Verifiable & Private Inference

Methods beyond ZK & FHE

Erhan Tezcan

4.9.2025

Introduction

whoami

- Lead Developer at **Dria**[1]
- We are building:
  - A permissionless peer-to-peer network of locally-served LLMs (~27k nodes).
  - Special-purpose *tiny* models[2].
  - A custom-built inference engine & compiler for distributed serving of LLMs.
  - A decentralized AI application layer over the network.
- Interested in zero-knowledge cryptography for the last 3 years, with a focus on developer tooling & Circom.

This presentation is written in **Typst**, and is open-source at the link at the bottom of the slide.

[1] https://dria.co
[2] https://huggingface.co/driaforall

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.

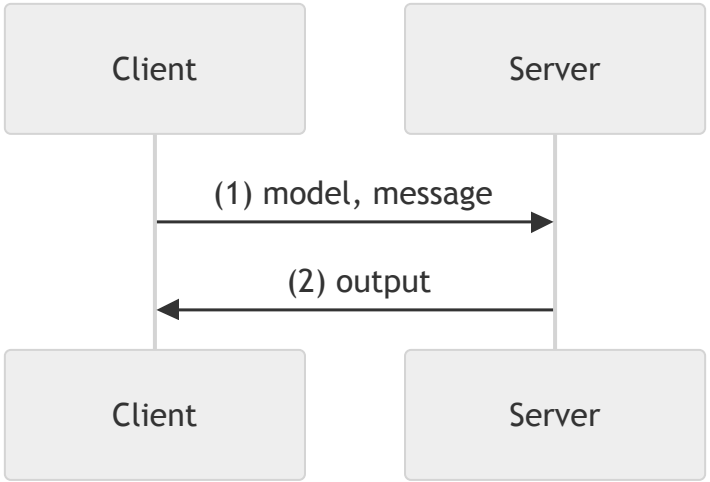


Figure 1: Inference

There are two problems with such an inference:

- Can we make sure that **Server** is really using `model` and `message` to generate its `output`? (Verifiable Inference)
- Can we make sure that **Server** does not see what is written in our `message` and `output`? (Privacy-Preserving Inference)

There are two problems with such an inference:

- Can we make sure that **Server** is really using `model` and `message` to generate its `output`? (Verifiable Inference)
- Can we make sure that **Server** does not see what is written in our `message` and `output`? (Privacy-Preserving Inference)

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`.

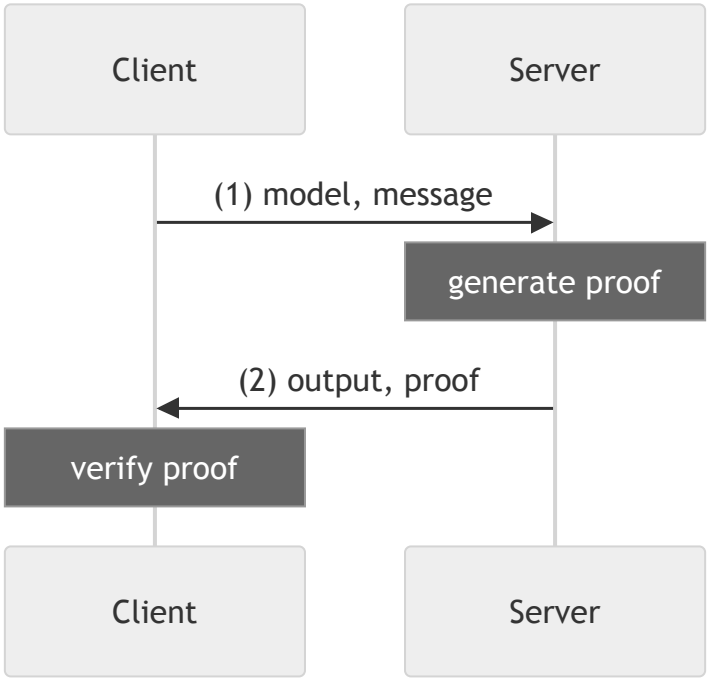


Figure 2: Inference with ZK

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.

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

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

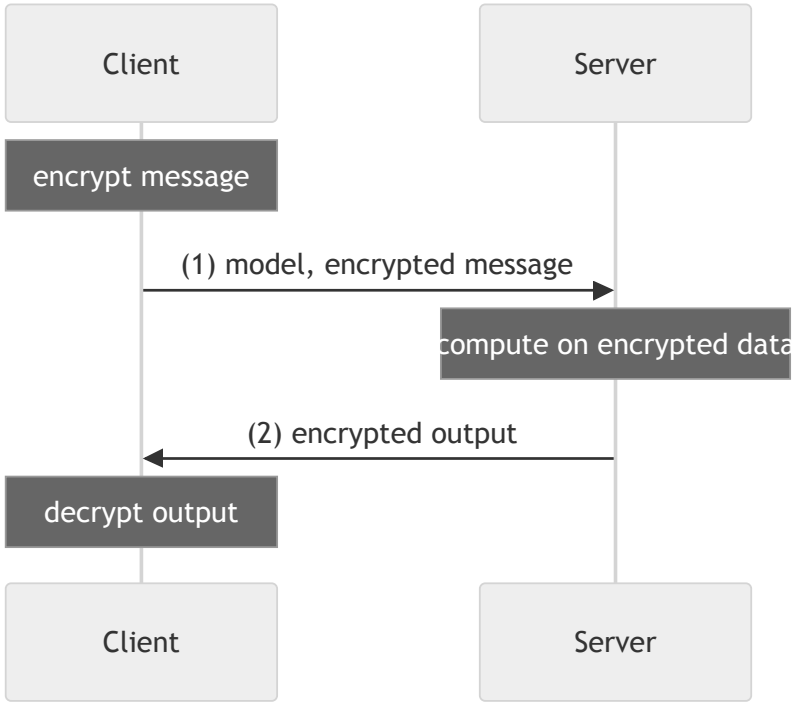


Figure 3: Inference with FHE

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.

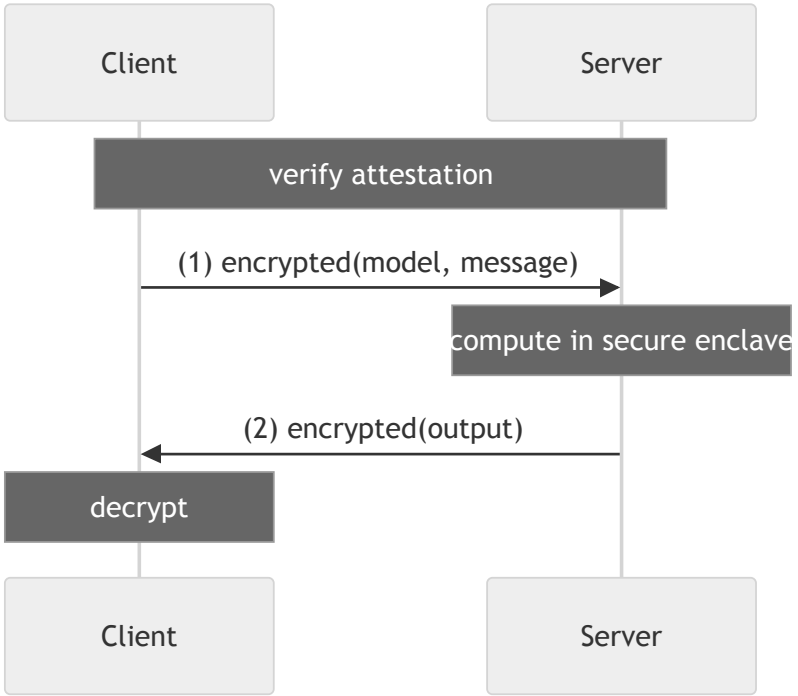


Figure 4: Inference with TEE

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, bad for the finite field arithmetic in ZK and FHE.
- TEEs require a trusted hardware environment. Furthermore, they operate in rather memory and compute constrained environments; yet still are open to side-channel attacks [3].
- These make it even harder for practical deployment in real-world applications, especially on consumer-grade devices.

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, and much more.

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.

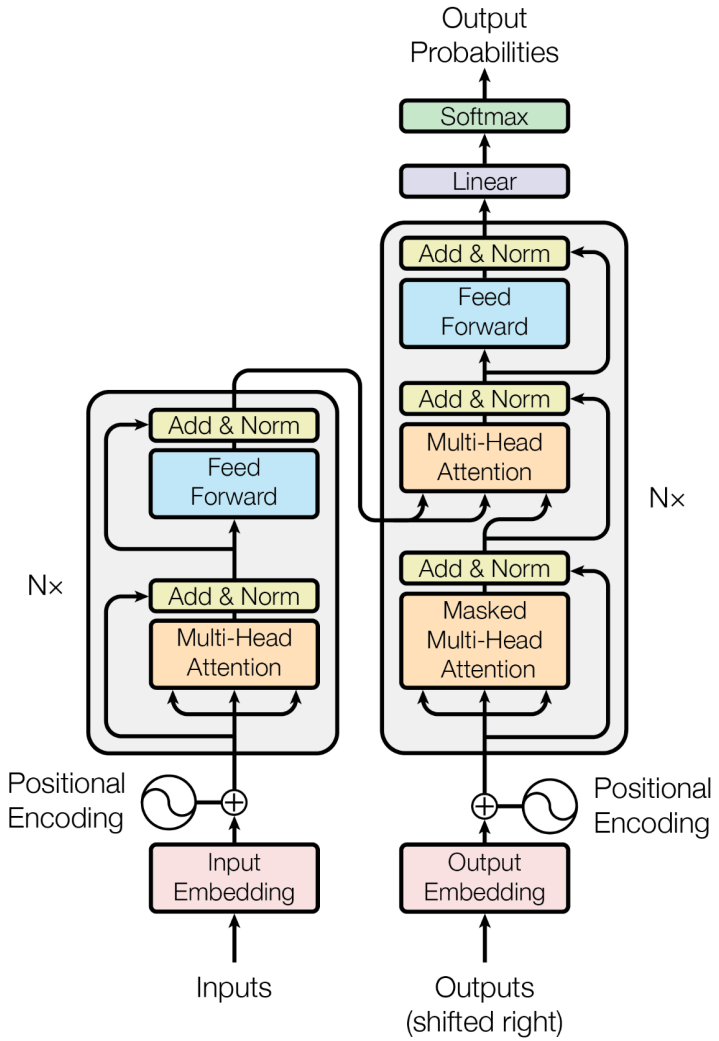
This brings **emergent capabilities**: reasoning, knowledge synthesis, code generation, and much more.

The dominant architecture today is the Transformer, introduced in "Attention is All You Need" (2017) [4].

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

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

If you have a model provider that you would like to check for simple compliance (especially for smal models), a better-than-nothing is to use a simple challenge & verify method.

If you have a model provider that you would like to check for simple compliance (especially for smal models), a better-than-nothing is to use a simple challenge & verify method.

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 [5].
- 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.

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 3 dogs and 2 birds?"
```

- Compare the model's response with the expected output using static analysis.

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.
- Models can be funny: “How many legs are there among 0 snakes and 0 snakes?”, the model says: “Of course none at all hahah!” instead of “0”.

Veri-Split

Secure & Practical Offloading of ML Inference

Splitting with Noise

One of the early examples of privacy-preserving inference offloading comes with Veri-Split, where they do $y = Wx + b$ as follows (for a 2-device 1 user setting):

- Generate random masks δ and β
- Send $\frac{W+\delta}{2}, \frac{b+\beta}{2}$ to one party
- Send $\frac{W-\delta}{2}, \frac{b-\beta}{2}$ to other party
- Receive $y_1 = \frac{W+\delta}{2} * x + \frac{b+\beta}{2}$
- Receive $y_2 = \frac{W-\delta}{2} * x + \frac{b-\beta}{2}$
- Add them together to obtain $y = Wx + b$

Other **non-linear** computations are kept on the user device.

Splitting with Noise

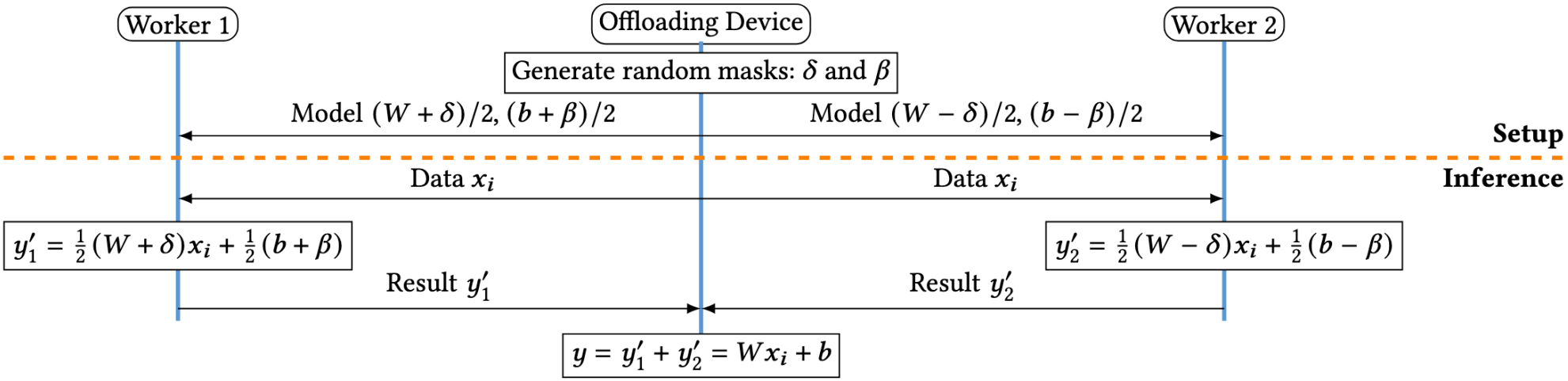


Figure 6: Split with Noise

Commitment with 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.

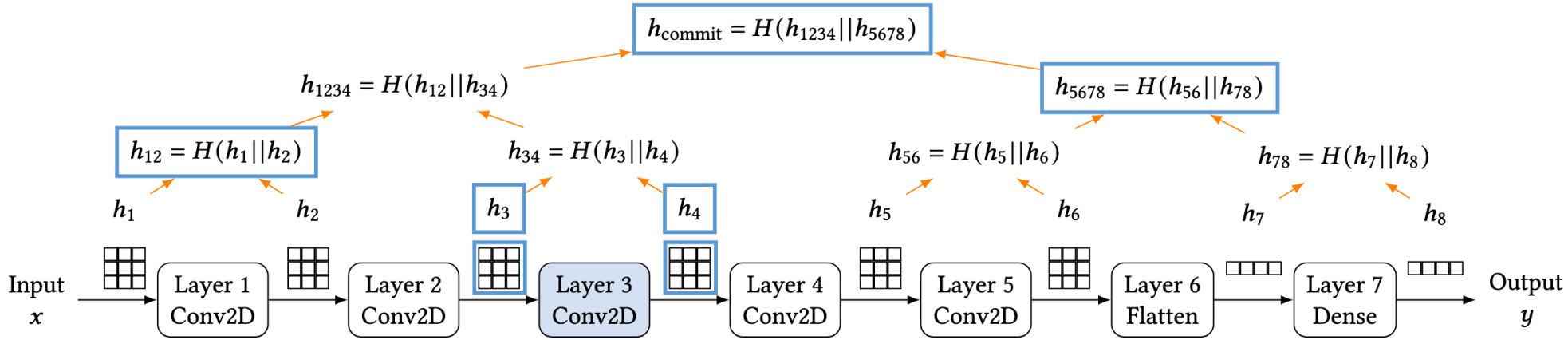


Figure 7: Merkle Tree for Intermediate Layer Commitment

However, this may suffer from floating-point inconsistencies due to the nature of the computations involved.

TOPLOC

Locality Sensitive Hashing

TOPLOC: Prover

TOPLOC [6] is a novel method of *committing* to an LLM-inference. Like Veri-Split [1], it wants to commit to inner computations of the transformer.

During inference, the prover creates a commitment to the **last hidden state** every few (32) tokens. The hidden state has dimensions `(batch_size, seq_length, hidden_size)`, and for a single sequence (`batch_size = 1`) we have a tensor of shape `(seq_length, hidden_size)`.

TOPLOC: Prover

TOPLOC [6] is a novel method of *committing* to an LLM-inference. Like Veri-Split [1], it wants to commit to inner computations of the transformer.

During inference, the prover creates a commitment to the **last hidden state** every few (32) tokens. The hidden state has dimensions `(batch_size, seq_length, hidden_size)`, and for a single sequence (`batch_size = 1`) we have a tensor of shape `(seq_length, hidden_size)`.

- `seq_length` is the number of tokens in the input sequence
- `hidden_size` is the size of the hidden state vector for each token. (can be large, e.g. GPT-3 has dimension 12288)

TOPLOC works over the flattened hidden state, **committing** to the top-k values in it.

TOPLOC: Prover

TOPLOC uses a polynomial interpolation over the top-k values this array.

- The *values* are floating-point (FP16), so they are "treated as" 2-byte integers.
- The *indices* are positions of these values, which may be larger than 2-bytes (65536); to map them to unique 2-byte values TOPLOC uses an *injective modulus* m

TOPLOC: Prover

TOPLOC uses a polynomial interpolation over the top-k values this array.

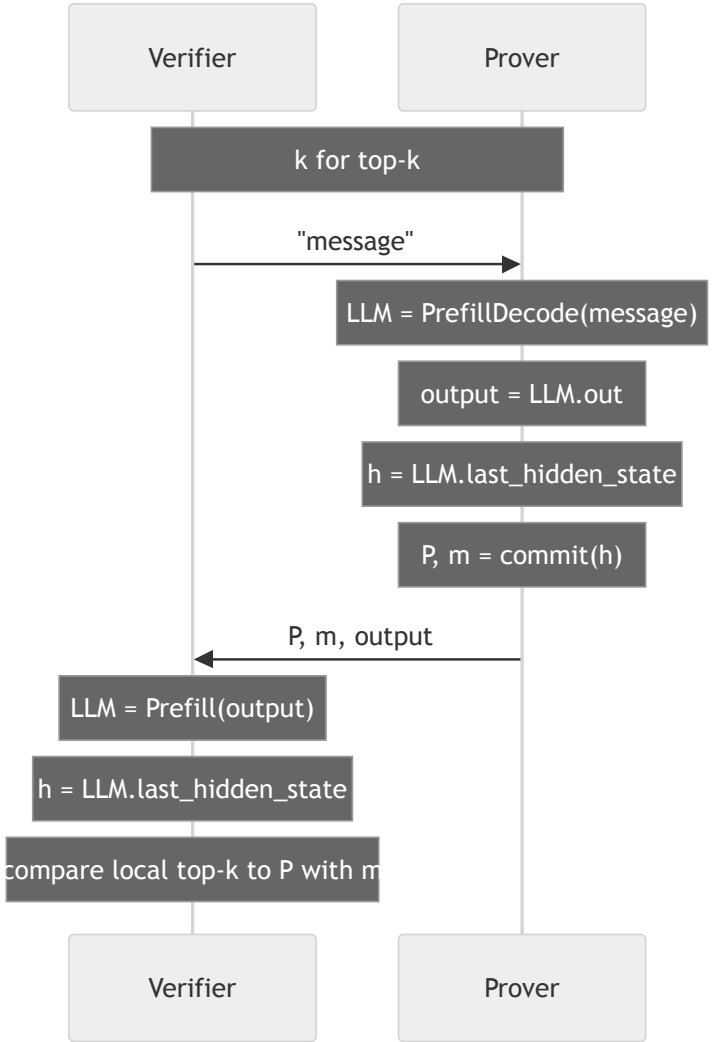
- The *values* are floating-point (FP16), so they are "treated as" 2-byte integers.
- The *indices* are positions of these values, which may be larger than 2-bytes (65536); to map them to unique 2-byte values TOPLOC uses an *injective modulus* m

In the end, we have 2-byte indices and values, which we interpolate the polynomial over. The prover sends the coefficients of this polynomial, along with k and injective modulus m.

TOPLOC: Verifier

The verifier receives the output sequence along with the proof (coefficients of the polynomial, k, and m). It does a single **prefill** with this sequence to compute the hidden state locally, and extracts the top-k values from it.

Then, it evaluates the polynomial at the indices of these top-k values, and checks if the results *approximately* match the actual values for verification.



TOPLOC

TOPLOC is vulnerable against **speculative decoding** [7]: using a smaller & more efficient model for *decoding* phase (to make guesses), and then using the actual model for the *prefill* over the generated tokens (to make corrections).

TOPLOC

TOPLOC is vulnerable against **speculative decoding** [7]: using a smaller & more efficient model for *decoding* phase (to make guesses), and then using the actual model for the *prefill* over the generated tokens (to make corrections).

It is also susceptible to floating-point errors in some edge cases, causing honest proofs to fail. In another case, a model with FP8 values can be verified as FP16 too.

TOPLOC

TOPLOC is vulnerable against **speculative decoding** [7]: using a smaller & more efficient model for *decoding* phase (to make guesses), and then using the actual model for the *prefill* over the generated tokens (to make corrections).

It is also susceptible to floating-point errors in some edge cases, causing honest proofs to fail. In another case, a model with FP8 values can be verified as FP16 too.

Nevertheless, it is an **efficient** and **reliable** method for model verification!

TOPLOC has a version 2 as well, which brings support for *pipeline parallelism*.

STIP

Secure Transformer Inference Protocol

Permutation Equivariance

STIP (Secure Transformer Inference Protocol) [8] is a novel approach to secure inference in transformer models. The main idea comes from “Permutation Equivariance of Transformers” [9].

Permutation Equivariance

STIP (Secure Transformer Inference Protocol) [8] is a novel approach to secure inference in transformer models. The main idea comes from "Permutation Equivariance of Transformers" [9].

If you represent the transformer model with $y = F(x)$; if you have a permutation π you can actually do $\pi y = \pi F(x) = F(\pi x)$.

Permutation Equivariance

STIP (Secure Transformer Inference Protocol) [8] is a novel approach to secure inference in transformer models. The main idea comes from "Permutation Equivariance of Transformers" [9].

If you represent the transformer model with $y = F(x)$; if you have a permutation π you can actually do $\pi y = \pi F(x) = F(\pi x)$.

- Row-permutations simply change the order of the input tokens, which in turn changes the order of the output tokens in a predictable way.
- Linear operations are not affected by row-permutations, they simply "carry" the permutation to their results.
- Non-linear operations are *element-wise*, so they are also not affected by row-permutations.

Semi-Symmetrical Permutations

The key insight of STIP is to permute matrices **semi-symmetrically**, in such a way that the operations cancel each-other out!

Suppose $y = xA$. We generate 2 permutations: π, π_c and permute the input and the matrix as follows:

- $x' = x\pi$
- $A' = \pi^T A \pi_c$
- $y' = x'A' = x(\pi\pi^T)A\pi_c = xA\pi_c$

The receiver of y' can re-permute to obtain $y = (x'A')\pi_c^T$

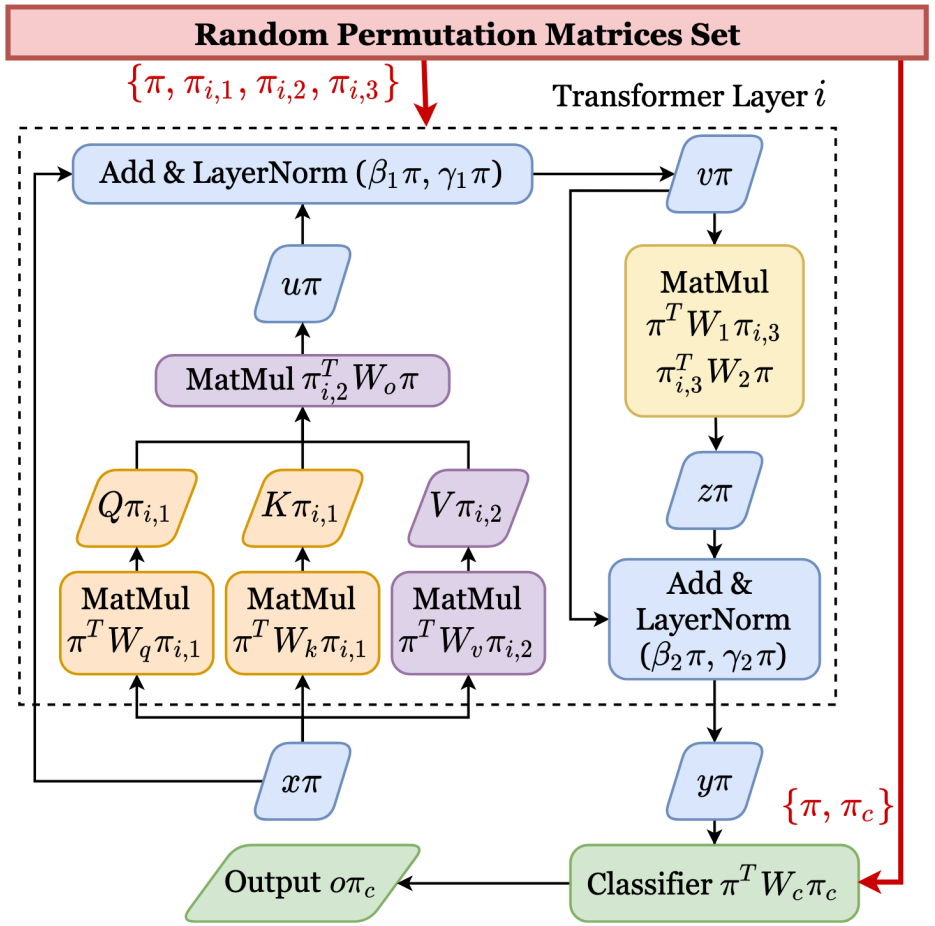


Figure 9: STIP Permutations

Semi-Symmetrical Permutations

STIP generates permutations π, π_c for input and output, and $\pi_{i,1}, \pi_{i,2}, \pi_{i,3}$ for each layer i.

Then, they permute all layers of the LLM with different semi-symmetrical permutations. The user receives π, π_c and the model server receives permuted weights.

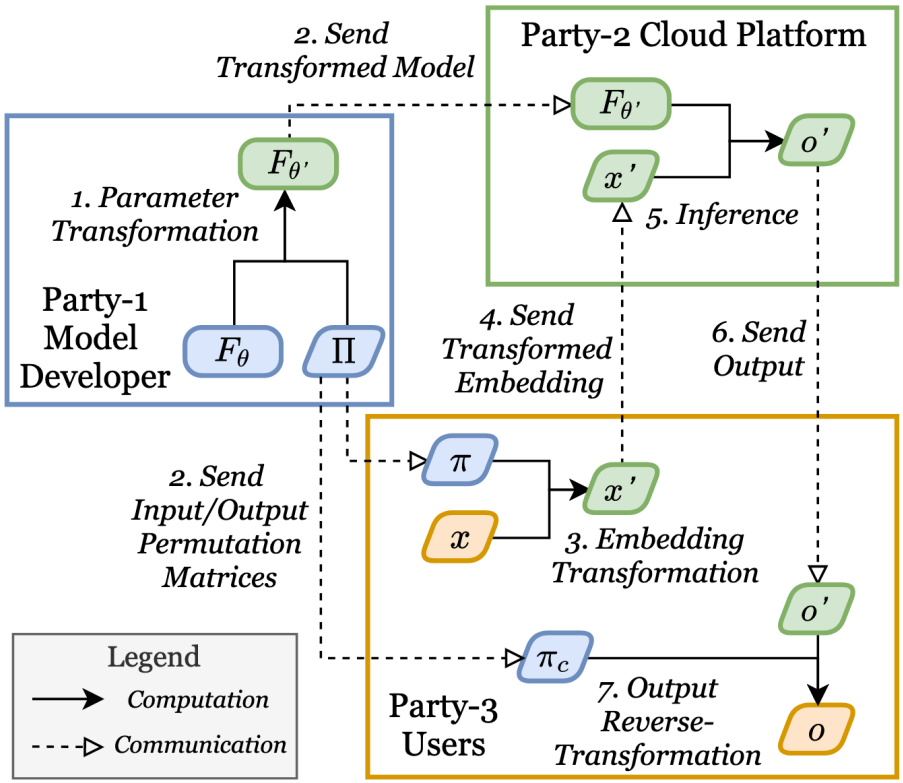


Figure 10: STIP Overview

Too Good To Be True?

There are some attacks, such as user-become-server cases; but they can be solved with efficient re-permutations[1] and one-more-than-once permutations per model for multiple users and servers.

[1] to re-permute a permuted matrix $M\pi$ with π', just send $\pi^T\pi'$ to it!

Too Good To Be True?

There are some attacks, such as user-become-server cases; but they can be solved with efficient re-permutations[1] and one-more-than-once permutations per model for multiple users and servers.

Sadly, a recent paper [10] has shown how to break the privacy guarantees of STIP, by pointing out an error in their statistical approach. They are able to break the permutation to recover the tokens as if they were unpermuted.

[1] to re-permute a permuted matrix $M\pi$ with π', just send $\pi^T\pi'$ to it!

Too Good To Be True?

There are some attacks, such as user-become-server cases; but they can be solved with efficient re-permutations[1] and one-more-than-once permutations per model for multiple users and servers.

Sadly, a recent paper [10] has shown how to break the privacy guarantees of STIP, by pointing out an error in their statistical approach. They are able to break the permutation to recover the tokens as if they were unpermuted.

Nevertheless, they show that with addition of **noise** to the equation, one can still achieve privacy at the cost of accuracy using these permutations.

It's important to note that another paper [11] shows that privacy-preserving LLM inference **MUST** come with some kind of accuracy loss, i.e. "No Free Lunch".

[1] to re-permute a permuted matrix $M\pi$ with π', just send $\pi^T\pi'$ to it!

Future

What Next?

Future

- It is obvious that ZK / FHE / TEE require some forms of breakthrough to make it into every-day usage.

Future

- It is obvious that ZK / FHE / TEE require some forms of breakthrough to make it into every-day usage.
- However, we can come up with methods that are specifically tailored for LLM inference & transformers, and make use of the properties of the models themselves.
- Permutation based methods with noise & per-token hidden state commitments are promising, and they are **compatible** with each other; so maybe we can get privacy preserving & verifiable inference in the end.

Future

- It is obvious that ZK / FHE / TEE require some forms of breakthrough to make it into every-day usage.
- However, we can come up with methods that are specifically tailored for LLM inference & transformers, and make use of the properties of the models themselves.
- Permutation based methods with noise & per-token hidden state commitments are promising, and they are **compatible** with each other; so maybe we can get privacy preserving & verifiable inference in the end.
- Even further, these methods are *pipeline-parallelism* safe, so multiple devices can work together to create these proofs for their "portions" of the whole LLM.

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] X. Li, B. Zhao, G. Yang, T. Xiang, J. Weng, and R. H. Deng, “A Survey of Secure Computation Using Trusted Execution Environments.” [Online]. Available: https://arxiv.org/abs/2302.12150

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

[5] H. Kydlíček, “Math-Verify: Math Verification Library.” [Online]. Available: https://github.com/huggingface/math-verify

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

[7] Y. Leviathan, M. Kalman, and Y. Matias, “Fast Inference from Transformers via Speculative Decoding.” [Online]. Available: https://arxiv.org/abs/2211.17192

[8] M. Yuan, L. Zhang, and X.-Y. Li, “Secure Transformer Inference Protocol.” [Online]. Available: https://arxiv.org/abs/2312.00025

[9] 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

[10] R. Thomas, L. Zahran, E. Choi, A. Potti, M. Goldblum, and A. Pal, “An Attack to Break Permutation-Based Private Third-Party Inference Schemes for LLMs.” [Online]. Available: https://arxiv.org/abs/2505.18332

[11] X. Zhang *et al.*, “No Free Lunch Theorem for Privacy-Preserving LLM Inference.” [Online]. Available: https://arxiv.org/abs/2405.20681

Q & A

slides — github.com/erhant/cryptist-2025-inference

reach out — x.com/0xerhant