

Introduction to Protokit

Building privacy-enabled blockchains using a Typescript zkDSL

@proto_kit

👉 You can find these slides on our twitter

From zero to zkHero

All you need to know to start building with zk!

What is a zk-proof?

Mathematical **proof of computation**^[1] using public or private inputs

Might include a public output too

Computation might include **signature verification, merkle tree inclusion**, and more!

To compute a proof, **you need a proof system, we use Kimchi**^[2] (same as MINA L1)

Proof is a **result of circuit execution** - which determine what constraints create a valid proof

When a circuit is compiled, we get a prover and a verifier/verification key

Validity of a proof can be cheaply verified using a verification key

Zero-knowledge-ness is achieved by not exposing the private inputs to the verifier

[Wikipedia: Zero-knowledge proofs](#) ↗

[Mina book: Kimchi](#) ↗

Meet o1js, zkDSL for writing circuits

```
1  import { Experimental } from "o1js";
2
3  const counter = Experimental.ZkProgram({
4    publicInput: Field,
5    publicOutput: Field
6    methods: {
7      increment: {
8        privateInputs: [UInt64, UInt64],
9        method: (publicInput: Field, a: UInt64, b: UInt64): Field => {
10          const hash = Poseidon.hash(a.toFields());
11          publicInput.assertEquals(hash, "Uh-oh, you're using the wrong number!");
12          const incremented = a.add(b);
13          return Poseidon.hash(incremented.toFields());
14        }
15      },
16    }
17  })
```

Running our circuit

```
1  await counter.compile(); // creates prover & verifier artifacts
2
3  const add = UInt64.from(1);
4  let state = UInt64.from(0);
5  let publicInput = Poseidon.hash(state.toFields());
6
7  const proofOfIncrement = await counter.increment(publicInput, state);
8
9  // keep track of the state, so we can run counter.increment(...) again
10 publicInput = proofOfIncrement.publicOutput; // h(state + 1)
11 state = state.add(add); // state + 1
12
13 const verified = await counter.verify(proofOfIncrement); // true
```

Power of recursive proofs

```
1  const credential = Experimental.ZkProgram({ ... });
2  class CredentialProof extends Experimental.ZkProgram.Proof(credential) {}
3
4  const counter = Experimental.ZkProgram({
5    publicInput: Field,
6    publicOutput: Field
7    methods: {
8      add: {
9        privateInputs: [CredentialProof, UInt64, UInt64],
10       method: (
11         publicInput: Field,
12         credentialProof: Proof<Field, Field>,
13         a: UInt64,
14         b: UInt64
15       ): Field => {
16         credentialProof.verify(); // yet another computation to prove
17         // ... rest of our add() method
18       }
19     }
20   })
21 })
```

How is this helpful for blockchains?

Blockchain is effectively a **spicy distributed database with consensus**

Nodes execute user transactions - authorized intent to perform an action on top of the chain data

Results of the transactions are bundled into a block

Contents of a **block can be verified by re-executing the contents** of it (transactions)

This approach imposes certain *(high?)* **hardware requirements, which in turn impacts decentralization**

MINA's protocol^[1] takes a novel approach, by **turning the aforementioned processes into circuits**

Block is now a single recursive block proof 🎉

This allows MINA nodes to **verify the computation that resulted into a block(proof)** very cheaply

[Mina protocol architecture](#) ➡

MINA's native L1 smart contracts

Smart contracts on Mina (1/3)

Built using o1js's `SmartContract`` (also known as zkApps)

Smart contract is just a **circuit that produces a proof of its own execution**


Executed off-chain, keeps the L1 node hardware requirements to the "minimum"

Transactions **don't tell the node "what code to execute" anymore**

Instead a **transaction contains a list of pre-authorized account updates**

Pre-authorized either by a signature, or a proof

Block producer just verifies the signature/proof and applies account updates to the chain state

Applying all account updates results into a block proof 

This approach results into **potential race conditions**, since users have no built-in way of organizing off-chain

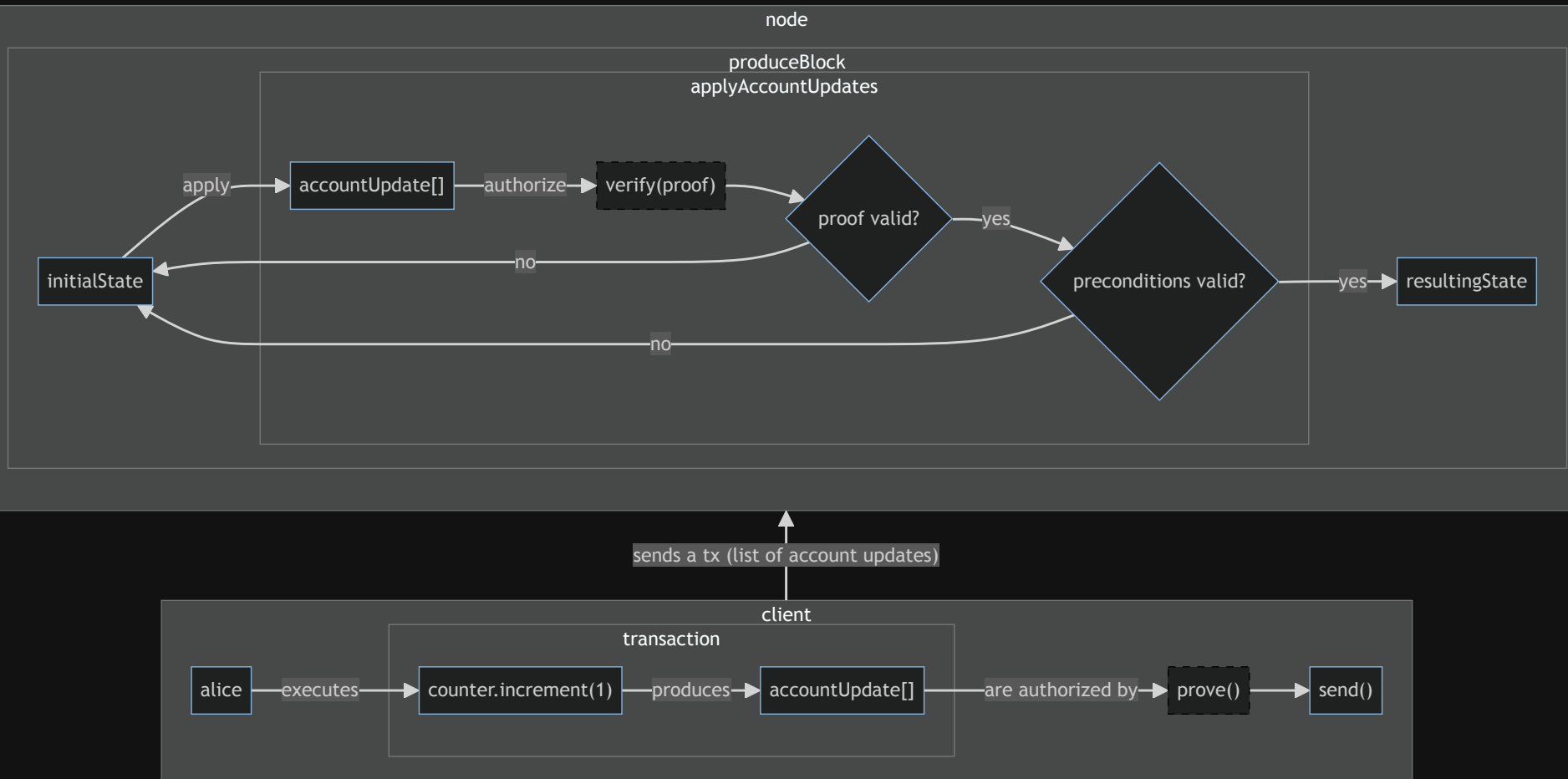
Smart contracts on MINA (2/3)

```
1  class Counter extends SmartContract {
2      // ⚠ storage worth of 8 Fields is available (8*256bits total)
3      @state(Field) public commitment = State();
4
5      @method()
6      public increment(a: UInt64, b: UInt64) {
7          const hash = Poseidon.hash(a.toFields());
8          const commitment = this.commitment.get();
9          this.commitment.assertEquals(commitment); // create a precondition
10
11         commitment.assertEquals(hash, "Uh-oh, you're using the wrong number!");
12
13         const incremented = a.add(b);
14         const incrementedCommitment = Poseidon.hash(incremented.toFields());
15
16         this.commitment.set(incrementedCommitment); // update the on-chain state
17     }
18 }
```

Smart contracts on MINA (3/3)

```
1  const counter = new Counter(address);
2
3  // create account updates for the transaction, by executing the contract code
4  const tx = Mina.transaction(feePayer, () => {
5    counter.increment(UInt64.from(1));
6  });
7  // account updates:
8  // [{ address, preconditions: [commitment], state: [incrementedCommitment] authorization: proof }]
9
10 // prove the code execution
11 await tx.prove();
12
13 // send it to the node/mempool
14 await tx.send();
```

Smart contracts on MINA (4/4)



Key-value storage in smart contracts

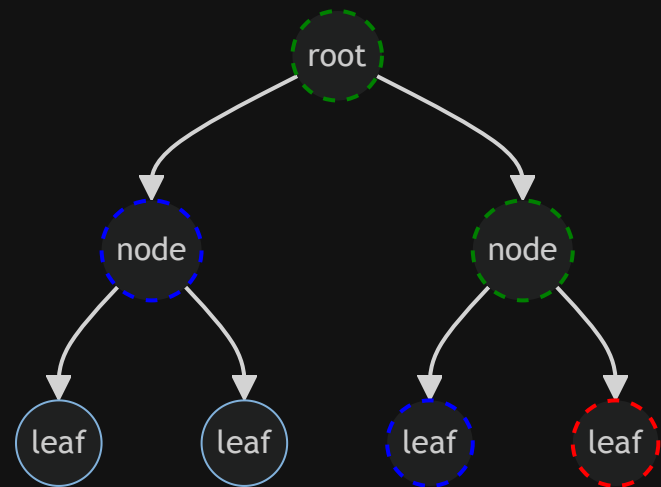
Map-like storage is not possible with the L1 smart contracts out of the box:

```
1  class Counters extends SmartContract {
2    // ❌ this is NOT possible with the L1 smart contract
3    @state(PublicKey, UInt64) public counters = State();
4
5    @method()
6    increment(for: PublicKey, by: UInt64) {}
7  }
```

You can **store a commitment to a merkle tree's root hash** instead:

```
1  class Counters extends SmartContract {
2    // ✅ this is possible with the L1 smart contract
3    @state(Field) public countersCommitment = State();
4
5    @method()
6    increment(for: PublicKey, by: UInt64) {}
7  }
```

What is a merkle tree?



blue = provided by a witness, **red** = provided by you, **green** = computed by you

TL;DR; a data structure that allows us to **cryptographically verify**

that a leaf/node is a part of a tree, by computing the root hash using a merkle witness.

MerkleMap to the rescue

Merkle tree leafs can be identified by an index

If you use a merkle tree with a *depth* of 256 (size), it means there's 2^{256} leafs

This means the leaf index ranges from 0- 2^{256} , therefore can be represented by a single field

You can now think of a tree like of a key-value storage, where key = field, value = field, root = field

This key-value API is called MerkleMap

```
1  const map = new MerkleMap();
2  const key = Poseidon.hash(PrivateKey.random().toPublicKey().toFields()) // Field
3  const value = Poseidon.hash(UInt64.from(1000).toFields()); // Field
4
5  // store a hash of values in the tree
6  map.set(key, value);
7
8  const witness = map.getWitness(key); // MerkleMapWitness
9  const [computedRoot, computedKey] = witness.computeRootAndKey(value); // [Field, Field]
```

Reading tree data

```
1  class Counters extends SmartContract {
2      @state(Field) public countersCommitment = State();
3
4      @method()
5      increment(for: PublicKey, by: UInt64) {
6          const countersCommitment = this.countersCommitment.get();
7          this.countersCommitment.assertEquals(); // precondition to countersCommitment
8
9          const forKey = toField(for); // returns h(for)
10         const [forCounter, forCounterWitness] = getCounter(forKey); // returns [UInt64, MerkleMapWitness]
11
12         const [computedCountersCommitment, computedForKey] =
13             forCounterWitness.computeRootAndKey(toField(forCounter));
14
15         // check if the provided data is valid, by checking if its part of the tree
16         countersCommitment.assertEquals(countersCommitment, "Uh-oh, invalid root hash!");
17         forKey.assertEquals(computedForKey, "Uh-oh, invalid key!");
18
19         // ... rest of the increment logic
20     }
```


Updating tree data

```
1  class Counters extends SmartContract {
2      @state(Field) public countersCommitment = State();
3
4      @method()
5      increment(for: PublicKey, by: UInt64) {
6          // ... our prior increment logic is here
7
8          const newForCounter = forCounter.add(by);
9          // compute a new tree root hash, using the new balance as a leaf value
10         const [newCountersCommitment] = forCounterWitness.computeRootAndKey(toField(newForCounter));
11
12         // update the on-chain commitment
13         this.countersCommitment.set(newCountersCommitment);
14
15         // ... rest of the increment logic
16     }
```

Design limitations

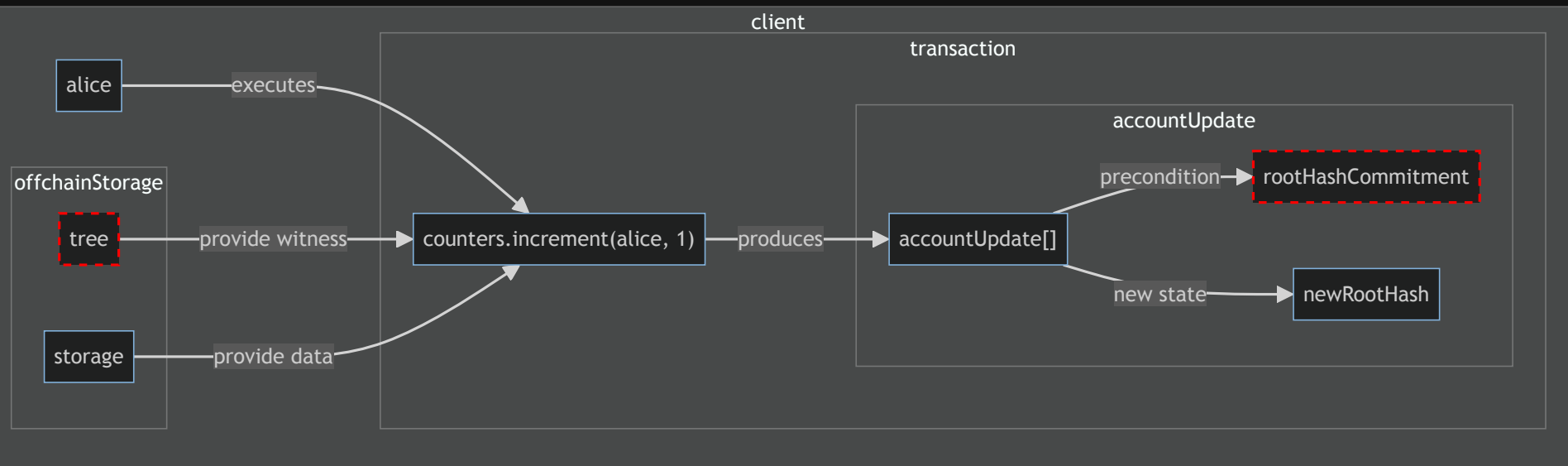
Users **execute smart contracts on the client side**, independently of other clients

This **leads to preconditions pointing to the root hash of the tree**

If there are **two users transferring at the same time, their preconditions will be the same**

After **one transaction succeeds, the on-chain state is updated** (commitment changes)

This **invalidates the second transaction, since it now has an outdated precondition**



Building a private airdrop on MINA

Architecture

Airdrop merkle tree, stores all addresses eligible for airdrop

Everyone receives the same amount of tokens from the airdrop

Airdrop smart contract, that tokens can be claimed from only if **the user can prove they are part of the airdrop tree**

In order to prevent double-claims, we must also **use a cryptographically sound nullifier** that **can't be traced back to the user's address**.

Smart Contract (1/2)

```
1  class Airdrop extends SmartContract {
2      @state(Field) public airdropTreeCommitment = State()
3      @state(Field) public nullifierTreeCommitment = State()
4
5      @method()
6      public claim(airdropWitness: MerkleMapWitness, nullifier: Nullifier, nullifierWitness: MerkleMapWitness) {
7          const airdropTreeCommitment = this.airdropTreeCommitment.getAndAssertEquals();
8          const nullifierTreeCommitment = this.nullifierTreeCommitment.getAndAssertEquals();
9
10         // check if user is part of the airdrop
11         const [computedAirdropTreeCommitment, computedAirdropKey] = airdropWitness.computeRootAndKey(
12             Bool(true).toField()
13         );
14
15         airdropTreeCommitment.assertEquals(computedAirdropTreeCommitment)
16         computedAirdropKey.assertEquals(Poseidon.hash(nullifier.getPublicKey().toFields()))
17
18         // ... rest of the claim logic
19     }
20 }
```

Smart Contract (2/2)

```
1  class Airdrop extends SmartContract {
2      @method()
3      public claim(airdropWitness: MerkleMapWitness, nullifier: Nullifier, nullifierWitness: MerkleMapWitness) {
4          // ... earlier claim logic
5
6          // check the nullifier has not been yet used
7          const [computedNullifierTreeCommitment, computedNullifierKey] = nullifierWitness.computeRootAndKey(
8              Field(0)
9          );
10
11         nullifierTreeCommitment.assertEquals(computedNullifierTreeCommitment)
12         nullifier.key().assertEquals(computedNullifierKey)
13
14         const [updatedNullifierTreeCommitment] = nullifierWitness.computeRootAndKey(
15             Bool(true).toField()
16         );
17
18         this.nullifierTreeCommitment.set(updatedNullifierTreeCommitment);
19         // airdrop the tokens
20         this.send({ to: this.sender, amount: UInt64.from(1000) });
21     }
22 }
```

Protokit

Protocol development framework for privacy enabled application chains
(a.k.a. zkRollups)

```
$ npx degit proto-kit/starter-kit#develop my-app-chain
```

What is Protokit?

Typescript based framework for building zkChains

Not another zkEVM, uses a **custom-tailored zkVM** instead

Hybrid execution model, both off and on chain

Off-chain execution = **client side zk-proofs** 🧐

End to end proven execution, no sequencer operator shenanigans

Customizable and modular (not monolithic 🗑️)

Virtually no learning curve (you should still pay attention tho 🙄)

Build your own zkRollup

Recursive zk-rollup is a series of **recursive circuits**, that prove transaction execution and rollup state transitions

L1 offers ***sequence state*** and ***actions*** to help you build zk-rollups, but **you still have to write your own circuits!**

Protokit enables developers to **build zero-knowledge, interoperable and privacy preserving application chains** with a minimal learning curve.

You can **roll your own zkRollup** in a few lines of code.

zkRollups are a **superset of zkApps**, combining **opt-in privacy features with on-chain execution** for the best-in-class user experience.

What is a Protokit zkRollup made of?

Runtime = application logic

Protocol = underlying VM, block production, transaction fees, account state, ...

Sequencer = mempool, block production, settlement to L1

L1 settlement contract - an L1 contract validating your rollup's block proofs

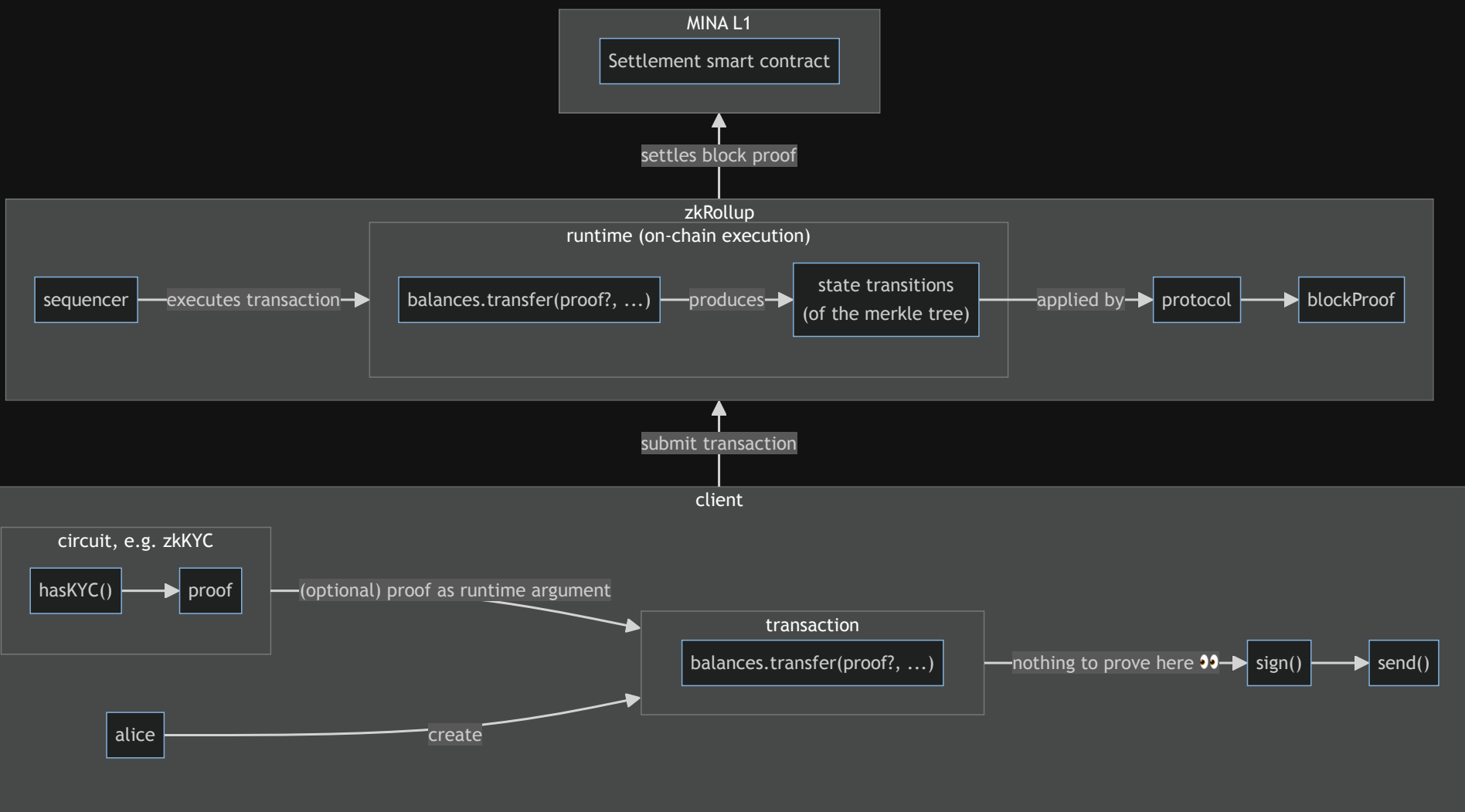
👉 You can replace any module of your app-chain, want a different mempool implementation? Just write one!

Features

- 🔌 Supercharged developer experience, it's just Typescript (and a little bit of o1js)
- 📖 Reusable modules, found an open-source module you like? ``npm install <module-name>``
- 🏎️ No data race conditions, thanks to sequenced transactions (throughput goes brrr)
- 💿 "Bottomless" on-chain storage, including a key-value storage (``State/StateMap``)
- 👤 Preserved privacy using off-chain proofs as arguments for on-chain transactions
- 🔧 Focus on business logic, we've done the heavy lifting for you. (looking at you, merkle trees 🌴)
- 👛 Integrated with user wallets, thanks to ``mina-signer``
- 🤖 Not another zkEVM, but a succinct zkVM instead
- 🧐 You can implement the entire zkApps MIP if you want, or just customize the runtime to fit your needs
- 🔥 Iterate your on-chain logic faster than with the L1, feature is king

soon™:

- 🏠 Interoperable with L1s and other L2s (with varying degrees of security guarantees)
- 🍷 Data availability (don't tell anyone just yet)



Writing your own runtime

```
1  @runtimeModule()
2  class Balances extends RuntimeModule<unknown> {
3    // "unlimited" on-chain state
4    @state() public balances = StateMap.from(PublicKey, UInt64);
5
6    // on-chain executed business logic
7    @runtimeMethod()
8    public transfer(from: PublicKey, to: PublicKey, amount: UInt64) {
9      const fromBalance = this.balances.get(from); // produces a state transition
10     const toBalance = this.balances.get(to);
11
12     const fromBalanceIsSufficient = fromBalance.greaterThanOrEqual(amount);
13
14     assert(fromBalanceIsSufficient, errors.fromBalanceInsufficient()); // soft-fails circuit execution
15
16     const newFromBalance = fromBalance.sub(amount);
17
18     this.balances.set(newFromBalance); // produces a state transition
19
20     // ... rest of the transfer logic
21   }
22 }
```

Testing your runtime

```
1  const alicePrivateKey = PrivateKey.random();
2  const alicePublicKey = alicePrivateKey.toPublicKey();
3
4  const bobPrivateKey = PrivateKey.random();
5  const bobPublicKey = bobPrivateKey.toPublicKey();
6
7  const appChain = TestingAppChain.fromRuntime({
8    modules: {
9      Balances,
10    },
11    config: {
12      Balances: {},
13    },
14  });
15
16  await appChain.start();
17  appChain.setSigner(alicePrivateKey);
```

Executing transactions

```
1  const balances = appChain.runtime.resolve("Balances");
2  const tx = appChain.transaction(alicePublicKey, () => {
3    balances.transfer(alice, bob, UInt64.from(100));
4  });
5
6  await tx.sign();
7  await tx.send();
8
9  await appChain.produceBlock();
10
11 const bobBalance =
12   await appChain.query.runtime.Balances.balances.get(bobPublicKey);
13 // ^ UInt64 = 100, assuming we minted sufficient balance for Alice before (not shown)
```

Building a private airdrop runtime

```
$ npx degit proto-kit/starter-kit#develop private-airdrop
```

Additional content

Examples of runtime usage

Modularity examples

Query api modularization

In memory signer / Auro signer

Protocol module examples

tx fees

Account state

What to expect in the future