Verifiable & Private Inference

Methods beyond ZK & FHE

Erhan Tezcan

Lead Developer @ Dria

4.9.2025

Introduction

whoami

- Lead Developer at **Dria**
- We are building a peer-to-peer network of

State of AI & LLMs

Artificial Intelligence (AI), and Large Language Models (LLMs) in particular, are revolutionizing the world. Close to %10 of the entire world population is using ChatGPT alone[1].

New models are coming out every week, smashing the existing records on numerous benchmarks, with an ever increasing performance demand[2].

We are actually progressing faster than we though we were, as noted by powerhouse's such as OpenAI, Anthropic, and Google DeepMind[3].

[1] https://backlinko.com/chatgpt-stats

[2] https://hai.stanford.edu/ai-index/2025-ai-index-report

[3] https://80000hours.org/agi/guide/when-will-agi-arrive/

Within this talk, we are specifically interested in the **inference** part of the AI/LLM stack. Inference is the process of running a trained model to make predictions or generate outputs based on new input data, as shown in Figure 1.

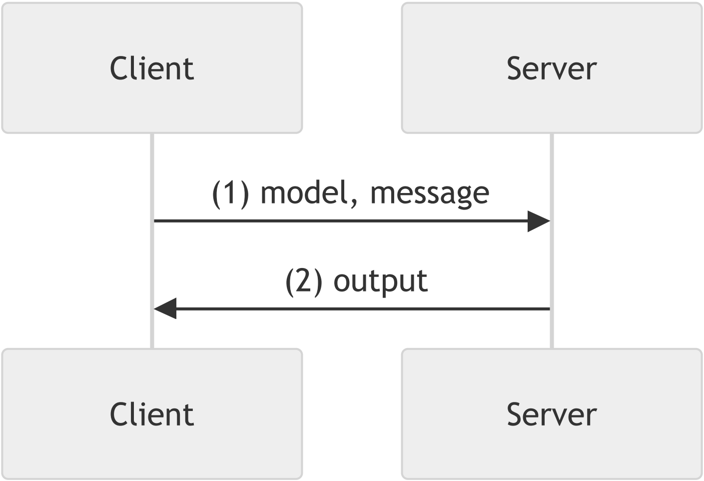


Figure 1: Inference

State of AI & LLMs

There are two problems with such an inference:

- Is the **Server** really using `model` and `message` to generate the `output`?
- Is the **Server** peeking into user `message`?

These problems are denoted as **verifiable inference** and **private inference**, respectively.

Problem Setting

We would like to focus on **consumer-grade** model providers in particular, as they can:

- Locally serve models on their own hardware, utilizing their idle-compute on open-source models
- Join a permissionless network & earn from their services
- Decentralize the inference market, which is currently dominated by a few big players

ZK & FHE & TEE

Zero-Knowledge Proofs (ZK)

A zero-knowledge proof proves knowledge of a secret information without revealing it. For example, you can tell “I know the preimage to some SHA256 digest”.

Zero-Knowledge Proofs (ZK)

A zero-knowledge proof proves knowledge of a secret information without revealing it. For example, you can tell "I know the preimage to some SHA256 digest".

It has three notable properties:

- **Completeness**: If the statement is `true`, an honest prover can convince any verifier.
- **Soundness**: If the statement is `false`, no dishonest prover can convince the verifier.
- **Zero-Knowledge**: If the statement is `true`, the verifier learns nothing beyond it being `true`.

Zero-Knowledge Proofs (ZK)

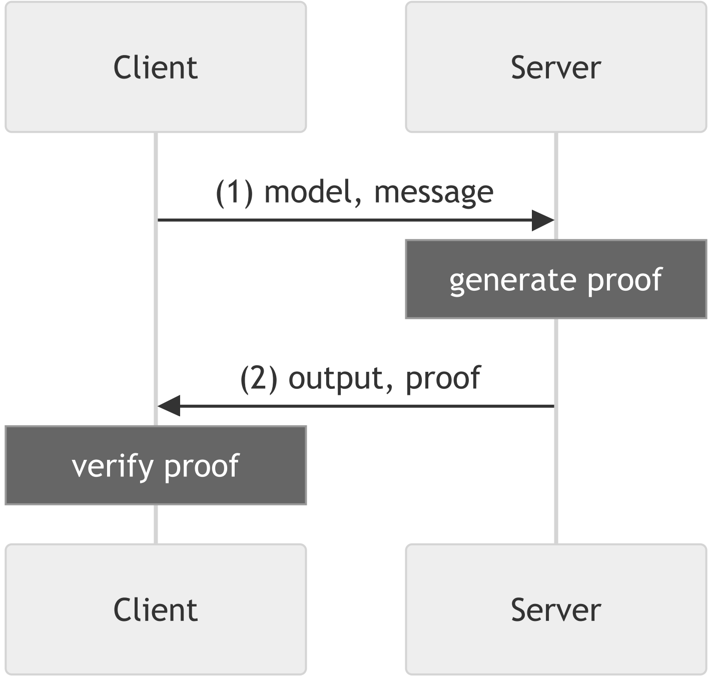


Figure 2: Inference with ZK

Zero-Knowledge Proofs (ZK)

Within LLM inference, a ZK-proof tells you that indeed your chosen model was used for inference (Figure 2). It does NOT hide the user prompt though, because the proof **requires** processing the prompt.

Fully Homomorphic Encryption (FHE)

FHE allows computation on encrypted data without decrypting it.

- $\mathrm{Enc}(a) + \mathrm{Enc}(b) = \mathrm{Enc}(a + b)$
- $\mathrm{Enc}(a) * \mathrm{Enc}(b) = \mathrm{Enc}(a * b)$

This allows one to hide the input & output data during inference.

Fully Homomorphic Encryption (FHE)

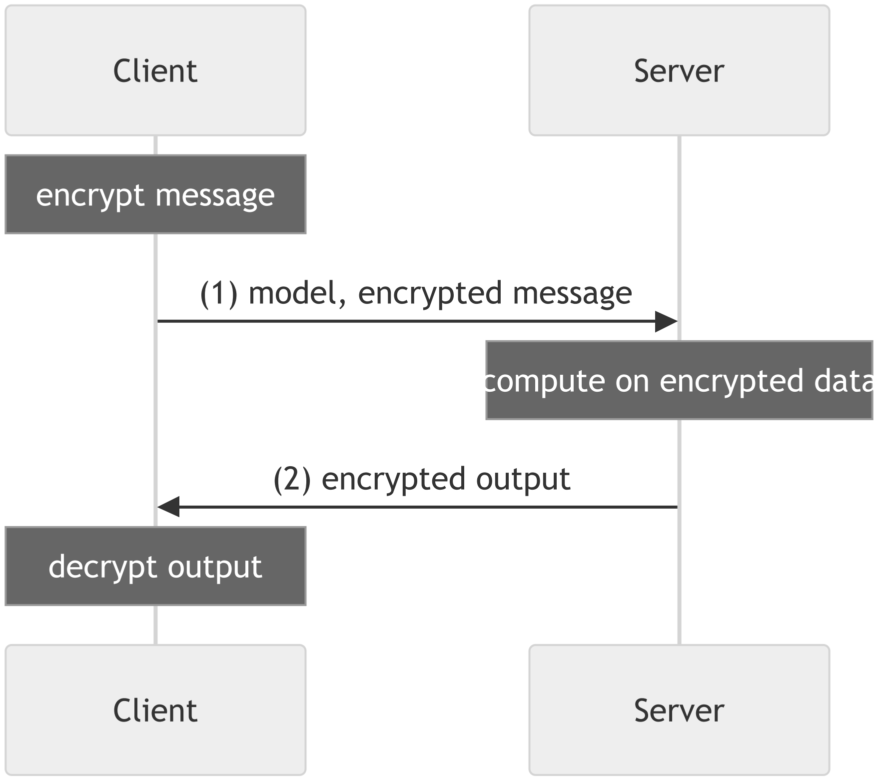


Figure 3: Inference with FHE

Trusted Execution Environments (TEE)

TEEs are hardware-based secure enclaves (Intel SGX, ARM TrustZone, AMD SEV). They ensure isolated execution with memory encryption, along with a remote attestation to prove code integrity.

We get both the privacy and verifiability at once here, at the cost of a trusted attestion service and hardware.

Trusted Execution Environments (TEE)

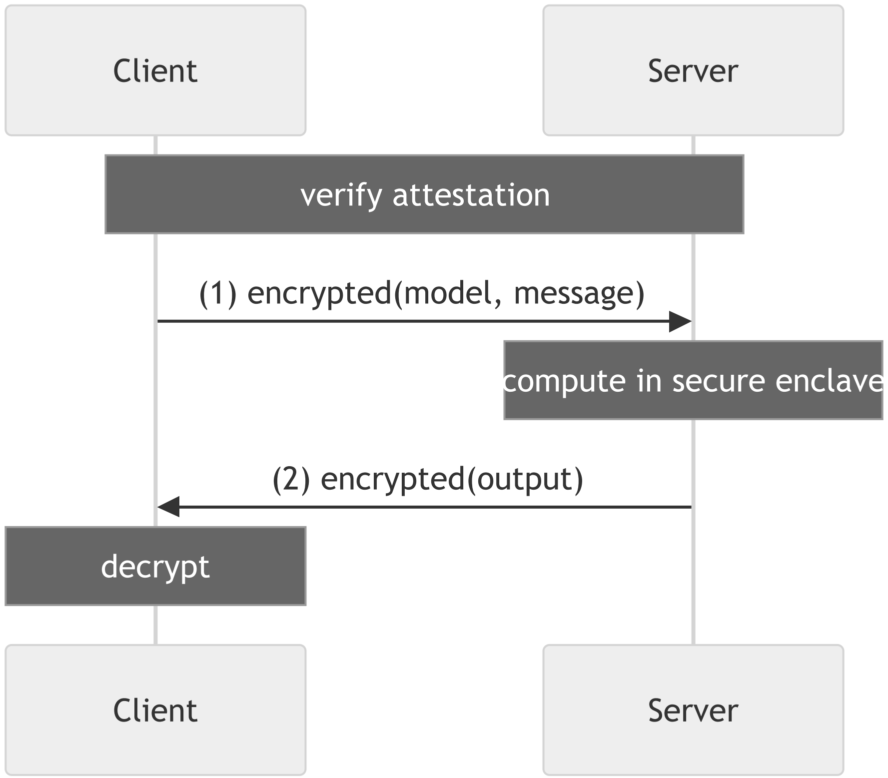


Figure 4: Inference with TEE

Problems with ZK & FHE & TEE

All of these methods so far have drawbacks that limit their applicability in real-world scenarios [1], [2].

Problems with ZK & FHE & TEE

All of these methods so far have drawbacks that limit their applicability in real-world scenarios [1], [2].

- LLMs make use of floating point arithmetic, to the dismay of finite field arithmetic in ZK and FHE.
- TEEs require a trusted hardware environment, which is beyond a "consumer-grade" hardware. Furthermore, they operate in rather memory and compute constrained environments; yet still are open to side-channel attacks.

Transformers

The Transformer

LLMs are **neural networks** trained on massive text datasets with a sole purpose: to predict the next token in a sequence.

The Transformer

LLMs are **neural networks** trained on massive text datasets with a sole purpose: to predict the next token in a sequence.

With that, they can generate text by outputting one token at a time, and feeding it back as input, in a loop.

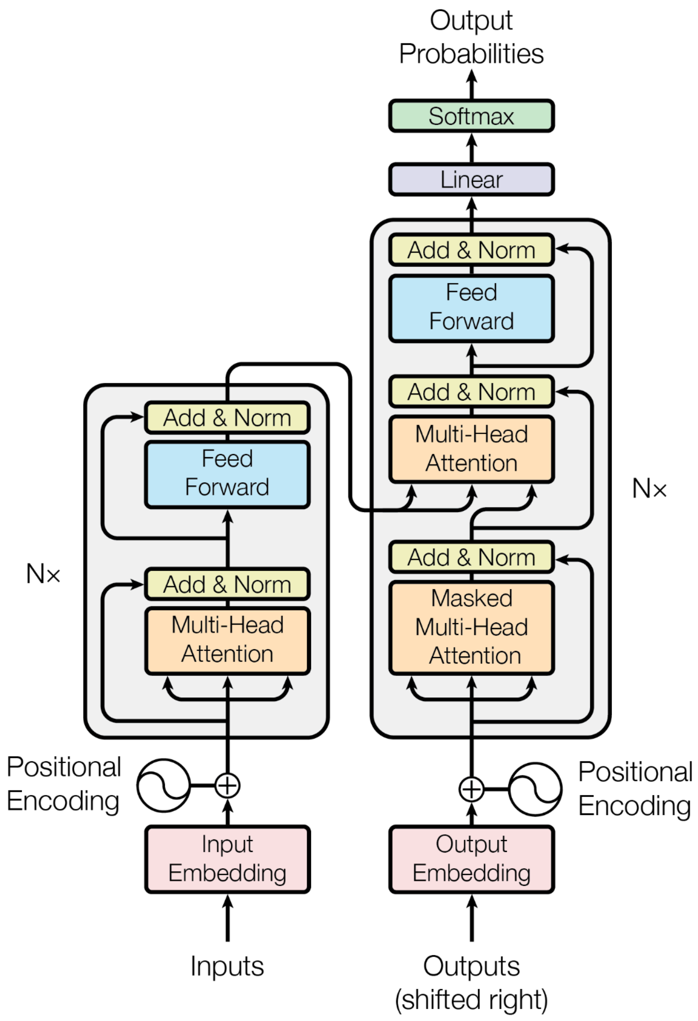
This brings **emergent capabilities**: reasoning (?), knowledge synthesis, code generation, …

The Transformer

The dominant architecture today is the Transformer, introduced in "Attention is All You Need" (2017) [3].

- Replaced RNNs with **self-attention** mechanism.
- Parallel processing enables efficient training.

The Transformer



Key Components

- **Self-Attention**: allows tokens to attend to all other tokens in sequence; multiple attention heads capture different relationships.
- **Feed-Forward Networks**: position-wise fully connected layers; provides non-linear transformations.
- **Layer Normalization & Residual Connections**: stabilizes training and enables deeper networks.

Encoder-Decoder

The original Transformer describes both encoder and decoder:

- **Encoder**: processes input sequence, generates contextualized representations.
- **Decoder**: generates output sequence autoregressively.

This has to do with the *mask* used within the self-attention block.

Encoder-Decoder

The original Transformer describes both encoder and decoder:

- **Encoder**: processes input sequence, generates contextualized representations.
- **Decoder**: generates output sequence autoregressively.

This has to do with the *mask* used within the self-attention block.

Modern LLMs are decoder-only:

- `GPT`, `LLaMA` use decoder-only architecture.
- Simpler design, better scaling properties.
- Trained with causal (left-to-right) attention mask.

We are focused on **decoder-only** models in this talk!

Prefill vs. Decoding

There are two main steps in autoregressive inference:

- **Prefill**: initialize the context with a prompt or previous tokens. While this is compute intensive, it is highly parallelisable!
- **Decoding**: generate the output sequence one token at a time. This step is inherently sequential and cannot be parallelized.

This will be important later on.

Challenge & Verify

Methodology

If you have a model provider that you would like to check for simple compliance, a better-than-nothing is to use **vanilla verification**.

Every few requests, you can send a procedurally generated prompt with a known output, and check if the provider returns the correct result.

- For mathematical reasoning, there are static analysis tools like MathVerify[1].
- For code generation, you can use a sandboxed unit-test to check if the generated code works as expected.
- For text generation, you can use a set of known question-answer pairs; or simply do a string inclusion check.

[1] https://github.com/huggingface/Math-Verify

Example

A simple example:

- Have a list of animals:

`["cat", "dog", "bird", "snake", "fish"]`)

- Pick 2 animals, and pick 2 numbers, and generate a prompt:

“How many legs are there in total for `N` `animal1`s and `M` `animal2`s?”

- Compare the model’s response with the expected output.

Problems

There are multiple problems with this approach:

Problems

There are multiple problems with this approach:

False-negatives: sometimes the model is just not clever, and fails the challenge. Other times, the verification process may be faulty. You should provide a **margin of error**, instead of banning on first failure.

Problems

There are multiple problems with this approach:

False-negatives: sometimes the model is just not clever, and fails the challenge. Other times, the verification process may be faulty. You should provide a **margin of error**, instead of banning on first failure.

A clever attacker could detect the challenge, and **only** run honest model for those challenges.

Problems

There are multiple problems with this approach:

False-negatives: sometimes the model is just not clever, and fails the challenge. Other times, the verification process may be faulty. You should provide a **margin of error**, instead of banning on first failure.

A clever attacker could detect the challenge, and **only** run honest model for those challenges.

You need different challenges for each model, to avoid false-positives on small models acting like large ones while passing the easy challenges.

Problems

There are multiple problems with this approach:

False-negatives: sometimes the model is just not clever, and fails the challenge. Other times, the verification process may be faulty. You should provide a **margin of error**, instead of banning on first failure.

A clever attacker could detect the challenge, and **only** run honest model for those challenges.

You need different challenges for each model, to avoid false-positives on small models acting like large ones while passing the easy challenges.

story time

Veri-Split

Secure and Practical Offloading of Machine Learning Inferences

Splitting with Noise

- for non-linear parts, they use the device itself
- expensive matrix multiplication is off-loaded

https://arxiv.org/html/2405.20681v1 says privacy with noise MUST incur loss

Veri-Split: Merkle Trees

Veri-Split paper also proposes using a Merkle Tree for comitting to intermediate layers, followed by a random interactive protocol to request Merkle Proofs.

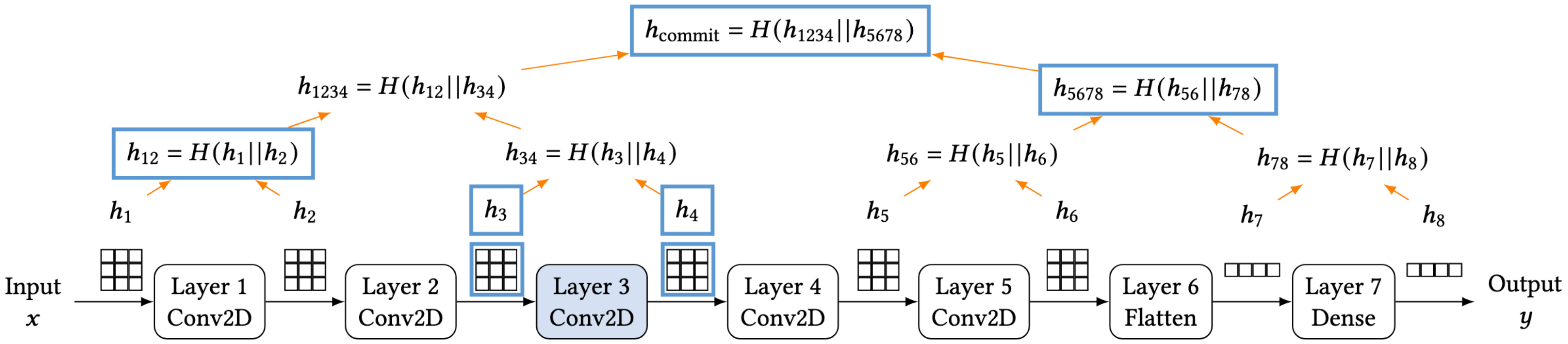


Figure 6: Merkle Tree for Intermediate Layer Commitment

STIP

Secure Transformer Inference Protocol

STIP

STIP (Secure Transformer Inference Protocol) is a novel approach to secure inference in transformer models. It leverages permutation-based techniques to achieve privacy.

The main idea comes from [4]

TOPLOC

A Locality Sensitive Hashing Scheme for

Trustless Verifiable Inference

TOPLOC [5] is a novel method of committing to an LLM-inference.

It is vulnerable against speculative decoding: using a smaller & more efficient model for decoding phase, and then prefilling with the actual model.

Bibliography

[1] H. Zhang, Z. Wang, M. Dhamankar, M. Fredrikson, and Y. Agarwal, “VeriSplit: Secure and Practical Offloading of Machine Learning Inferences across IoT Devices.” [Online]. Available: https://arxiv.org/abs/2406.00586

[2] M. Labs, "The Cost of Intelligence: Proving Machine Learning Inference with Zero-Knowledge." Jan. 2023.

[3] A. Vaswani *et al.*, "Attention Is All You Need." [Online]. Available: https://arxiv.org/abs/1706.03762

[4] H. Xu, L. Xiang, H. Ye, D. Yao, P. Chu, and B. Li, "Permutation Equivariance of Transformers and Its Applications." [Online]. Available: https://arxiv.org/abs/2304.07735

[5] J. M. Ong *et al.*, "TOPLOC: A Locality Sensitive Hashing Scheme for Trustless Verifiable Inference." [Online]. Available: https://arxiv.org/abs/2501.16007

Thank You!