zklora: Verifiable Multi-Party Inference for Evaluating Private LoRA Weights

Anonymous Authors

January 11, 2025

Abstract

In decentralized computing environments, one party may wish to fine-tune a publicly available base model with Low-Rank Adaptation (LoRA) but keep those adapter weights private, while another party wants to evaluate performance on new data. We introduce `zklora`, a library designed to help the *Base Model User verifiably assess* the quality of a private LoRA module for a target task—for instance, to run a validation experiment without directly accessing the private weights. By adopting a multi-party inference approach and generating *verifiable computations*, `zklora` enables the *Base Model User* to confirm that remote LoRA computations are correct, all without seeing the private weights themselves. We demonstrate the feasibility of `zklora` via benchmark experiments, showing that our approach remains computationally viable across various model sizes, with notably fast verification times.

1 Introduction

Large Language Models (LLMs) excel at a variety of tasks but come with significant computational and storage costs [1, 2]. Low-Rank Adaptation (LoRA) [4] addresses these costs by introducing a small number of trainable parameters into the model, drastically reducing memory overhead for fine-tuning [3].

When two entities, a *LoRA Owner* and a *Base Model User*, collaborate on a shared LLM, they may have conflicting constraints. The *LoRA Owner* wants to keep custom LoRA weights private (e.g., for competitive or regulatory reasons), while the *Base Model User* wants to *test* or *validate* these LoRA weights on a target task before deciding whether to deploy them at scale. Under normal circumstances, either the LoRA must be fully shared (sacrificing privacy), or the Base Model User must trust the LoRA Owner's claims about performance with little evidence.

`zklora` bridges this gap by orchestrating multi-party inference: the base model user remains in control of the main layers, whereas the LoRA Owner privately applies LoRA transformations. The user obtains a zero-knowledge proof demonstrating that the LoRA transformations are indeed correct, thus verifying the final performance metrics. Once satisfied that the LoRA weights meet their needs, the Base Model User can incorporate them at scale through whatever inference arrangement they prefer.

2 Preliminary Results

We benchmarked `zklora` by measuring the creation and verification times for our verifiable computations across multiple base models and private LoRA configurations. For each model, we computed a forward pass for a batch of 3 items with sequence length 5. Each experiment simulates the workflow where the *Base Model User* uses local data to evaluate the base model + LoRA combination, requesting remote LoRA computations from the *LoRA Owner*. We report total and average proof-generation times (`total_prove`, `avg_prove`) and verification times (`total_verify`, `avg_verify`), as well as relevant model parameters.

base_model	**#lora**	**total_params**	**avg_params**	**total_prove**	**avg_prove**	**total_verify**	**avg_verify**
distilgpt2	24	589824	24576.0	759.33	31.64	16.56	0.69
gpt2	48	2359296	49152.0	1675.58	34.91	32.79	0.68
Llama-3.2-1B	32	851968	26624.0	991.93	31.00	24.91	0.78
Llama-3.3-70B-Instr.	80	11796480	147456.0	3749.76	46.87	123.11	1.54
Llama-3.1-8B-Instr.	32	5242880	163840.0	1527.40	47.73	35.79	1.12
Mixtral-8x7B-Instr.	32	10485760	327680.0	2357.61	73.68	44.30	1.38

Table 1: Benchmark results for `zklora` showing multi-party inference overhead on various model + LoRA combinations. Times are in seconds.

Notably, although proof-generation times can grow with model size, verification remains comparatively fast (e.g., an average of around 1–2 seconds per proof even for 70B-scale models). This result demonstrates that `zklora` allows frequent correctness checks without excessive overhead, which is crucial for real-world "trial runs." In larger-scale usage scenarios, once trust in the LoRA module is established, `zklora` may no longer be needed.

3 Multi-Party Inference: zklora for Verifiable Evaluation

Our aim is to enable the *Base Model User* to run a *validation experiment* on their local data, incorporating the private LoRA module from the *LoRA Owner*. The Base Model User can measure task performance (e.g., cross-entropy loss, accuracy) to decide whether the LoRA meets performance requirements.

Figure 1 illustrates a simple scenario in which:

1. **Base Model Forward Pass (Base Model User).** The *Base Model User* runs layers without LoRA parameters locally, generating partial activations.
2. **Forward to LoRA Adapter (Base Model User).** Whenever the next layer includes a LoRA transform, the *Base Model User* sends these partial activations to the *LoRA Owner*, who holds the private LoRA module.
3. **LoRA Computation & Proof Generation (LoRA Owner).** The *LoRA Owner* applies the LoRA transformation to the activations and returns the updated activations alongside a *cryptographic proof* of correctness.
4. **Resuming the Forward Pass (Base Model User).** The *Base Model User* injects the updated activations back into the local model state and continues the forward pass.
5. **Verification & Validation (Base Model User).** The *Base Model User* verifies the LoRA proof. If valid, the outputs from the combined base + LoRA model are guaranteed to match the LoRA Owner's claimed computation. The Base Model User can now measure final metrics (loss, accuracy, etc.) to assess *quality* on the target task.

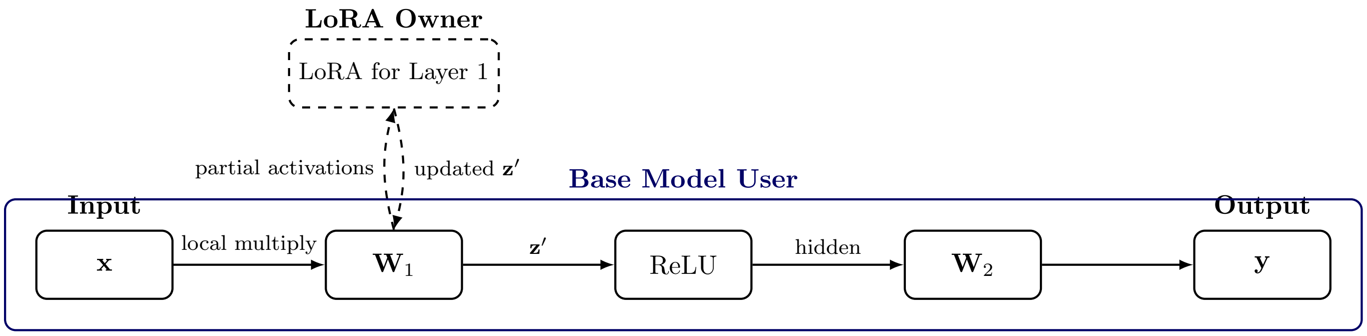

Figure 1: A simplified 2-layer MLP with LoRA on the first layer. The Base Model User (blue box) runs most computations locally but calls the LoRA Owner (dashed box) to update partial activations. After verifying correctness, the Base Model User can measure final metrics to decide whether to adopt the private LoRA more extensively.

4 Generating the Zero-Knowledge Proof

A central piece of our methodology is producing a cryptographic proof that the LoRA transformations were correctly computed—without ever disclosing the LoRA parameters themselves. We outline the proof-generation process below, referencing the flow in the accompanying code. For simplicity, we assume the LoRA Owner has already exported the LoRA-modified layer(s) as an ONNX model and that intermediate activations are saved in a JSON file. Each pairing of "model file" and "JSON input data" yields a distinct proof.

4.1 Preparing Model, Inputs, and Circuit

Circuit and ONNX File. The LoRA Owner provides an ONNX model that includes the relevant LoRA weights. We then compile this model into a zero-knowledge friendly circuit format. This step:

- Parses the graph structure from the ONNX file.
- Determines how many constraints or "gates" are needed.
- Produces a circuit representation that can be used for further proof-generation steps.

Input Activations (JSON). When the *Base Model User* runs the partial forward pass locally, the intermediate activations for that partial pass are recorded in a JSON file. This JSON effectively captures the inbound tensor(s) that will go through the remote LoRA layers.

4.2 Generating Setup and Keys

Settings and Keys. To make the cryptographic proof work, the system needs:

- **Settings File:** Summarizes the circuit's structure (e.g., log rows, public vs. private parameters).
- **Proving/Verification Keys (PK/VK):** Analogous to a public-private key system, enabling one party to prove a statement and another to verify.

We produce these keys from the compiled circuit. If the protocol also requires a structured reference string (SRS), we generate that as well. This SRS encapsulates certain group elements or domain parameters for polynomial commitments, usually saved in a file.

4.3 Generating the Witness

Witness Creation. With the circuit in place, we feed the JSON data for intermediate activations into the compiled circuit. This step:

- Performs the actual forward pass inside the circuit, substituting the LoRA parameters from the ONNX file.
- Produces a "witness file," which is a low-level representation of all wire values in the circuit.

In practice, this might be asynchronous or batched, ensuring large inputs can be processed efficiently.

4.4 Proof Generation

Prove. Once we have:

- The circuit representation.
- The proving key (PK).
- The witness file.

we invoke a "prove" routine that executes polynomial commitment schemes and solves the constraint system. This yields a standalone proof file that can be shared with the *Base Model User*.

Result. The final proof cryptographically certifies that the LoRA Owner used the declared ONNX model and input JSON to compute the updated activations *exactly* as claimed, with no tampering or hidden changes.

4.5 Proof Verification

Checking the Proof. The *Base Model User*, upon receiving the proof, uses:

- The verification key (VK).
- The structured reference string (SRS).
- The proof file.

A single verification call then confirms or denies the proof's correctness. If correct, it implies that the LoRA transformations were faithfully applied.

Timing. In our benchmarks, this verification step is typically on the order of 1–2 seconds, even for large models, as shown in Table 1. The minimal overhead of verification makes `zklora` feasible for continuous or repeated "trial runs" to confirm LoRA quality.

5 Related Work

5.1 Low-Rank Adaptation

LoRA [4] has emerged as a popular way to efficiently fine-tune LLMs by injecting small, low-rank matrices into specific layers. This approach scales well, requiring less GPU memory than full fine-tuning. Various frameworks build on LoRA for GPT-2, GPT-Neo, LLaMA, and others [3]. By confining most updated parameters to low-rank adapter matrices, LoRA drastically reduces memory overhead during fine-tuning.

5.2 Incrementally Verifiable Computation

Valiant [7] introduced the notion of Incrementally Verifiable Computation (IVC), which offers updatable, compact proofs of correctness over multi-step computations. Follow-up work [6, 5] refines these ideas, leveraging folding schemes and zero-knowledge properties to enable scaling of verifiable computations. While prior research has investigated IVC for inference or specialized tasks, `zklora` adapts such cryptographic proof techniques for verifying private LoRA transformations in multi-party inference.

6 Conclusion

`zklora` enables a *Base Model User* to verify the quality of a private LoRA module without revealing the LoRA weights. Our benchmarks show that proof generation remains feasible for multiple model sizes, and verification overhead is small enough to allow iterative experiments. This design provides strong assurances of correctness while preserving LoRA privacy. Future directions include optimizing circuit compilation, supporting multi-owner LoRA composition, and investigating broader multi-party computation frameworks that incorporate private data alongside private model parameters.

References

[1] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners, 2020.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[3] Ning Ding, Zhuosheng Zheng, Fei Tan, Yuxian Chen, Xipeng Xie, Zhiyang Liu, Xinze Dai, and et al. Delta tuning: A comprehensive study of parameter efficient methods for pre-trained language models, 2022.

[4] Edward Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.

[5] Rahul Kothapalli et al. Hypernova: High-degree polynomial folding via customizable constraint systems, 2024.

[6] Abhiram Kumar, Mary Maller, Pratyush Mishra, Shashank Siri, Giovanni Tessaro, Pratik Vasudevan, and Yupeng Zhao. Nova: Recursive zero-knowledge arguments from folding schemes, 2022.

[7] Leslie G Valiant. Incrementally verifiable computation or ivc. https://dash.harvard.edu/handle/1/5026950, 2008. Harvard University, Technical Report.