

Session 2

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
- Development workflow and tools
- Techniques and useful patterns

Slido [Link](#)

What's new

See January [Blog](#)

- Bitwise [operations](#)
- Foreign Field [Arithmetic](#)
- [Keccak](#)
- [ECDSA](#)

Improved prover times by caching existing static data structures, see

Twitter [post](#)

Discord [post](#)

Github [issue](#)

Interesting things coming up

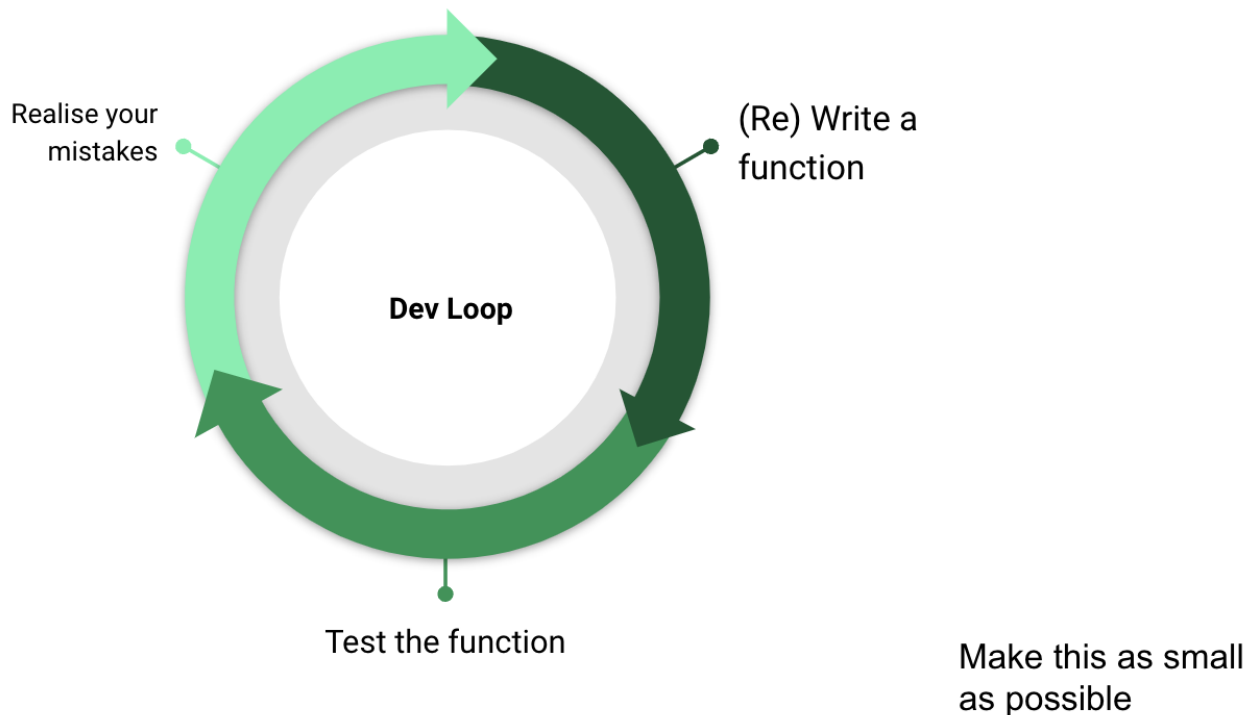
- Asynchronous [circuits](#)

Mina [roadmap](#)

General Blockchain directions

- Modularity
 - Still a focus on scalability
 - Privacy
 - Interoperability (See [Mina Bridge](#))
-

Development Workflow



Testing your zkApp

See [Docs](#)

Testing principles you used in web2 still apply, such as TDD.

You can use

```
npm run coverage
```

to get a coverage report.

You should be writing unit and integration tests.

Since we are using typescript, the Jest framework can be used to write tests.

See [Docs](#)

It is recommended to use `describe` blocks to delineate

the areas of functionality.

For example

```
describe('dex.test.ts', () => {  
  describe('deposit()', () => {  
    ...  
  });  
  describe('swap()', () => {  
    ...  
  });  
});
```

The usual `beforeAll` and `beforeEach` functions are available.

It is good practice to check pre and post conditions for your functions.

Disable proofs

When initially (unit) testing our code, we may just be looking to see that certain logic is correct, and we may be less concerned about proofs.

We have the option to enable / disable proofs with the `proofsEnabled` parameter when starting a local blockchain, or by calling

```
Local.setProofsEnabled(x: boolean)
```

Disabling proofs can shorten the dev / test iteration time, the default value is enabled.

In Web3 we have additional steps as we will be deploying to a network.

Starting with a mock blockchain

```
const Local = Mina.LocalBlockchain();  
Mina.setActiveInstance(Local);
```

This supplies some accounts for us, for example

```
let feePayer =  
Local.testAccounts[0].privateKey;
```

Example Integration test using the mock blockchain

```
describe('Add smart contract integration  
test', () => {  
  let feePayer: PrivateKey,  
      zkAppAddress: PublicKey,  
      zkAppPrivateKey: PrivateKey,  
      zkAppInstance: Add,  
      currentState: Field,  
      txn;  
  
  beforeAll(async () => {  
    // setup local blockchain  
    let Local = Mina.LocalBlockchain();  
    Mina.setActiveInstance(Local);  
  
    // Local.testAccounts is an array of 10  
    test accounts that have been pre-filled with  
    Mina  
  
    feePayer =
```

```
Local.testAccounts[0].privateKey;
```

```
    // zkapp account
    zkAppPrivateKey = PrivateKey.random();
    zkAppAddress =
zkAppPrivateKey.toPublicKey();
    zkAppInstance = new Add(zkAppAddress);

    // deploy zkapp
    txn = await Mina.transaction(feePayer, ()
=> {
        AccountUpdate.fundNewAccount(feePayer);
        zkAppInstance.deploy({ zkappKey:
zkAppPrivateKey });
    });
    await txn.send();
});

it('sets initial state of num to 1', async
() => {
    currentState = zkAppInstance.num.get();
    expect(currentState).toEqual(Field(1));
});

it('correctly updates num from initial state
to 3', async () => {
    txn = await Mina.transaction(feePayer, ()
=> {
        zkAppInstance.update();
        zkAppInstance.sign(zkAppPrivateKey);
```

```
});  
txn.send();  
  
currentState = zkAppInstance.num.get();  
expect(currentState).toEqual(Field(3));  
});  
});
```


Deploying to a lightweight local network.

There is a docker [image](#) you can use to spin up a local network, this has 1000 pre funded accounts.

You can set up a multi node network with this.

Deploying to testnet

[Docs](#)

You can use zk config to create a [deploy alias](#) this will take you through the details you need to add such as

- Alias name
- URL
- transaction deploy fee
- fee account

Once the config is set up, you can use

```
zk deploy
```

to deploy your contract


You can get test Mina from the [faucet](#) or [minanft faucet](#)

Development Tools

- Usual IDEs - VSCode etc.
- CLI

Mina Playground


See [Site](#)

 Mina Playground

All-in-One Platform for Mina Protocol

With Mina Playground you can create, test and run zkApps/Smart Contracts, deploy your own Smart Contracts and follow interactive tutorials.

[Follow a tutorial →](#)

 Mina Playground

Introduction

Functions

Functions

Functions work as you would expect in TypeScript. For example:

```
function addOneAndDouble(x: Field): Field {  
  return x.add(1).mul(2);  
}
```

Here we take the input `x`, add 1 to the Field value, multiply it by two and return the new Field.

Tutorial


Let's first call the `addOneAndDouble` function with our defined Field `x` and call the result `y`:

```
const y = addOneAndDouble(x);
```

Now check if our `y` field is equal to the value of 4:

```
const isEqual = y.equals(4);
```

```
1 import { Field, Provable } from "o1js";  
2  
3 let x = new Field(1); // x = 1  
4  
5 const addOneAndDouble = (x: Field): Field => {  
6   return x.add(1).mul(2);  
7 };  
8
```

 Run

```
~/mina  
> 
```

Smart Contracts

Public/private inputs

Public and private inputs

While the state of a zkApp is **public**, method parameters are **private**.

When a smart contract method is called, the proof it produces uses zero-knowledge to hide inputs and details of the computation.

The only way method parameters can be exposed is when the computation explicitly exposes them, as in the last example where the input was directly stored in the public state: `this.x.set(xPlus1);`

For example where this is not the case, define a new method called `incrementSecret()`:

```
class HelloWorld extends SmartContract {
  @state(Field) x = State<Field>();

  // ...

  @method incrementSecret(secret: Field) {
    const x = this.x.get();
    this.x.assertEquals(x);

    Poseidon.hash(secret).assertEquals(x);
    this.x.set(Poseidon.hash(secret.add(1)));
  }
}
```

```
1 import {
2   SmartContract,
3   PrivateKey,
4   Field,
5   method,
6   AccountUpdate,
7   Mina,
8   state,
9   State,
10  Provable,
11  Poseidon,
12 } from "o1js";
13
```

 Run

```
~/mina
> □
```

There is also project to create an online IDE using docker

see [repo](#)

Block Explorers

[Mina explorer](#)

[Minataur](#)

Mina Use Cases

The blockchain world has become very focussed on DeFi, it is worth looking at other use cases.

When considering Mina use cases, think of Mina's strengths, for example

- Verifiable off chain computation
- Private inputs
- Recursion in contracts
- Succinct blockchain

There are many successful use cases around

- Verification of private data
- Games involving limited information
- zk Oracles
- Identity / Login
- Tooling

Privacy of inputs allows us more flexibility, for example without private inputs we have additional security requirements.

It is a good opportunity to explore some of the other ideas from the program.

Why projects fail

- Not feasible
 - technically
 - economically
 - scope too large
 - Mina not needed
 - Boilerplate projects
 - Team issues
 - Lack of experience
 - Lack of time
-

Mina patterns

Composability

A general pattern, very widely used in DeFi.

See Session 1 notes, here is some example [code](#)

Recursion

More of this in the next session.

As a pattern, it can help with a problem that zk frameworks face that of dynamic circuit size.

A recursive function has two parts, a step and a base condition.

For example

```
import { SelfProof, Field, ZkProgram, verify
} from 'o1js';

const AddOne = ZkProgram({
  name: "add-one-example",
  publicInput: Field,

  methods: {
    baseCase: {
      privateInputs: [],

      method(publicInput: Field) {
```

```
        publicInput.assertEquals(Field(0));
    },
},

step: {
    privateInputs: [SelfProof],

    method(publicInput: Field,
earlierProof: SelfProof<Field, void>) {
        earlierProof.verify();

earlierProof.publicInput.add(1).assertEquals(
publicInput);
    },
},
},
});
```

We can then recurse through this with

```
const { verificationKey } = await  
AddOne.compile();
```

```
const proof = await  
AddOne.baseCase(Field(0));
```

```
const proof1 = await AddOne.step(Field(1),  
proof);  
const proof2 = await AddOne.step(Field(2),  
proof1);
```

finally verifying this in a contract

```
@method foo(proof: Proof) {  
    proof.verify().assertTrue();  
  
}
```

You could use this approach if you (potentially) have a variable number of arguments that you want to send to a function . You can split the function into parts and recurse through those parts.

See [this discussion](#)
and example

[Recursion example](#)

Nullifier Pattern

This is a general pattern used in cryptography, where we wish to show that an event has happened or that a resource has been consumed. It for example allows us to prevent double spends in confidential tokens such as ZCash.

The general pattern is

- You have some means of identifying a resource, for example by taking a hash
- When that resource is consumed, you create a nullifier from the hash. This is stored as part of the application state and indicates that the resource has been consumer.
- The hiding properties of a hash function allow some confidentiality in the process.

An example of this is in this [project](#)

Stateless Contracts

This is a contract without state variables, only methods, you can have non state 'variables' in your contract

With this you have more flexibility of multiple transactions in one block.

See code from [Mina NFT](#)

See [discussion](#)

This allowed many [transactions](#) in one block

Contracts or programs

A simple way to look at this is to say that contracts have some extra functionality (concerned with blockchain context) over zkProgram.

See this [discussion](#) for some great details







Be aware of what needs to be proved and what doesn't

- Think of a sudoku result prover
 - Be aware of under constrained code
-

Mina / o1js Techniques

Packing Variables

See [o1js-pack](#)

 PackedBool.test.ts
 PackedBool.ts
 PackedString.test.ts
 PackedString.ts
 PackedUInt32.test.ts
 PackedUInt32.ts

Example [test](#) showing strings being packed and unpacked

Checking if an account exists

```
const zkApp = new
MinaNFTNameServiceContract(nameService.address);
const tokenId = zkApp.token.id;
await fetchAccount({ publicKey: address,
tokenId });
const hasAccount = Mina.hasAccount(address,
tokenId);

const transaction = await Mina.transaction(
  { sender, fee: await MinaNFT.fee(),
```

```
memo: "minanft.io", nonce },  
    () => {  
        if (!hasAccount)  
            AccountUpdate.fundNewAccount(sender);  
        zkApp.mint(mintData);  
    }  
);
```

Time restrictions

Examples are given in the [docs](#)

Context from the network

A number of values are available about the context in which your code is running.

```
// current UNIX time in milliseconds, as  
// measured by the block producer  
this.network.timestamp.get(): UInt64;  
// length of the blockchain, also known as  
// block height  
this.network.blockchainLength.get(): UInt32;  
// total minted currency measured in units of  
// 1e-9 MINA  
this.network.totalCurrency.get(): UInt64;  
// slots since genesis / hardfork -- a "slot"  
// is the Mina-native time unit of 3 minutes  
this.network.globalSlotSinceGenesis.get():  
UInt32;
```

```
this.network.globalSlotSinceHardFork.get():  
UInt32;  
// hash of the snarked ledger -- i.e., the  
// state of Mina included in the blockchain  
// proof  
this.network.snarkedLedgerHash.get(): Field;  
// minimum window density in our consensus  
// algorithm  
this.network.minWindowDensity.get(): UInt32;  
// consensus data relevant to the current  
// staking epoch  
this.network.stakingEpochData.ledger.hash.get()  
(): Field;  
this.network.stakingEpochData.ledger.totalCur  
rency.get(): UInt64;  
this.network.stakingEpochData.epochLength.get  
(): UInt32;  
this.network.stakingEpochData.seed.get():  
Field;  
this.network.stakingEpochData.lockCheckpoint.  
get(): Field;  
this.network.stakingEpochData.startCheckpoint  
.get(): Field;  
// consensus data relevant to the next,  
// upcoming staking epoch  
this.network.nextEpochData.ledger.hash.get():  
Field;  
this.network.nextEpochData.ledger.totalCurren  
cy.get(): UInt64;  
this.network.nextEpochData.epochLength.get():
```

```
UInt32;  
this.network.nextEpochData.seed.get(): Field;  
this.network.nextEpochData.lockCheckpoint.get()  
(): Field;  
this.network.nextEpochData.startCheckpoint.ge  
t(): Field;
```

Making a commitment to a file

1. Create a hash of the file off chain. (See example code [here](#))
2. Verifiably create a hash of the hash from 1 in a contract (or program).
3. Store the hash from 2 on chain.

ETH integration techniques

See ECDSA item above.

Using Ethereum addresses

Converting to ETH address see [discussion](#)

Example code :

Ethereum addresses are 20 bytes long and can be fitted into one field:

```
const address =  
"0xb3Edf83eA590F44f5c400077EBd94CCFE10E4Bb0";
```



```
const field = Field.from(address);

convert back:
const address = "0x" +
field.toBigInt().toString(16);
// to get lower-upper case right:
const addressWithUpperLowerCase =
ethers.utils.getAddress(address);
```

Using token balance as storage

See this [discussion](#)

Here is a [code example](#)

Meta data about your contract

See [docs](#)

You can use the `analyzeMethods` method on your contract to give for example details about the circuit. It returns

- `rows` the size of the constraint system created by this method
- `digest` a digest of the method circuit
- `hasReturn` a boolean indicating whether the method returns a value
- `actions` the number of actions the method dispatches

- `gates` the constraint system, represented as an array of gates

Rosetta API

This alternative to GraphQL can be used for

- Querying historical data from the Mina blockchain
- Integrating the Mina blockchain with your exchange
- Building blockchain applications, such as explorers and wallets

See [docs](#)

Resources

Navigator program [details](#)

Developer [resources](#)

[Mina Tutorials](#)

[o1js reference](#)