step that depends on the weight matrix applied to it. Our algorithm works as follows. For each matrix $\mathbf{W}$, we define $L_{\mathbf{W}} = \max_i \sum_k |w_{ik}^{\text{int}}|$ to be the largest row sum of the quantized weights, and fix a bound $A_{\mathbf{W}}$ such that $A_{\mathbf{W}} \cdot L_{\mathbf{W}} \leq (q-1)/2$. An input $\mathbf{x}$ of $\mathbf{W}$ is then quantized with the step $s_{\mathbf{x}} = \max_k |x_k|/A_{\mathbf{W}}$, which maps its entries to integers $x_k^{\text{int}} = \text{round}(x_k/s_{\mathbf{x}}) \in [-A_{\mathbf{W}}, A_{\mathbf{W}}]$; we write $\mathbf{x}^{\text{int}}$ for the resulting integer vector. Since $A_{\mathbf{W}} \cdot L_{\mathbf{W}} \leq (q-1)/2$, the product $\mathbf{W}^{\text{int}}\mathbf{x}^{\text{int}}$ cannot wrap around.

Rescaling the result. The server returns the integer vector $\mathbf{y}^{\text{int}} = \mathbf{W}^{\text{int}}\mathbf{x}^{\text{int}}$. The client recovers the floating-point result by multiplying every entry by the two steps, $y_i \approx s_i s_{\mathbf{x}} y_i^{\text{int}}$, and evaluates the next nonlinear operation on it in floating point. The recovered value is close to the product $\sum_k w_{ik}x_k$ of the original model but not equal to it: the integer product $\mathbf{y}^{\text{int}}$ is exact, but $s_i w_{ik}^{\text{int}}$ and $s_{\mathbf{x}} x_k^{\text{int}}$ only approximate w_{ik} and x_k, since both were rounded. Because the client evaluates the nonlinear operations, it returns to floating point after every MVM and quantizes the next input from scratch. Quantization thus affects one MVM at a time. In a system that evaluates the entire forward pass in the field, by contrast, the output of one MVM is the input of the next, so the bound on the integers is multiplied by $L_{\mathbf{W}}$ at every MVM, which tends to require a much larger field to avoid a wrap-around. In Maverick, this growth does not occur, because the input of every MVM is quantized from scratch; a 31-bit field is therefore large enough.

7.2.3 Implementation and optimizations

We implement Maverick in about 38,000 lines of Rust and 2,000 lines of CUDA, with support for Qwen3, Qwen2.5, and Llama 4 model families. Our implementation includes optimizations to reduce the number of round trips in the privacy mode, compresses the server's outputs in the verification-only mode, and to optimally leverage a small GPU on the client that is too small to hold the entire model.

Reducing round complexity. In the privacy mode, every MVM costs one round of interaction, and the rounds are sequential, since the input of an MVM depends on the outputs of the preceding ones. We reduce the number of rounds by exploiting that several weight matrices of a transformer layer are applied to the same input: the query, key, and value projections of the attention, and the gate and up projections of the feed-forward network. We stack the matrices of each group vertically into one matrix, $\mathbf{W}_{\text{qkv}}$ and $\mathbf{W}_{\text{gu}}$ respectively, and delegate it as a single MVM. A layer then takes 4 rounds instead of 7, and a forward pass of Qwen3-32B takes 257 rounds instead of 449.

Reducing communication. In the verification-only mode, the entries of the streamed vectors $\mathbf{y}^{\text{int}}$ are typically far smaller in magnitude than the modulus, so the server stores each block of 32 entries at the bit width of its largest entry; an entry then takes about 2.9 bytes on average instead of 4. (In the privacy mode, the server's outputs are masked, hence uniformly distributed, and cannot be compressed.)

GPU-equipped clients. Maverick targets computationally weak clients, including those with a GPU that is too small to hold the model and thus cannot run it efficiently on their own. Nonetheless, such a client can hold some of the weight matrices and evaluate their MVMs itself, which saves communication and verification. The question is how to split the work optimally between the client and the server under the client's memory budget. Our evaluation shows that GPU clients are bound by the network, so we use a simple heuristic that maximizes the reduction in bandwidth. Storing a matrix with m rows and n columns on the client's GPU costs $O(mn)$ memory and saves m values per token in bandwidth, so the bandwidth saved per byte of memory is proportional to $1/n$. The client therefore takes the matrices in increasing order of n, i.e., the narrowest matrices first (for Qwen3-32B, $\mathbf{W}_{\text{qkv}}$ and $\mathbf{W}_{\text{gu}}$, then $\mathbf{W}_o$, then $\mathbf{W}_{\text{down}}$), until its budget is exhausted. Looking ahead to Section 7.3.1, on a desktop with a 16 GB GPU, the client holds 71 of the 256 per-layer matrices of Qwen3-32B in 11.1 GB, which halves the communication from 13.8 to 7.1 MB per generated token.

Another way to use the client's GPU is to perform the prefill stage on the client. If the client can afford to keep a copy of the quantized weights on disk, it can evaluate the entire prefill pass efficiently, because prefill processes all prompt tokens in one batch, with matrix–matrix products instead of MVMs. The weight matrices are loaded from disk once and applied to all prompt tokens, so the cost of loading them is paid once per prompt. In the decode stage, by contrast, every token would require loading all the weights again, which is much slower than delegating.

7.3 Evaluation

We evaluate Maverick in three parts: the verification-only mode on weak clients, against local inference with llama.cpp

(Section 7.3.1); the privacy mode, against MOSAIC [98] (Section 7.3.2); and the accuracy cost of quantization (Section 7.3.3).

7.3.1 Verification-only mode

To demonstrate that a computationally weak client can verify the inference of large models by an untrusted server, we evaluate Maverick on two such clients: a CPU-only client and a client with a desktop GPU (called the *GPU client*). For both clients we evaluate two models: Qwen3-32B [75] and Llama 4 [79], both with 8-bit weights. For Qwen3-32B, which the clients can barely run locally, Maverick generates tokens 81–270× faster than llama.cpp. For Llama 4, which they cannot run at all, the CPU and GPU clients generate 7–15 and 24–33 tokens/s.

Setup. Our CPU client is a Lenovo ThinkPad X9-15 Gen 1 laptop with an Intel Core Ultra 7 268V processor (8 cores), 30 GB of memory and no discrete GPU. The GPU client is a desktop with an RTX 4080 (16 GB of GPU memory), limited to 16 GB of memory and 4 cores, so that its total memory, 32 GB, is about that of the CPU client.

The servers run in data centers. For Qwen3-32B, we use an A100 SXM4 with 80 GB of GPU memory, rented from a commercial cloud provider and reached over a WAN; we measure a round-trip time of 10 ms to the client and a bandwidth of 1 Gb/s. For Llama 4, we use four H200s in a university research cluster, reached over a WAN with a round-trip time of 8–9 ms and a bandwidth of 1 Gb/s. Unless otherwise specified, every test is run once to warm up and three times measured, and we report the mean.

Baselines. The baseline is llama.cpp [43] on the same client. llama.cpp is a widely used open-source engine for local inference, optimized with SIMD kernels and batched prompt processing. We run it unmodified (commit `669204e` on the CPU client and `7071ba3`, built with CUDA, on the GPU client) with its 8-bit format `Q8_0`. For llama.cpp, we run each generation test once, and we compute the time to first token from its prompt-processing throughput. We have no baseline for Llama 4, which is far too large to run on either client.

Benchmarks and metrics. To compare Maverick with baselines, we benchmark token generation with varying context lengths. We use the convention of `llama-bench` to denote benchmark runs: `tgM` produces M tokens one at a time after a one-token prompt; `ppN+tgM` processes an N-token prompt and then generates M tokens. We use `tg128` and `ppN+tg128` for N = 512, 1,024, 2,048 and 4,096. In all tests, decoding is greedy: the next token is the one with the largest logit; alternatively, the next token can be sampled pseudorandomly using a preshared seed wit no additional overhead.

For each test, we report the two quantities that a user experiences. The **time to first token** (TTFT) is the time from sending the prompt until the first token of the response is available and verified. The **generation throughput** (tokens/s) is the number of generated tokens over the time to generate them, after the first token. All times are online times and exclude one-time setup.

Results for Qwen3-32B. Fig. 4 presents the evaluation results for Qwen3-32B. Run locally, the model is not usable on either client: because the model does not fit in memory, each forward pass reads part of the model from disk, which drastically limits the performance. For instance, a 128-token response takes close to an hour on the CPU client and about ten minutes on the GPU client. With Maverick, the same clients generate 8 and 18 tokens/s, a speedup of more than 200× and of 81–110×, respectively.

The evaluation shows that Maverick is network-bound for short context lengths. For the CPU client, the server sends the output of all matrix–vector products, about 13.8 MB per generated token, which takes 0.12 s on the client's 1 Gb/s link. The GPU client evaluates 71 of the 256 per-layer matrices itself (Section 7.2); this halves the traffic and, accordingly, doubles the throughput. Being network-bound also explains why the throughput is almost independent of the prompt length: the client's computation becomes the bottleneck only on the CPU client, and only after prompts of 2,048 tokens or more.

The TTFT behaves differently on the two clients. On the CPU client, the TTFT is similarly network-bound: the outputs for all prompt tokens must be transmitted to the client. Even so, Maverick still produces the first token about 2× sooner than llama.cpp. The GPU client can avoid this transfer by evaluating the prefill pass itself (Section 7.2), much faster than llama.cpp. The reason is that llama.cpp places as many layers as fit on the GPU, in order, and evaluates the remaining layers on the CPU, whereas Maverick's client needs only one layer on the GPU at a time for the prompt.

Optimistic verification. The client can also defer the verification to the end of the response (Section 7.2). We repeat the Qwen3-32B experiments in this setting. The difference is small, because a verification is cheap: it takes 8.5 ms on the CPU client and 4.8 ms on the GPU client. The TTFT is therefore practically unchanged, although the first token is then displayed before it is verified. The throughput is also unchanged when generation is network-bound, since the client's verification time is hidden by the transmission time. It improves only when the CPU client's computation is the bottleneck, after prompts of 2,048 and 4,096 tokens, and then by 6–7%. The rest of the evaluation therefore reports real-time verification unless stated otherwise.

Results for Llama 4. Llama 4 cannot run on either client at all, since its weights take about 400 GB, so there is no local baseline. With Maverick, both clients can nevertheless use it (Fig. 5): across the prompt lengths, the CPU client generates 7–15 tokens/s and the GPU client 24–33 tokens/s, which is fast enough for interactive use. The GPU client is again at least twice as fast, because it evaluates part of the model itself. Interestingly, both clients are faster than with Qwen3-32B, a model 12× smaller, because Llama 4 is a mixture

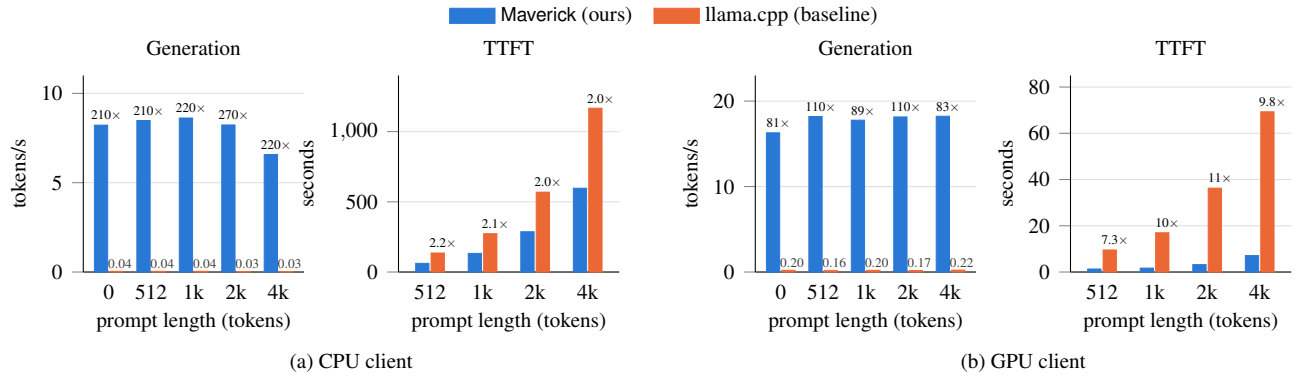


Figure 4: Qwen3-32B in the verification-only mode: generation throughput and TTFT of Maverick and llama.cpp on both clients, for `tg128` (prompt length 0) and `ppN+tg128`. The numbers above the bars are the speedups of Maverick.

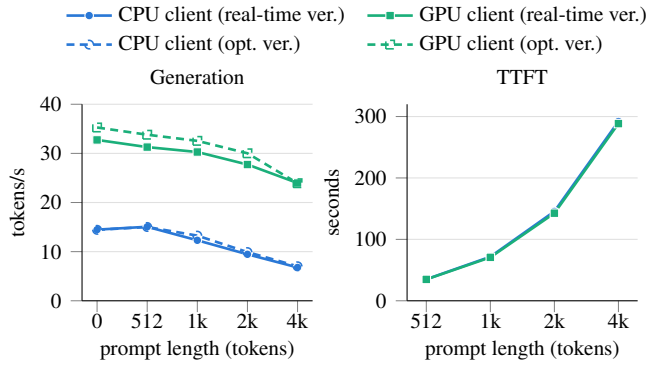


Figure 5: Llama 4 in the verification-only mode: generation throughput and TTFT of Maverick on both clients, for `tg128` (prompt length 0) and `ppN+tg128`. There is no local baseline. Real-time ver. = real-time verification; opt. ver. = optimistic verification (Section 7.2).

of experts [82], in which each token uses only a fraction of the parameters, so the server sends less data per token to the client. The throughput decreases with the prompt length as the client's computation becomes the bottleneck. Unlike with Qwen3-32B, it is infeasible for the GPU client to evaluate the prefill pass itself, so the TTFT is the same on both clients. With optimistic verification, the GPU client generates up to 8% faster when client is computation-bound.

For comparison, FlashProve [6], the state-of-the-art zero-knowledge proof system for LLM inference, evaluated on the same model, reports 58.8 s on an H100 to prove one next token after a 1,024-token prompt, and about 21 s to verify it. With Maverick, the server does no work beyond the inference, and the throughput above includes the client's verification.

Cost of refreshing the challenges. Recall that the client verifies a claim $\mathbf{y} = \mathbf{M}\mathbf{x}$ with a secret sparse challenge $\mathbf{e}$, by checking $(\mathbf{e}^\top\mathbf{Q})\mathbf{x} = \mathbf{e}^\top\mathbf{G}^\top\mathbf{y}$, where $\mathbf{Q} = \mathbf{G}^\top\mathbf{M}$ is computed in the preprocessing (Section 5.1). In the experiments above, the client computes the projections $\mathbf{e}^\top\mathbf{Q}$ ahead of time and reuses them for all requests. The client may need to refresh its challenges when it detects server cheating, or, as a layer of defense in depth, may want to refresh them proactively after every request. We measure the overhead of proactive refreshing.

The baseline of comparison is the standard way to obtain a fresh challenge: sample a uniform vector $\mathbf{c}$ and compute its projection $\mathbf{c}^\top\mathbf{M}$ for all $\mathbf{M}$. This is used in, e.g., Slalom [87].

In our experiment, we repeat the Qwen3-32B experiment on the GPU client with refreshing after each request. With both methods, the client computes new challenges in the background, in parallel to the current request; in optimistic verification, this is to compute the challenge for the current request; in real-time verification, this is to compute the challenge for the *next* request. If the refresh takes longer than the current request, the next request must wait for it. The two are identical in performance, so we present optimistic verification. We measure the TTFT and the throughput that the client can sustain over consecutive requests. The refresh takes a fixed time per request, so it matters most for short requests. We therefore add four short tests (`pp16+tg32`, `pp32+tg64`, `pp128+tg64` and `pp1024+tg64`) to `pp512+tg128`.

Fig. 6 reports the results. Maverick completes a refresh in about 1.5 s, less than the shortest request takes. Refreshing is therefore free. The standard method needs more than 7 s per refresh, dominated by the time to read the 32 GB of weights from the SSD. Requests that are shorter than this must wait. This is because the standard method involves reading the entire model and, for large models that cannot fit in memory, reading it from disk. In Maverick, computing $\mathbf{e}^\top\mathbf{Q}$ only involves reading a few rows of $\mathbf{Q}$. $\mathbf{Q}$ is larger, so this is a storage–performance tradeoff. In our experiment, using a Reed–Solomon code of rate 1/2 (with 40 bits of soundness), $\mathbf{Q}$ takes 8 bytes per weight (256 GB for Qwen3-32B), and a refresh reads 747 MB of it (the model weights required by the standard method are 32 GB).

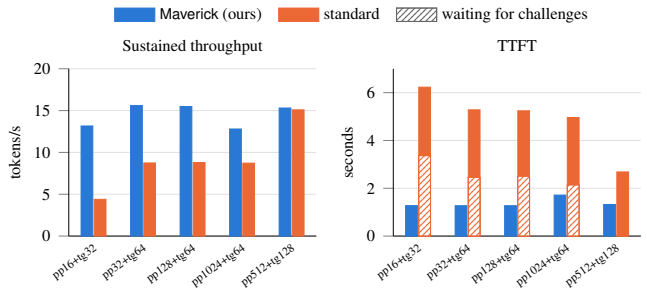


Figure 6: Qwen3-32B on the GPU client with fresh challenges for every request: sustained throughput over consecutive requests, and TTFT. The hatched part of the TTFT is the time that a request waits for its challenges; the rest is the prompt pass, which the standard method slows down by refreshing the next challenges at the same time.

On the four short tests, the sustained throughput of Maverick is 1.5–3× higher. For longer requests, such as `pp512+tg128`, the two methods reach the same throughput. The standard method also worsens the TTFT, for two reasons. A short request first waits for its challenges (the hatched part in Fig. 6). Its prompt pass then takes about twice as long, because the refresh for the next request runs at the same time and competes with it for the SSD. Together, the TTFT of a short request is 3–5× longer than with Maverick.

Storage and memory usage. When challenges are reused, the client's storage is small, because it can compute the challenge projections $\mathbf{e}^\top\mathbf{Q}$ (one per weight matrix) ahead of time and then delete the model weights. For Qwen3-32B, the CPU client stores 1.65 GB: 84 MB of challenge projections and quantization steps, and 1.57 GB for the embedding table. For Llama 4, whose weights take about 400 GB, the client stores only 3.3 GB: 1.2 GB of challenge projections and quantization steps, and 2.1 GB for the embedding table. The memory usage is moderate, peaking at 6.5 GB and 7.0 GB in the Qwen3-32B and Llama 4 experiments, respectively.

To leverage its GPU, the GPU client stores the part of the model that it evaluates itself. In our experiment, for Qwen3-32B, it keeps a copy of all 8-bit weights (31 GB) for the prefill pass; for Llama 4, it stores 28 GB for the non-expert weights and 11 GB of setup data. (Experts are run on the server.) To support refreshing the challenges, a client needs to store $\mathbf{Q}$ on disk as discussed above.

7.3.2 Privacy mode

The privacy mode involves many round trips between the client and the server, so it is only feasible when the two are connected by a low-latency link. It is most similar to the recent system MOSAIC [98] (CCS'26), in which the client and the servers reside in the same data center (connected by a low-latency link such as NVLink) but the client is trusted (e.g., it runs in a TEE) while the servers are not. In this setting, trusted compute is the scarce resource, and the goal of outsourcing is for a trusted GPU to increase its throughput with the help of *untrusted* accelerators [98]. The relevant cost is therefore the time that the trusted side spends per token, in addition to the end-to-end throughput.

	Maverick	MOSAIC
GPUs (trusted + untrusted)	1 + 1	1 + 3
Prompt processing at 1k / 4k / 7k (t/s)	1,010 / 895 / 645	211 / 261 / 247
Generation at 1k / 4k / 7k context (t/s)	21.6 / 18.9 / 18.4	1.84 / 1.83 / 1.83
Trusted side per generated token (ms)	19.1 / 25.7 / 27.0	42.8 / 43.1 / 42.8
Verifiability	yes	no
Model privacy	no	yes
Outputs exact	yes	no

Table 5: Privacy mode: Maverick and MOSAIC, at about 1k, 4k and 7k tokens of context.

MOSAIC is the natural point of comparison: it targets the same deployment and, like Maverick, outsources the linear operations of the model to untrusted GPUs. The two systems make different trade-offs in what they guarantee: Maverick verifies the server's results and keeps the outputs exact, whereas MOSAIC also hides the model from the server, but it does not verify the results and adds noise to the outputs. The comparison therefore shows how efficiently each design delivers its guarantees in the same setting.

Setting. We run both systems on the hardware MOSAIC targets, H200 GPUs of one machine connected by NVLink, and on Qwen2.5-72B, a model of the same size (72B) as the one its paper evaluates. We run MOSAIC's code [97] with its own benchmark, in its faster mode, with one trusted and three untrusted H200s (three GPUs are required because its masked weights take about 290 GB). Maverick runs in the privacy mode with optimistic verification, with the client on one H200 and the server on a second. It uses the boost mode, in which the client prepares its one-time masks ahead of time, while MOSAIC generates its masks within the request. Each system uses its own benchmark's prompts: 1,024, 4,096 and 7,168 tokens for Maverick, and for MOSAIC a system prompt of 1,456, 4,519 or 6,703 tokens followed by a question of 45–67 tokens, processed in two passes. We refer to these as 1k, 4k and 7k of context. We report online time.

Results. Table 5 reports the results. Maverick processes prompts 2.6–4.8× faster than MOSAIC and generates 10–12× faster, while using one untrusted GPU instead of three. The gap in generation comes from the untrusted side: MOSAIC's untrusted GPUs take 479 ms per generated token, whereas Maverick's server takes 22 ms. The trusted side, the scarce resource in this setting, also does less work with Maverick: 19–27 ms per generated token, against 43 ms with MOSAIC. Recall, however, that the two systems offer different guarantees: Maverick provides verifiability and does not introduce noise into its outputs, whereas MOSAIC hides the

Model	Reference	Maverick	Ratio
Qwen3-32B	7.9435	7.9464	1.00036
Qwen2.5-72B	3.6744	3.6748	1.00009
Llama 4	5.1508	5.1414	0.99817

Table 6: Perplexity on the WikiText-2 test set with the released weights (reference) and with Maverick's quantization.

model. Moreover, Maverick runs in the boost mode: its times are online times and exclude the preparation of its masks, whereas MOSAIC's times include its mask generation.

7.3.3 Accuracy

Maverick evaluates a model on quantized weights and inputs (Section 7.2), so we measure how much accuracy the quantization costs. By construction, no result can wrap around the modulus, so rounding is the only source of error.

Methodology. We run each model twice on the WikiText-2 test set [67], the standard corpus for perplexity comparisons of quantized models. The reference run uses released weights without any further quantization: the official FP8 checkpoint of Qwen3-32B, which stores 8-bit floating-point weights, the official BF16 checkpoint of Qwen2.5-72B, and, for Llama 4, the FP8 checkpoint published by Red Hat AI. Throughout, Qwen2.5-72B refers to the instruction-tuned release, Qwen2.5-72B-Instruct. The second run applies Maverick's quantization to the same weights. We compare the two runs by their perplexity ratio, which measures whether the quantization makes the model worse at predicting the text.

Computing the perplexity. Perplexity is computed from how well the model predicts each token of the text given the tokens before it, its context. The result depends on how much context each token receives, so for each model we follow the convention of the baseline it is compared with. For Qwen3-32B and Llama 4 we follow llama.cpp. The text is cut into separate chunks of 512 tokens, and each chunk is given to the model as a separate input, so that the model sees nothing of the other chunks. The perplexity is computed from the predictions for the second half of each chunk with the first half as context. For Qwen2.5-72B we follow MOSAIC. A window of 2,048 tokens slides over the text and advances by 512 tokens at a time. The perplexity is computed from the predictions for the last 512 tokens of each window, with the preceding 1,536 tokens as context.

Results. Table 6 reports the results. For the two Qwen models, the perplexity with Maverick's quantization is within 0.04% of the reference. For Llama 4, it is within 0.2% of the reference. Maverick's quantization therefore costs practically no accuracy.

8 Conclusions and Future Directions

We develop Maverick, a protocol for private and verifiable LLM inference that delegates the model's linear operations using a new matrix–vector multiplication primitive with transparent preprocessing and efficient client verification. When only verification is required, our implementation serves models of up to 400B parameters to clients that cannot run them on their own: a laptop and a desktop with a 16 GB GPU generate tokens 81–270× faster on `Qwen3-32B` than with local inference, and they use `Llama 4`, a 400B-parameter model, at 7–15 and 24–33 tokens per second. With input privacy, Maverick generates tokens 10–12× faster than the concurrent system MOSAIC in a data-center setting, with precomputed masks. An important direction for future work is reducing the communication rounds for private inference.

Acknowledgements

Wenhao Wang is supported in part by a research grant from IC3.

References

[1] Kasra Abbaszadeh, Hossein Hafezi, Jonathan Katz, and Sarah Meiklejohn. Single-server private outsourcing of zk-snarks. *Cryptology ePrint Archive*, 2025.

[2] Carlos Aguilar-Melchor, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, and Gilles Zémor. Efficient encryption from random quasi-cyclic codes. *IEEE Transactions on Information Theory*, 64:3927–3943, 2016.

[3] Pariya Akhiani and Yupeng Zhang. Distance of RAA codes over large finite fields (with applications in zk-SNARKs and PCGs). Cryptology ePrint Archive, Paper 2026/524, 2026.

[4] Michael Alekhnovich. More on average case vs approximation complexity. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '03, page 298, USA, 2003. IEEE Computer Society.

[5] Sarah Allen, Pranay Anchuri, James Austgen, Maryam Bahrani, Samuel Breckenridge, Aaron Buchwald, Christian Cachin, Andrés Fábrega, Jared Fernandez, James Hsin yu Chiang, Marwa Mouallem, Roi Bar-Zur, Neil DeSilva, Ittay Eyal, Giulia Fanti, Ari Juels, Andrew Miller, Christian Sillaber, Dani Vilardell, Pramod Viswanath, Wenhao Wang, Matt Weinberg, Sen Yang, Jianzhu Yao, and Fan Zhang. Crypto x AI, AI x crypto: A survey. Technical report, Initiative for CryptoCurrencies and Contracts (IC3), June 2026. Version 1.0.

[6] Anthropic. FlashProve: Provable inference. https://www.anthropic.com/research/provable-inference, 2026.

[7] Anthropic and Pattern Labs. Confidential inference systems: Design principles and security risks. White paper, Anthropic and Pattern Labs, June 2025. Version 1.0.

[8] Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Santosh Ghosh, Shay Gueron, Tim Güneysu, et al. Bike: bit flipping key encapsulation, 2022.

[9] Diego F. Aranha, Anamaria Costache, Antonio Guimarães, and Eduardo Soria-Vazquez. Heliopolis: Verifiable computation over homomorphically encrypted data from interactive oracle proofs is practical. In Kai-Min Chung and Yu Sasaki, editors, *Advances in Cryptology – ASIACRYPT 2024*, pages 302–334, Singapore, 2025. Springer Nature Singapore.

[10] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In Luca Aceto, Monika Henzinger, and Jiří Sgall, editors, *Automata, Languages and Programming*, pages 403–415, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[11] David Balbás, Dario Fiore, Maria Isabel González Vasco, Damien Robissout, and Claudio Soriente. Modular sumcheck proofs with applications to machine learning and image processing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1437–1451, New York, NY, USA, 2023. Association for Computing Machinery.

[12] Louay Bazzi, Mohammad Mahdian, and Daniel A. Spielman. The minimum distance of turbo-like codes. *IEEE Transactions on Information Theory*, 55(1):6–15, 2009.

[13] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 420–432, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[14] Fabrice Benhamouda, Caicai Chen, Shai Halevi, Yuval Ishai, Hugo Krawczyk, Tamer Mour, Tal Rabin, and Alon Rosen. Encrypted matrix-vector products from secret dual codes. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, CCS '25, page 394–408, New York, NY, USA, 2025. Association for Computing Machinery.

[15] Huck Bennett, Karthik Gajulapalli, Alexander Golovnev, and Evelyn Warton. Matrix Multiplication Verification Using Coding Theory. In Amit Kumar and Noga Ron-Zewi, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2024)*, volume 317 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 42:1–42:13, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[16] Wyatt Benno, Alberto Centelles, Antoine Douchet, and Khalil Gibran. Jolt atlas: Verifiable inference via lookup arguments in zero knowledge, 2026.

[17] Avrim Blum, Merrick Furst, Michael Kearns, and Richard J. Lipton. Cryptographic primitives based on hard learning problems. In Douglas R. Stinson, editor, *Advances in Cryptology — CRYPTO' 93*, pages 278–291, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[18] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *J. ACM*, 50(4):506–519, July 2003.

[19] J. L. Bordewijk. Inter-reciprocity applied to electrical networks. *Applied Scientific Research B: Electrophysics, Acoustics, Optics, Mathematical Methods*, 6:1–74, 1956. URL: http://cr.yp.to/bib/entries.html#1956/bordewijk.

[20] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector ole. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 896–912, New York, NY, USA, 2018. Association for Computing Machinery.

[21] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Nicolas Resch, and Peter Scholl. Correlated pseudorandomness from expand-accumulate codes. In *Advances in Cryptology – CRYPTO 2022: 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15–18, 2022, Proceedings, Part II*, page 603–633, Berlin, Heidelberg, 2022. Springer-Verlag.

[22] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent ot extension and more. In *Advances in Cryptology – CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III*, page 489–518, Berlin, Heidelberg, 2019. Springer-Verlag.

[23] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from ring-lpn. In *Advances in Cryptology – CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part II*, page 387–416, Berlin, Heidelberg, 2020. Springer-Verlag.

[24] Mark Braverman and Stephen Newman. Practical secure delegated linear algebra with trapdoored matrices. In *Theory of Cryptography Conference*, pages 97–118. Springer, 2025.

[25] Martijn Brehm, Binyi Chen, Ben Fisch, Nicolas Resch, Ron D. Rothblum, and Hadas Zeilberger. Blaze: Fast snarks from interleaved raa codes. In *Advances in Cryptology – EUROCRYPT 2025: 44th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Madrid, Spain, May 4–8, 2025, Proceedings, Part IV*, page 123–152, Berlin, Heidelberg, 2025. Springer-Verlag.

[26] Ida Bruhns, Sebastian Berndt, Jonas Sander, and Thomas Eisenbarth. Slalom at the carnival: Privacy-preserving inference with masks from public knowledge. *IACR Communications in Cryptology*, 1(3), 2024.

[27] Ignacio Cascudo, Anamaria Costache, Daniele Cozzo, Dario Fiore, Antonio Guimarães, and Eduardo Soria-Vazquez. Verifiable computation for approximate homomorphic encryption schemes. In *Annual International Cryptology Conference*, pages 643–677. Springer, 2025.

[28] Center for Research on Foundation Models. The foundation model transparency index.

[29] Sylvain Chatel, Christian Knabenhans, Apostolos Pyrgelis, Carmela Troncoso, and Jean-Pierre Hubaux. Veritas: Plaintext encoders for practical verifiable homomorphic encryption. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 2520–2534, New York, NY, USA, 2024. Association for Computing Machinery.

[30] Lingjiao Chen, Matei Zaharia, and James Zou. How is chatgpt's behavior changing over time? *arXiv preprint arXiv:2307.09009*, 2023.

[31] Jung Hee Cheon and Daehyun Jang. Cryptanalysis on lightweight verifiable homomorphic encryption. In Goichiro Hanaoka and Bo-Yin Yang, editors, *Advances in Cryptology – ASIACRYPT 2025*, pages 366–397, Singapore, 2026. Springer Nature Singapore.

[32] Jalen Chuang, Alex Seto, Nicolas Berrios, Stephan Van Schaik, Christina Garman, and Daniel Genkin. TEE.Fail: Breaking Trusted Execution Environments via DDR5 Memory Bus Interposition. In *2026 IEEE Symposium on Security and Privacy (SP)*, pages 4408–4426. IEEE Computer Society, May 2026.

[33] DeepSeek-AI. DeepSeek-V3 technical report, 2024.

[34] Design Arena. Kimi K3's design secret may be in its thinking traces. The Intelligence Company, jul 2026.

[35] Jean-Guillaume Dumas and Vincent Zucca. Prover efficient public verification of dense or sparse/structured matrix-vector multiplication. In Josef Pieprzyk and Suriadi Suriadi, editors, *Information Security and Privacy*, pages 115–134, Cham, 2017. Springer International Publishing.

[36] duola15. Deploy guide & hardware requirements for Kimi-K3. Hugging Face discussion, 2026.

[37] Boyuan Feng, Lianke Qin, Zhenfei Zhang, Yufei Ding, and Shumo Chu. ZEN: An optimizing compiler for verifiable, zero-knowledge neural network inferences. Cryptology ePrint Archive, Paper 2021/087, 2021.

[38] Dario Fiore and Rosario Gennaro. Publicly verifiable delegation of large polynomials and matrix computations, with applications. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 501–512, New York, NY, USA, 2012. Association for Computing Machinery.

[39] Dario Fiore, Rosario Gennaro, and Valerio Pastro. Efficiently verifiable computation on encrypted data. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 844–855, New York, NY, USA, 2014. Association for Computing Machinery.

[40] Dario Fiore, Anca Nitulescu, and David Pointcheval. Boosting verifiable computation on encrypted data. In *Public-Key Cryptography – PKC 2020: 23rd IACR International Conference on Practice and Theory of Public-Key Cryptography, Edinburgh, UK, May 4–7, 2020, Proceedings, Part II*, page 124–154, Berlin, Heidelberg, 2020. Springer-Verlag.

[41] Rusins Freivalds. Probabilistic machines can use less running time. In *IFIP Congress*, 1977.

[42] Nicolas Gailly, Ismael Hishon-Rezaizadeh, Tianyi Liu, Nicholas Mainardi, Dimitrios Papadopoulos, Charalampos Papamanthou, Christodoulos Pappas, Shravan Srinivasan, Zack Youell, and Yupeng Zhang. DeepProve: Verifiable end-to-end large language model inference. Cryptology ePrint Archive, Paper 2026/1112, 2026.

[43] Georgi Gerganov and the llama.cpp contributors. llama.cpp: LLM inference in C/C++. https://github.com/ggml-org/llama.cpp, 2023.

[44] Zahra Ghodsi, Tianyu Gu, and Siddharth Garg. SafetyNets: Verifiable execution of deep neural networks on an untrusted cloud. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[45] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 201–210, New York, New York, USA, 20–22 Jun 2016. PMLR.

[46] 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, Paper 2023/1269, 2023.

[47] Kanav Gupta, Jonathan Katz, and Ian Miers. Private and verifiable outsourcing of open-weight LLM inference. Cryptology ePrint Archive, Paper 2026/1849, 2026.

[48] Venkatesan Guruswami, Atri Rudra, and Madhu Sudan. Essential coding theory, 2025.

[49] Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, Guowen Xu, and Tianwei Zhang. Iron: Private inference on transformers. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pages 15718–15731, 2022.

[50] Jiaxing He, Kang Yang, Guofeng Tang, Zhangjie Huang, Li Lin, Changzheng Wei, Ying Yan, and Wei Wang. Rhombus: Fast homomorphic matrix-vector multiplication for secure two-party inference. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 2490–2504, New York, NY, USA, 2024. Association for Computing Machinery.

[51] Zhicong Huang, Wen jie Lu, Cheng Hong, and Jiansheng Ding. Cheetah: Lean and fast secure Two-Party deep neural network inference. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 809–826, Boston, MA, August 2022. USENIX Association.

[52] Alexandre Graell i Amat and Eirik Rosnes. On the minimum distance properties of weighted nonbinary repeat multiple-accumulate codes. In *2011 Information Theory and Applications Workshop*, pages 1–5, 2011.

[53] IEEE. IEEE standard for floating-point arithmetic, 2019. IEEE Std 754-2019 (Revision of IEEE 754-2008).

[54] 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. In *32nd Annual Network and Distributed System Security Symposium (NDSS)*, 2025.

[55] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security)*, pages 1651–1669, 2018.

[56] Andes Y. L. Kei and Sherman S. M. Chow. Shaft: Secure, handy, accurate and fast transformer inference. *IACR Cryptol. ePrint Arch.*, 2025:2324, 2025.

[57] Majid Khabbazian. Sharp minimum-distance lower tails for RAA codes. Cryptology ePrint Archive, Paper 2026/1075, 2026.

[58] Yongsang Kim, Kyungwhoon Cheun, and Hyuntack Lim. Performance of weighted nonbinary repeat-accumulate codes over gf(q) with q-ary orthogonal modulation. *IEEE Transactions on Communications*, 59(5):1208–1212, 2011.

[59] Kimi Team. Kimi K2: Open agentic intelligence, 2025.

[60] Jörg Kliewer, Kamil Sh. Zigangirov, Christian Koller, and Daniel J. Costello Jr. Coding theorems for repeat multiple accumulate codes. *CoRR*, abs/0810.3422, 2008.

[61] Kyeongtae Lee, Byeongkyu Han, Jihye Kim, and Hyunok Oh. LAMP: Linear verification of matrix multiplication via proximity testing. Cryptology ePrint Archive, Paper 2026/1571, 2026.

[62] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 619–631, New York, NY, USA, 2017. Association for Computing Machinery.

[63] Jing Liu and Liang Feng Zhang. Privacy-preserving and publicly verifiable matrix multiplication. *IEEE Transactions on Services Computing*, 16(3):2059–2071, 2023.

[64] Tianyi Liu, Xiang Xie, and Yupeng Zhang. zkCNN: Zero knowledge proofs for convolutional neural network predictions and accuracy. In *Proceedings of the*

2021 ACM SIGSAC Conference on Computer and Communications Security (CCS), pages 2968–2985, 2021.

[65] Hidde Lycklama, Alexander Viand, Nikolay Avramov, Nicolas Küchler, and Anwar Hithnawi. Artemis: Efficient commit-and-prove SNARKs for zkML. In *34th USENIX Security Symposium*, 2025.

[66] James Massey. Shift-register synthesis and bch decoding. *IEEE transactions on Information Theory*, 15(1):122–127, 1969.

[67] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations (ICLR)*, 2017.

[68] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2505–2522. USENIX Association, August 2020.

[69] Payman Mohassel and Yupeng Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning . In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38, Los Alamitos, CA, USA, May 2017. IEEE Computer Society.

[70] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. BOLT: Privacy-preserving, accurate and efficient inference for transformers. In *2024 IEEE Symposium on Security and Privacy (S&P)*, pages 4753–4771, 2024.

[71] Kabir Peshawaria, Zeyu Liu, Ben Fisch, and Eran Tromer. Laminate: Succinct SIMD-friendly verifiable FHE. Cryptology ePrint Archive, Paper 2025/2285, 2025.

[72] E. Prange. The use of information sets in decoding cyclic codes. *IRE Transactions on Information Theory*, 8(5):5–9, 1962.

[73] Wenjie Qu, Yanpei Guo, Yue Ying, and Jiaheng Zhang. VerfCNN, Optimal Complexity zkSNARK for Convolutional Neural Networks . In *2026 IEEE Symposium on Security and Privacy (SP)*, pages 515–532, Los Alamitos, CA, USA, May 2026. IEEE Computer Society.

[74] Wenjie Qu, Yijun Sun, Xuanming Liu, Tao Lu, Yanpei Guo, Kai Chen, and Jiaheng Zhang. zkGPT: An efficient non-interactive zero-knowledge proof framework for LLM inference. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 2045–2063, 2025.

[75] Qwen Team. Qwen3-32B-FP8. Model files, https://huggingface.co/Qwen/Qwen3-32B-FP8, 2025.

[76] Srinivasan Raghuraman, Peter Rindal, and Titouan Tanguy. Expand-convolute codes for pseudorandom correlation generators from lpn. In *Advances in Cryptology – CRYPTO 2023: 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20–24, 2023, Proceedings, Part IV*, page 602–632, Berlin, Heidelberg, 2023. Springer-Verlag.

[77] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow2: Practical 2-party secure inference. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 325–342, New York, NY, USA, 2020. Association for Computing Machinery.

[78] Chiara Ravazzi and Fabio Fagnani. Spectra and minimum distances of repeat multiple-accumulate codes. *IEEE Trans. Inf. Theor.*, 55(11):4905–4924, November 2009.

[79] Red Hat AI. Llama-4-Maverick-17B-128E-Instruct-FP8. Model files, https://huggingface.co/RedHatAI/Llama-4-Maverick-17B-128E-Instruct-FP8, 2025.

[80] RedPill AI. Verify real end-to-end private AI. RedPill AI, 2026. Accessed August 3, 2026.

[81] Muhammad Husni Santriaji, Jiaqi Xue, Yancheng Zhang, Qian Lou, and Yan Solihin. DataSeal: Ensuring the Verifiability of Private Computation on Encrypted Data . In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 2378–2394, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.

[82] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations (ICLR)*, 2017.

[83] Haochen Sun, Jason Li, and Hongyang Zhang. zkLLM: Zero knowledge proofs for large language models. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 4405–4419, 2024.

[84] Qwen Team. Qwen3 technical report, 2025.

[85] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. *CoRR*, abs/1304.3812, 2013.

[86] Louis Tremblay Thibault, Michael Walter, and Jiapeng Zhang. Practical SNARGs for matrix multiplications over encrypted data. Cryptology ePrint Archive, Paper 2026/027, 2026.

[87] Florian Tramer and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. In *International Conference on Learning Representations*, 2019.

[88] Vinod Vaikuntanathan and Or Zamir. Improving algorithmic efficiency using cryptography: Trapdoored matrices and applications. In *Proceedings of the 2026 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2554–2574. SIAM, 2026.

[89] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[90] Alexander Viand, Christian Knabenhans, and Anwar Hithnawi. Verifiable fully homomorphic encryption. arXiv:2301.07041, 2023.

[91] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *2021 IEEE Symposium on Security and Privacy (S&P)*, pages 1074–1091, 2021.

[92] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. In *30th USENIX Security Symposium (USENIX Security)*, pages 501–518, 2021.

[93] Chenkai Weng, Kang Yang, Zhaomin Yang, Xiang Xie, and Xiao Wang. AntMan: Interactive zero-knowledge proofs with sublinear communication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2901–2914, 2022.

[94] Tianshi Xu, Wen-jie Lu, Jiangrui Yu, Yi Chen, Chenqi Lin, Runsheng Wang, and Meng Li. Breaking the layer barrier: remodeling private transformer inference with hybrid ckks and mpc. In *Proceedings of the 34th USENIX Conference on Security Symposium*, SEC '25, USA, 2025. USENIX Association.

[95] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2986–3001, 2021.

[96] Kyeongcheol Yang. Weighted nonbinary repeat-accumulate codes. *IEEE Transactions on Information Theory*, 50(3):527–531, 2004.

[97] James Hsin yu Chiang, Sheila Zingg, Kari Kostiainen, and Srdjan Capkun. MOSAIC: implementation. https://github.com/jachiang/mosaic, 2026. Commit f1207c7.

[98] James Hsin yu Chiang, Sheila Zingg, Kari Kostiainen, and Srdjan Capkun. Mosaic: Masked outsourcing of secure ai computations, 2026.

[99] Hadas Zeilberger, Binyi Chen, and Ben Fisch. Basefold: Efficient field-agnostic polynomial commitment schemes from foldable codes. In *Advances in Cryptology – CRYPTO 2024: 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2024, Proceedings, Part X*, page 138–169, Berlin, Heidelberg, 2024. Springer-Verlag.

[100] 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. Cryptology ePrint Archive, Paper 2024/136, 2024.

[101] Liang Zhang and Reihaneh Safavi-Naini. Protecting data privacy in publicly verifiable delegation of matrix and polynomial functions. *Designs, Codes and Cryptography*, 88, 04 2020.

[102] Xinxuan Zhang, Ruida Wang, Zeyu Liu, Binwu Xiang, Yi Deng, Ben Fisch, and Xianhui Lu. Phalanx: An fhe-friendly snark for verifiable computation on encrypted data. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, CCS '25, page 4035–4048, New York, NY, USA, 2025. Association for Computing Machinery.

A Learning Parity with Noise

We begin by defining the dual-LPN assumption over a field $\mathbb{F}$ with respect to a distribution of code generators $\mathcal{G}$, dimension n, block length $N > n$, and noise distribution $\mathcal{E}$.

Definition A.1 (Dual-LPN)**.** *The $(\mathcal{E}, \mathcal{G}, \mathbb{F})$-dual-LPN assumption for dimensions $n = n(\lambda)$ and $N = N(\lambda)$ states that*

$$(\mathbf{G}, \mathbf{Ge}) \approx_c (\mathbf{G}, \mathbf{u})$$

where $\mathbf{G} \leftarrow \mathcal{G}, \mathbf{e} \leftarrow \mathcal{E}$, and $\mathbf{u} \leftarrow \mathbb{F}^n$. Here, $\mathcal{G}$ samples code generators $\mathbf{G} \in \mathbb{F}^{n \times N}$ and $\mathcal{E}$ samples noise vectors $\mathbf{e} \in \mathbb{F}^N$. Further, we say $\mathcal{G}$ is LPN-friendly if this assumption is believed to hold under standard noise distributions $\mathcal{E}$.

Observe that $\mathbf{Ge}$ is not a *codeword*, but is instead a *syndrome* of the dual code. As such, the search problem version of dual-LPN is also referred to as *syndrome decoding* [21].

Equivalence to (primal) LPN. The dual-LPN assumption is equivalent to the LPN assumption [17], which informally states that

$$(\mathbf{H}, \mathbf{Hs}+\mathbf{e}) \approx_c (\mathbf{H}, \mathbf{u})$$

where $\mathbf{H} \in \mathbb{F}^{N \times n'}$ is a parity-check matrix for the code dual to $\mathbf{G}$ and $\mathbf{s} \leftarrow \mathbb{F}^{n'}, \mathbf{e} \leftarrow \mathcal{E}$, and $\mathbf{u} \leftarrow \mathbb{F}^N$. Equivalence holds for $n' = N - n$ by solving for $\mathbf{G} \in \mathbb{F}^{n \times N}$ such that $\mathbf{GH} = 0$ via row-reduction. It follows that

$$\mathbf{G}(\mathbf{Hs}+\mathbf{e}) = (\mathbf{GH})\mathbf{s} + \mathbf{Ge} = \mathbf{Ge}$$

If dual-LPN holds over a distribution of generators $\mathcal{G}$ of a dual code, primal-LPN holds over the corresponding distribution of parity-check matrices $\mathcal{H}$ for the primal code. See, e.g., [14, 22] for further details.

Selecting a noise distribution $\mathcal{E}$. For LPN-based constructions, exact noise is typically used [21], where $\mathcal{E} = \mathcal{E}_t^N$ is the uniform distribution over the set of t-sparse vectors in $\mathbb{F}^N$. However, alternative distributions such as Bernoulli noise or regular noise may be considered.

Selecting code generators $\mathcal{G}$. A natural question arises of which $\mathcal{G}$ are LPN-friendly. The standard LPN assumption instantiates $\mathcal{G}$ as the uniform distribution over matrices in $\mathbb{F}^{n \times N}$. More recently, two separate lines of work in code-based encryption [2, 8] and pseudorandom correlation generators [20–22, 76] have sought to instantiate LPN from families of codes which enable linear or quasilinear encoding time. Such code families include quasi-cyclic, Toeplitz, LDPC [4], expand-accumulate (EA) [21], expand-convolute (EC) [76], and repeat-accumulate-accumulate (RAA) [25], and more [14, 23]. In order to test the concrete security of various possibly LPN-friendly codes, [21] introduced the following unified framework.

Security against linear tests. Suppose $\mathcal{G}$ is a distribution over code generators $\mathbf{G}$ such that the code does not admit an efficient decoding algorithm. For instance, Reed-Solomon codes are highly algebraic and enable fast decoding via the Berlekamp-Massey algorithm [66], which trivially breaks the LPN assumption for these codes. Non-algebraic codes typically do not admit such decoding algorithms.

The key observation of [21] is that all known attacks on LPN over such codes reduce to finding a vector $\mathbf{v} \in \mathbb{F}^n$ (or equivalently a linear functional) such that LPN samples have some non-negligible bias with respect to $\mathbf{v}$. This procedure is called a *linear test* and is done by analyzing $\mathbf{v}^\top \mathbf{b}$ for $\mathbf{b} = \mathbf{Ge} \in \mathbb{F}^n$ and $\mathbf{e} \leftarrow \mathcal{E}$. Linear tests encapsulate information set decoding (ISD) attacks [72], the BKW algorithm [18], Gaussian elimination, and all other existing practical distinguishing attacks [21].

Then, [21] shows that the codes which are resistant to linear tests are precisely those which have high minimum distance, as this ensures that the inner product $\mathbf{v}^\top \mathbf{b} = (\mathbf{v}^\top \mathbf{G})\mathbf{e}$ has $\mathbf{v}^\top \mathbf{G}$ of large Hamming weight and sufficiently high probability of colliding with sparse $\mathbf{e}$. We formally state these ideas below.

Definition A.2 (Bias of a distribution)**.** *For a nonzero vector $\mathbf{v} \in \mathbb{F}^n$ and a distribution $\mathcal{D}$ over $\mathbb{F}^n$, the bias of $\mathcal{D}$ with respect to $\mathbf{v}$ is defined as*

$$\mathrm{bias}_{\mathbf{v}}(\mathcal{D}) := \max_{c \in \mathbb{F}} \left| \Pr_{\mathbf{x} \leftarrow \mathcal{D}} \left[\mathbf{v}^\top \mathbf{x} = c \right] - \frac{1}{|\mathbb{F}|} \right|.$$

The bias of $\mathcal{D}$ is defined as

$$\mathrm{bias}(\mathcal{D}) = \max_{\mathbf{v} \neq 0} \mathrm{bias}_{\mathbf{v}}(\mathcal{D}).$$

For a noise distribution $\mathcal{E} \subset \mathbb{F}^N$ and a code generator $\mathbf{G} \in \mathbb{F}^{n \times N}$, we let $\mathcal{E}_{\mathbf{G}} \subset \mathbb{F}^n$ denote the distribution of $\mathbf{Ge}$ where $\mathbf{e} \leftarrow \mathcal{E}$, as in [1]. Then, *linear testing* corresponds to the search problem where a PPT adversary must output some vector $\mathbf{v} \in \mathbb{F}^n$ such that $\mathrm{bias}_{\mathbf{v}}(\mathcal{E}_{\mathbf{G}}) > \mathsf{negl}(\lambda)$. As shown in [21, Lemma 2.6], if the minimum distance of the code generated by $\mathbf{G}$ is $\geq d$, then

$$\mathrm{bias}(\mathcal{E}_{\mathbf{G}}) = \max_{\substack{\mathbf{u} \in \mathbb{F}^N \\ \mathbf{u} = \mathbf{v}^\top \mathbf{G} \text{ for some } \mathbf{v} \neq 0}} \mathrm{bias}_{\mathbf{u}}(\mathcal{E}) \leq \max_{\substack{\mathbf{u} \in \mathbb{F}^N \\ \mathrm{wt}(\mathbf{u}) \geq d}} \mathrm{bias}_{\mathbf{u}}(\mathcal{E}).$$

For $\mathcal{E} = \mathcal{E}_t^N$ the uniform distribution over t-sparse vectors in $\mathbb{F}^N$, the bias for any $\mathbf{u} \in \mathbb{F}^N$ with Hamming weight at least d can be bounded by

$$\mathrm{bias}_{\mathbf{u}}(\mathcal{E}_t^N) \leq \left(1 - \frac{t}{N}\right)^d < e^{-td/N} = e^{-\delta t}.$$

where $\delta = d/N$ is the relative distance of the given code [21]. Since the search problem requires finding a suitable vector $\mathbf{u} \in \mathbb{F}^N$, we can attain $\lambda - \log_2 N$ bits of security as in [1, 21] by setting $e^{-\delta t} \leq 2^{-\lambda + \log_2 N}$, or equivalently,

$$t \geq \frac{(\ln 2)(\lambda - \log_2 N)}{\delta}$$

where δ depends on the field $\mathbb{F}$ and dimensions n and N. For an asymptotically good code with $\delta = \Theta(1)$, this justifies the parameter selection of $t = O_\lambda(1)$.

The linear-test framework does not account for when the noise distribution is highly structured and an adversary is given sufficiently many samples, which enables algebraic attacks such as Arora-Ge [10]. We ensure that noise is generated via the *exact distribution* $\mathcal{E}_t^N$ to mitigate this risk [2].

B Repeat-Accumulate-Accumulate Codes

We instantiate the dual-LPN assumption with $\mathcal{G}$ being a generator for Repeat-Accumulate-Accumulate (RAA) linear codes.

[2] [21] also remarks that when instantiating the linear-test framework over a ring, one must be wary of projection attacks onto ideals. These do not concern our construction since fields do not contain non-trivial ideals.

We provide a summary of these codes and previous work supporting the conjecture that RAA codes are LPN-friendly.

For any field $\mathbb{F}$, dimension n, block length N divisible by n, and constant $r = N/n$, an RAA code with generator $\mathbf{G}_{\mathrm{RAA}} \in \mathbb{F}^{n \times N}$ is constructed via the composition

$$\mathbf{G}_{\mathrm{RAA}} = \mathbf{R} \cdot \mathbf{\Pi}_1 \cdot \mathbf{V}_1 \cdot \mathbf{A} \cdot \mathbf{\Pi}_2 \cdot \mathbf{V}_2 \cdot \mathbf{A}.$$

Each of the component matrices is sampled and defined as follows:

- Repetition matrix $\mathbf{R} \in \mathbb{F}^{n \times N}$: repeats each entry of the input vector r times, given by

$$R_{i,j} = \begin{cases} 1 & \text{if } (i-1)r < j \leq ir \\ 0 & \text{otherwise} \end{cases}.$$

- Permutation matrix $\mathbf{\Pi} \in \mathbb{F}^{N \times N}$: samples and performs a uniformly random permutation $\pi : [N] \to [N]$ to the input vector, given by

$$\Pi_{i,j} = \begin{cases} 1 & \text{if } \pi(j) = i \\ 0 & \text{otherwise} \end{cases}.$$

- Randomization matrix $\mathbf{V} \in \mathbb{F}^{N \times N}$: samples a uniformly random nonzero vector $\mathbf{v} \leftarrow (\mathbb{F}^*)^N$ along the diagonal of a matrix

$$V_{i,j} = \begin{cases} v_i & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}.$$

- Accumulation matrix $\mathbf{A} \in \mathbb{F}^{N \times N}$: computes the prefix-sum of the input vector by applying the upper triangular matrix

$$A_{i,j} = \begin{cases} 1 & \text{if } i \leq j \\ 0 & \text{otherwise} \end{cases}.$$

Linear-time encoding. We observe that the encoding map $\mathbb{F}^n \to \mathbb{F}^N$ by $\mathbf{w} \mapsto \mathbf{G}_{\mathrm{RAA}}^\top \mathbf{w}$ can be computed in $O(N)$ field operations. By the transposition principle [19], the syndrome map $\mathbb{F}^N \to \mathbb{F}^n$ by $\mathbf{v} \mapsto \mathbf{G}_{\mathrm{RAA}} \mathbf{v}$ can similarly be computed in $O(N)$ field operations. We show that for each component matrix of $\mathbf{G}_{\mathrm{RAA}}$, matrix–vector multiplication can be computed in time $O(N)$.

- Repetition matrix $\mathbf{w} \mapsto \mathbf{R}^\top \mathbf{w}$: requires N copies of field elements—no field operations.
- Permutation matrix $\mathbf{w} \mapsto \mathbf{\Pi}^\top \mathbf{w}$: requires N copies of field elements—no field operations.
- Randomization matrix $\mathbf{w} \mapsto \mathbf{V}^\top \mathbf{w}$: requires N field multiplications.
- Accumulation matrix $\mathbf{w} \mapsto \mathbf{A}^\top \mathbf{w}$: requires N field additions to compute the prefix-sum.

Composing the above operations yields a circuit which computes encodings $\mathbf{w} \mapsto \mathbf{G}_{\mathrm{RAA}}^\top \mathbf{w}$ in at most $4N$ field operations. The concrete count for syndromes $\mathbf{v} \mapsto \mathbf{G}_{\mathrm{RAA}} \mathbf{v}$ sits similarly at $5N$ field operations.

LPN friendliness and asymptotic goodness. RAA codes are non-algebraic due to the accumulation and permutation matrices. As such, it is conjectured that no efficient decoding algorithm exists, and that RAA codes are LPN-friendly [1,14].

To motivate this conjecture, many previous papers have conducted analyses on the minimum distance of RAA codes, from which security against linear tests directly follows. Early analyses focused on asymptotic bounds for the minimum distance of RAA codes and limited their instantiation to binary fields [12,60,78]. Later works [52,57,58,96] showed similarly strong minimum-distance properties and asymptotic goodness for RAA codes over arbitrary prime fields.

Since asymptotic analysis is insufficient for instantiating cryptographic protocols with provable guarantees, further work has provided concrete bounds on the minimum distance of RAA codes over various parameters. Binary fields are analyzed in the Blaze SNARK [25] and proven to achieve constant relative distance, while [1] analyzes prime fields for $n \in [2^4, 2^{10}]$ using combinatorial techniques and extrapolates to larger n. More recently, [3] gives a rigorous prime-field analysis proving a constant relative distance which is larger than in the binary case.

Failure probabilities. Crucially, when considering code generators $\mathbf{G}_{\mathrm{RAA}}$ sampled from a distribution, guarantees of good minimum distance are probabilistic rather than exact. That is, for a specified dimension n, block length N, rate $r = N/n$, field $\mathbb{F}$, and relative distance δ, the code has a *failure probability* that a randomly sampled generator matrix $\mathbf{G}_{\mathrm{RAA}}$ from this distribution will not have relative distance at least δ. We denote this failure probability by $p_{\mathrm{fail}} = p_{\mathrm{fail}}(\mathcal{G}, N, \delta)$. The focus of prior work is to demonstrate that the failure probability can be made arbitrarily small under practical parameters.

In particular, [3] shows that for $\delta = 0.5$, the failure probability is $\widetilde{O}(N^{2-r}) + O(\frac{1}{q-1})$, and [57] sharpens this to $\widetilde{O}(N^{1-r})$ with no additive $1/(q-1)$ term, whenever $q - 1 = \Omega(N)$. Thus, setting $r = N/n \geq 3$ ensures that the failure probability decreases polynomially with N. Furthermore, a framework introduced in Blaze [25] enables publicly *testing* the distance of an RAA code in polynomial time $O(N^w)$ for any integer $w \in \mathbb{N}$ in order to reduce the failure probability to $\widetilde{O}(N^{(w+1)(2-r)})$ [3].

C Private Matrix Extension

Π_{pvMVMD} can be extended to support a client who holds a *private* matrix $\mathbf{M} \in \mathbb{F}^{m \times n}$ in addition to a private vector $\mathbf{x} \in \mathbb{F}^n$, and wishes to delegate $\mathbf{Mx}$ without revealing either operand to the compute server. In private LLM inference, this corresponds to outsourcing linear-layer computation for a propri-

etary model owned by the client.

We next formalize its syntax and security.

Definition C.1 (Private-Matrix pvMVMD)**.** *A private-matrix pvMVMD protocol* Π *is a tuple of PPT algorithms* (PREPROCESS,ENCRYPT,VERIFY,DECRYPT) *defined by:*

- $\textsc{Preprocess}(1^\lambda, 1^\kappa, \mathbf{M}) \to (\mathsf{pp}, \widehat{\mathbf{M}}, \mathsf{st}_{\mathbf{M}})$. *On input security parameter* λ*, statistical parameter* κ*, and matrix* $\mathbf{M} \in \mathbb{F}^{m\times n}$*, outputs public parameters* pp*, masked matrix* $\widehat{\mathbf{M}}$*, and private matrix state* $\mathsf{st}_{\mathbf{M}}$.
- $\textsc{Encrypt}(\mathsf{pp}, \mathbf{x}) \to (\widehat{\mathbf{x}}, \mathsf{st}_{\mathbf{x}})$. *On input public parameters* pp *and vector* $\mathbf{x} \in \mathbb{F}^n$*, outputs ciphertext* $\widehat{\mathbf{x}}$ *and private vector state* $\mathsf{st}_{\mathbf{x}}$.
- $\textsc{Verify}(\mathsf{pp}, \widehat{\mathbf{x}}, \widehat{\mathbf{y}}) \to \{0,1\}$. *On input public parameters* pp*, ciphertext* $\widehat{\mathbf{x}} \in \mathbb{F}^n$*, and server output* $\widehat{\mathbf{y}} \in \mathbb{F}^m$*, outputs* 1 *(accept) or* 0 *(reject).*
- $\textsc{Decrypt}(\mathsf{pp}, \mathsf{st}_{\mathbf{M}}, \mathsf{st}_{\mathbf{x}}, \widehat{\mathbf{y}}) \to \mathbf{y}$. *On input public parameters* pp*, private states* $\mathsf{st}_{\mathbf{M}}, \mathsf{st}_{\mathbf{x}}$*, and ciphertext* $\widehat{\mathbf{y}} \in \mathbb{F}^m$*, outputs plaintext vector* $\mathbf{y} \in \mathbb{F}^m$.

The protocol satisfies:

- ***Completeness****: For any* $\mathbf{M} \in \mathbb{F}^{m\times n}$ *and* $\mathbf{x} \in \mathbb{F}^n$*,*

$$\Pr\left[\mathbf{y} = \mathbf{M}\mathbf{x} \wedge b = 1 \;\middle|\; \begin{array}{l} (\mathsf{pp}, \widehat{\mathbf{M}}, \mathsf{st}_{\mathbf{M}}) \leftarrow \textsc{Preprocess}(1^\lambda, 1^\kappa, \mathbf{M}) \\ (\widehat{\mathbf{x}}, \mathsf{st}_{\mathbf{x}}) \leftarrow \textsc{Encrypt}(\mathsf{pp}, \mathbf{x}) \\ \widehat{\mathbf{y}} = \widehat{\mathbf{M}}\widehat{\mathbf{x}} \\ b \leftarrow \textsc{Verify}(\mathsf{pp}, \widehat{\mathbf{x}}, \widehat{\mathbf{y}}) \\ \mathbf{y} \leftarrow \textsc{Decrypt}(\mathsf{pp}, \mathsf{st}_{\mathbf{M}}, \mathsf{st}_{\mathbf{x}}, \widehat{\mathbf{y}}) \end{array}\right] = 1.$$

- ***Soundness****: There exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for any* $\mathbf{M} \in \mathbb{F}^{m\times n}$ *and* $\mathbf{x} \in \mathbb{F}^n$*,*

$$\Pr\left[\mathbf{y} \neq \mathbf{M}\mathbf{x} \wedge b = 1 \;\middle|\; \begin{array}{l} (\mathsf{pp}, \widehat{\mathbf{M}}, \mathsf{st}_{\mathbf{M}}) \leftarrow \textsc{Preprocess}(1^\lambda, 1^\kappa, \mathbf{M}) \\ (\widehat{\mathbf{x}}, \mathsf{st}_{\mathbf{x}}) \leftarrow \textsc{Encrypt}(\mathsf{pp}, \mathbf{x}) \\ \widehat{\mathbf{y}} \leftarrow \mathcal{A}(\mathsf{pp}, \widehat{\mathbf{M}}, \widehat{\mathbf{x}}) \\ b \leftarrow \textsc{Verify}(\mathsf{pp}, \widehat{\mathbf{x}}, \widehat{\mathbf{y}}) \\ \mathbf{y} \leftarrow \textsc{Decrypt}(\mathsf{pp}, \mathsf{st}_{\mathbf{M}}, \mathsf{st}_{\mathbf{x}}, \widehat{\mathbf{y}}) \end{array}\right] < \mathsf{negl}(\kappa).$$

- ***Client privacy****: There exists a PPT simulator* Sim *such that for any* $\mathbf{M} \in \mathbb{F}^{m\times n}$ *and* $\mathbf{x} \in \mathbb{F}^n$*,*

$$\mathsf{Sim}(1^\lambda, 1^\kappa, 1^m, 1^n) \approx_c \mathsf{View}^{\Pi}_{\mathrm{Server}}(\mathbf{M}, \mathbf{x}),$$

where $\mathsf{View}^{\Pi}_{\mathrm{Server}}(\mathbf{M}, \mathbf{x}) := (\mathsf{pp}, \widehat{\mathbf{M}}, \widehat{\mathbf{x}})$ *is generated by* $(\mathsf{pp}, \widehat{\mathbf{M}}, \mathsf{st}_{\mathbf{M}}) \leftarrow \textsc{Preprocess}(1^\lambda, 1^\kappa, \mathbf{M})$ *and* $(\widehat{\mathbf{x}}, \mathsf{st}_{\mathbf{x}}) \leftarrow \textsc{Encrypt}(\mathsf{pp}, \mathbf{x})$.

Our construction leverages the notion of a *trapdoored matrix* [24, 88], which is a pseudorandom matrix sampled alongside a trapdoor that enables fast MVMs by that matrix.

Definition C.2 (Trapdoored matrix [24, 88])**.** *A trapdoored matrix scheme* TDM *is a pair of PPT algorithms* (TDM.GEN, TDM.EVAL) *with the following syntax:*

- $\text{TDM.}\textsc{Gen}(1^\lambda, 1^m, 1^n) \to (\mathbf{M}', \mathsf{td}_{\mathbf{M}'})$ *outputs a matrix* $\mathbf{M}' \in \mathbb{F}^{m\times n}$ *and a trapdoor* $\mathsf{td}_{\mathbf{M}'}$.
- $\text{TDM.}\textsc{Eval}(\mathsf{td}_{\mathbf{M}'}, \mathbf{x})$ *takes trapdoor* $\mathsf{td}_{\mathbf{M}'}$ *and vector* $\mathbf{x} \in \mathbb{F}^n$*, and outputs a vector in* $\mathbb{F}^m$.

TDM *satisfies the following properties:*

- ***Correctness****: For every* $(\mathbf{M}', \mathsf{td}_{\mathbf{M}'})$ *output by* $\text{TDM.}\textsc{Gen}(1^\lambda, 1^m, 1^n)$ *and every* $\mathbf{x} \in \mathbb{F}^n$*,*

$$\text{TDM.}\textsc{Eval}(\mathsf{td}_{\mathbf{M}'}, \mathbf{x}) = \mathbf{M}'\mathbf{x}.$$

- ***Pseudorandomness****: The distribution of* $\mathbf{M}'$ *is computationally indistinguishable from the uniform distribution over* $\mathbb{F}^{m\times n}$ *to any PPT adversary that does not know* $\mathsf{td}_{\mathbf{M}'}$.

Braverman and Newman [24] and Vaikuntanathan and Zamir [88] both construct TDMs using recursion, where TDM.EVAL runs in quasilinear time $T_{\mathrm{TDM}}(m,n) = \widetilde{O}(m+n)$. Using dual-LPN-friendly linear-time codes, EMVP [14] improves the runtime to linear $T_{\mathrm{TDM}}(m,n) = O(m+n)$.

In our construction, the client masks $\mathbf{M}$ by sampling $(\mathbf{M}', \mathsf{td}_{\mathbf{M}'}) \leftarrow \text{TDM.}\textsc{Gen}$ and uploading $\widehat{\mathbf{M}} := \mathbf{M} + \mathbf{M}'$ to the server, storing $\mathsf{td}_{\mathbf{M}'}$ locally. The preprocessing terms $\mathbf{P} := \widehat{\mathbf{M}}\mathbf{G}_{\mathrm{x}}$ and $\mathbf{Q} := \mathbf{G}_{\mathrm{y}}^\top \widehat{\mathbf{M}}$ are generated as part of pp. For each query, the client masks $\mathbf{x}$ as $\widehat{\mathbf{x}} := \mathbf{x} + \mathbf{G}_{\mathrm{x}}\mathbf{e}_{\mathrm{x}}$, and the server computes $\widehat{\mathbf{y}} := \widehat{\mathbf{M}}\widehat{\mathbf{x}} = (\mathbf{M}+\mathbf{M}')(\mathbf{x}+\mathbf{x}')$. The client checks the claim using VERIFY and decrypts $\mathbf{y} := \widehat{\mathbf{y}} - \mathbf{P}\mathbf{e}_{\mathrm{x}} - \text{TDM.}\textsc{Eval}(\mathsf{td}_{\mathbf{M}'}, \mathbf{x})$. Protocol 4 presents the full protocol.

Theorem C.3. *Suppose that* TDM *satisfies correctness and pseudorandomness,* $\mathcal{G}_x$ *is a distribution over codes with relative distance at least* δ_x *satisfying the* $(\mathcal{E}_{t_x}^{N_x}, \mathcal{G}_x, \mathbb{F})$*-dual-LPN assumption for polynomially many independent samples, and the code generated by* $\mathbf{G}_y$ *has relative distance* δ_y*. Then* $\Pi^{\mathsf{privM}}_{\mathsf{pvMVMD}}$ *satisfies completeness, soundness, and client privacy as defined in Definition C.1 when instantiated with* $|\mathbb{F}| \geq 4$*,* $\ell = \left\lceil \frac{\kappa}{\log_2(|\mathbb{F}|-1)-1} \right\rceil$*, and* $t_y = \left\lceil \frac{\max(\log_2(|\mathbb{F}|-1), \kappa+1)}{-\log_2(1-\delta_y)} \right\rceil$.

Proof. For any vector $\widehat{\mathbf{y}}$, the decrypted output is $\mathbf{y} := \widehat{\mathbf{y}} - \mathbf{P}\mathbf{e}_{\mathrm{x}} - \text{TDM.}\textsc{Eval}(\mathsf{td}_{\mathbf{M}'}, \mathbf{x})$. Using $\mathbf{P} = \widehat{\mathbf{M}}\mathbf{G}_{\mathrm{x}}$, $\text{TDM.}\textsc{Eval}(\mathsf{td}_{\mathbf{M}'}, \mathbf{x}) = \mathbf{M}'\mathbf{x}$, and $\widehat{\mathbf{M}} = \mathbf{M} + \mathbf{M}'$, we derive

$$\begin{aligned} \mathbf{y} - \mathbf{M}\mathbf{x} &= \widehat{\mathbf{y}} - \widehat{\mathbf{M}}\mathbf{G}_{\mathrm{x}}\mathbf{e}_{\mathrm{x}} - \mathbf{M}'\mathbf{x} - \mathbf{M}\mathbf{x} \\ &= \widehat{\mathbf{y}} - \widehat{\mathbf{M}}\mathbf{G}_{\mathrm{x}}\mathbf{e}_{\mathrm{x}} - (\mathbf{M}+\mathbf{M}')\mathbf{x} \\ &= \widehat{\mathbf{y}} - \widehat{\mathbf{M}}(\mathbf{x} + \mathbf{G}_{\mathrm{x}}\mathbf{e}_{\mathrm{x}}) \\ &= \widehat{\mathbf{y}} - \widehat{\mathbf{M}}\widehat{\mathbf{x}}. \end{aligned}$$

Completeness. If the server honestly evaluates $\widehat{\mathbf{y}} = \widehat{\mathbf{M}}\widehat{\mathbf{x}}$, then the identity above yields $\mathbf{y} = \mathbf{M}\mathbf{x}$. Further, VERIFY accepts with probability 1 by Theorem 5.2.

Soundness. If the server returns an output with $\mathbf{y} \neq \mathbf{M}\mathbf{x}$, the identity above implies $\widehat{\mathbf{y}} \neq \widehat{\mathbf{M}}\widehat{\mathbf{x}}$. By Theorem 5.2, VERIFY rejects with probability at least $1 - 2^{-\kappa}$ for these parameters.

$\Pi^{\mathsf{privM}}_{\mathsf{pvMVMD}}$

Input parameters:

- Trapdoored matrix scheme TDM $=$ $(\mathrm{TDM.GEN}, \mathrm{TDM.EVAL})$ for matrices in $\mathbb{F}^{m\times n}$.
- Code distributions $\mathcal{G}_{\mathrm{x}}$ and $\mathcal{G}_{\mathrm{y}}$, sparsity parameters $t_{\mathrm{x}}, t_{\mathrm{y}}$, repetition parameter ℓ, and block lengths $N_{\mathrm{x}}, N_{\mathrm{y}}$ as in Protocol 3.
- Field oracle $\mathbb{F}$.

$\textsc{Preprocess}(1^{\lambda}, 1^{\kappa}, \mathbf{M}) \to (\mathsf{pp}, \widehat{\mathbf{M}}, \mathsf{st}_{\mathbf{M}})$:

- Sample $(\mathbf{M}', \mathsf{td}_{\mathbf{M}'}) \leftarrow \mathrm{TDM.GEN}(1^{\lambda}, 1^{m}, 1^{n})$
- Compute $\widehat{\mathbf{M}} := \mathbf{M} + \mathbf{M}' \in \mathbb{F}^{m\times n}$.
- Sample $\mathbf{G}_{\mathrm{x}} \leftarrow \mathcal{G}_{\mathrm{x}}$ and $\mathbf{G}_{\mathrm{y}} \leftarrow \mathcal{G}_{\mathrm{y}}$.
- Compute $\mathbf{P} := \widehat{\mathbf{M}}\mathbf{G}_{\mathrm{x}} \in \mathbb{F}^{m\times N_{\mathrm{x}}}$.
- Compute $\mathbf{Q} := \mathbf{G}_{\mathrm{y}}^{\top}\widehat{\mathbf{M}} \in \mathbb{F}^{N_{\mathrm{y}}\times n}$.
- Output $\mathsf{pp} := (\mathbf{G}_{\mathrm{x}}, \mathbf{G}_{\mathrm{y}}, \mathbf{P}, \mathbf{Q}, t_{\mathrm{x}}, t_{\mathrm{y}}, \ell)$, $\widehat{\mathbf{M}}$, and $\mathsf{st}_{\mathbf{M}} := \mathsf{td}_{\mathbf{M}'}$.

$\textsc{Encrypt}(\mathsf{pp}, \mathbf{x}) \to (\widehat{\mathbf{x}}, \mathsf{st}_{\mathbf{x}})$:

- Sample $\mathbf{e}_{\mathrm{x}} \leftarrow \mathcal{E}_{t_{\mathrm{x}}}^{N_{\mathrm{x}}}$ and compute $\mathbf{x}' := \mathbf{G}_{\mathrm{x}}\mathbf{e}_{\mathrm{x}} \in \mathbb{F}^{n}$.
- Output $\widehat{\mathbf{x}} := \mathbf{x} + \mathbf{x}'$ and $\mathsf{st}_{\mathbf{x}} := (\mathbf{e}_{\mathrm{x}}, \mathbf{x})$.

$\textsc{Verify}(\mathsf{pp}, \widehat{\mathbf{x}}, \widehat{\mathbf{y}}) \to \{0,1\}$:

- Compute $\mathbf{G}_{\mathrm{y}}^{\top}\widehat{\mathbf{y}} \in \mathbb{F}^{N_{\mathrm{y}}}$.
- Repeat ℓ times:
  - Sample $\mathbf{e}_{\mathrm{y}} \leftarrow \mathcal{E}_{t_{\mathrm{y}}}^{N_{\mathrm{y}}}$.
  - Compute $v_L := (\mathbf{e}_{\mathrm{y}}^{\top}\mathbf{Q})\widehat{\mathbf{x}}$ and $v_R := \mathbf{e}_{\mathrm{y}}^{\top}(\mathbf{G}_{\mathrm{y}}^{\top}\widehat{\mathbf{y}})$.
  - If $v_L \neq v_R$, output 0 (reject) and halt.
- If all ℓ checks pass, output 1 (accept).

$\textsc{Decrypt}(\mathsf{pp}, \mathsf{st}_{\mathbf{M}}, \mathsf{st}_{\mathbf{x}}, \widehat{\mathbf{y}}) \to \mathbf{y}$:

- Parse $\mathsf{st}_{\mathbf{M}} = \mathsf{td}_{\mathbf{M}'}$ and $\mathsf{st}_{\mathbf{x}} = (\mathbf{e}_{\mathrm{x}}, \mathbf{x})$.
- Compute $\mathbf{y}_1 := \mathbf{P}\mathbf{e}_{\mathrm{x}}$ and $\mathbf{y}_2 := \mathrm{TDM.EVAL}(\mathsf{td}_{\mathbf{M}'}, \mathbf{x})$.
- Output $\mathbf{y} := \widehat{\mathbf{y}} - \mathbf{y}_1 - \mathbf{y}_2 \in \mathbb{F}^{m}$.

Protocol 4: Protocol $\Pi^{\mathsf{privM}}_{\mathsf{pvMVMD}}$ for private-matrix and private-vector delegation.

Client privacy. We prove client privacy via two hybrid steps. In the first hybrid, we replace the trapdoored matrix $\mathbf{M}'$ with a uniformly random matrix $\mathbf{U}_{\mathbf{M}} \leftarrow \mathbb{F}^{m\times n}$, which is computationally indistinguishable by the pseudorandomness of TDM. After this replacement, $\widehat{\mathbf{M}} = \mathbf{M} + \mathbf{U}_{\mathbf{M}}$ is uniformly distributed over $\mathbb{F}^{m\times n}$ and independent of $\mathbf{M}$. In the second hybrid, we replace $\mathbf{G}_{\mathrm{x}}\mathbf{e}_{\mathrm{x}}$ with an independently sampled uniform vector $\mathbf{u} \leftarrow \mathbb{F}^{n}$, which is computationally indistinguishable under the $(\mathcal{E}_{t_{\mathrm{x}}}^{N_{\mathrm{x}}}, \mathcal{G}_{\mathrm{x}}, \mathbb{F})$-dual-LPN assumption. After this replacement, $\widehat{\mathbf{x}} = \mathbf{x} + \mathbf{u}$ is uniformly distributed over $\mathbb{F}^{n}$ and independent of $\mathbf{x}$. The resulting distribution is simulated by a PPT simulator $\mathsf{Sim}(1^{\lambda}, 1^{\kappa}, 1^{m}, 1^{n})$ that samples uniform $\widetilde{\mathbf{M}} \leftarrow \mathbb{F}^{m\times n}$ and $\widetilde{\mathbf{x}} \leftarrow \mathbb{F}^{n}$, computes $\widetilde{\mathsf{pp}}$ from $\widetilde{\mathbf{M}}$, and outputs $(\widetilde{\mathsf{pp}}, \widetilde{\mathbf{M}}, \widetilde{\mathbf{x}}) \approx_c \mathsf{View}^{\Pi}_{\mathrm{Server}}(\mathbf{M}, \mathbf{x})$. Computational indistinguishability follows from these two hybrids. □

Efficiency. The online client computes TDM.EVAL in some time $T_{\mathrm{TDM}}(m,n)$ and all other ENCRYPT, VERIFY, and DECRYPT steps in time $O_{\lambda,\kappa}(m+n)$ by Section 6.3. Choosing a linear evaluation time TDM (e.g., in [14]) provides asymptotically optimal online time $O_{\lambda,\kappa}(m+n)$. The server performs exactly one field MVM, and preprocessing requires $O(mn)$ field operations to compute $\mathbf{P}$ and $\mathbf{Q}$.

Drawback on preprocessing. In the public-matrix setting pvMVMD, preprocessing is *transparent* and can be computed once by the server and published globally for all clients. In contrast, for private-matrix delegation, each client must sample their own independent trapdoor $\mathsf{td}_{\mathbf{M}'}$ and mask the matrix $\widehat{\mathbf{M}}$ privately, requiring $O(mn)$ *private* preprocessing time. This construction is therefore most attractive when the preprocessing cost can be amortized across many queries.

D Soundness Composition

We show that any private delegation protocol can be made sound by composing with a sound verifiable delegation protocol. This is the same technique we use in constructing Π_{pvMVMD} from Π_{vMVMD} in Protocol 3, and this demonstrates that a lightweight verification layer can be applied to a private delegation protocol.

Let Π_{vMVMD} be a complete and sound vMVMD protocol, and let Π_{pvMVMD} be a complete and private but possibly unsound private delegation protocol. We informally define the protocol Π'_{pvMVMD} as follows:

- PREPROCESS: run Π_{vMVMD}.PREPROCESS and Π_{pvMVMD}.PREPROCESS.
- ENCRYPT: run Π_{pvMVMD}.ENCRYPT.
- VERIFY: run Π_{vMVMD}.VERIFY.
- DECRYPT: run Π_{pvMVMD}.DECRYPT.

Completeness and client privacy of Π'_{pvMVMD} follow from the corresponding properties of Π_{vMVMD} and Π_{pvMVMD}.

For soundness, suppose the adversary returns $\widehat{\mathbf{y}}$ such that $\mathbf{y} \neq \mathbf{M}\mathbf{x}$, where $\mathbf{y} = \Pi'_{\mathsf{pvMVMD}}.\mathrm{DECRYPT}(\mathsf{pp}, \mathsf{st}, \widehat{\mathbf{y}})$. By the completeness of Π_{pvMVMD}, if $\widehat{\mathbf{y}} = \mathbf{M}\widehat{\mathbf{x}}$ then $\mathbf{y} = \mathbf{M}\mathbf{x}$; taking the contrapositive, if $\mathbf{y} \neq \mathbf{M}\mathbf{x}$, it must hold that $\widehat{\mathbf{y}} \neq \mathbf{M}\widehat{\mathbf{x}}$. The VERIFY algorithm of Π_{vMVMD} is applied to the claim

$$\widehat{\mathbf{y}} \stackrel{?}{=} \mathbf{M}\widehat{\mathbf{x}}.$$

By soundness of Π_{vMVMD}, VERIFY detects this inconsistency except with negligible probability.

E Attack on Slalom at the Carnival

Slalom at the Carnival (S@C) by Bruhns et al. [26] proposes an addition to Slalom [87] which reduces the client online running time required to encrypt and decrypt vectors in a pvMVMD protocol. They claim to achieve client privacy;

however, we show a flaw in their approach which breaks client privacy and enables a semi-honest server to uncover projections of the client's input vectors.

S@C protocol. Fix a matrix $\mathbf{M} \in \mathbb{F}^{m \times n}$ and a parameter $\ell \ll n$ which is dependent on the security parameter. In the S@C preprocessing, the server samples ℓ vectors $\mathbf{r}_1, \ldots, \mathbf{r}_\ell \leftarrow \mathbb{F}^n$ and computes the corresponding projections $\mathbf{u}_1, \ldots, \mathbf{u}_\ell \in \mathbb{F}^m$, where $\mathbf{u}_i = \mathbf{M}\mathbf{r}_i$. These projections are sent to the client. When the client wishes to delegate an MVM for a vector $\mathbf{x}$, they first sample $\mathbf{k} \leftarrow \{0,1\}^\ell$, compute the masking term

$$\mathbf{x}' := \sum_{i=1}^{\ell} k_i \mathbf{r}_i \in \mathbb{F}^n,$$

and encrypt $\widehat{\mathbf{x}} := \mathbf{x} + \mathbf{x}'$. This is sent to the server, which returns $\widehat{\mathbf{y}} = \mathbf{M}\widehat{\mathbf{x}}$. The client decrypts by subtracting the projections:

$$\mathbf{y} := \widehat{\mathbf{y}} - \sum_{i=1}^{\ell} k_i \mathbf{u}_i.$$

The attack. The S@C protocol is efficient only when $\ell < n$. However, in this parameter range, the server can use Gaussian elimination to solve for a projection matrix $\mathbf{\Pi} \in \mathbb{F}^{(n-\ell) \times n}$ annihilating the vectors $\mathbf{r}_1, \ldots, \mathbf{r}_\ell$ (i.e., $\mathbf{\Pi}\mathbf{r}_i = 0$ for each i).

For a client input $\widehat{\mathbf{x}}$, the server applies $\mathbf{\Pi}$ and recovers

$$\mathbf{\Pi}\widehat{\mathbf{x}} = \mathbf{\Pi}\left(\mathbf{x} + \sum_{i=1}^{\ell} k_i \mathbf{r}_i\right) = \mathbf{\Pi}\mathbf{x} \in \mathbb{F}^{n-\ell}.$$

This enables the server to distinguish $\widehat{\mathbf{x}}$ from truly random and precludes the existence of a privacy simulator that works for all $\mathbf{x}$.