Lesson 15 - Identity Solutions / zkML / Oracles

Identity Solutions

Polygon ID

See site

Polygon ID has the following properties:

- Blockchain-based ID for decentralised and self-sovereign models
- Zero-knowledge native protocols for ultimate user privacy
- Scalable and private on-chain verification to boost decentralised apps and DeFi
- Open to existing standards and ecosystem development

It uses the Iden3 and Circom toolkit

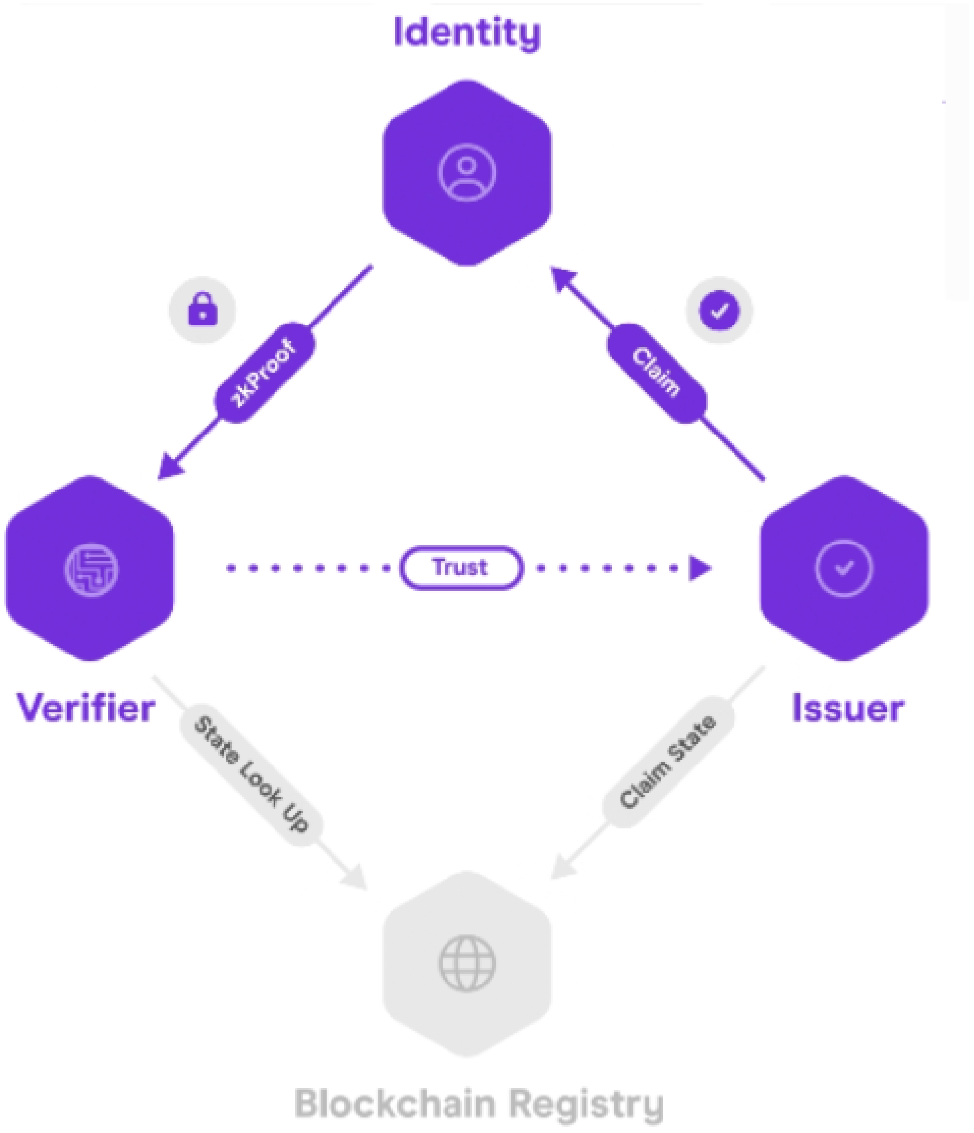
In comparison with NFTs and VCs

NFTs are not private, and have high minting costs.
While VCs offer some degree of privacy with selective disclosure and ZK add-on, their limitations are in the expressibility and composability, which are required for applications. Verifying VCs on-chain is prohibitively expensive.

There is an ID client toolkit to facilitate onboarding
On chain verification uses zkProof Request Language, which allows applications to specify which requested private attributes a user needs to prove.

Roles in Polygon ID

1. Identity Holder: An entity that holds claims in its Wallet. A claim is issued by an Issuer to the Holder. The Identity holder creates the zero-knowledge proofs of the claims issued and presents these proofs to the Verifier (which verifies the correctness and authenticity of the claim). A Holder is also called Prover as it needs to prove to the Verifier that the credential it holds is authentic and matches specific criteria.
2. Issuer: An entity (person, organisation, or thing) that issues claims to the Holders. Claims are cryptographically signed by the Issuer. Every claim comes from an Issuer.
3. Verifier: A Verifier verifies the claims presented by a Holder. It requests the Holder to send proof of the claim issued from an Issuer and on receiving the zero-knowledge proofs from the Holder, verifies it. The verification process includes checking the veracity of the signature of the Issuer. The simplest real-world examples of a Verifies can be a recruiter that verifies your educational background or a voting platform that verifies your age.


Identity
zkProof
Claim
Trust
Verifier
Issuer
State Look Up
Claim State
Blockchain Registry

Semaphore

Documentation
Repo
Semaphore is a zero-knowledge protocol that allows you to cast a signal (for example, a vote or endorsement) as a provable group member without revealing your identity.
Use cases include private voting, whistleblowing, anonymous DAOs and mixers.

"Encode/ZKPCourse/LatestCourse/img/Screenshot 2022-08-18 at 14.57.43.png" is not created yet. Click to create.

Circuit

The Semaphore circuit is the heart of the protocol and consists of three parts:

- Proof of membership
- Nullifier hash
- Signal

They provide tools to create and verify proofs.

Setup

The semaphore CLI will set up a hardhat project

```
npx @semaphore-protocol/cli@latest create my-app
```

Example contract

WorldID

World ID is a digital passport that lets a user prove they are a unique and real person while remaining anonymous.
This happens through zero knowledge proofs and other privacy-preserving cryptographic mechanisms.

It works with a project as follows

1. The user gets their World ID in a compatible wallet
2. The user receives credentials in their World ID. The flagship credential is biometric verification, currently available by using the Orb. The user can also verify their phone number to obtain the respective credential.
3. A Project integrates with WorldID
4. The user connects their World ID to authenticate, and optionally prove they are a unique human doing something only once. The user's wallet will generate a zero knowledge proof to accomplish this.
5. The project verifies the proof either by using the API or by verifying it on-chain.

Clique

[See] (https://clique.social/)

Clique builds identity-oracles for web2 user behaviour data. We provide legacy data pipelines for web3 protocols, so that they can incentivise and engage users on existing web2 platforms.

Clique uses the standardized Twitter and Discord O-Auth tokens to get public information about an account.
None of the private account information is revealed to the protocols.

Axiom

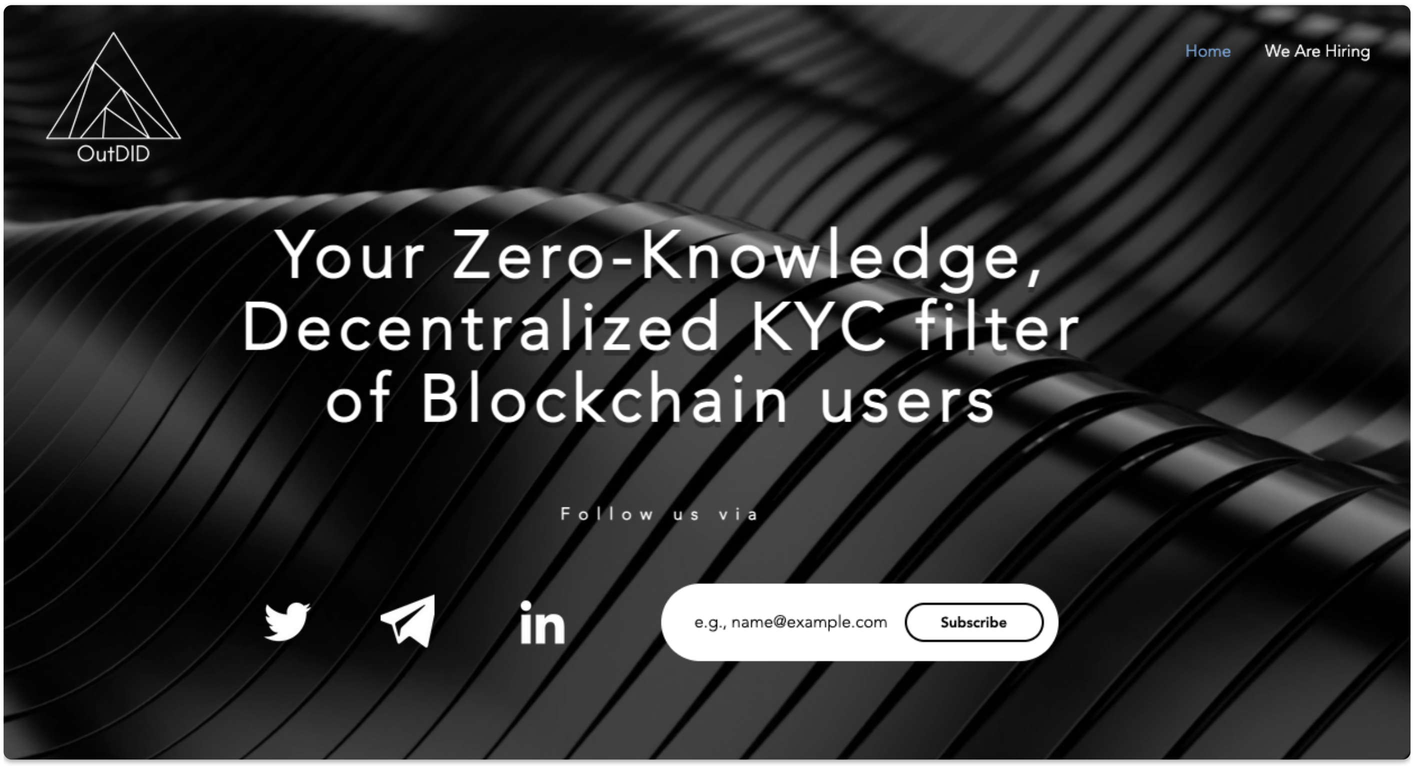
See Demo
Axiom is a ZK coprocessor designed for Ethereum. It allows smart contracts to access on-chain data in a trustless manner and perform various computations on that data. Developers can submit queries to Axiom and utilise the ZK-verified results directly in their smart contracts.

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.

OutDID.io

See docs



Providing KYC and Identity using Circom with for example passport data.

zCloak Network

zCloak Network provides Zero-Knowledge Proof as a Service based on the Polkadot Network

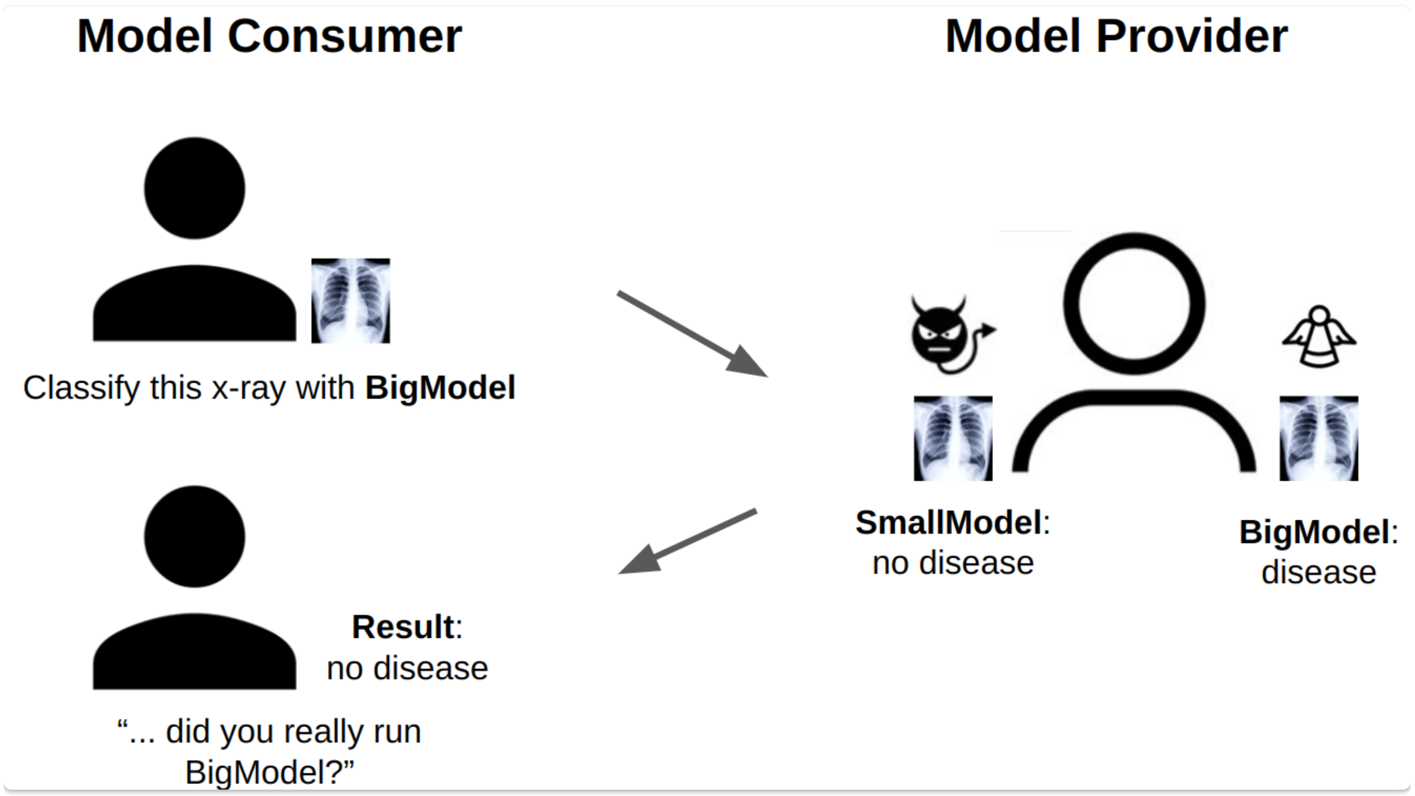
In the 'Cloaking Space' you control your own data and you can run all sorts of computation without sending your data away.
Note that the data stored in the Cloaking Space is not just some arbitrary data on your device, but they are attested by some credible network/organization to garantee its authenticity.
The type of computation can range from

- a regular state transition of a blockchain,
- a check of your income for a bank loan to
- an examination of your facial features to pass an airport checkpoint.

zCloak uses the Distaff VM
Distaff is a zero-knowledge virtual machine written in Rust.
For any program executed on Distaff VM, a STARK-based proof of execution is automatically generated. This proof can then be used by anyone to verify that a program was executed correctly.

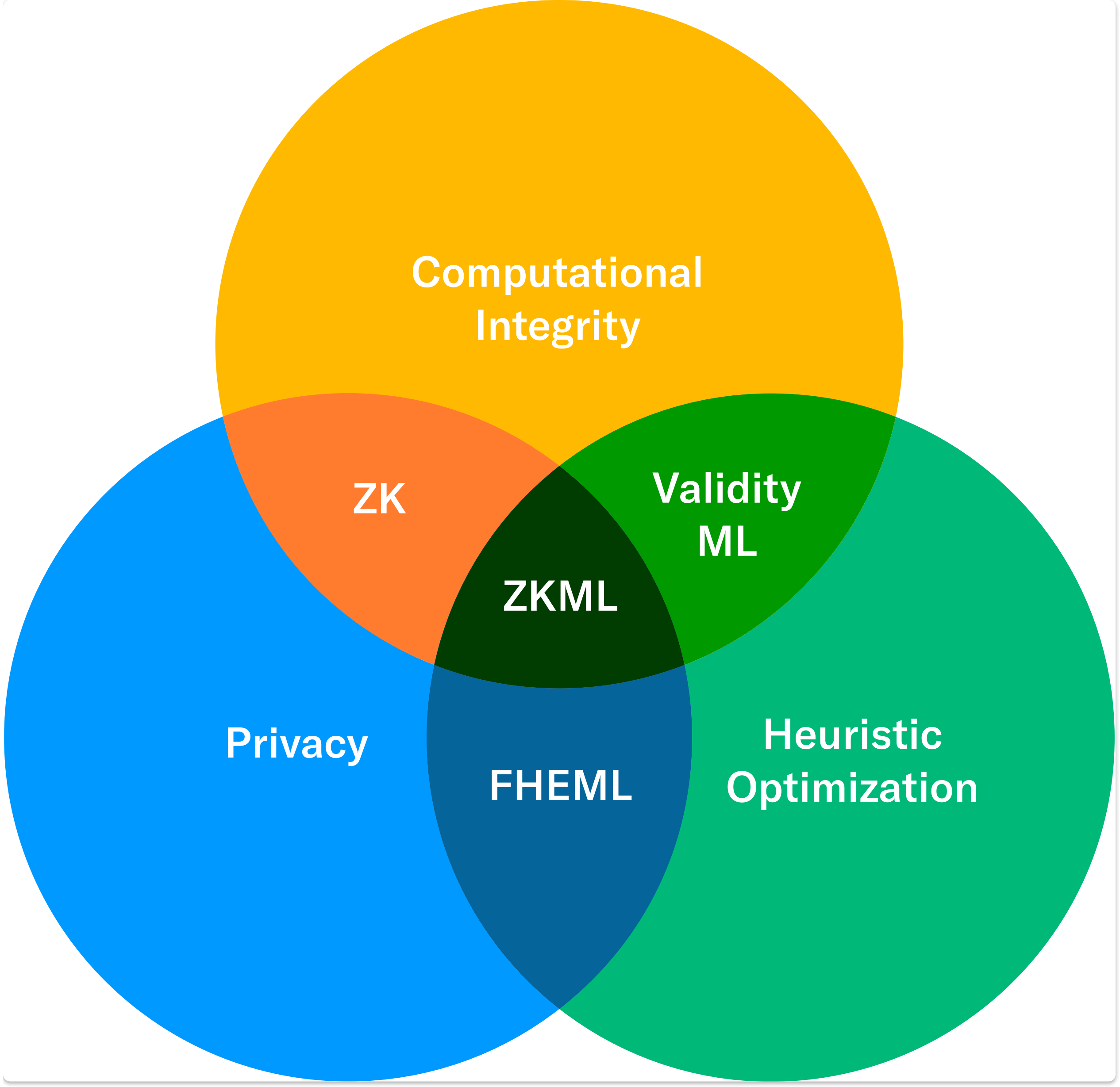
ZK-ML

Example use case



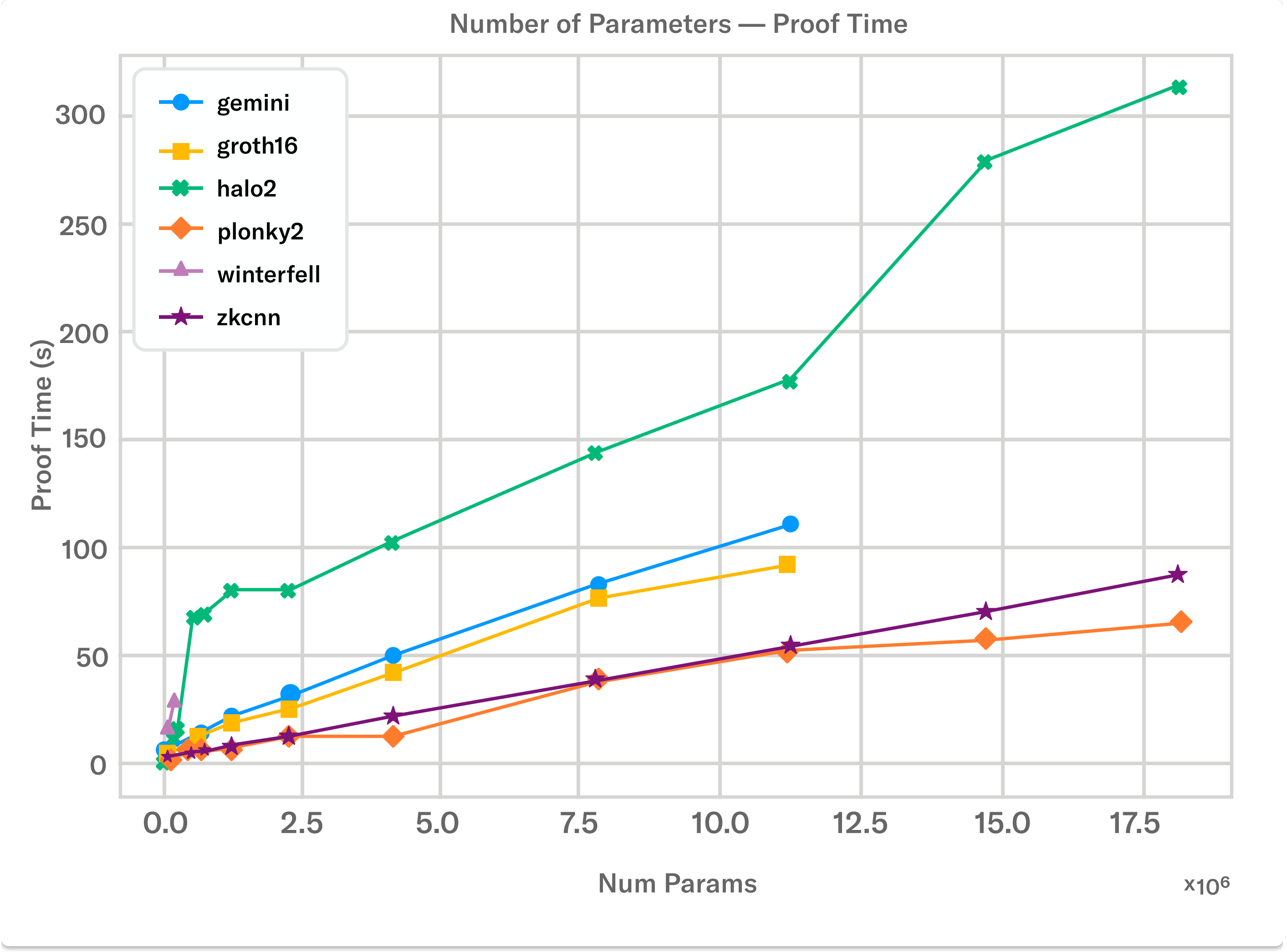
Model Consumer
Model Provider
Classify this x-ray with **BigModel**
SmallModel: no disease
BigModel: disease
Result: no disease
"... did you really run BigModel?"

From introduction blog



The current state of zero-knowledge systems, even with high-performance hardware, is insufficient to handle large language models. But progress is being made with smaller models.

The Modulus Labs team recently released a paper titled “The Cost of Intelligence”, where they benchmark existing ZK proof systems against a wide range of models of different sizes. It is currently possible to create proofs for models of around 18M parameters in about 50 seconds running on a powerful AWS machine using a proving system like plonky2. Figure 1 illustrates the scaling behaviour of different proving systems as the number of parameters of a neural network are increased:"



Modulus Labs

Modulus Labs are working on a number of use cases :

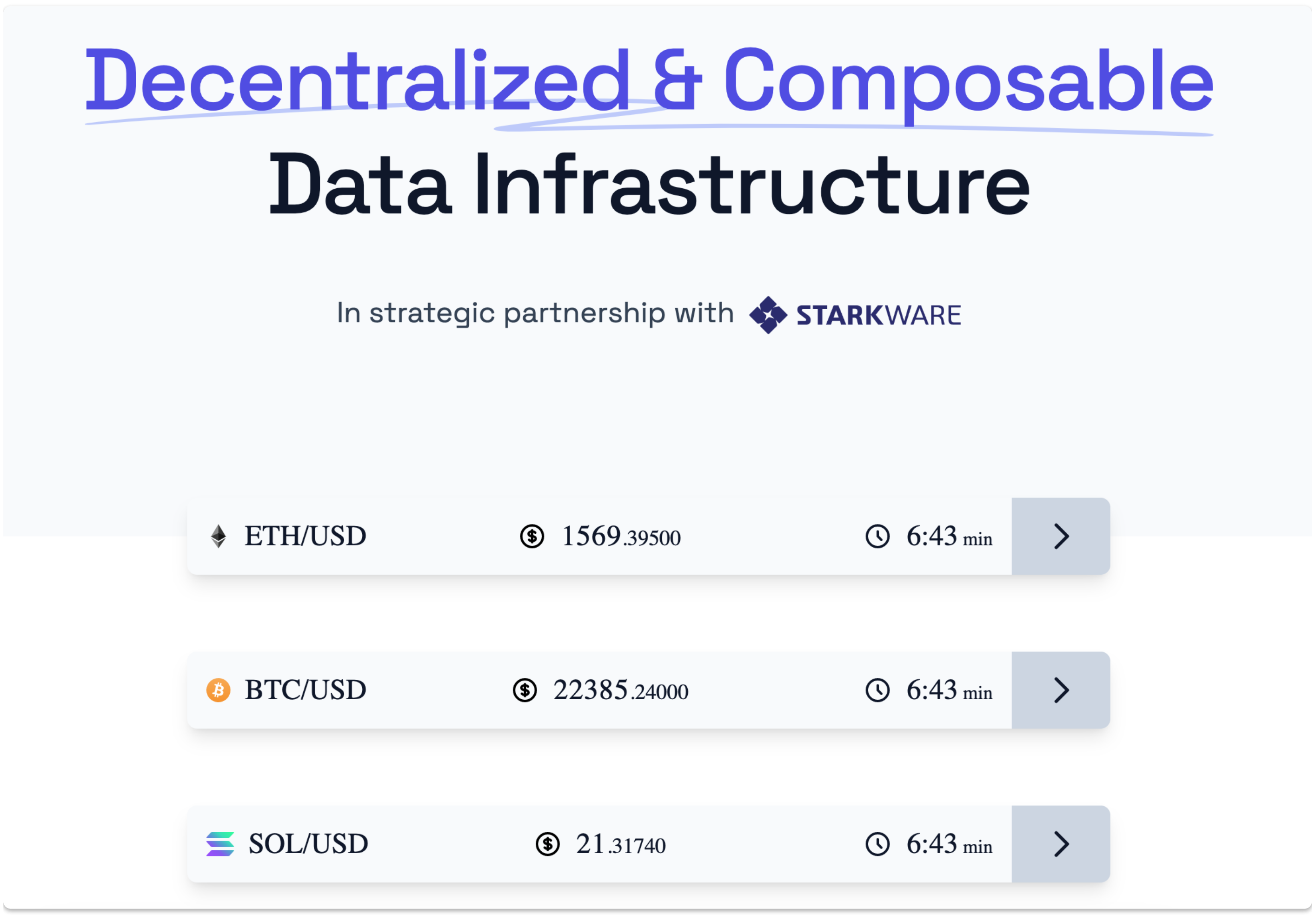
- On-chain verifiable ML trading bot - RockyBot
- Blockchains that self-improve vision
- Enhancing the Lyra finance options protocol AMM with intelligent features
- Creating a transparent AI-based reputation system for Astraly (ZK oracle)
- Working on the technical breakthroughs needed for contract-level compliance tools using ML for Aztec Protocol (a zk-rollup with privacy features)

Resources

dcbuilder has many resources
Good Introduction
Zero Knowledge Podcast
Trustless verification
Image redacting using zkSNARKS paper
ZK ML community [calls](- ZKML community call #0) on telegram

Oracle Solutions

Empiric


Decentralized & Composable
Data Infrastructure

In strategic partnership with STARKWARE

ETH/USD	1569.39500	6:43 min
BTC/USD	22385.24000	6:43 min
SOL/USD	21.31740	6:43 min

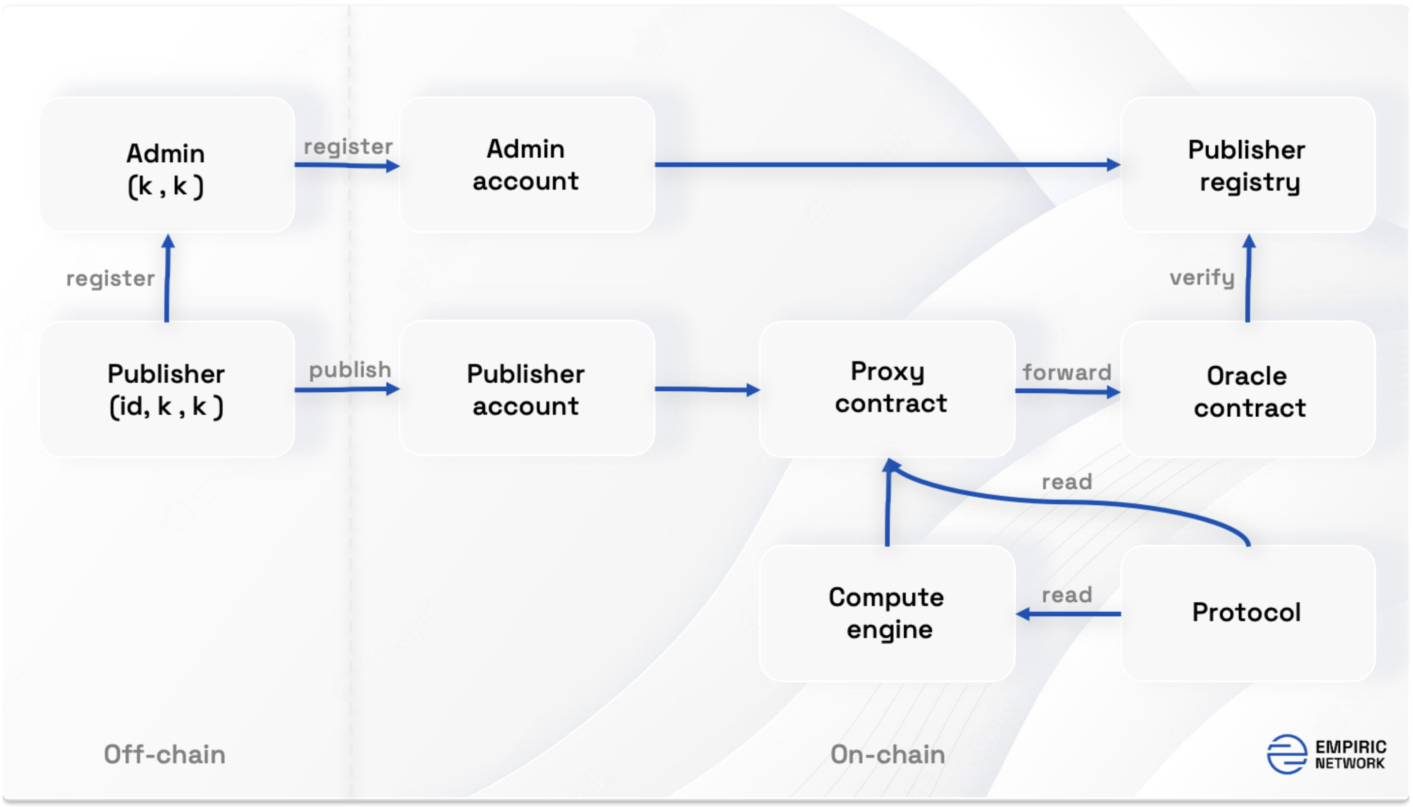
Empiric is available on Starknet and coming soon to Consensys zkEVM and already integrated with leading protocols such as ZKLend, Magnety, Serity, CurveZero, Canvas, and FujiDAO.

Assets supported on Starknet

- BTC/USD,
- BTC/EUR,
- ETH/USD,
- ETH/MXN
- SOL/USD,
- AVAX/USD,
- DOGE/USD,
- SHIB/USD,
- BNB/USD,
- ADA/USD,
- XRP/USD,
- MATIC/USD,
- USDT/USD,
- DAI/USD,

- USDC/USD,
- TUSD/USD,
- BUSD/USD,

Contracts



Code snippet

```
 %lang starknet

from starkware.cairo.common.cairo_builtins import HashBuiltin

# Oracle Interface Definition
const EMPIRIC_ORACLE_ADDRESS =
0x012fadd18ec1a23a160cc46981400160fbf4a7a5eed156c4669e39807265bcd4
const KEY = 28556963469423460  # str_to_felt("eth/usd")
const AGGREGATION_MODE = 120282243752302  # str_to_felt("median")

@contract_interface
namespace IEmpiricOracle:
    func get_value(key : felt, aggregation_mode : felt) -> (
        value : felt,
        decimals : felt,
        last_updated_timestamp : felt,
        num_sources_aggregated : felt
    ):
    end
end
```

```
# Your function
@view
func my_func{
    syscall_ptr : felt*,
    pedersen_ptr : HashBuiltin*,
    range_check_ptr
}() -> ():
    let (eth_price,
        decimals,
        last_updated_timestamp,
        num_sources_aggregated) = IEmpiricOracle.get_value(
            EMPIRIC_ORACLE_ADDRESS, KEY, AGGREGATION_MODE
        )
    # Your smart contract logic!
    return ()
end
```

Empiric SDK

Installation
`pip install empiric-network`

Computational Feeds

You can compose and program data with Cairo, in order to get the right computed data for your protocol.
Empiric has designed compute engines that use the same raw market data underlying our price feeds, but calculate different metrics to produce feeds of processed data.
For example

- Realised Volatility see Docs
- Yield Curve , see Docs

DECO

DECO

A novel privacy-preserving oracle protocol, created by students and faculty at IC3.

Deco

- works with *modern TLS versions*
- requires *no trusted hardware*
- requires *no server-side modifications*

See paper

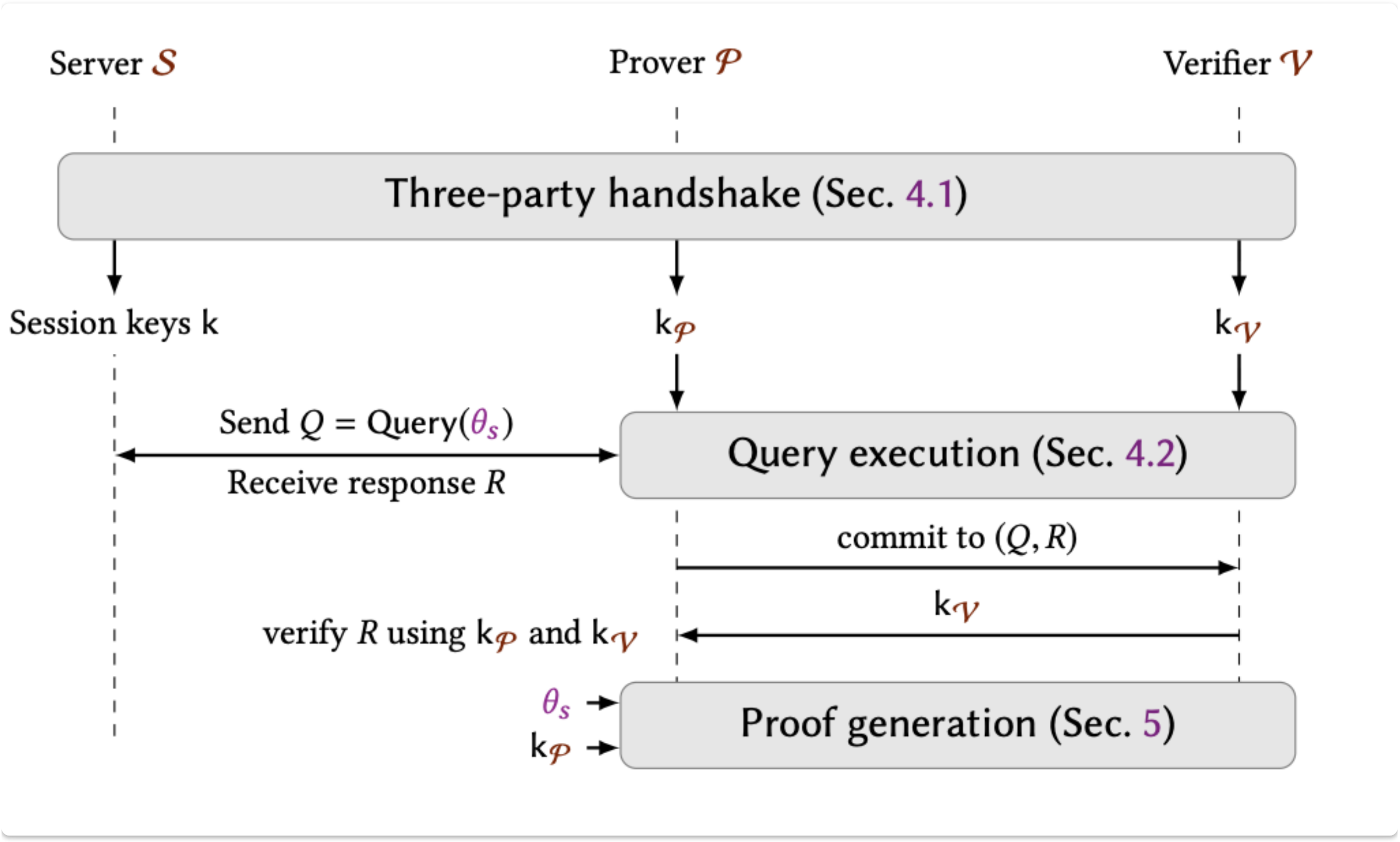
DECO Short for decentralized oracle, DECO is a new cryptographic protocol that enables a user (or oracle) to prove statements in zero knowledge about data obtained from HTTPS-enabled servers. DECO consequently allows private data from unmodified web servers to be relayed safely by oracle networks. (It does not allow data to be sent by a prover directly on chain.)
DECO has narrower capabilities than Town Crier, but unlike Town Crier, does
not rely on a trusted execution environment.

DECO can also be used to power the creation of decentralized identity (DID) protocols such as CanDID, where users can obtain and manage their own credentials, rather than relying on a centralized third party.
Such credentials are signed by entities called issuers that can authoritatively associate claims with users such as citizenship, occupation, college degrees, and more. DECO allows any existing web server to become an issuer and provides key-sharing management to back up accounts, as well as a privacy-preserving form of Sybil resistance based on definitive unique identifiers such as Social Security Numbers (SSNs).

ZKP solutions like DECO benefit not only the users, but also enable traditional institutions and data providers to monetize their proprietary and sensitive datasets in a confidential manner. Instead of posting the data directly on-chain, only attestations derived from ZKPs proving facts about the data need to be published.
This opens up new markets for data providers, who can monetize existing datasets and increase their revenue while ensuring zero data leakage. When combined with Chainlink Mixicles, privacy is extended beyond the input data executing an agreement to also include the terms of the agreement itself.

Process

Diagram from the white paper



Three party handshake

Essentially, P and V jointly act as a TLS client. They negotiate a shared session key with S in a secret shared form

Query Execution

Since the session keys are secret-shared, as noted, P and V execute an interactive protocol to construct a TLS message encrypting the query. P then sends the message to S as a standard TLS client.

P commits to the session data before receiving V's key share, making the commitment unforgeable. Then P can verify the integrity of the response, and prove statements about it.

Proof Generation

With unforgeable commitments, if P opens the commitment to the messages completely (i.e., reveals the encryption key) then V could easily verify the authenticity of the messages by checking MACs on the decryption.
Revealing the encryption key for the messages , however, would breach privacy: it would reveal all session data exchanged between P and S. In theory, P could instead prove any statement about the messages in zero knowledge (i.e., without revealing the encryption key).
Generic zero-knowledge proof techniques, though, would be prohibitively expensive for many natural choices of the statement.

DECO instead introduces two techniques to support efficient proofs for a broad, general class of statement, namely selective opening of a TLS session transcript, see the white paper for details.

A web server itself could assume the role of an oracle, e.g., by simply signing data. However, server-facilitated oracles would not only incur a high adoption cost, but also put users at a disadvantage: the web server could impose arbitrary constraints on the oracle capability.

- Thus a single instance of DECO could enable anyone to become an oracle for any website
- Importantly, DECO does not require trusted hardware, unlike alternative approaches that could achieve a similar vision

DECO end-to-end performance depends on the available TLS ciphersuites, the size of private data, and the complexity of application specific proofs.
It takes about 13.77s to finish the protocol, which includes the time taken to generate unforgeable commitments (0.50s), to run the first stage of two-stage parsing (0.30s), and to generate zero-knowledge proofs (12.97s).

Roadmap

- Chainlink plans to do an initial PoC of DECO, with a focus on decentralized finance applications such as Mixicles.

For more details see their blog

Arkworks

`arkworks` is a Rust ecosystem for zkSNARK programming. Libraries in the `arkworks` ecosystem provide efficient implementations of all components required to implement zkSNARK applications, from generic finite fields to R1CS constraints for common functionalities.

Tutorial includes Exercises for

1. Merkle Tree
2. Validating a single transaction
3. Writing a rollup circuit

Halo2

See tutorial
See documentation
See Halo2 Book

Halo 2 is a proving system that combines the Halo recursion technique with an arithmetisation based on PLONK, and a polynomial commitment scheme based around the Inner Product Argument

Chips

Using our API, we define chips that "know" how to use particular sets of custom gates. This creates an abstraction layer that isolates the implementation of a high-level circuit from the complexity of using custom gates directly.

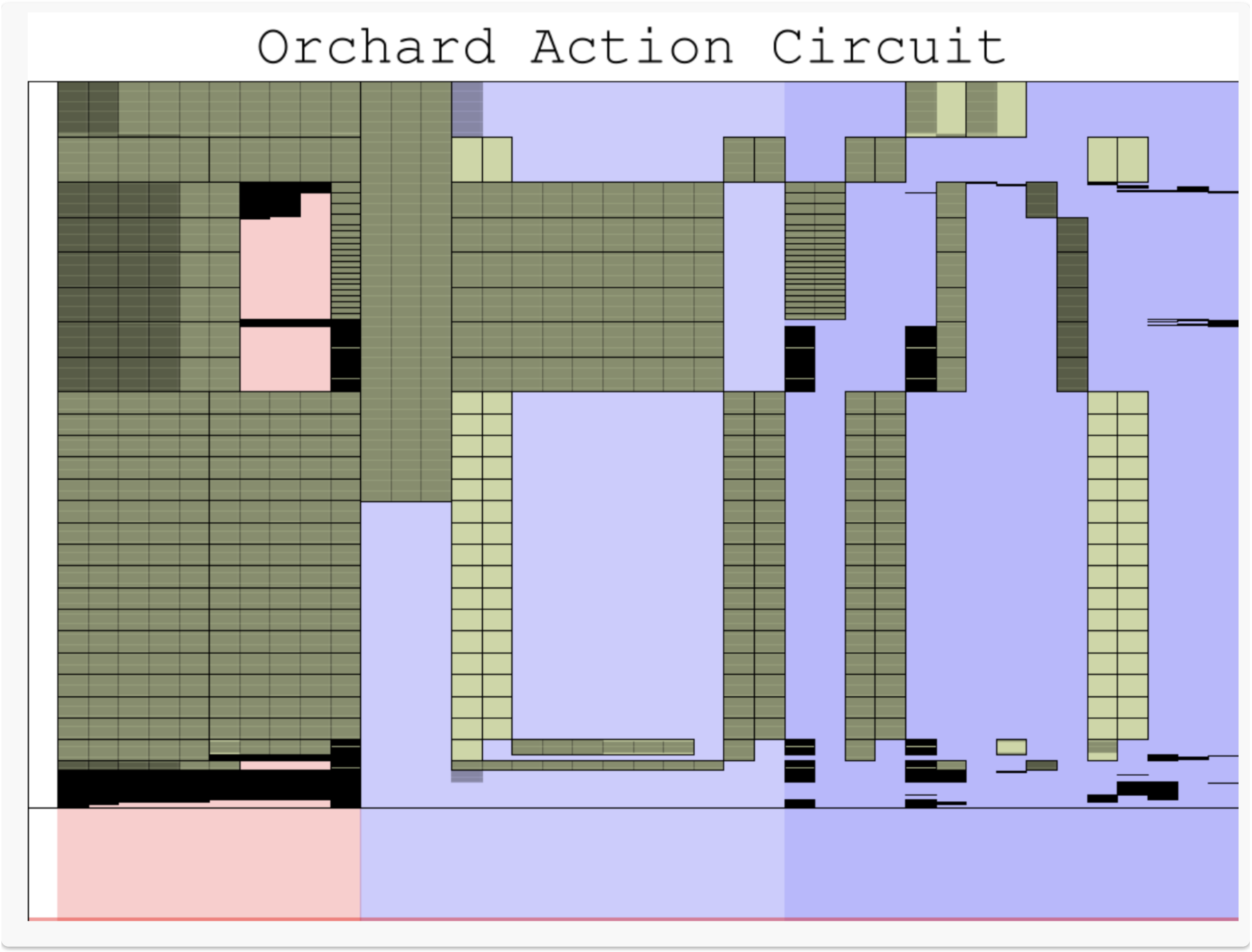
Example Simple Circuit

```rust
trait NumericInstructions<F: Field>: Chip<F> {
    /// Variable representing a number.
    type Num;

    /// Loads a number into the circuit as a private input.
    fn load_private(&self, layouter: impl Layouter<F>, a: Value<F>) ->
Result<Self::Num, Error>;

    /// Loads a number into the circuit as a fixed constant.
    fn load_constant(&self, layouter: impl Layouter<F>, constant: F) ->
Result<Self::Num, Error>;

    /// Returns `c = a * b`.
    fn mul(
        &self,
        layouter: impl Layouter<F>,
        a: Self::Num,
        b: Self::Num,
    ) -> Result<Self::Num, Error>;

    /// Exposes a number as a public input to the circuit.
    fn expose_public(
        &self,
        layouter: impl Layouter<F>,
        num: Self::Num,
        row: usize,
    ) -> Result<(), Error>;
}
```

Halo 2 circuits are two-dimensional: they use a grid of "cells" identified by columns and rows, into which values are assigned.
Constraints on those cells are grouped into "gates", which apply to every row simultaneously,

and can refer to cells at relative rows.
To enable both low-level relative cell references in gates, and high-level layout optimisations, circuit developers can define "regions" in which assigned cells will preserve their relative offsets.

Example from ZCash



In the example circuit layout pictured, the columns are indicated with different backgrounds.
The instance column in white;
advice columns in red;
fixed columns in light blue; and
selector columns in dark blue.
Regions are shown in light green, and assigned cells in dark green or black.

Column Types

- Instance columns contain per-proof public values, that the prover gives to the verifier.
- Advice columns are where the prover assigns private (witness) values, that the verifier learns zero knowledge about.
- Fixed columns contain constants used by every proof that are baked into the circuit.
- Selector columns are special cases of fixed columns that can be used to selectively enable gates.

Attacks against improperly implemented FS

See paper

Example with Schnoor protocol

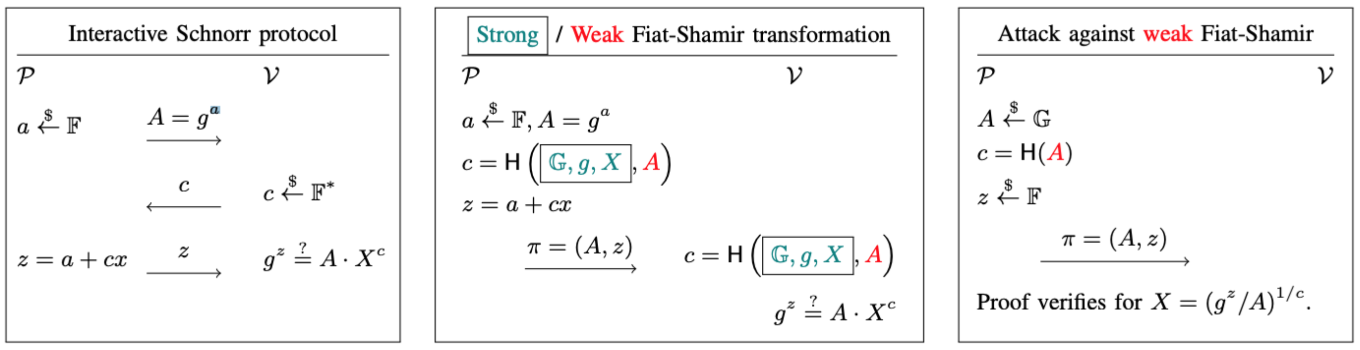
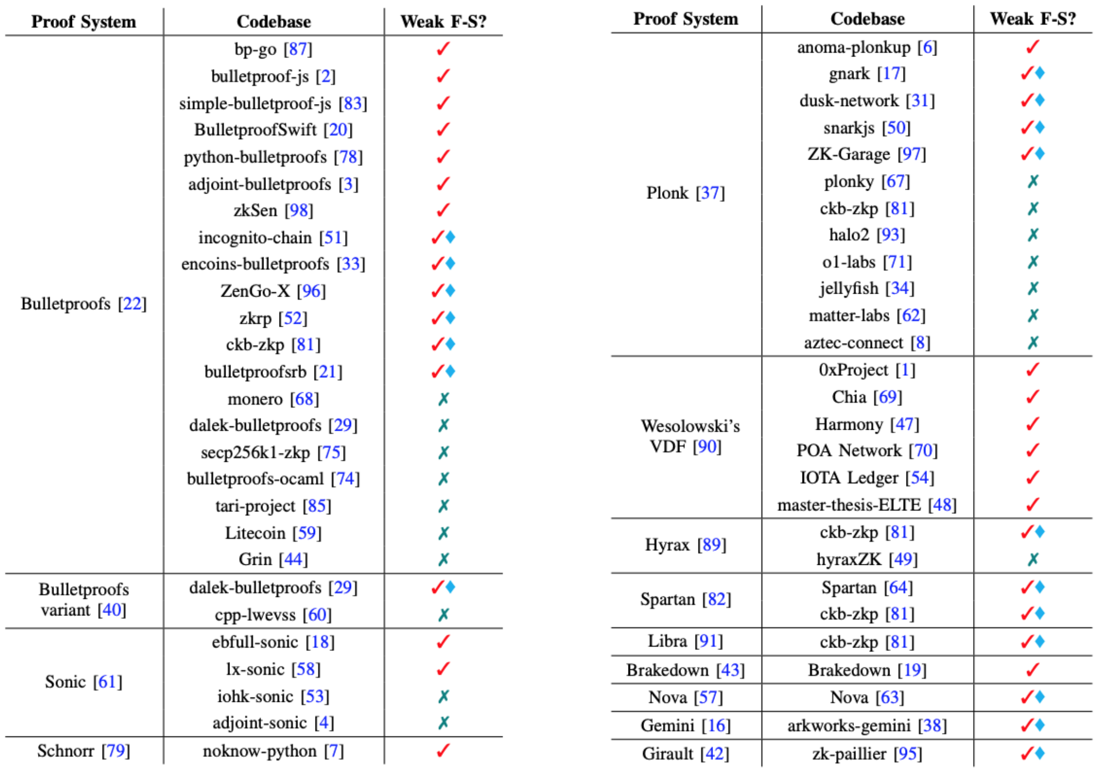

Interactive Schnorr protocol
$\mathcal{P}$
$\mathcal{V}$
$a \xleftarrow{\$} \mathbb{F}$
$A = g^a$
c
$c \xleftarrow{\$} \mathbb{F}^*$
$z = a + cx$
z
$g^z \overset{?}{=} A \cdot X^c$
Strong / Weak Fiat-Shamir transformation
$a \xleftarrow{\$} \mathbb{F}, A = g^a$
$c = \mathsf{H}\left(\boxed{\mathbb{G}, g, X}, A\right)$
$z = a + cx$
$\pi = (A, z)$
$c = \mathsf{H}\left(\boxed{\mathbb{G}, g, X}, A\right)$
$g^z \overset{?}{=} A \cdot X^c$
Attack against weak Fiat-Shamir
$A \xleftarrow{\$} \mathbb{G}$
$c = \mathsf{H}(A)$
$z \xleftarrow{\$} \mathbb{F}$
$\pi = (A, z)$
Proof verifies for $X = (g^z/A)^{1/c}$.

Fig. 1: Example weak Fiat-Shamir attack against Schnorr proofs for relation $\{((\mathbb{G}, g), X; x) \mid X = g^x\}$

Bulletproofs and Plonk can be vulnerable, "using weak F-S leads to attacks on their soundness when the prover can choose the public inputs adaptively, as a function of the proof. Importantly, our results do not invalidate the security proofs for these schemes—when given explicitly, soundness proofs for non-interactive, weak F-S variants of these protocols provide only non-adaptive security"



Proof System	Codebase	Weak F-S?
Bulletproofs [22]	bp-go [87]	✓
	bulletproof-js [2]	✓
	simple-bulletproof-js [83]	✓
	BulletproofSwift [20]	✓
	python-bulletproofs [78]	✓
	adjoint-bulletproofs [3]	✓
	zkSen [98]	✓
	incognito-chain [51]	✓♦
	encoins-bulletproofs [33]	✓♦
	ZenGo-X [96]	✓♦
	zkrp [52]	✓♦
	ckb-zkp [81]	✓♦
	bulletproofsrb [21]	✓♦
	monero [68]	✗
	dalek-bulletproofs [29]	✗
	secp256k1-zkp [75]	✗
	bulletproofs-ocaml [74]	✗
	tari-project [85]	✗
	Litecoin [59]	✗
	Grin [44]	✗
Bulletproofs variant [40]	dalek-bulletproofs [29]	✓♦
	cpp-lwevss [60]	✗
Sonic [61]	ebfull-sonic [18]	✓
	lx-sonic [58]	✓
	iohk-sonic [53]	✗
	adjoint-sonic [4]	✗
Schnorr [79]	noknow-python [7]	✓
Plonk [37]	anoma-plonkup [6]	✓
	gnark [17]	✓♦
	dusk-network [31]	✓♦
	snarkjs [50]	✓♦
	ZK-Garage [97]	✓♦
	plonky [67]	✗
	ckb-zkp [81]	✗
	halo2 [93]	✗
	o1-labs [71]	✗
	jellyfish [34]	✗
	matter-labs [62]	✗
	aztec-connect [8]	✗
Wesolowski's VDF [90]	0xProject [1]	✓
	Chia [69]	✓
	Harmony [47]	✓
	POA Network [70]	✓
	IOTA Ledger [54]	✓
	master-thesis-ELTE [48]	✓
Hyrax [89]	ckb-zkp [81]	✓♦
	hyraxZK [49]	✗
Spartan [82]	Spartan [64]	✓♦
	ckb-zkp [81]	✓♦
Libra [91]	ckb-zkp [81]	✓♦
Brakedown [43]	Brakedown [19]	✓
Nova [57]	Nova [63]	✓♦
Gemini [16]	arkworks-gemini [38]	✓♦
Girault [42]	zk-paillier [95]	✓♦

TABLE I: Implementations surveyed. We include every proof system with at least one vulnerable implementation, and survey all implementations for each one (except classic protocols like Schnorr and Girault). ♦ = has been fixed as of May 15, 2023.