Session 3

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
- Recursion use case
- Recursion
- Custom Tokens
- ECDSA example

Slido Link

What's New

- Testworld-2 has completed
- Punk Poll Censorship resistant voting platform
- Security audit of Pickles

Recursion

Possibilities coming from blockchain recursion

Recursion is at the heart of Mina and allows a succinct blockchain.
This allows wider involvement since the hardware requirements to verify the chain are so much lower than other chains.

Recent article by Vitalik

- Transaction fees a significant driver of blockchain use
- ZK an essential technology

Transaction cost has a lower limit determined by the cost of computation, so here also Mina has an advantage.
With low transaction fees comes the possibility of a more diverse ecosystem.

Aggregation of Proofs

The implementation of zk rollups has been made more feasible with the introduction of aggregating smaller proofs.

For example a batch of transactions can be processed in parallel / as they arrive, and the resulting proofs aggregated into a single proof at the end of the batch.

ZK Recursion research

The last couple of years have seen a great deal of research into recursion, particularly following the folding scheme approach used by Nova and derivatives.

Recursion program use cases.

From the program design perspective recursion allows us to split a computation into smaller parts, and join these using recursion.
Furthermore it allows flexibility to respond to demand dynamically by increasing the number of times we recurse.

This also allows for the case where we could have a variable number of arguments to a function

Recursion in o1js

ZkProgram

The module that allows us to perform recursion is Zk Program

See also Self Proof

We can use ZkProgram to specify our computation and embed it in a Smart Contract method that verifies the execution.

Inputs

Inputs are private by default, and you can chose the shape of the inputs to the methods.

Simple example

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

const SimpleExample = ZkProgram({
  name: "Zkprogram-example",
  publicInput: Field,

  methods: {
    run: {
      privateInputs: [],

      method(publicInput: Field) {
```

```
        publicInput.assertEquals(Field(42));
      },
    }
  }
});
```

We can compile and run this

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

const proof = await
SimpleProgram.run(Field(42));
```

This gives us a proof which we can verify within our Smart Contract, and use the result of the computation / state within our ZkProgram

```
@method check_answer(proof:
SimpleProgram.Proof) {
proof.verify().assertTrue();
const answer: Field = proof.value;

}
```

Using ZkProgram for recursion

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);
      },
```

```
    },
  },
});
```

See gist

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();

}
```

Further example

SelfProof methods

With SelfProof we can use the `verifyIf` method for conditional verification.
We also have the `dummy` for for ZkPrograms that handle the base case in the same method as the inductive case, using a pattern like this:

```
method(proof: SelfProof<I, O>, isRecursive:
Bool) {
proof.verifyIf(isRecursive);
// ...
}
```

To use such a method in the base case, you need a dummy proof:

```
let dummy = await MyProof.dummy(publicInput,
publicOutput, 1);
await myProgram.myMethod(dummy, Bool(false));
```

Using recursion in a game

See this example

Tree based program design

The ability to use recursion gives us a useful design pattern, that of representing a computation as a tree. This example shows a rollup adding numbers

Proving and verifying off chain

We can of course verify the proof we create off chain.

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

const proof = await MyProgram.base(Field(0));
```

```
import { verify } from 'o1js';

const ok = await verify(proof.toJSON(),
verificationKey);
```

These can be used as a shortcut when testing if we want to test some logic and don't need to interact with the blockchain.

Recursion problems

You do need to ensure that the recursion will have a final step, otherwise you will get a `maximum call stack size exceeded` error.

The recursion steps we take need to be the same (in terms of the circuit) other we can get an error like

`Error: the permutation was not constructed correctly: final value`

There is currently a bug when using recursion with an ECDSA circuit, see this issue

Further reading

See this post by Brandon comparing Pickles and Nova
Illya's Blog including a calculator example

Custom Tokens

See Docs

Token Accounts

See Docs

Token accounts are like regular accounts, but they hold a balance of a specific custom token instead of MINA. A token account is created from an existing account but is specified by a public key *and* a token id.
If an existing account receives a transaction that is specified by a custom token, a token account for that public key and token id is created if it does not exist.

Token accounts are specific for each type of custom token, meaning that a single public key can have many different types of token accounts.

A token account is automatically created for a public key whenever an existing account receives a transaction denoted with a custom token.

Aside from sending custom tokens, custom tokens can be minted and burned by a **token owner account**.
A token owner account is the governing zkApp account for a specific custom token.

See o1js reference

token

• `get` **token**(): `Object`

Token of the SmartContract.

Returns

`Object`

Name	Type
`id`	`Field`
`parentTokenId`	`Field`
`tokenOwner`	`PublicKey`
`burn`	(`__namedParameters`: { `address`: `PublicKey` | `AccountUpdate` | `SmartContract`; `amount`: `number` | `bigint` | `UInt64` }) => `AccountUpdate`
`mint`	(`__namedParameters`: { `address`: `PublicKey` | `AccountUpdate` | `SmartContract`; `amount`: `number` | `bigint` | `UInt64` }) => `AccountUpdate`
`send`	(`__namedParameters`: { `amount`: `number` | `bigint` | `UInt64`; `from`: `PublicKey` | `AccountUpdate` | `SmartContract`; `to`: `PublicKey` | `AccountUpdate` | `SmartContract` }) => `AccountUpdate`

tokenSymbol

• `get` **tokenSymbol**(): `Object`

`Deprecated`

use `this.account.tokenSymbol`

Returns

`Object`

Name	Type
`set`	(`tokenSymbol`: `string`) => `void`

Defined in

lib/zkapp.ts:988

Token Owner

See. Docs

A token owner is an account that creates, facilitates, and governs how a custom token is to be used. Concretely, the token owner is the account that created the custom token and is the only account that can mint and burn tokens.
Additionally the token owner is the only account that can approve sending tokens between two accounts.
If two accounts want to send tokens to each other, the token owner must approve the transaction.
The token owner generates the changes the two accounts want to make and can then make assertions about those changes.
The token owner can approve the transaction with

- signature
- proof.
  See below for details of using these

TokenID

See Docs

Token ids are unique identifiers that are used to distinguish between different types of custom tokens.
Custom token identifiers are globally unique across the entire network.
The token ID can be found with

`this.token.id`

on a zkApp

Creating a token

The first step is to create a 'manager' contract for the token. This contract will enforce the logic around the common token functionality such as minting and burning.
The contract will also hold meta data about the token such as the symbol.

Interacting with the token

Minting

See Docs
Minting adds new tokens, the minted tokens can be sent to any existing account in the ledger.

```
class MintExample extends SmartContract {

@method mintNewTokens(receiverAddress:
PublicKey) {
    this.token.mint({
      address: receiverAddress,
      amount: 100_000,
    });
  }
}
```

Burning

Burning removes tokens, for example

```
class BurnExample extends SmartContract {

@method burnTokens(addressToDecrease:
PublicKey) {
    this.token.burn({
      address: addressToDecrease,
      amount: 100_000,
    });
  }
}
```

Sending Tokens

We can transfer tokens to other accounts using the `send()` method

For example

```
class SendExample extends SmartContract {

@method sendTokens(
    senderAddress: PublicKey,
    receiverAddress: PublicKey,
    amount: UInt64
  ) {
    this.token.send({
      to: receiverAddress,
```

```
      from: senderAddress,
      amount,
    });
  }
}
```

Authorisation

Clearly this functionality cannot be open to anyone for any token.
If an account wishes to interact with a token it did not create, it needs authorisation from the token owner.
This authorisation can be in the form of a signature or a proof.

Authorisation with a signature

Although this is the simplest approach, it is the least flexible. A token owner is required to sign the transaction sending the token before the transaction is submitted to the network.
The token contract must implement a `send()` method as above.

Signing the transaction example

```
let tx = await Mina.transaction(feePayerKey,
() => {
  zkapp.sendTokens(senderAddress,
```

```
receiverAddress, UInt64.from(1_000)); //
Sends 1,000 tokens from `senderAddress` to
`receiverAddress`
  zkapp.sign(zkappKey); // Signs the
transaction with the token owner key
});
await tx.send();
```

The lack of flexibility lies in the coordination required.

Authorisation with a proof

More flexible is the asynchronous approach of providing a proof.
To allow for proof authorization by the token owner, the child zkApp that is requesting authorization must provide a way for the token owner to inspect the changes it wants to make and verify that they are valid.
Token owner contracts have the power to inspect child account updates to enforce custom token rules. For example, a token owner contract could enforce that a child zkApp can send tokens only to a specific address.

Token owner contracts can inspect the updates that a child zkApp wants to make by using a combination of `Experimental.Callback` and `this.approve`.

1. The token contract generates the required account updates that a child zkApp wants to make.

2. The child zkApp wraps a function around `Experimental.Callback` which contain the changes.
3. The token owner can then execute the function with `this.approve` and inspect the changes that the child zkApp wants to make.

Example of proof authorisation
The result of this example is `zkAppB` sending tokens to `account1Address` and `account1Address` receiving tokens from `zkAppB`. The transaction is approved by the token owner without the token owner having to sign the transaction.

For another example of how to approve a transaction with a zkApp, see this authorization example provided.

Token Examples

A simple example
To interact with this token see this code

Extending a simple contract

Fuller example (take note of comments)

From the same project the last 2 tests show an example of sending from a normal account to smart contract

Whitelisted Token

Example token contract

Interacting with the token

ECDSA

See Docs

The ECDSA gadget is used to verify ECDSA signatures. The gadget takes as input the message, the signature, and the public key of the signer.
It outputs a `Bool` indicating whether the signature is valid.

Before you can use the gadget, you need to initiate it with the curve configuration.

```
// create a secp256k1 curve
class Secp256k1 extends
createForeignCurve(Crypto.CurveParams.Secp256
k1) {}
```

The choices of Curve Params we have are

```
// predefined curve parameters
CurveParams: {
  Secp256k1: CurveParams;
  Pallas: CurveParams;
  Vesta: CurveParams;
}
```

Once we have initialised the gadget we can create an instance.

```
// create an instance of ECDSA over
secp256k1, previously specified
class Ecdsa extends createEcdsa(Secp256k1) {}
```

Signing messages

This is achieved with the `sign` function from the ECDSA class. Signing is not provable

```
// a private key is a random scalar of
secp256k1
let privateKey = Secp256k1.Scalar.random();
let publicKey =
Secp256k1.generator.scale(privateKey);

// create a message, for a detailed
explanation of `Bytes` take a look at the
Keccak overview
let message = Bytes32.fromString('cat');

// sign a message - this is not a provable
method!
let signature = Ecdsa.sign(message.toBytes(),
privateKey.toBigInt());
```

The provable part is achieved with the `verify` method.

```
// verify the signature, returns a Bool
indicating whether the signature is valid or
not
let isValid: Bool = signature.verify(message,
publicKey);
```

See Example

ECDSA API

See API