Session 7

Navigator sessions

Date	Topics	
15/12/23	o1js review	
19/1/24	Development workflow, design approaches, techniques and useful patterns	
26/1/24	Recursion	
9/2/24	Application storage solutions	
1/3/24	Utilising decentralisation	
15/3/24	zkOracles and decentralised exchanges	
5/4/24	Ensuring security	
26/4/24	Review Session	

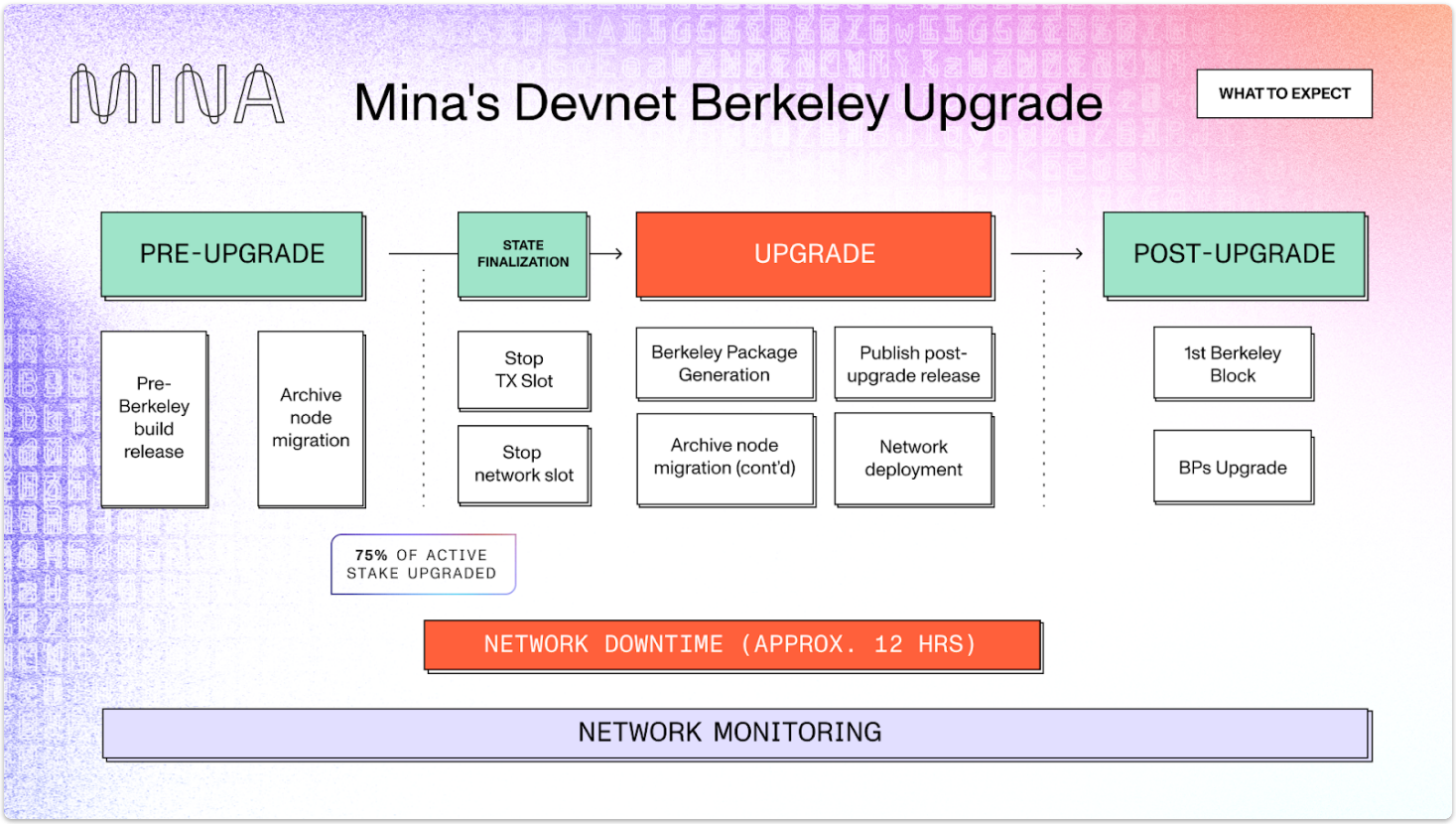
Today's topics

- What's new
- Security
- Identity
- zkML

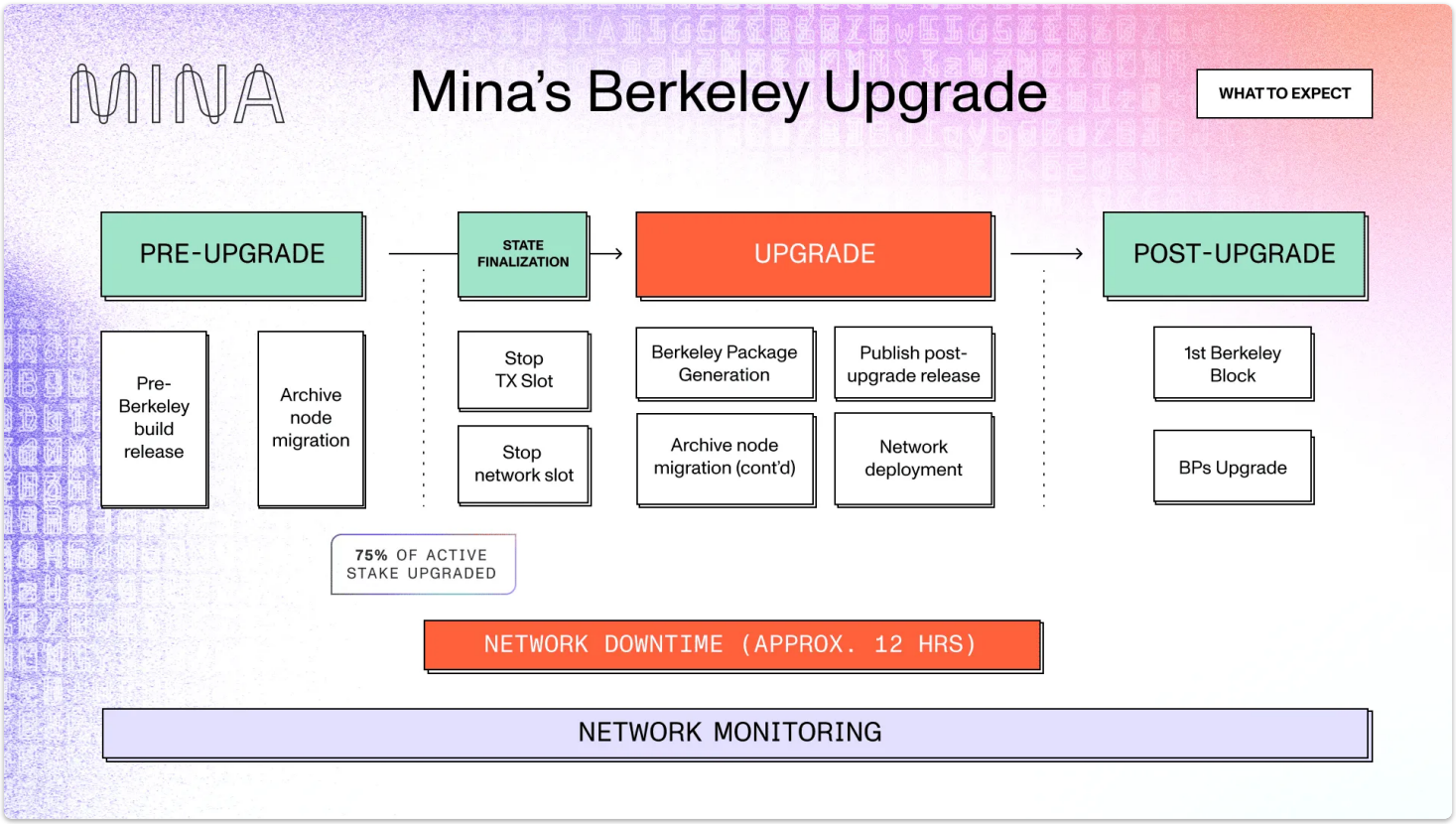
What's new

Devnet upgrade

Scheduled for April 9th



Berkley upgrade



Updates include

- Easier zkApp programmability (MIP 4)
- Kimchi, a more powerful proof system (MIP 3)
- Removal of Supercharged Rewards (MIP 1)

Security and Decentralisation

In Web2 / closed systems, security is often implemented by controlling access to the system
In Web3 / open systems we use

- Cryptography
- Cryptoeconomics
  to keep the systems open to access, but to guide the behaviour of the participants.

Security then becomes the issue of avoiding / preventing "unexpected" behaviour.
Access control is still used, but it is complicated because of decentralised identities / pseudo anonymity

Access control / Open systems

A general architectural consideration is that of access control based on some form of identity as opposed to making a system open.
As developers it is easier to implement access control based on identity, but this can be seen as a centralising / closing design.

Permissions

See Permissions documentation

Permissions are checked each time an account update tries to update state.

Permissions versus Authorisation

Permissions tell us who can execute an action, whereas authorisation defines which resources can be accessed.

Account permissions

See Docs

- `editState`: The permission describing how the zkApp account's eight on-chain state fields are allowed to be manipulated.
- `send`: The permission corresponding to the ability to send transactions from this account. For example, this permission determines whether someone can send a transaction to transfer MINA from this particular account.
- `receive`: Similar to `send`, the `receive` permission determines

whether a particular account can receive transactions, for example, depositing MINA.

- `setDelegate`: The permission corresponding to the ability to set the delegate field of the account. The delegate field is the address of another account that this account is delegating its MINA for staking.
- `setPermissions`: The permission corresponding to the ability to change the permissions of the account. As the name suggests, this type of permission describes how already set permissions can be changed.
- `setVerificationKey`: The permission corresponding to the ability to change the verification key of the account. Every smart contract has a verification key stored on-chain. The verification key is used to verify off-chain proofs. This permission essentially describes if the verification key can be changed; you can also think of it as the "upgradeability" of smart contracts.

- `setZkappUri`: The permission corresponding to the ability to change the `zkappUri` field of the account that stores metadata about the smart contract, for example, link to the source code.
- `editActionsState`: The permission that corresponds to the ability to change the actions state of the associated account. Every smart contract can dispatch actions that are committed on-chain. This type of permission describes who can change the actions state.
- `setTokenSymbol`: The permission corresponding to the ability to set the token symbol for this account.
  The `tokenSymbol` field stores the symbol of a token.
- `incrementNonce`: The permission that determines whether to increment the nonce with an account update and who can increment the nonce on this account with a transaction.
- `setVotingFor`: The permission corresponding to the ability to set the chain

hash for this account. The `votingFor` field is an on-chain mechanism to set the chain hash of the hard fork this account is voting for.

- `access`: This permission is more restrictive than all the other permissions combined! It corresponds to the ability to include any account update for this account in a transaction, even no-op account updates. Usually, the access permission is set to require no authorization. However, for token manager contracts (custom tokens), `access` requires at least proof authorization so that token interactions are approved by calling one of the token manager's methods.
- `setTiming`: The permission corresponding to the ability to control the vesting schedule of time-locked accounts.

Authorization

When a transaction goes to the system, it will want to make updates to multiple accounts, so we need to be sure that those updates are authorised.
To do this, there is an authorisation field, there are 2 possibilities

- If the `authorization` field has a proof attached, it means the transaction is authorized by a proof that is checked against the verification key of the account.
- If the `authorization` field has a signature, it means the account update is authorised by a signature.

Types of authorisations

- `none`: Everyone has access to fields with permission set to `none` - and therefore can manipulate the fields as they please.
- `impossible`: If a field permission is set to `impossible`, nothing can ever change this field!
- `signature`: these can only be manipulated by account updates that are accompanied

and authorized by a valid signature.

- `proof`: Fields that have their permission set to `proof` can be manipulated only by account updates that are accompanied and authorised by a valid proof.
  Proofs are generated by proving the execution of a smart contract method.
  A proof is checked against the verification key of the account to ensure that state is changed only if the user generated a valid proof by executing a smart contract method correctly.
- `proofOrSignature`:
  An authorisation set to `proofOrSignature` will accept either a valid signature or a valid proof.

With our contracts we want the authorisation set to `proof` so that a proof needs to be supplied to change the state of the contract.

When we deploy a contract we get a default set of permissions, as described in the [docs]

Require signature

Use this command if this account update should be signed by the account owner, instead of not having any authorisation.

If you use this and are not relying on a wallet to sign your transaction, then you should use the following code before sending your transaction:

```
let tx = Mina.transaction(...); // create transaction as usual, using `requireSignature()` somewhere
tx.sign([privateKey]); // pass the private key of this account to `sign()`!
```

Example of an unsecure contract

```
class UnsecureContract extends
SmartContract {
  init() {
    super.init();
    this.account.permissions.set({
      ...Permissions.default(),
      send: Permissions.none(),
    });
  }

  @method withdraw(amount: UInt64) {
    this.send({ to: this.sender, amount
});
  }
}
```

Note the permissions specified in the `init()` method have set the `send` permission to `Permissions.none()`. Because `none` means you don't have to provide *any* form of authorisation, a malicious actor can easily drain all funds from the smart contract.

Here is an example of code that could do that

```
tx = await
Mina.transaction(account1Address, () => {
  let withdrawal =
AccountUpdate.create(zkappAddress);
  withdrawal.send({ to: account1Address,
amount: 1e9 });
});
await tx.sign([account1Key]).send();
```

To fix this we can change the permissions as follows

```
send: Permissions.signature(),
```

However this is not what we would generally want, usually we would want a proof instead.

```
send: Permissions.proof(),
```

Even so, we should also still secure the `@method withdraw(amount: UInt64)` to specify who can withdraw funds.

Permissions and upgradability

You can use the `setVerificationKey` permission to control whether a contract is upgradable or not. `setVerificationKey: Permissions.impossible(),` will mean that the contract cannot be upgraded. If this is what you intend, you should also set the permission `setPermissions` to `impossible`

Signatures

See Signature class

- create method creates a signature from a private key and message
- the verify method checks a signature for a message and public key

An example of using a signature

```
const signedNum1 = Int64.from(-3);
const signedNum2 = Int64.from(45);

const signedNumSum =
signedNum1.add(signedNum2);
const zkAppPrivateKey =
PrivateKey.random();
const zkAppPublicKey =
zkAppPrivateKey.toPublicKey();


const data2 =
char1.toFields().concat(str1.toFields());
const signature =
Signature.create(zkAppPrivateKey, data2);

const verifiedData2 =
```

```
signature.verify(zkAppPublicKey,
data2).toString();
```

Identity Standards

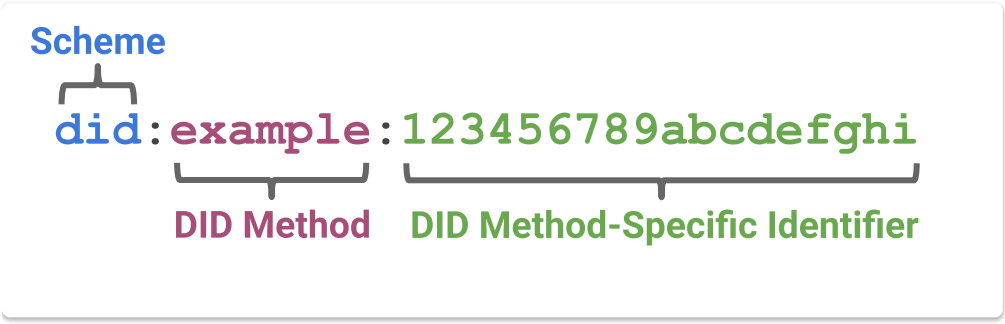
Decentralised Identifiers

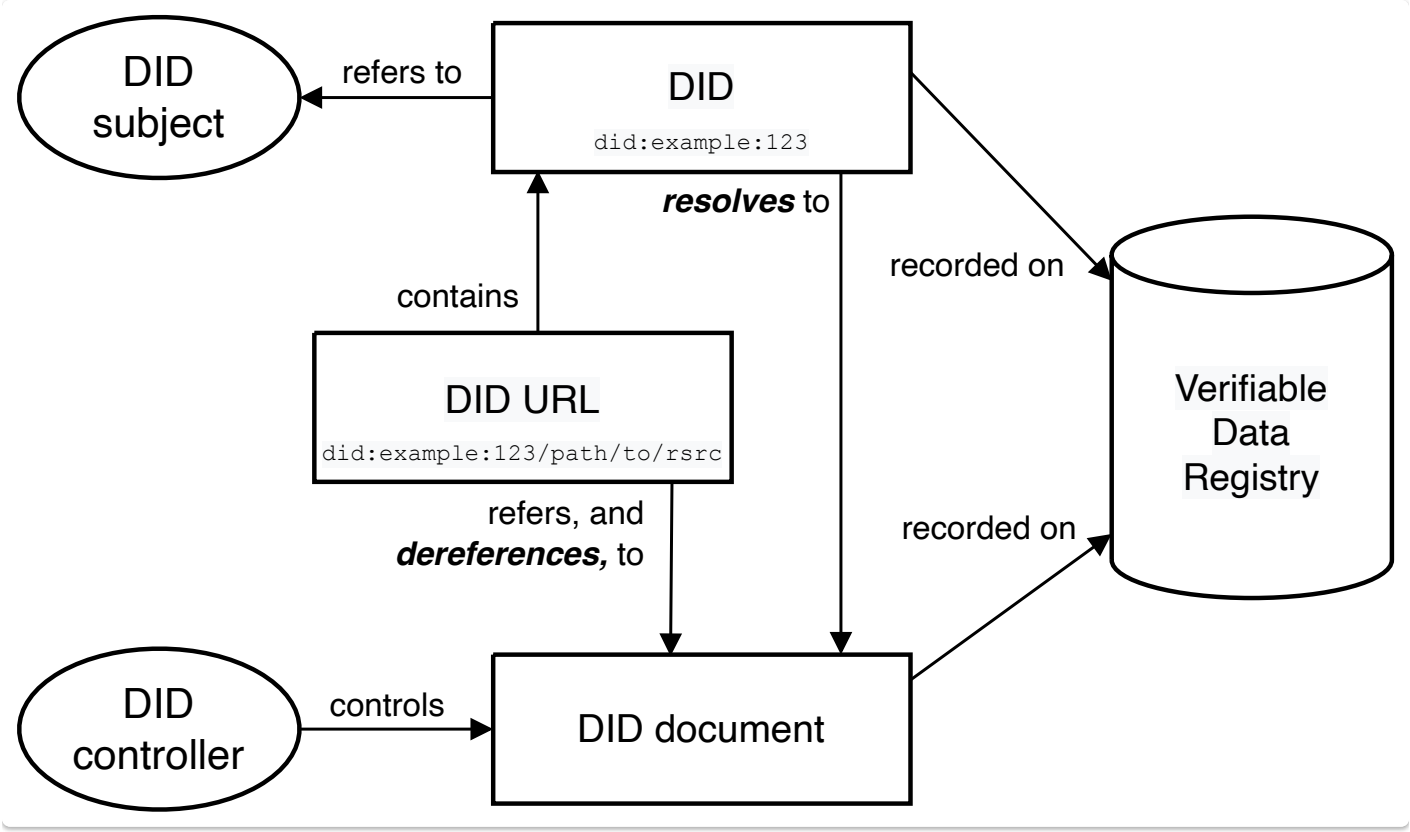
Decentralised Identity standards exist in the form of

- DIDs
- Verifiable credentials
  These are generic standards and not dependent on blockchain / zkp.
  There is a W3 standard for DIDs

The DID consists of 3 parts

1. the `did` URI scheme identifier,
2. the identifier for the DID method, and
3. the DID method-specific identifier.



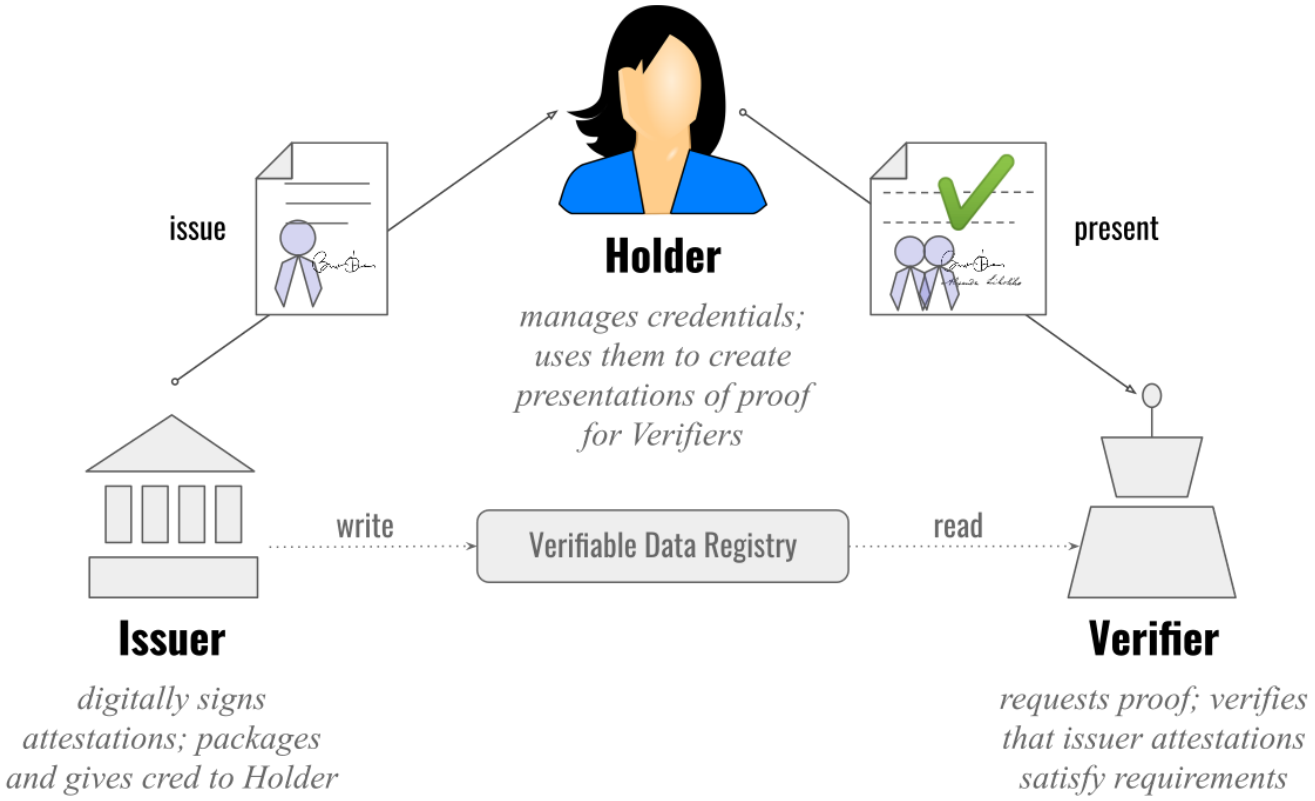


DIDs follow the Privacy by Design principle, with privacy as the default setting and privacy embedded in the design.

Verifiable Credentials

See Standard

A verifiable credential is a tamper-evident credential that has authorship that can be cryptographically verified. Verifiable credentials can be used to build verifiable presentations, which can also be cryptographically verified.


issue
present
Holder
manages credentials; uses them to create presentations of proof for Verifiers
write
Verifiable Data Registry
read
Issuer
digitally signs attestations; packages and gives cred to Holder
Verifier
requests proof; verifies that issuer attestations satisfy requirements

Mina Identity Solutions

See Video

Concerns / requirements

- Control of Identity / Data
- Differential Privacy
- Multiple Identities
- Ease of update
- Verifiable Identity / Liveness
- Composability

Zero Knowledge technology can be used to provide these requirements.

Composable Identity

See article
Listen to Twitter Space

ZKP Building blocks

0xPARC has produced some building blocks in this space

- Semaphore
- zk Nullifier Signature

Example Mina Identity Projects

From recent cohorts

zk SmartID

Aims to provide as an zkOracle linking to an national ID service

- Proof of unique human
- Proof of adulthood
- Proof of not being on the international sanctions list

Decentralized identity credentials

allows users to add verifiable reputation credentials and aggregated scores to their wallet accounts

KimlikDAO Pass

The KimlikDAO Pass comes with 3 verifiable credentials by default.

- Government ID
    - such as Passport scan
- Contact Info:
    - Verified email, phone number
- HumanIDv1:

    - Anonymous unique ID, signed by 7 independent KimlikDAO network nodes
- HumanIDv2
    - fully anonymous unique ID which gives a usable nullifier

[Pass3]

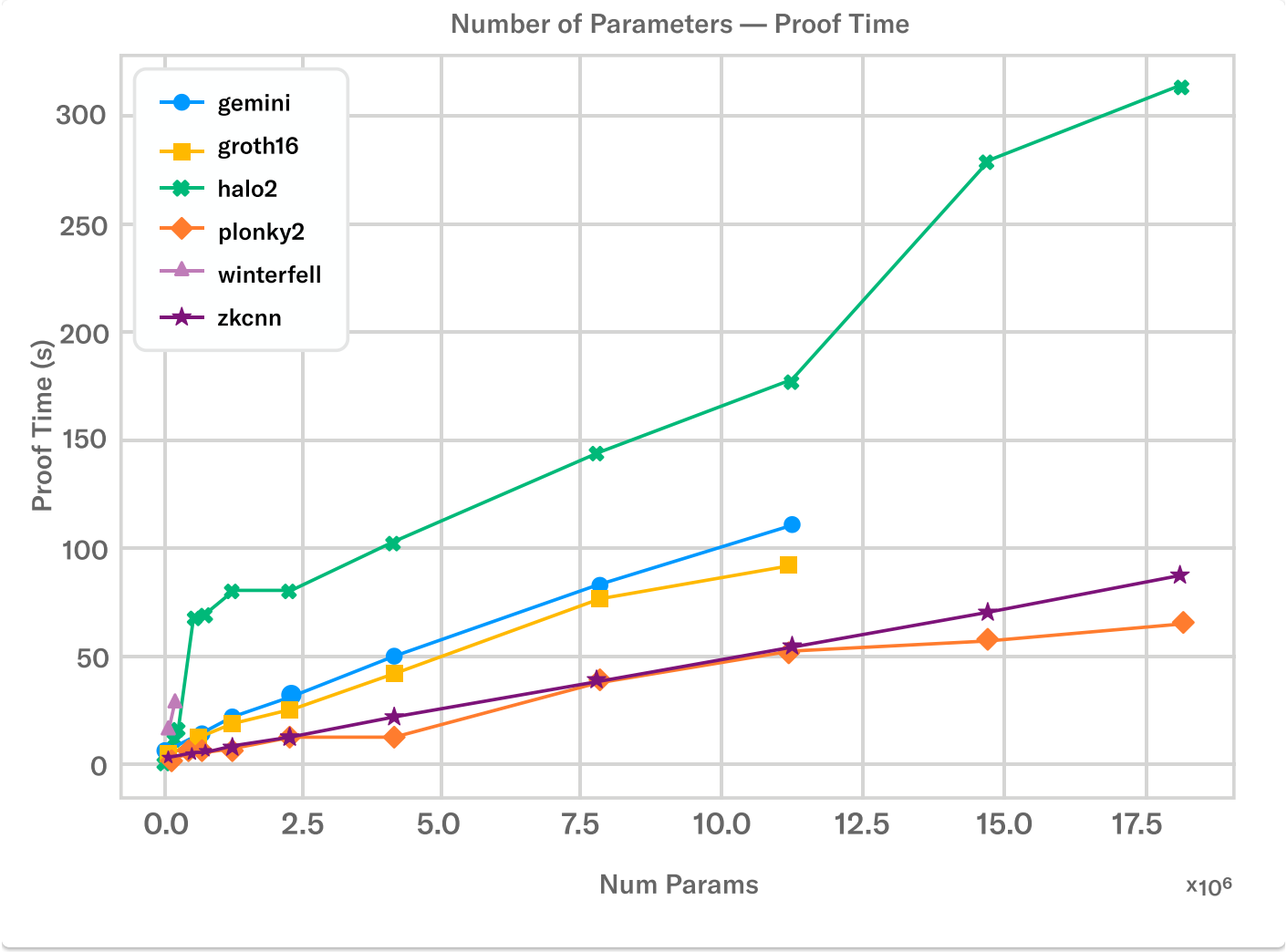
Pass3 is designed to be an umbrella solution to the existing concepts of identity.
Pass3 creates an identity layer for a public address, able to integrate the existing identity solutions to its scope and provide the consumers of the data, the applications, a seamless way of querying a public address regarding its scope.

zk Machine Learning

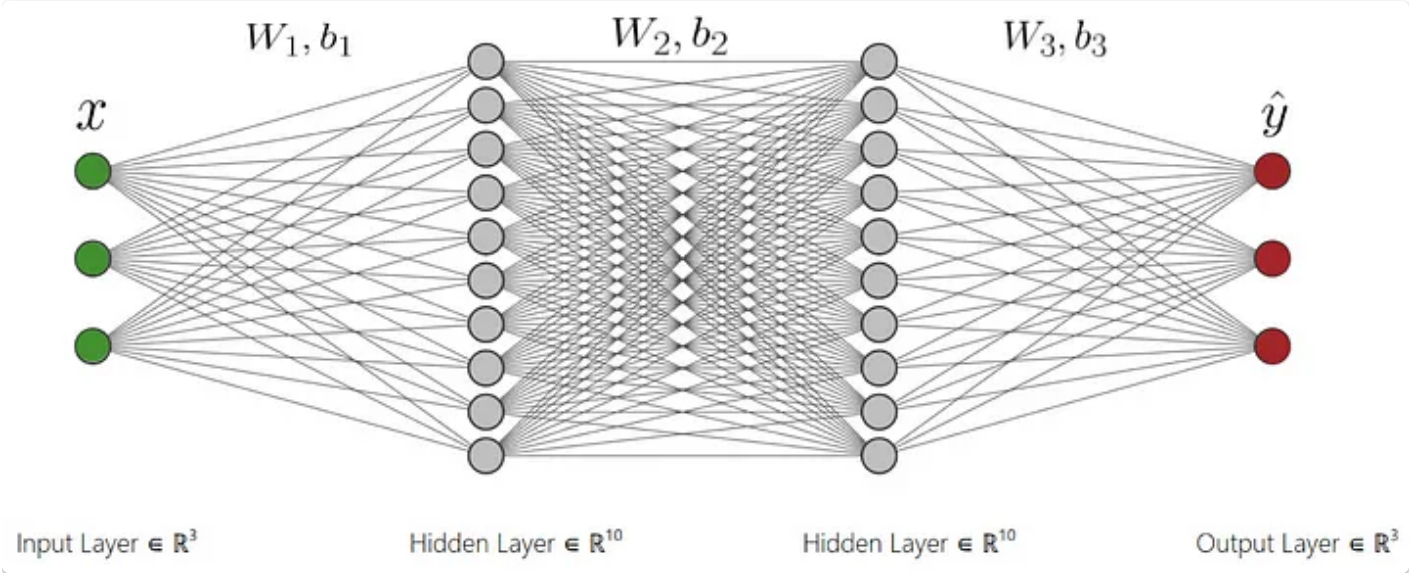
Introduction

Although there is definite appetite to bring machine learning to blockchain, or more recently to perform machine learning in zero knowledge, this has proved difficult to achieve.

There is a fundamental difference between the types of maths used for zk (integer operations on finite fields) and ml (matrix multiplication using floating point numbers)
Converting between the two can be costly in terms of performance.



Typical Neural Net Architecture



What we are trying to do, is to make the inference step of the network provable (training is considered to costly to prove at the moment).

Taking a naive approach, implementing the neural net directly as a circuit, leads to poor scalability for example $O(n^3)$ for a square matrix of side n

Other techniques have managed to improve this, for example using FFTs some some of the operations involved rather than matrix multiplication.

An approach used by Giza is to create an ONNX format for a model built using traditional tools such as Tensor Flow, and then transpile this into a provable language such as Cairo.

A fundamentally different approach is that taken by the zero gravity project (See Repo) who build neural networks without using weights.

Progress is being made, recently Modulus Labs announced the first on chain LLM

We have seen a number of proposals in cohorts to build zkML on Mina
For example
Veritas
They try to overcome the difficulties of proving the inference, by making a commitment to the

> inference, and then allow people to challenge that as necessary.