Session 1

Course overview

Each Session will be 90 mins, we will provide also challenges which will be submitted every month.

Date	Topics
15/12/23	o1js review
12/1/24	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

Q&A on this sli.do

Intro to Extropy

We've been working with Mina since 2019

This is a meetup in 2022

Session1 topics

- Decentralisation
- Roadmap
- Basic and advanced data types
- Permissions
- Bitwise operations
- Composability
- Action / Reducer

Decentralisation

It is important to remember why we are developing on blockchains.

Problems with centralised systems

Monetary System

- Bank closure / insufficient capital reserves
- greek debt crisis in 2015 ? banks closed and people lost savings, insurance schemes meant nothing, lead to an increase in Bitcoin use in Greece
- Availability of banks
- Inflation - money supply controlled by central authority
- Merchant accounts may be shut down
- Control of money for political reasons - wikileaks funding shutdown

There are layers of access control built into our banking systems to prevent fraudulent transactions, effectively security is achieved by closing the network.

Goals of decentralisation

- Participation
- Diversity
- Conflict resolution
- Flexibility
- Moving power to the edge (user)

Scalability and centralisation

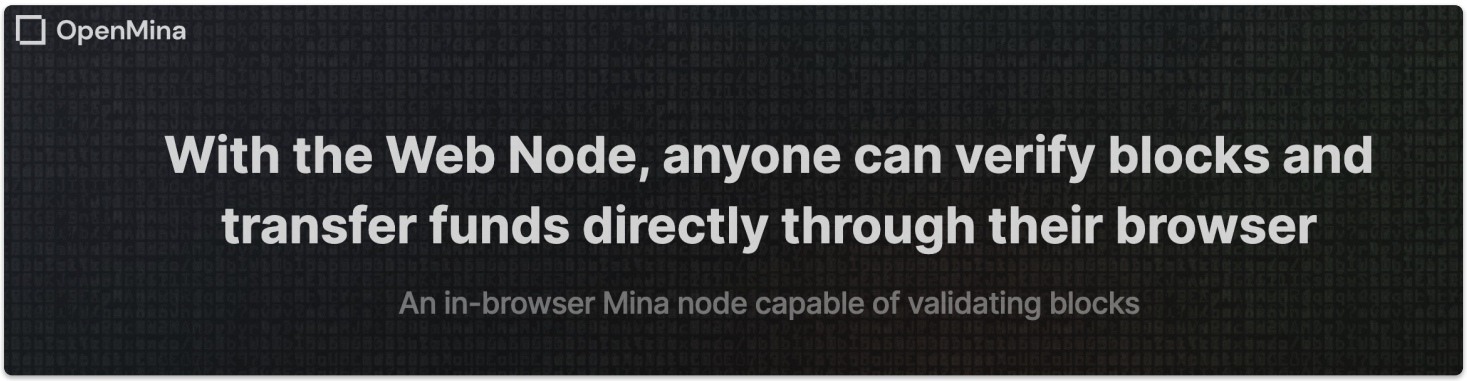
Even though this is ostensibly the goal for all blockchains, in practice participation may not be feasible for many people.

As the hardware requirements to run a node increase, nodes that fail to meet the requirements will be unable to process transactions quickly enough, they will fall behind, lose consensus and effectively fall off the network, leading to an increase in centralisation.

From Roadmap trust minimisation documentation

"We imagine a world where everyone through their phones and other digital devices plays a role in democratically securing Mina. That way, we can each play a part in securing and decentralising the services we rely upon and use."

We need to champion the opportunities which Mina provides.



Blog post about Web Node

Mina Roadmap

See roadmap

zkRollups on Mina

See article

See zkFusion from Trivio

Minaverse

> Minaverse is also about building an SDK for composable platforms, making it easy to build zkRollups and zkAppChains that recursively verify each other, while being able to rely on the security provided by Mina as a settlement layer. We want it to be as easy as possible for a developer to launch a new chain – and for that chain to be interoperable with Mina and the rest of crypto, while inheriting Mina's security
>
> By the end, we want Mina to both be fully interoperable with all digital systems, and make it easy to spin up new systems on top of it. That way we can give as much flexibility and connectivity to developers as possible.

Example privacy project

o1js updates

Monthly news about o1js updates are published in the blog

Latest updates

- Prover key caching - meaning faster compile times
- Custom gates and lookup tables

Programming for ZK Systems - why is it hard ?

- Relatively new area
- Performance
  - waiting for proof construction and verification
- Intrusion of cryptography into the language
  - Working with finite fields
  - Limitations on circuits
    Circuits in o1js have a fixed maximum size. Each operation performed in a function counts towards this maximum size. This maximum size is equivalent to:

    - about 5,200 hashes on two fields
    - about 2,600 hashes on four fields
    - about `2^17` field multiplies
    - about `2^17` field additions

    If a program is too large to fit into these constraints, it can be broken up into multiple recursive proof verifications

See this question about code and circuits

Underconstrained code

In zk applications a major security problem can come from under constrained code. This is why it is important to have sufficient assertions so that our proof is complete.

On the other hand we can have code that is not part of our proof.

o1js Basics

Starting tutorials / documentation

How zkApps work

o1js basic concepts

Developer tutorials

Mina playground

Data Types

Basic types

There are five basic types derived from Field

- Bool
- UInt32
- UInt64
- Int64
- Character

Advanced types

- CircuitString
- PrivateKey
- PublicKey
- Signature

String Manipulation

We can use the `CircuitString` API to gives us dynamic length Strings

For example we can create from a String literal

```
const mystr = CircuitString.fromString('Hi Navigators');
```

Private and Public Keys

Private Key and Public Keys are custom types
To generate a public / private key pair we can use

```
const myPK = PrivateKey.random();
const myPubKey = myPK.toPublicKey();
```

Signature

See Docs

- 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

```ts
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();
```

Merkle Trees and Merkle Maps

Useful data structures particularly when we have large amounts of data. In Session 4 we will look at some of the options around data storage.

Merkle Trees

See Documentation

You can import the `MerkleTree` type and create one of a specific height, for height `h` you will get 2^{h-1} leaves

A common pattern is to store the root of the tree in our contract.

Checks for inclusion of a leaf, or updates to that leaf require details of the path through the tree to the leaf, such as specifying the index of the leaf.

Example update code

```ts
// initialize the zkapp
const zkApp = new BasicMerkleTreeContract(basicTreeZkAppAddress);
await BasicMerkleTreeContract.compile();

// create a new tree
const height = 20;
const tree = new MerkleTree(height);
class MerkleWitness20 extends MerkleWitness(height) {}
```

```
...

const incrementIndex = 522n;
const incrementAmount = Field(9);

// get the witness for the current tree
const witness = new
MerkleWitness20(tree.getWitness(incrementIndex
));

// update the leaf locally
tree.setLeaf(incrementIndex, incrementAmount);

// update the smart contract
const txn1 = await
Mina.transaction(senderPublicKey, () => {
  zkApp.update(
    witness,
    Field(0), // leafs in new trees start at a
state of 0
    incrementAmount
  );
});
await txn1.prove();
const pendingTx = await
txn1.sign([senderPrivateKey,
zkAppPrivateKey]).send();
await pendingTx.wait();

// compare the root of the smart contract tree
```

```
to our local tree
console.log(
  `BasicMerkleTree: local tree root hash after
send1: ${tree.getRoot()}`
);
console.log(
  `BasicMerkleTree: smart contract root hash
after send1: ${zkApp.treeRoot.get()}`
);
```

In the contract we have

language-ts

```
  @state(Field) treeRoot = State<Field>();

  @method initState(initialRoot: Field) {
    this.treeRoot.set(initialRoot);
  }

  @method update(
    leafWitness: MerkleWitness20,
    numberBefore: Field,
    incrementAmount: Field
  ) {
    const initialRoot = this.treeRoot.get();
    this.treeRoot.assertEquals(initialRoot);

    incrementAmount.assertLt(Field(10));
```

```
    // check the initial state matches what we
expect
    const rootBefore =
leafWitness.calculateRoot(numberBefore);
    rootBefore.assertEquals(initialRoot);

    // compute the root after incrementing
    const rootAfter =
leafWitness.calculateRoot(
      numberBefore.add(incrementAmount)
    );

    // set the new root
    this.treeRoot.set(rootAfter);
  }
```

Merkle Map

See Docs

This works in a similar way to the merkle tree, but allows to to specify the leaf by a key rather than an index.

Example code

```ts
const map = new MerkleMap();

const rootBefore = map.getRoot();

const key = Field(100);

const witness = map.getWitness(key);

// update the smart contract
const txn1 = await
Mina.transaction(deployerAccount, () => {
  zkapp.update(
    contract.update(
      witness,
      key,
      Field(50),
      Field(5)
    );
  );
});
```

Permissions

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.

Authorisation

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:

```ts
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

```ts
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.

Bitwise operations

See Docs
It is sometimes more efficient to implement certain operations in special collections of custom gates called gadgets. Bitwise operations are one such operation.
The bitwise operations include

- and()
- not()
- xor()
- Left shifts
  Shifts the number left by a number of bits, any bits that fall off the left side are discarded. 0s are padded in from the right.
  - leftShift32()
  It returns a new `Field` element that is range checked to 32 bits.You can use rangeCheck32
  - leftShift64()
  It returns a new `Field` element that is range checked to 64 bits. You can use rangeCheck64
  Right Shift
  It moves the bits of a binary number to the right by the specified number of bits. Any bits that fall off the right side are discarded. 0s are padded in from the left. It returns a new `Field` element.

- rightShift64()

You can use rangeCheck64

Rotations are similar to left or right shifts, but instead of bits being discarded, then wrap round.

- rotate32()
- rotate64()

Composability

Example

See repo

From the example we have 2 contracts

```ts
// contract which can add 1 to a number
class Incrementer extends SmartContract {
  @method increment(x: Field): Field {
    return x.add(1);
  }
}
```

```ts
// contract which can add two numbers, plus 1,
and return the result
// incrementing by one is outsourced to
another contract (it's cleaner that way, we
want to stick to the single responsibility
principle)
class Adder extends SmartContract {
  @method addPlus1(x: Field, y: Field): Field
{
    // compute result
    let sum = x.add(y);
    // call the other contract to increment
    let incrementer = new
Incrementer(incrementerAddress);
    return incrementer.increment(sum);
  }
}
```

```ts
// contract which calls the Adder, stores the result on chain & emits an event
class Caller extends SmartContract {
  @state(Field) sum = State<Field>();
  events = { sum: Field };

  @method callAddAndEmit(x: Field, y: Field) {
    let adder = new Adder(adderAddress);
    let sum = adder.addPlus1(x, y);
    this.emitEvent('sum', sum);
    this.sum.set(sum);
  }
}
```

By calling method `callAddAndEmit` we call also call the other contracts.

Action / Reducer

See Docs

Motivation / Use case

A common pattern to handle off chain state is to have a merkle tree holding state, update the state in the tree, producing a new state root, and then publish the root on chain.

This runs into problems when we want concurrent state updates, and given Mina's block time, it is likely that any particular block will contain multiple updates.
This is an obvious concern for useful applications running off chain, trying to keep a consistent view of state without concurrency problems, such as the first update succeeding, and subsequent updates in the same block failing.
From Reducers are Monoids

"Manipulation of state in large applications quickly gets hairy. As an application grows, it becomes a real challenge to be sure that state mutations affect only the components you want it to. One mitigation is to centralize all of your state manipulation as best as you can — ideally to one function or one file or one module.
To do so, we can decouple an intention to change state (or an action) from the actual state change itself.
Reducers are one way to cleanly declare atomic chunks of

state manipulation in response to these actions. Smaller reducers can be composed into bigger ones as our application's state management grows in scope."

To continue with our earlier example, we send a number of `pending actions` which show our intentions, then at a later time you roll these up using a `reducer` to apply the actions sequentially, then finally we would update the state on chain.

Events and actions are not stored in the ledger and exist only on the transaction.

Examples

You can see an example here

```ts
// compute the new counter and hash from pending actions
    let pendingActions = this.reducer.getActions({
      fromActionState: actionState,
    });

    let { state: newCounter, actionState: newActionState } =
      this.reducer.reduce(
        pendingActions,
        // state type
        Field,
        // function that says how to apply an action
        (state: Field, _action: Field) => {
```

```
          return state.add(1);
        },
        { state: counter, actionState }
      );

    // update on-chain state
    this.counter.set(newCounter);
    this.actionState.set(newActionState);
  }
}
```

Difference beween getActions and fetchActions

- getActions works with the blockchain network
- fetchActions works with archive nodes

Drawbacks to the Action / Reducer model

See Issue
This doesn't scale well, we need to have a reducer process continually running in order to rollup any actions that happen. Also the reducer is limited in what it can due because of the circuit size limit, see comment

Workarounds / Alternatives

There is an interesting discussion here

Some thoughts from that

- The reducer has a recursive proof to change the state. The zkApp gets all pending actions from the archive node, reduce them recursively using ZkProgram, and

feeds the proof to the reducer on-chain as an argument to change the state.

A proposal from Gregor is
to have two kinds of actions - normal ones and those big state-storing ones. The reducer would ignore the big state actions and for reading state we would ignore the normal actions

Projects providing alternative patterns

- Protokit
- Zeko
- Anomix
- nft-zkapp