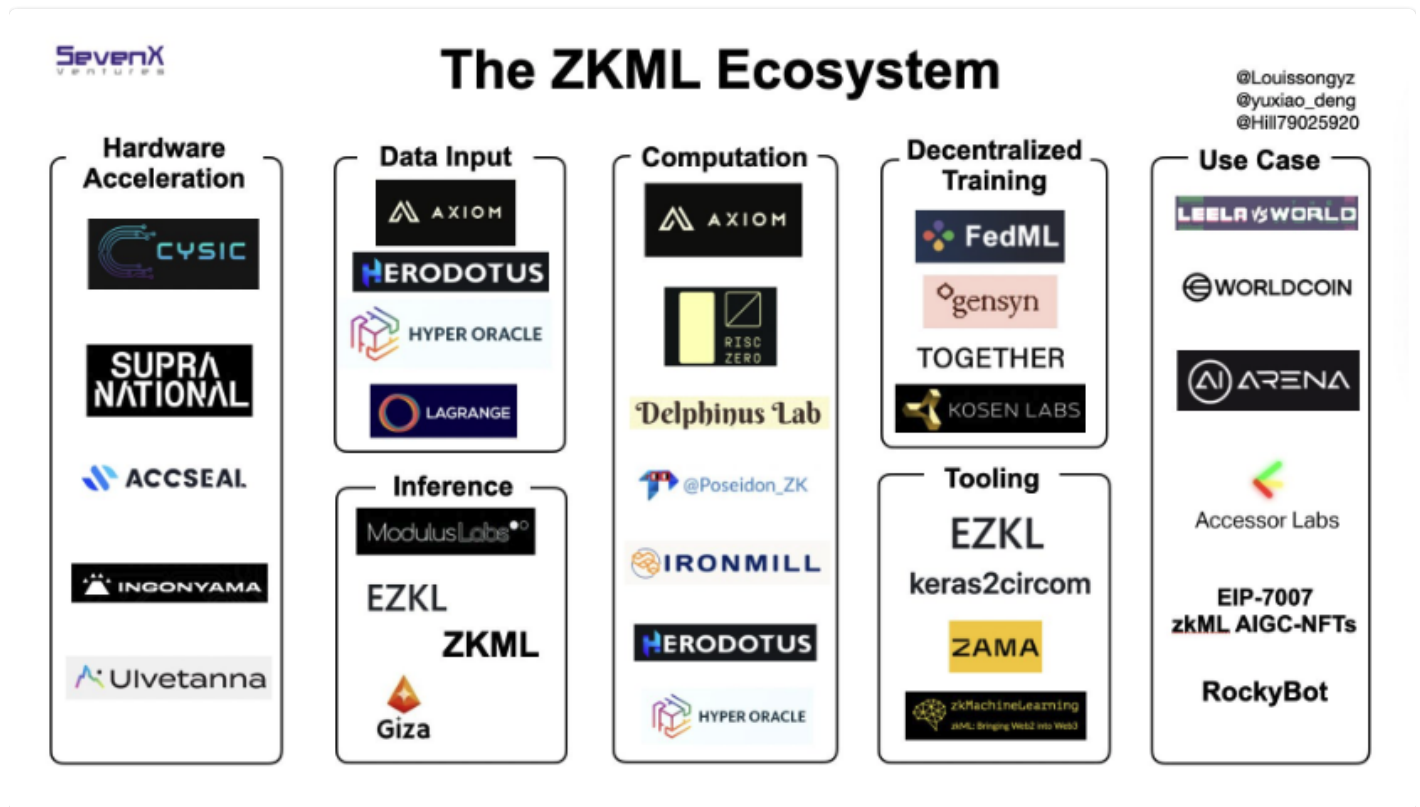


Lesson 10

Today's topics

- Axiom
- ERC-7007: Verifiable AI-Generated Content Token
- Hardware for zkML
- Netron
- Taceo
- Modulus - GPT2

Ecosystem



From : D. Kriesel – A Brief Introduction to Neural Networks (ZETA2-EN)

"Experiments showed that a human can recognize the picture of a familiar object or person in ≈ 0.1 seconds, which corresponds to a neuron switching time of $\approx 10^{-3}$ seconds in ≈ 100 discrete time steps of parallel processing.

A computer following the von Neumann architecture, however, can do practically nothing in 100 time steps of sequential processing, which are 100 assembler steps or cycle steps."

Axiom

The ZK Coprocessor for Ethereum.



See [Demo](#)

Axiom is a ZK coprocessor designed for Ethereum. It allows smart contracts to access on-chain data in a trustless manner and perform various computations on that data. Developers

can submit queries to Axiom and utilise the ZK-verified results directly in their smart contracts.

Axiom operates in three steps:

1. Read: Axiom employs ZK proofs to securely retrieve data from block headers, states, transactions, and receipts in past Ethereum blocks. Since all on-chain data is stored in one of these forms, Axiom can access any information available to archive nodes.
2. Compute: Once the data is obtained, Axiom applies verified compute operations on top of it. These operations range from basic analytics like sum, count, max, and min, to cryptographic tasks such as signature verification and key aggregation, as well as machine learning algorithms like decision trees, linear regression, and neural network inference. Each compute operation's validity is confirmed through a ZK proof.
3. Verify: Axiom provides a ZK validity proof with the result of each query, ensuring two things:
(1) the input data was accurately fetched

from the chain, and

(2) the compute operations were correctly executed.

This ZK proof is then verified on-chain within the Axiom smart contract, making the final result securely available for use by other smart contracts downstream.

Use cases - Trustless on-chain async calls

On-chain applications can delegate data-rich computations to the Axiom ZK coprocessor.

This allows smart contracts to perform more complex on-chain actions without compromising on trust. A few ideas in this direction are:

- Fully trustless ML-based parameter adjustments for DeFi protocols. Axiom allows zkML systems to trustlessly use historic on-chain data as inputs, thereby allowing smart contracts to adjust their parameters based on the outcome of machine learning models.
- A batch auction-based decentralized exchange where off-chain solvers compute order matchings and prove their optimality.

Axiom empowers solvers to prove the validity of on-chain data accesses entering into their matchings.

Verifiable AI-Generated Content Token

See [proposal](#)

Abstract

The verifiable AI-generated content (AIGC) non-fungible token (NFT) standard is an extension of the [ERC-721](#) token standard for AIGC.

It proposes a set of interfaces for basic interactions and enumerable interactions for AIGC-NFTs.

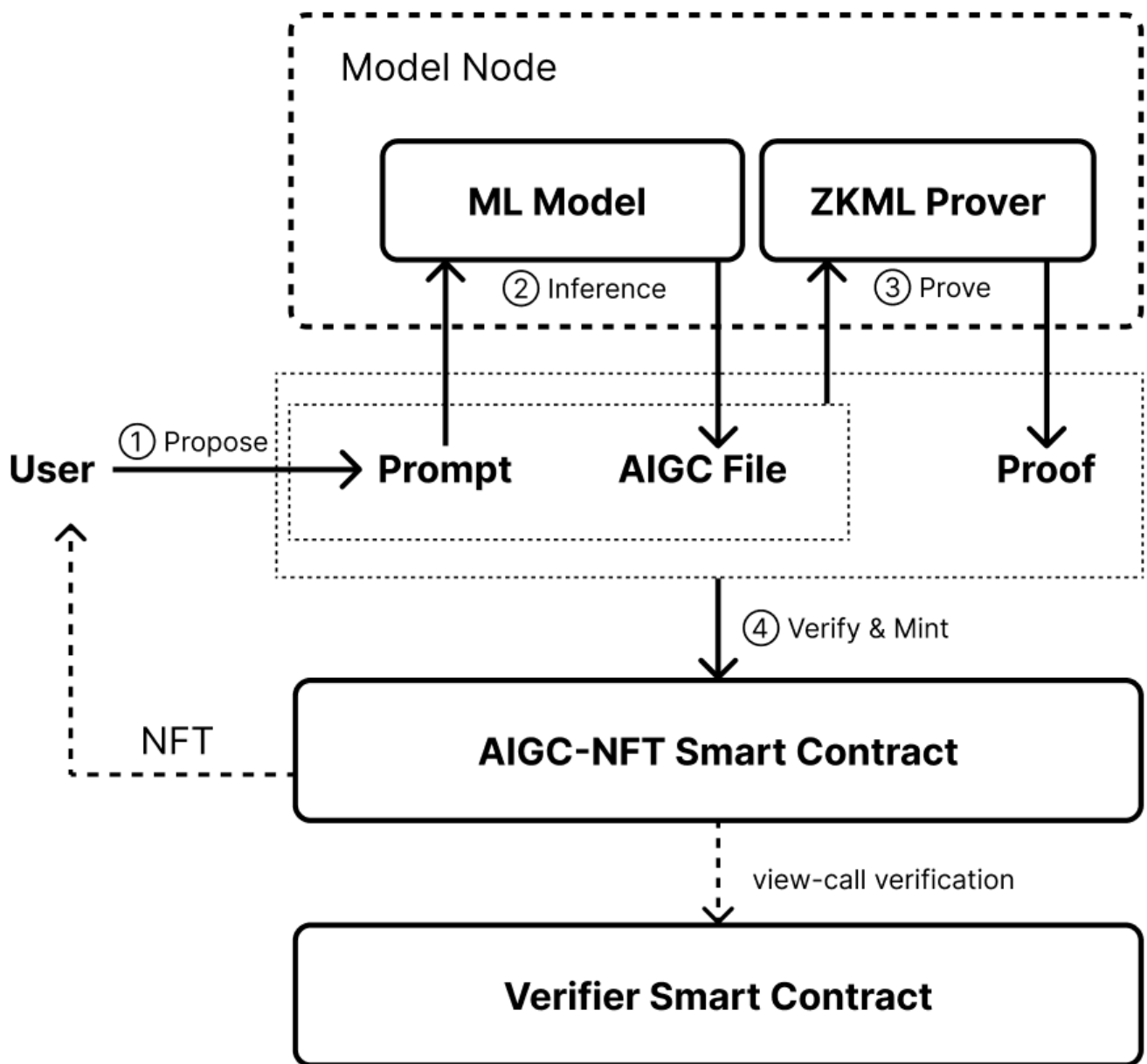
The standard includes

a `mint` and `verify` function interface, a new `Mint` event,

optional `Enumerable` and `Updatable` extensions, and a JSON schema for AIGC-NFT metadata.

Additionally, it incorporates Zero-Knowledge Machine Learning (zkML) and Optimistic Machine Learning (opML) capabilities to enable verification of AIGC data correctness.

In this standard, the `tokenId` is indexed by the `prompt`.



- ML model - contains weights of a pre-trained model; given an inference input, generates the output
- zkML prover - given an inference task with input and output, generates a ZK proof
- AIGC-NFT smart contract - contract compliant with this proposal, with full [ERC-721](#) functionalities

- Verifier smart contract - implements a `verify` function, given an inference task and its ZK proof, returns the verification result as a boolean

`IERC7007`: Defines a `mint` function and a `Mint` event for minting AIGC-NFTs. Defines a `verify` function to check the validity of the combination of prompt and aigcData using zkML/opML techniques.

```
pragma solidity ^0.8.18;

/**
 * @dev Required interface of an ERC7007
 * compliant contract.
 * Note: the ERC-165 identifier for this
 * interface is 0x7e52e423.
 */
interface IERC7007 is IERC165, IERC721 {
    /**
     * @dev Emitted when `tokenId` token
     * is minted.
     */
    event Mint(
```



```
        address indexed to,  
        uint256 indexed tokenId,  
        bytes indexed prompt,  
        bytes aigcData,  
        string uri,  
        bytes proof  
    );
```

```
/**
```

```
    * @dev Mint token at `tokenId` given  
    `to`, `prompt`, `aigcData`, `uri`, and  
    `proof`. `proof` means that we input the  
    ZK proof when using zkML and byte zero  
    when using opML as the verification  
    method.
```

```
    *
```

```
    * Requirements:
```

```
    * - `tokenId` must not exist.'
```

```
    * - verify(`prompt`, `aigcData`,  
    `proof`) must return true.
```

```
    *
```

```
    * Optional:
```

```
    * - `proof` should not include  
    `aigcData` to save gas.
```

```

    */
function mint(
    address to,
    bytes calldata prompt,
    bytes calldata aigcData,
    string calldata uri,
    bytes calldata proof
) external returns (uint256 tokenId);

/**
 * @dev Verify the `prompt`,
`aigcData`, and `proof`.
 */
function verify(
    bytes calldata prompt,
    bytes calldata aigcData,
    bytes calldata proof
) external view returns (bool
success);
}

```

While this standard does not describe the Machine Learning model publication stage, it is natural and recommended to publish the commitment of the Model to Ethereum

separately, before any actual `mint` actions. The model commitment schema choice lies on the AIGC-NFT project issuer party. The commitment should be checked inside the implementation of the `verify` function.

Unique Token Identification

This specification sets the `tokenId` to be the hash of its corresponding `prompt`, creating a deterministic and collision-resistant way to associate tokens with their unique content generation parameters. This design decision ensures that the same prompt (which corresponds to the same AI-generated content under the same model seed) cannot be minted more than once, thereby preventing duplication and preserving the uniqueness of each NFT within the ecosystem.

Generalization to Different Proof Types

This specification accommodates two proof types: validity proofs for zkML and fraud proofs for opML. Function arguments in `mint` and `verify` are designed for generality, allowing for compatibility with both proof systems. Moreover, the specification includes

an updatable extension that specifically serves the requirements of opML.

`verify` interface

We specify a `verify` interface to enforce the correctness of `aigcData`. It is defined as a view function to reduce gas cost. `verify` should return true if and only if `aigcData` is finalized in both zkML and opML. In zkML, it must verify the ZK proof, i.e. `proof`; in opML, it must make sure that the challenging period is finalized, and that the `aigcData` is up-to-date, i.e. has been updated after finalization.

Additionally, `proof` can be *empty* in opML.

`mint` interface

We specify a `mint` interface to bind the prompt and `aigcData` with `tokenId`. Notably, it acts differently in zkML and opML cases. In zkML, `mint` should make sure `verify` returns `true`. While in opML, it can be called before finalization. The consideration here is that, limited by the proving difficulty, zkML usually targets simple model inference tasks in practice, making it possible to provide a proof within an acceptable time frame. On the

other hand, opML enables large model inference tasks, with a cost of longer confirmation time to achieve the approximate same security level. Mint until opML finalization may not be the best practice considering the existing optimistic protocols.

Reference Implementation

See [Gist](#)




Reference

Taken from the specification from Cathie So ([@socathie](#)), Xiaohang Yu ([@xhyumiracle](#)), Conway ([@0x1cc](#)), Lee Ting Ting ([@tina1998612](#)), Kartin [kartin@hyperoracle.io], [<mailto:kartin@hyperoracle.io>], "ERC-7007: Verifiable AI-Generated Content Token [DRAFT]," *Ethereum Improvement Proposals*, no. 7007, May 2023. [Online serial]. Available: <https://eips.ethereum.org/EIPS/eip-7007>.

Expensive operations

From Ingonyama [video](#)

	Arithmetic	Compute	Input interface	Output interface	Constant memory	Dataflow complexity
MSM	EC	$O(n)$	High	Low	High (setup)	Med
NTT	FF	$O(n \log n)$	High	High	High (twiddle)	High
Hash	FF	$O(n^2)$	High	Low	High	Med

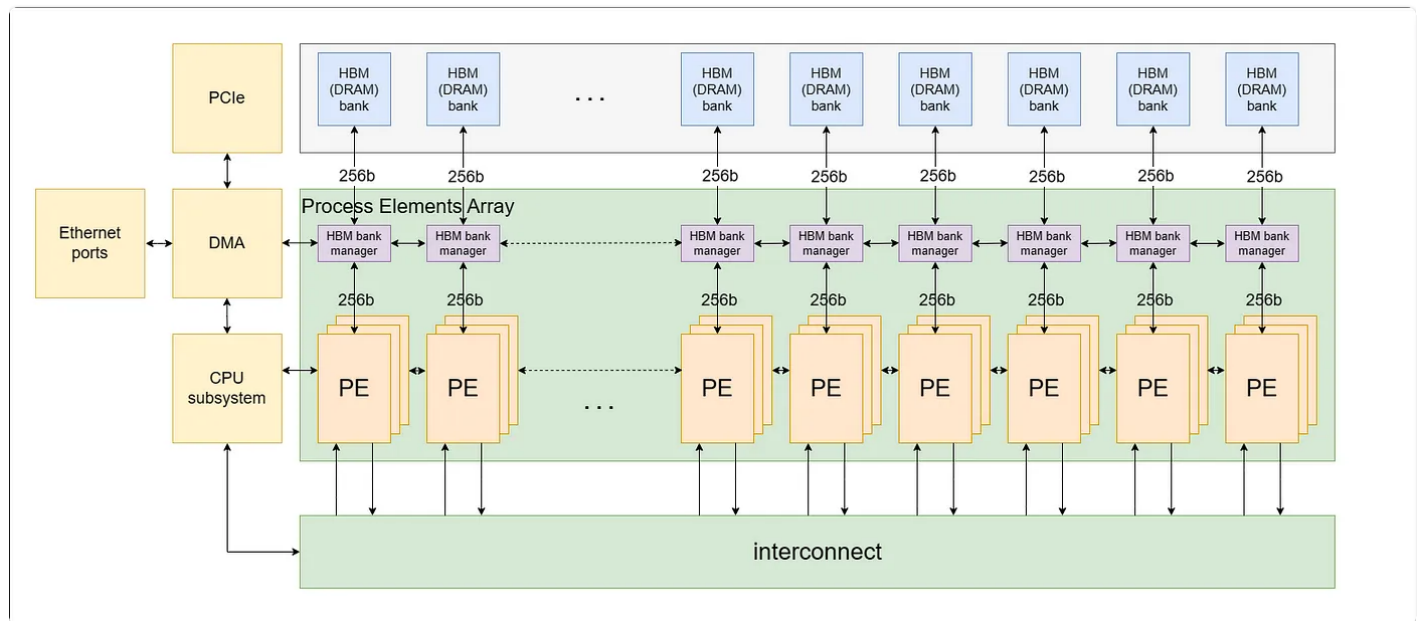
	CPU 	GPU 	FPGA 
Thread parallelism	Low	High	High
Scheduling	OS	RTOS	Deterministic
Memory scheme	Cache-based	Mixed	Deterministic
Power/security	High	High	High
Arithmetic	FP64	FP32	Fixed point
Programming	Easy	Med	Hard



ZPU Architecture

See [article](#)

The ZPU architecture features an interconnected network of Processing Elements (PEs), defined by an Instruction Set Architecture (ISA).



Each PE is designed with a core that includes a modulo multiplier, an adder, and a subtractor.

ZPU architecture: what's important?

- Large-integer modular arithmetic supported natively
- CT butterfly supported natively
- Large close-to-compute memories (L1)
- MIMD multiprocessors
- Fast, low-power, dynamic interconnect
- High IO bandwidth
- Large constant memory (nice-to-have)

For benchmarks see [article](#)

Poseidon Merkle Trees in Hardware

See [Article](#)

Many ZKPs in production now, however, use alternative hash functions designed for recursive ZKP verification, such as Poseidon, which is ~5x slower than Keccak on modern CPUs.

Arithmetization-friendly hash functions require substantial computational resources. For example, [FPGA implementations of Keccak](#) can be attained using a few thousand FPGA slices, orders of magnitude smaller than Poseidon.

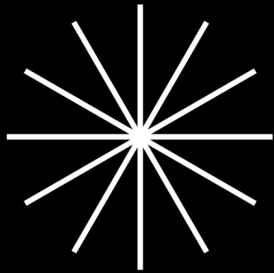
The design in the article improves on this with a 1.53x speedup compared to a CPU implementation when running the Polygon zkEVM benchmark.

The same team have created the Binius SNARK to take advantage of hardware acceleration.

See [article](#) and [zkpodcast episode](#)

Supranational

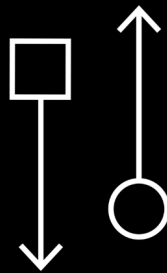
See [Site](#)



BLST

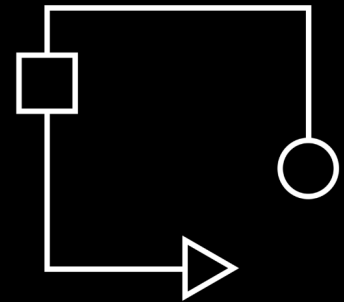
BLST is an IETF-compliant BLS12-381 signature library focused on security and performance.

Available [here](#).



'Proofs-as-a-Service'

A simple API for generating VDF and SNARK proofs. Powered by fast, open source implementations running on a high-availability cloud.



Crypto Accelerator

An arbitrary-precision arithmetic accelerator for cryptographic operations including VDFs, SNARKs, polynomial commitments, accumulators, and more.

Under development.

System Verilog implementation of the [MinRoot](#) VDF using the [Pasta Curves](#). The intended usage is within an ASIC to be developed for the Ethereum 2 and Filecoin protocols.

See [Repo](#)

Netron

See [Repo](#)

Example ONNX format [model](#)

Taceo zkML

From [Blog](#)

"There are a variety of great projects out there that try to link between the ZK world and Machine Learning (ML)."

"... it remains impractical to rebuild an NN from scratch inside a circuit, up to almost impossible, when looking at GPT-like models."

"On the one hand, we at TACEO plan to leverage the LLVM compiler toolchain to produce circuits from pre-trained, existing NN, eliminating the need to rewrite models inside a SNARK (or your preferred general-purpose ZKP).

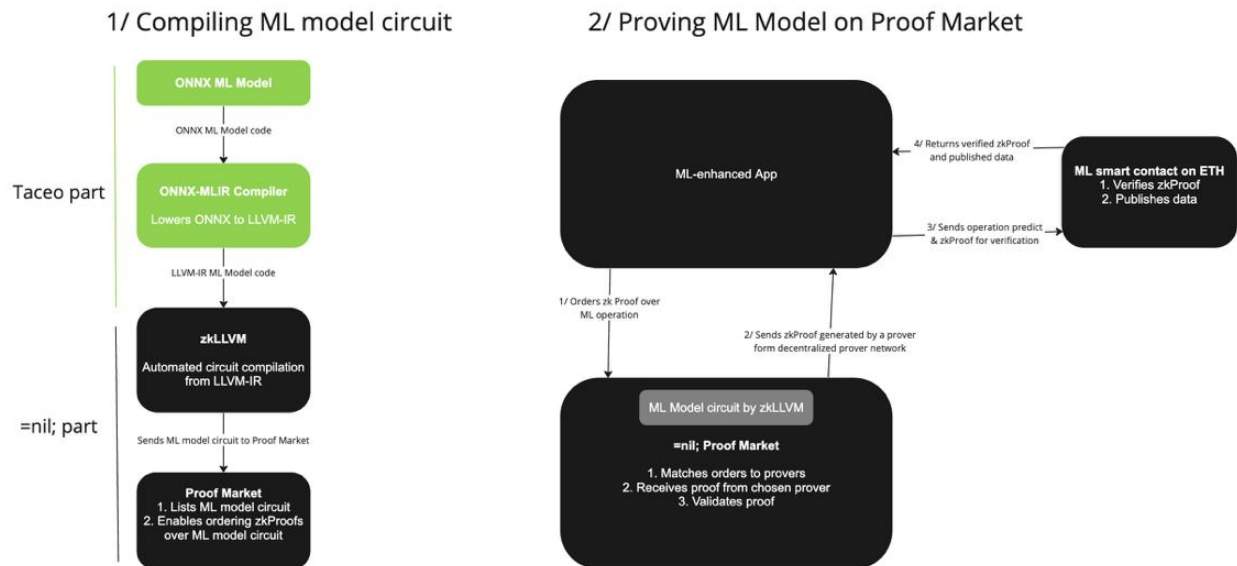
On the other hand, the =nil; team is focused on achieving efficient, accessible, and automated circuit compilation from mainstream language input (C++/Rust) through zkLLVM using the Placeholder proof system."

zkLLVM

zkLLVM is an LLVM-based circuit compiler capable of proving computations in whatever mainstream development languages LLVM supports: C++, Rust, JavaScript/TypeScript, and

other languages. Considering that the circuit is just another form of a program, zkLLVM directly transforms the code into a different representation.

zk ML



ONNX-MLIR Framework

MLIR is an extensible compiler infrastructure that allows the creation of domain-specific compilers, among many other use cases.

ONNX-MLIR is a compiler written with the MLIR compiler framework that can lower ONNX models to LLVM-IR. And if we can lower it to LLVM-IR, we can use zkLLVM!

To summarize, starting with a neural network in ONNX format, like GPT-2, we will use a tweaked version of ONNX-MLIR to apply the already

defined optimization and lowering passes of ONNX-MLIR. But instead of producing LLVM-IR, we will lower it to zkLLVM, which will allow us to compile circuits directly from ONNX! ML developers then can build circuits from their existing models without any cryptographic background.

Tutorial

See [Tutorial article](#)

The tutorial goes through the process of compiling an ONNX model into an intermediate representation, generating a circuit and an extended witness from the intermediate representation, and creating a proof from the circuit and the extended witness.

Proof Market

A collaboration between Taceo and
=nil;Foundation

Proof Market as the place where proof requesters and proof producers meet and create a free, open, and self-sustaining market.

[Demo](#)

There are three primary roles (parties) in the Proof Market:

- **Proof requesters** are applications that require zero-knowledge proofs, and make requests for them on the Proof Market.
- **Proof producers** are owners of computational infrastructure, who generate proofs for the requests made by proof requesters.
- **Circuit developers** make zero-knowledge circuits, that are used to generate requests and subsequent proofs.

The Proof Market toolchain has tools for proof requesters and producers. To learn more about

the Proof Market and these roles, read the [Proof Market documentation](#).

If you're interested in circuit development, check out the [zkLLVM compiler](#) and [zkLLVM template project](#).

If you wish to provide proofs, a [toolchain](#) is available.

On chain LLM

See [article](#)



On March 13th, Modulus Labs finally completed the ZK proving of the [full 1.5 billion parameter GPT2-XL](#).

This is seen particularly as a milestone as it has passed the billion parameter level.

"GPT2-XL is built with a relatively straightforward architecture, consisting of just

48 uniformly-sized decoder blocks which feed sequentially into one another (see [this](#) wonderful visualization!), making it easy to circuitize"

To achieve this, they broke the decoder blocks in the model into sub blocks that were easier to prove.

They used Halo2 as a backend and recursively aggregated the correct SNARK verifications, compressing the results for final on-chain settlement.

The results

- The total time for generating the proving key was 327,916 seconds — over **91 hours** when run on a single machine with 128 core CPUs and 1TB RAM
- These 144 proving keys occupied a disk space over **10TB**
- The total proving time of the 144 sub-blocks was 322,774 seconds — just shy of **90 hours** (when run on the same single machine)

Comparing Modulus versus general frameworks

generic provers

~100,000x overhead

>\$100M spent in
optimizations

Modulus

~100x overhead

<\$1M spent in
optimizations