Lesson 8 - Mina

Today's topics

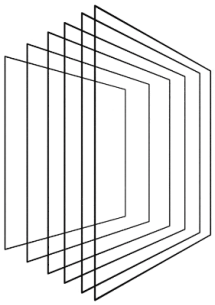
- Mina overview
- How proofs are created
- O1js library

Mina

Mina Blockchain

22KB[1]

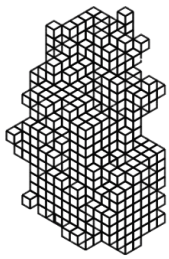
FIXED SIZE



Other Blockchains

300GB[2]

INCREASING SIZE



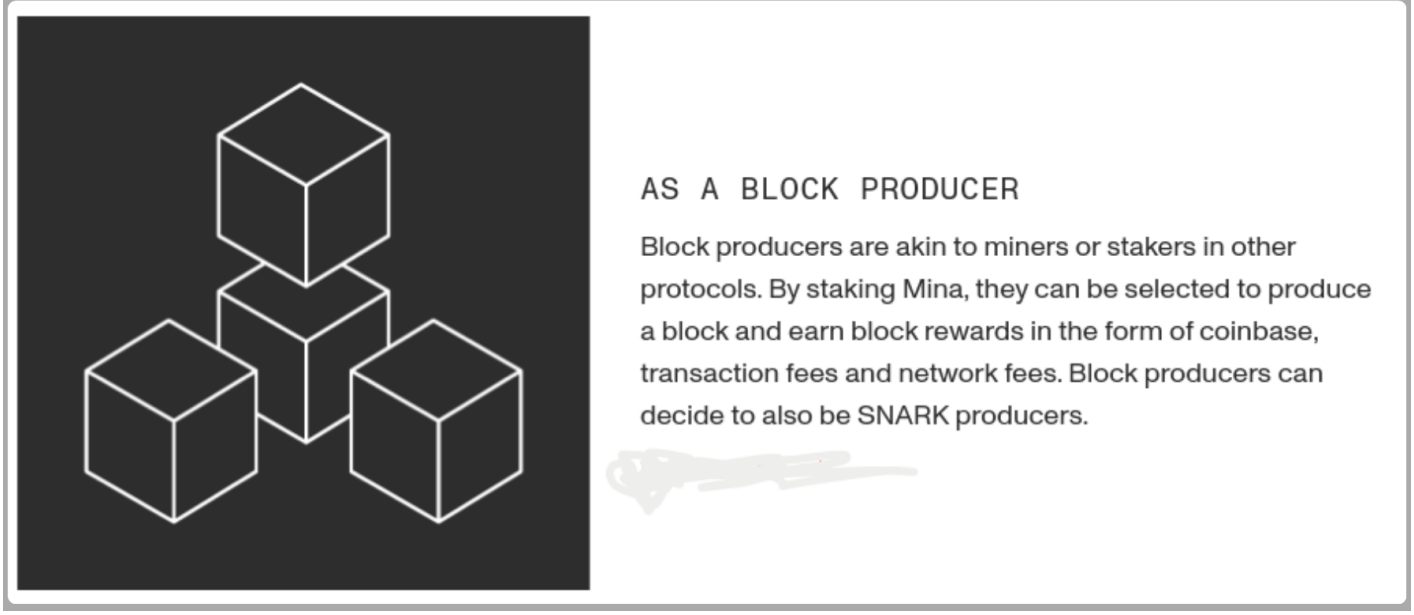
Mina is the first cryptocurrency protocol with a **succinct** blockchain. With Mina, no matter how much the usage of the network grows, the blockchain always stays the same size **- about 22kb**.

What data is needed for usable representation of the blockchain?

- A clear, usable representation of the basic data being queried of a state - accounts balances
- Data that a node needs in order to verify that this state is real in a trustless manner
- The ability to broadcast transactions on the network to make a transfer

Roles in Mina

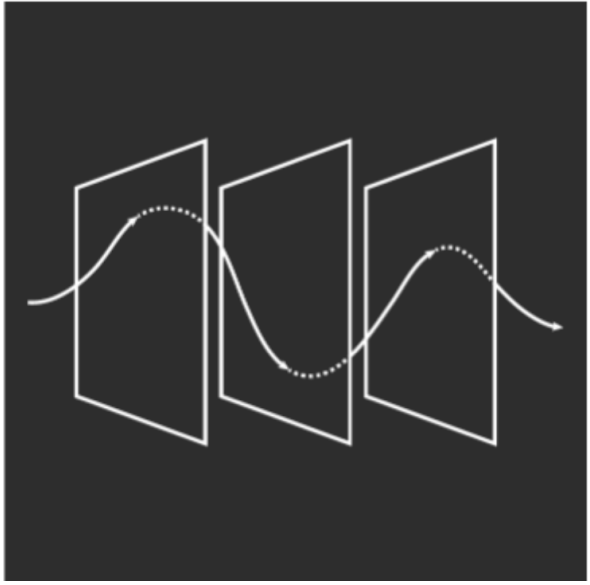
Block producer



A block in Mina is constituted of:

- Protocol state
  - Genesis state hash
  - Blockchain state
  - Consensus state
  - Consensus constants
- Protocol state proof
- Staged ledger diff
- Delta transition chain proof
- Current protocol version
- Proposed protocol version

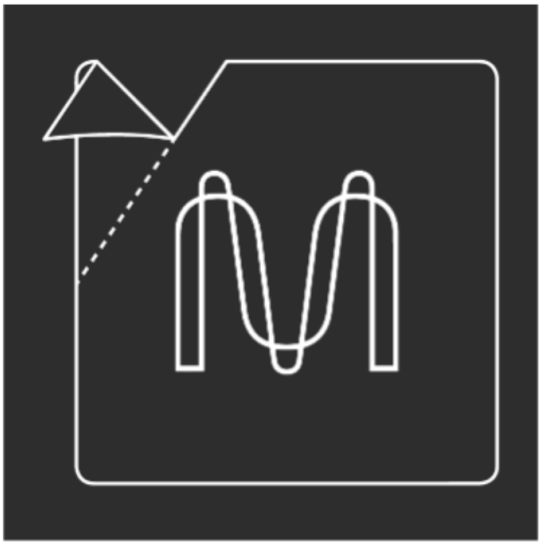
Snark worker



AS A SNARK PRODUCER

The second type of consensus node operator on Mina, snark producers help compress data in the network by generating SNARK proofs of transactions. They then sell those proofs to block producers in return for a portion of the block rewards.

Professional block producer



AS A PROFESSIONAL BLOCK PRODUCER

Because staking requires nodes to be online, some may choose to delegate their Mina to staking pools. These groups run staking services in exchange for a fee, which is automatically deducted when the delegator gets selected to be a block producer.

See guide

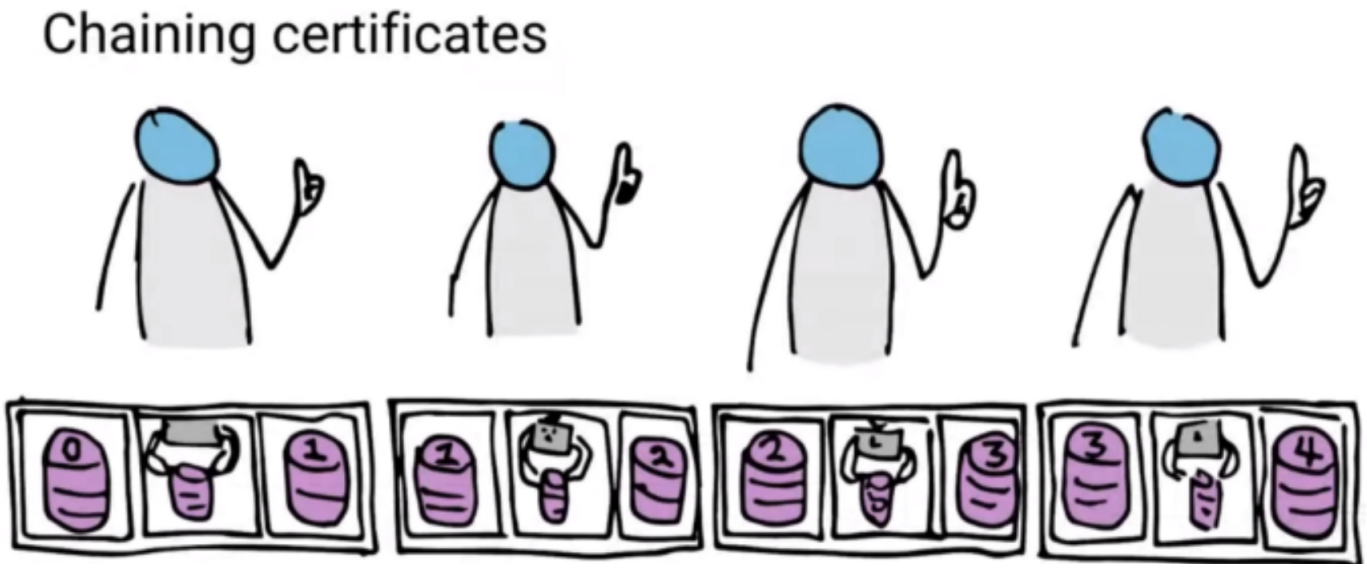
Mina's consensus mechanism

Mina's consensus mechanism is an implementation of Ouroboros Proof-of-Stake. Due to Mina's unique compressed blockchain, certain aspects of the algorithm have diverged from the Ouroboros papers, and the version Mina uses is called **Ouroboros Samisika** VRFs are used to decide whether to produce a block, the probability is proportional to the producer’s stake.

RECURSIVE COMPOSITION OF PROOF (SUCCINCTNESS)

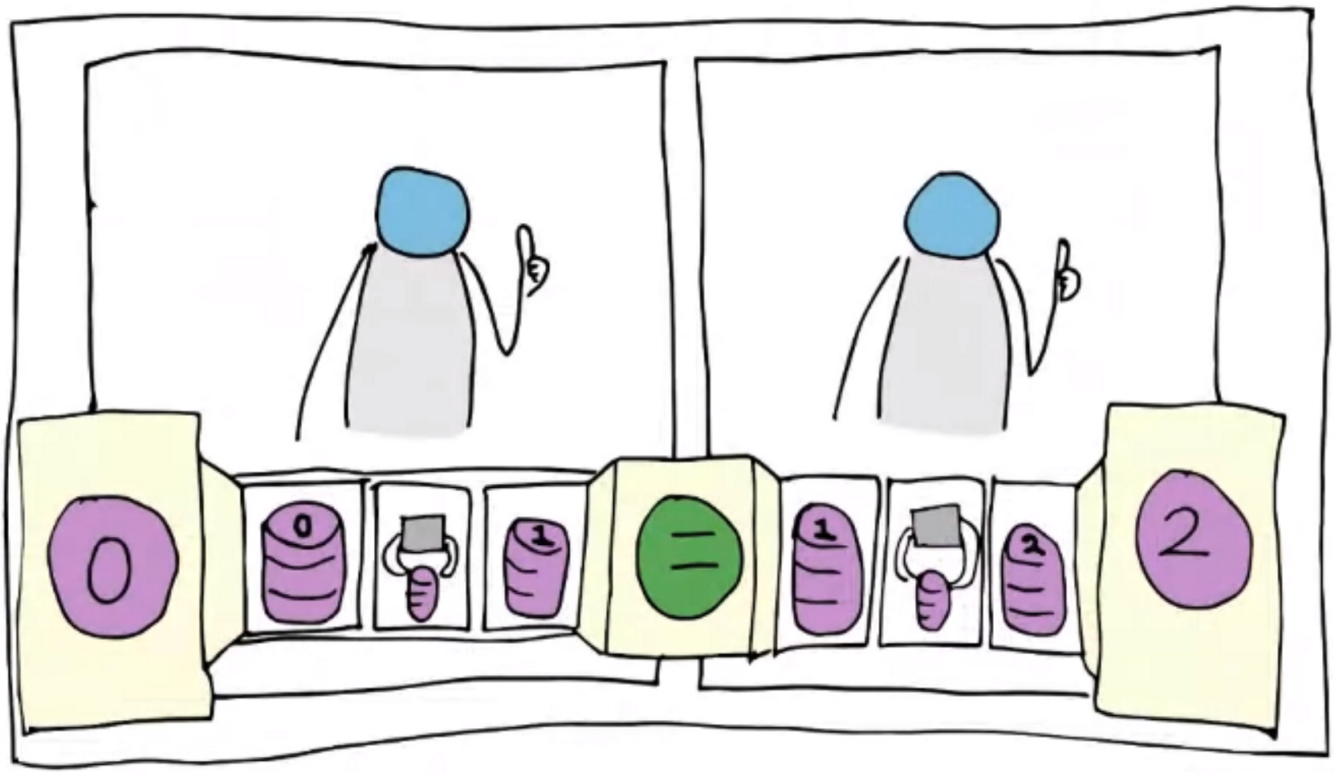
The blockchain is dynamic and new blocks keep getting added to it. However, we would like to ensure succinctness at any given point in time. Therefore, as the blockchain “grows”, we compute a new SNARK proof that not only validates the new blocks, but also the existing SNARK proof itself. The notion of a SNARK proof that attests to the verifiability of another SNARK proof is the notion of “incrementally-computable SNARK”

from Mina: Decentralized Cryptocurrency at Scale (whitepaper)



Better to use recursive proofs :

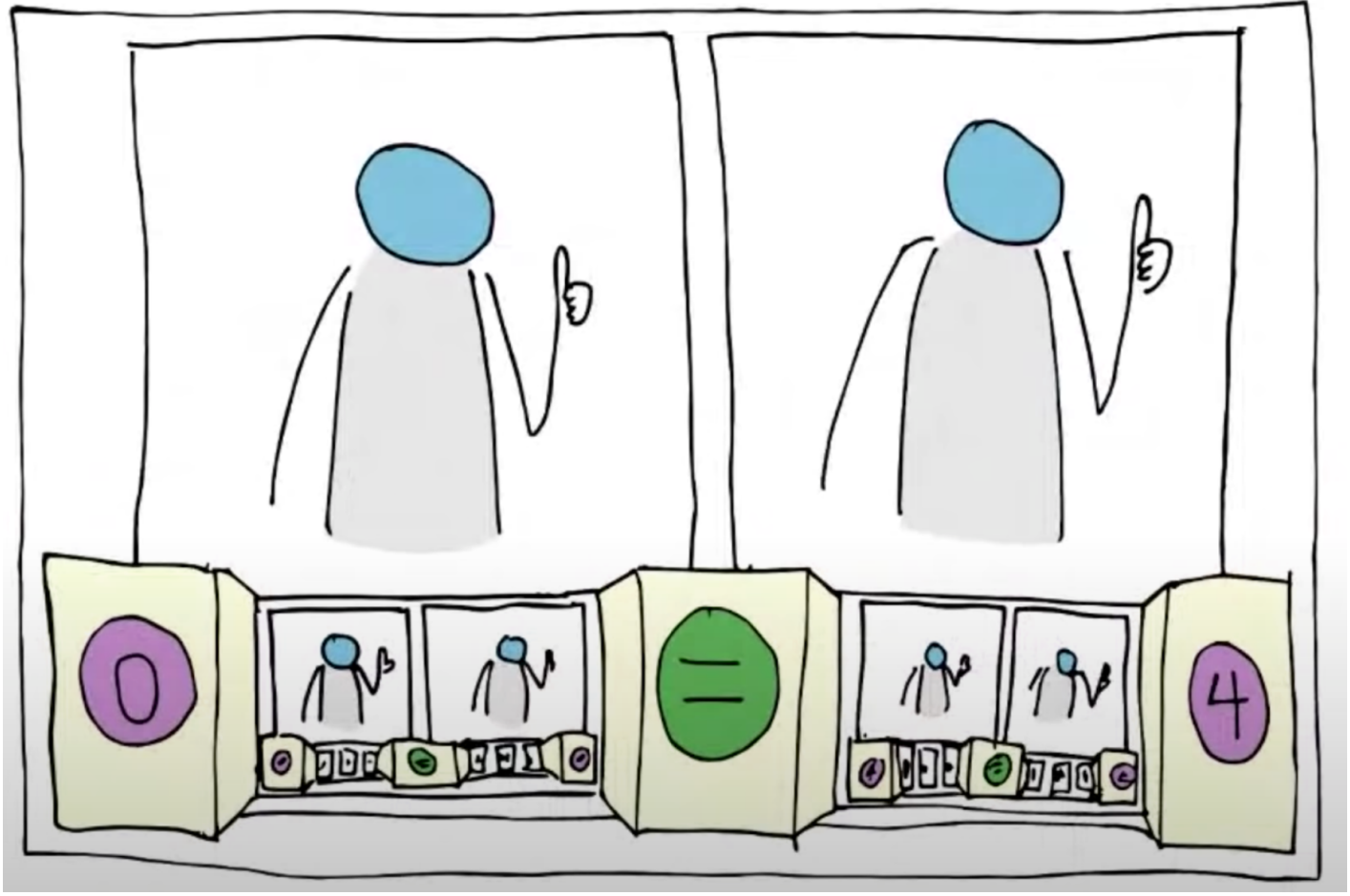
Thus, “You can get from **DB 0** to **DB 2**”



It’s a SNARK, so still ~1 kB

“You can get from
DB 0 to **DB 2**”

“You can get from
DB 2 to **DB 4**”

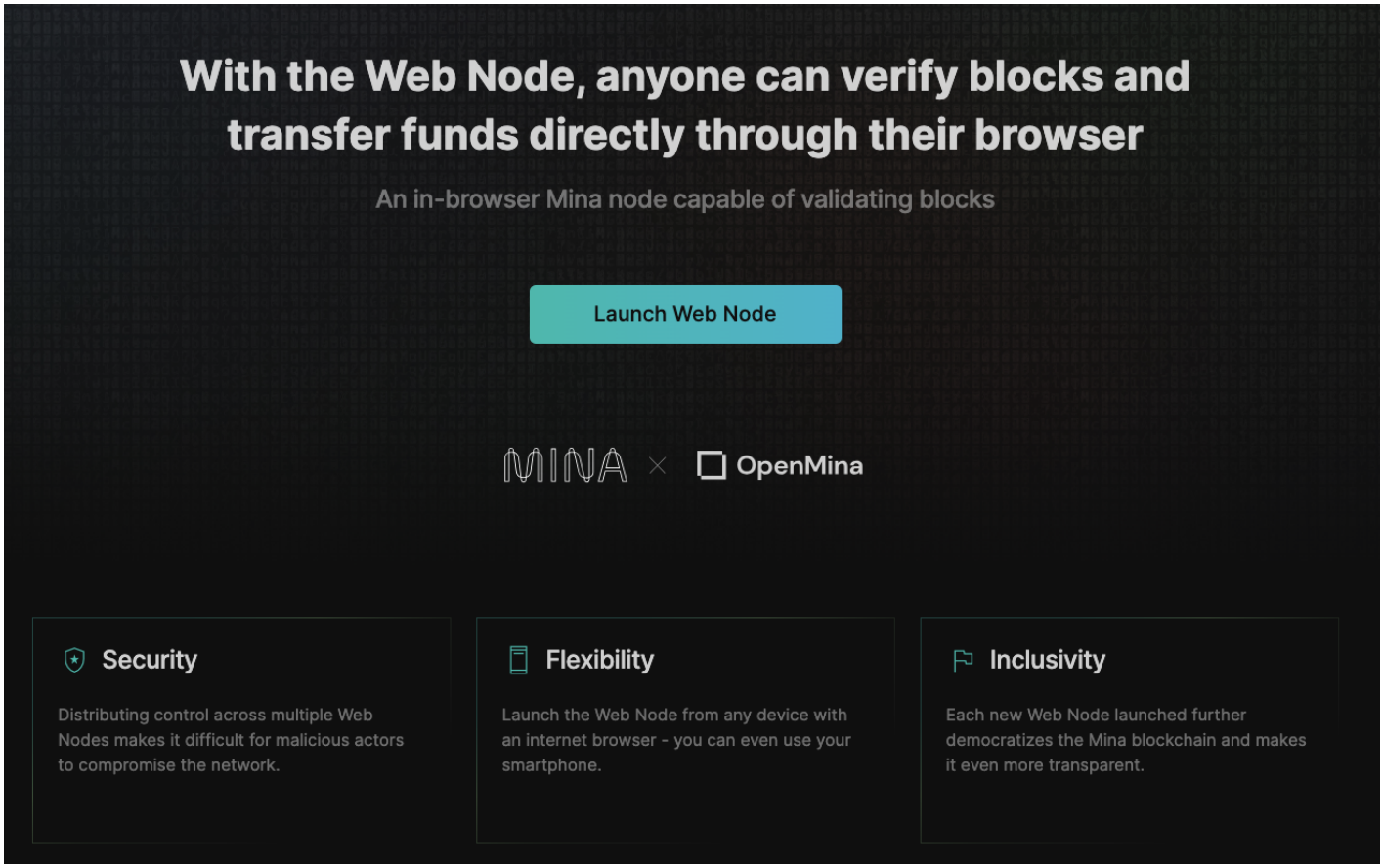


See video

Running Mina in a browser

See Open Mina

See article



Community

https://awesome.mina.tools/

https://minacrypto.com/

Mina Road Map

See Map

Navigator Program

See Details

There will be three phases of the Navigator program:

1. **Hackathon:** October 9 through November 10
2. **Learn-to-earn challenges:** November 2023 through April 2024
3. **Final contribution grant:** April 30, 2024

Phase Two: Build and Learn-to-Earn (Or should we say, Navigate-to-Earn?)

This is when the fun begins. Following the hackathon, Navigators will get to collaborate on their own or other's hackathon projects while being eligible for monthly grants by participating in a 6-month series of challenges to accelerate their learning about ZK, Mina, and o1js.

Phase Three: Navigator Final Contribution Grant Award

After six months of dedication and hard work, Navigators will have the chance to showcase their

contributions to a panel of community electors for grand prizes of MINA grants. Prizes will not be project-specific and will focus on individual contributions across multiple projects and GitHub repos.

The Mina ecosystem is in a dynamic state of growth. The Navigator Program emerges at a crucial juncture, synergizing with other impactful initiatives:

- **Mainnet Upgrade & Incentivized Testnet:** Mina is in its final testing stages before enabling easier programmable zkApps and other advancements in the protocol go live on mainnet.
- **zkIgnite Cohorts:** Forging a path for ZK businesses on Mina.
- **Cross-chain integration:** LambdaClass, =nil; Foundation, Protokit, and Zeko are all working towards building infrastructure to make Mina and its powerful ZK properties interoperable with other chains

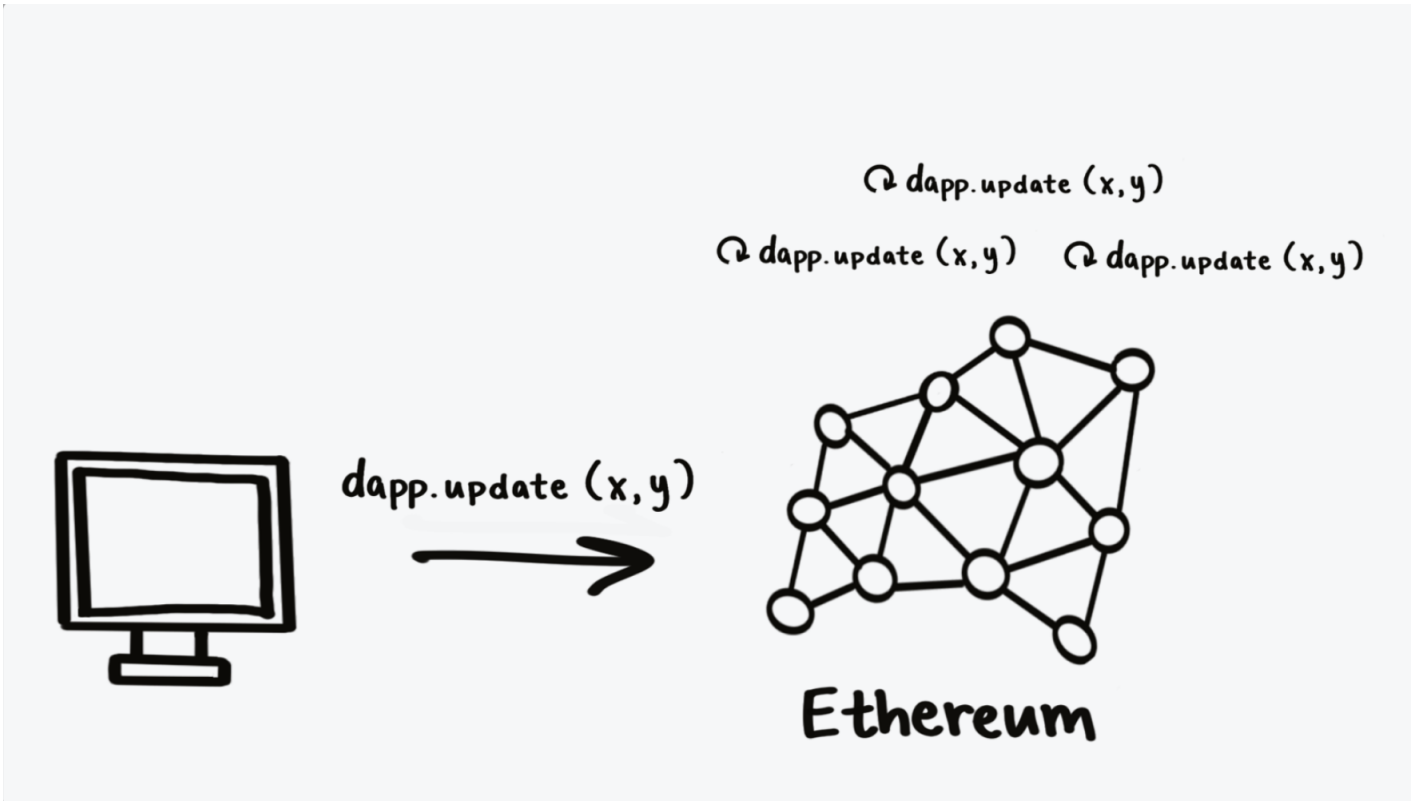
Staking on Mina

See guide

zkApps

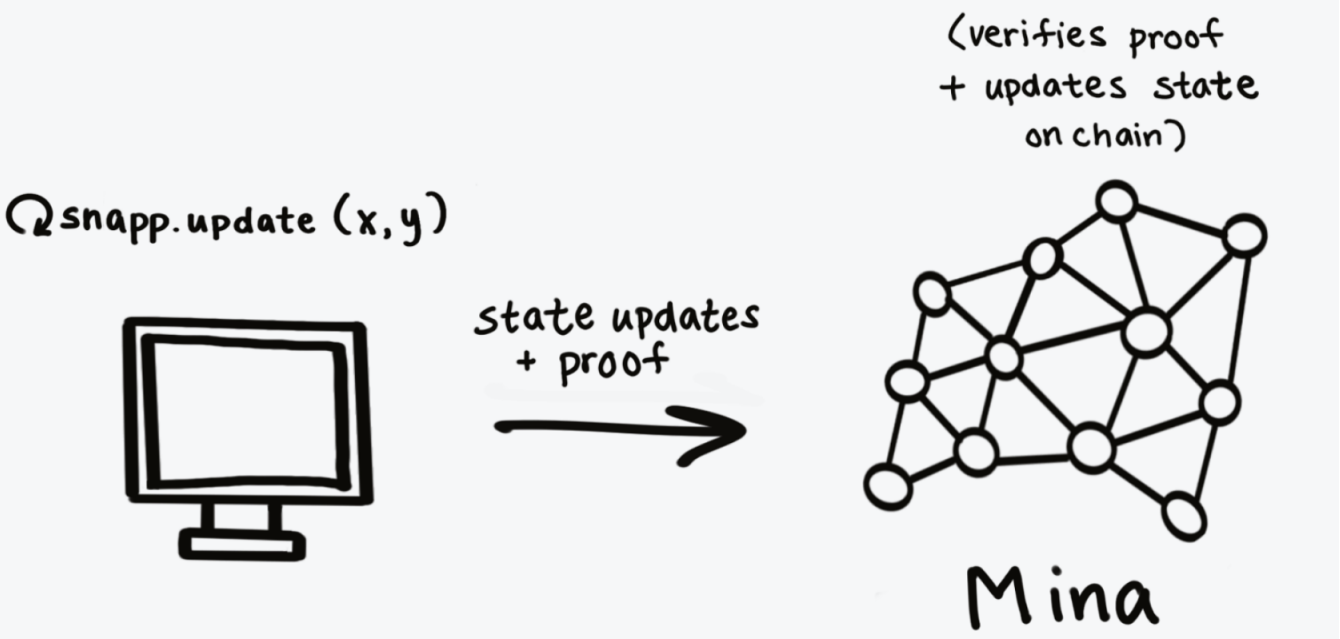
zkApps ("zero-knowledge apps") are Mina Protocol's smart contracts that use an off-chain execution and mostly off-chain state model.
This allows for private computation and state that can be either private or public.

Ethereum dapps

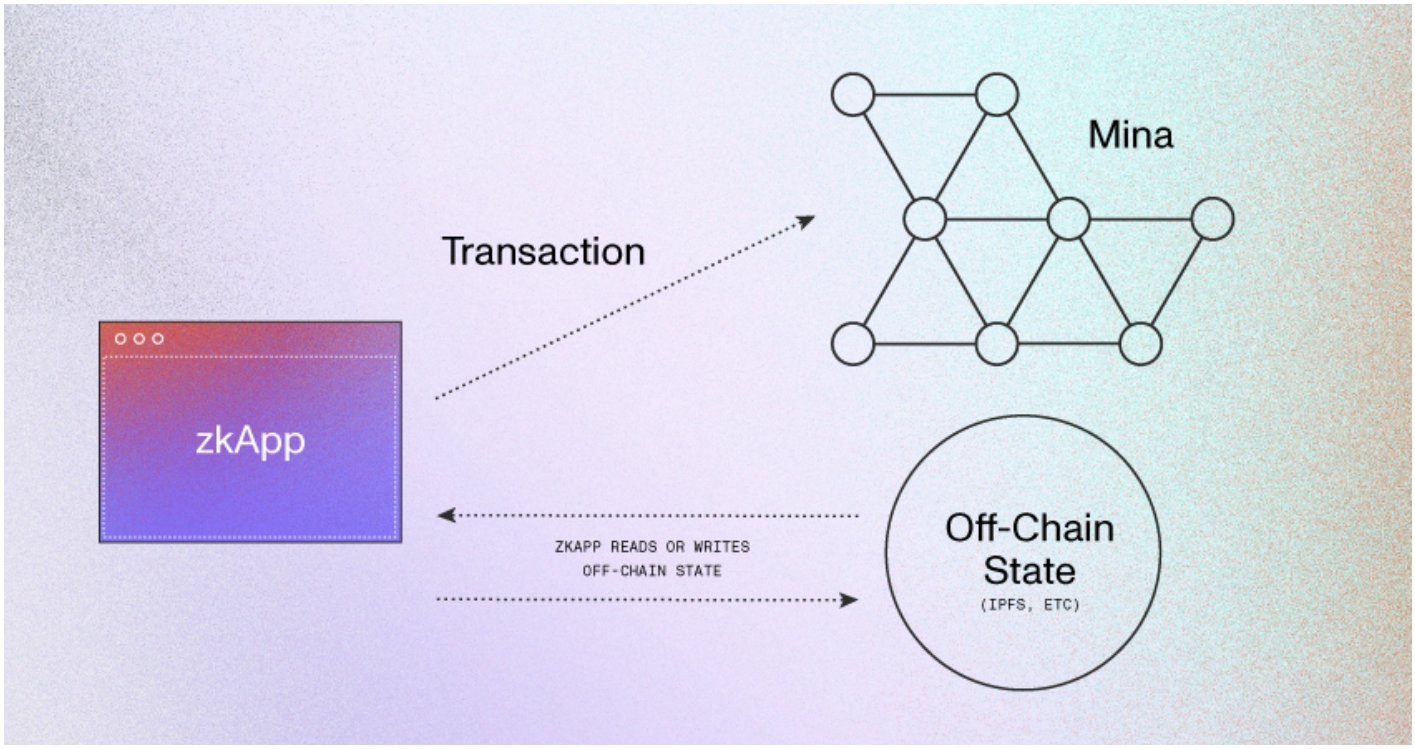


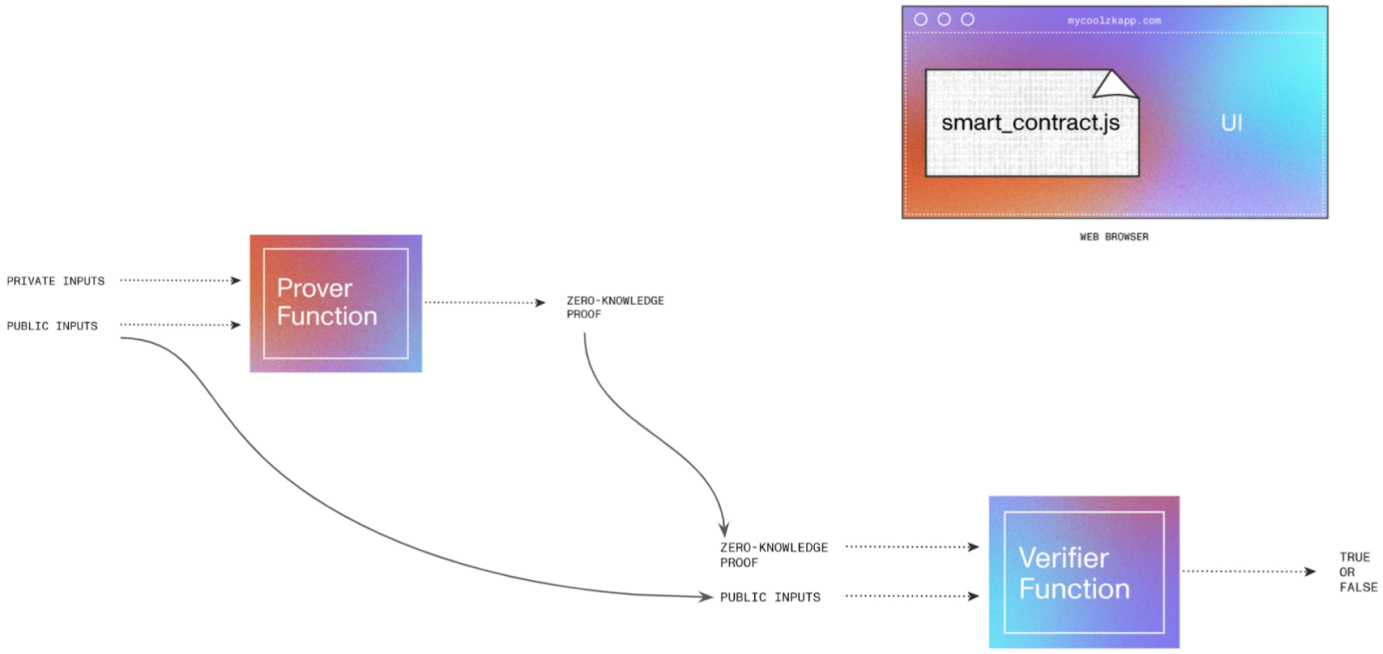
Ethereum uses **on-chain computation**.

Mina zkApps


(verifies proof
+ updates state
on chain)
snapp.update (x, y)
state updates
+ proof
Mina

Mina uses **off-chain computation & on-chain verification**.





zkApps, short for "zero-knowledge applications," are decentralized applications (dApps) built on the Mina Protocol using zero-knowledge proofs. These proofs allow the verification of the correctness of a computation without revealing the actual inputs or outputs. This leads to increased privacy and scalability compared to other blockchain-based dApps.

Mina Protocol leverages zk-SNARKs (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge) to achieve this. zk-SNARKs enable the network to maintain a small, fixed-sized proof called a "succinct blockchain" instead of a large, ever-growing ledger. This results in lower storage and processing requirements, making it easier for users to run nodes and participate in the network.

Developing a zkApp involves the following steps:

1. Write the logic: Define the zkApp's business logic using SnarkyJS, a high-level JavaScript library for writing zero-knowledge proofs. SnarkyJS provides functions and data types for expressing the application's rules and creating zk-SNARK proofs.

2. Deploy the zkApp: Use the Mina CLI (Command Line Interface) tool to deploy your zkApp on the Mina network. This process involves uploading the compiled SnarkyJS code, specifying the on-chain values, and paying a deployment fee.
3. Interact with the zkApp: Users can interact with the zkApp by sending transactions to it. These transactions will include the necessary zk-SNARK proofs to ensure the privacy and correctness of the computation. The zkApp will update its state based on these transactions.

Advanced SnarkyJS features:

- On-chain values: zkApps can store on-chain values using a key-value storage system. These values can be updated by proving the correctness of an update operation using zk-SNARKs.
- Custom tokens: zkApps can create and manage custom tokens, enabling developers to build a wide range of financial applications, including stablecoins, governance tokens, and more.

- Time-locked accounts: To build applications that require time-based conditions, SnarkyJS can be used to create time-locked accounts. These accounts allow for the release of funds only after a specific period has passed.

In summary, zkApps are decentralized applications built on the Mina Protocol that leverage zero-knowledge proofs to ensure privacy and scalability. Developers can use SnarkyJS to write the application logic, and deploy it on the Mina network using the Mina CLI tool. Advanced features such as on-chain values, custom tokens, and time-locked accounts enable the creation of a wide range of dApps with various use cases.

Deployment:

To deploy a zkApp on the Mina network, developers need to follow these steps:

1. Compile the SnarkyJS code: Before deploying, the SnarkyJS code must be compiled into a format that can be executed on the Mina Protocol. This is usually done using the SnarkyJS compiler.

2. Upload the compiled code: After compiling the code, it needs to be uploaded to the Mina network. This is done using the Mina CLI (Command Line Interface) tool. The developer specifies the compiled code's file path and provides a public key for the zkApp.
3. Specify on-chain values: During deployment, the developer must provide initial on-chain values for the zkApp. These values are crucial for setting up the application state and can be updated later by submitting transactions containing zk-SNARK proofs.
4. Pay the deployment fee: To deploy a zkApp, the developer must pay a deployment fee in Mina tokens. This fee is used to incentivize network validators for processing and storing the zkApp.

Interaction:

Users interact with zkApps by sending transactions containing zk-SNARK proofs. These proofs ensure the privacy and correctness of the computation. The zkApp updates its state based on the submitted transactions, while network validators verify the proofs and update the blockchain accordingly.

Limitations:

1. Performance: While zk-SNARKs offer privacy and scalability benefits, generating and verifying proofs can be computationally intensive. This may limit the complexity and efficiency of some zkApps compared to traditional smart contracts.
2. Development complexity: Writing zkApps requires understanding zero-knowledge proofs and using specialized languages like SnarkyJS. This may create a steeper learning curve for developers compared to more established languages like Solidity.
3. Ecosystem maturity: The Mina Protocol and the zkApp ecosystem are relatively new compared to more established platforms like Ethereum. This may result in fewer tools, libraries, and community resources available for developers.

Difference between zkApps and Solidity smart contracts:

1. Privacy: zkApps leverage zero-knowledge proofs to ensure privacy, while Solidity smart contracts on Ethereum are transparent by

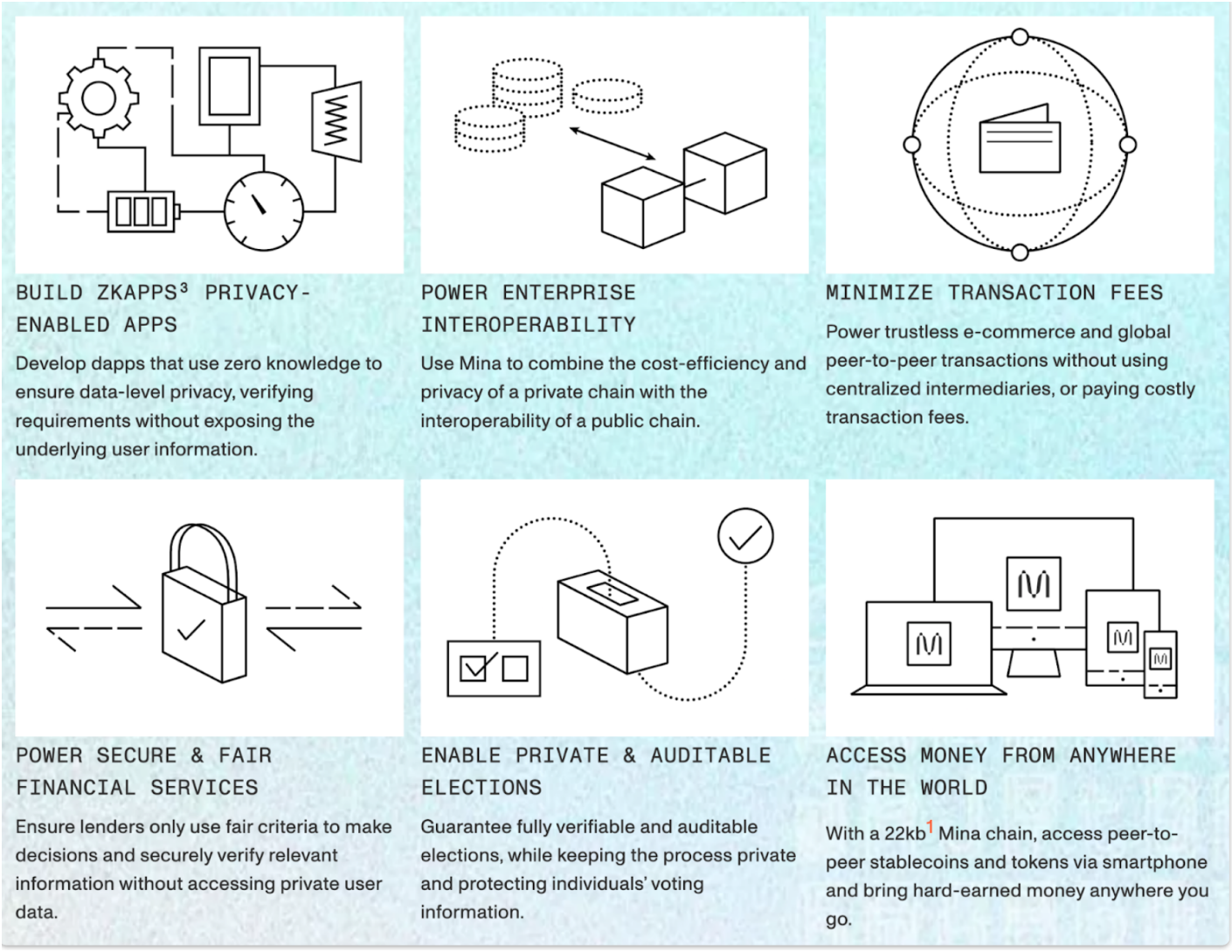
design. This makes zkApps more suitable for applications that require confidential transactions or data privacy.

2. Scalability: zkApps use zk-SNARKs to maintain a fixed-sized blockchain, reducing storage and processing requirements. In contrast, Ethereum smart contracts are part of a growing ledger, which can lead to scalability issues.
3. Language and development: zkApps are written using SnarkyJS, a specialized language for zero-knowledge proofs, while Solidity is used for Ethereum smart contracts. This difference in language and development paradigms may require developers to learn new skills and techniques.
4. Network: zkApps are built on the Mina Protocol, a dedicated platform for zero-knowledge applications, while Solidity smart contracts are built on Ethereum, a general-purpose blockchain platform.

In conclusion, zkApps and Solidity smart contracts differ in terms of privacy, scalability, development languages, and the underlying network. While zkApps offer increased privacy and scalability, they

also come with some limitations, including performance, development complexity, and ecosystem maturity.

zkApps use cases



zkBridge ETH & WMina example

The bridge is currently only one way- i.e MINA state can be read on Ethereum , but not the other way around.

How does the bridge works ?

- Retrieve MINA state proof and the verification key.
- Generate an Auxiliary proof for the state received.
- Post the Auxiliary proof to Ethereum blockchain , where a smart contract will update only if valid.

[ETH] A Wrapped MINA contract on ETH will be created which will be the minter. This contract has a MINA address where a user will deposit funds into.

[MINA] User deposits funds into address above on MINA blockchain.

[MINA] Deposit transaction gets committed.

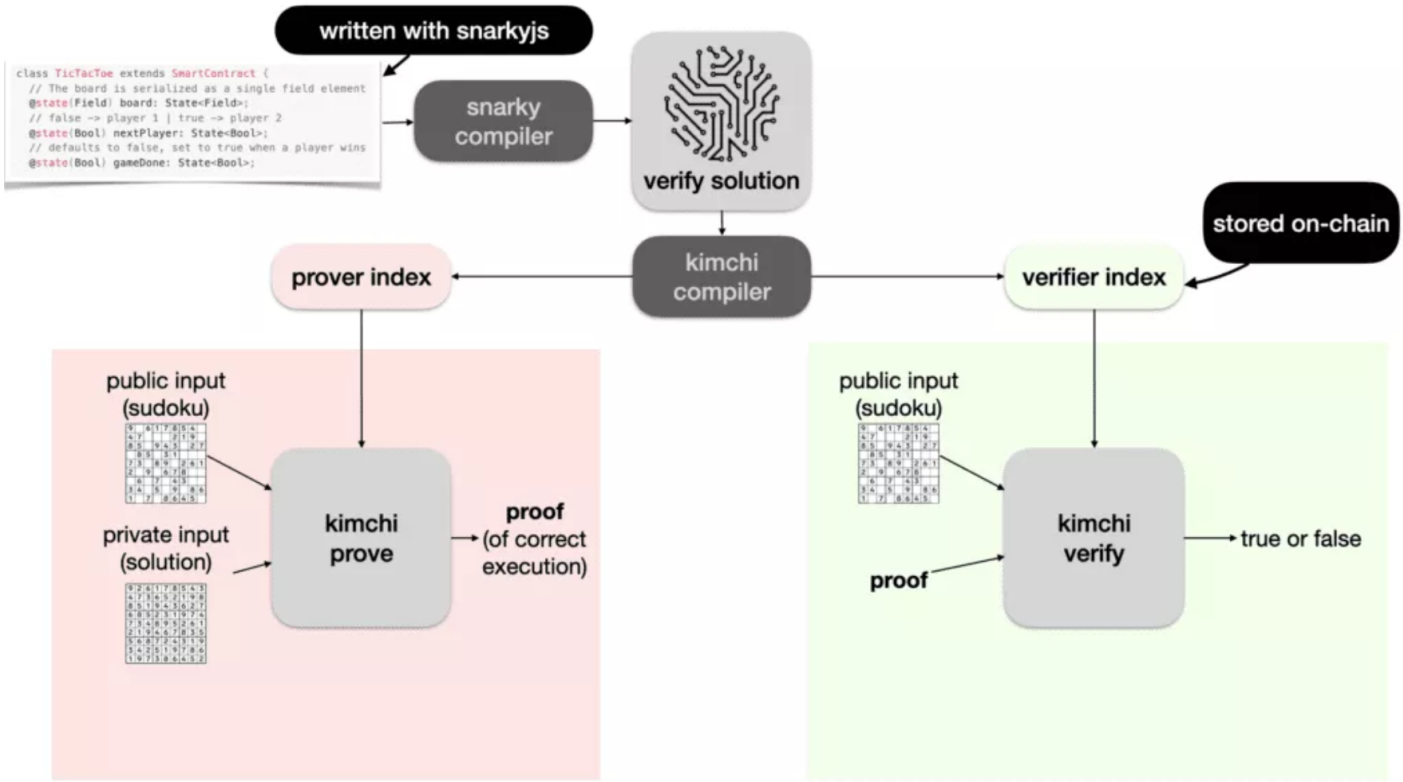
[ETH] Auxiliary Proof of MINA state is submitted to ETH.

[ETH] MINA state proof is validated and confirmed.

[ETH] Wrapped MINA Contract is called which uses the proof and validates the balance and mints Wrapped MINA on Ethereum.

How Mina creates proofs.

Pickles and Kimchi



Pickles

Pickles has two components:

- core zk-SNARK,
- developer toolkit (containing a wide array of library functionality and the Pickles Inductive Proof System)

Kimchi

The heart of the Mina proving system is Kimchi, this is the protocol that creates the proofs.

Kimchi is based on the Plonk family of zkSNARK proof systems.

One notable improvement over Plonk is that Kimchi doesn't require a trusted setup.
This is achieved by adding a polynomial commitment to the setup (similar to Bulletproofs)

Other Improvements in Kimchi

- The gates used in the circuit have changed from being generic gates (which could represent any functionality) to specific gates to target known functionality such as Posiedon Hashes.
- More than 2 inputs are allowed for gates.
- The use of lookup tables for public data or simple boolean operations.

How does Pickles compare to other proof systems ?

This table is a little out of date, but gives a general idea

System	Setup	Programmability	Proof Size	Prover Speed	Recursion Ready	Open Source
Pickles	Trustless	Full	8kB	Fast	Arbitrary	Yes
AIR STARKs	Trustless	Partial, complex constraint system model limiting to succinct circuits	100kB	Very fast	No	No
Halo	Trustless	Full	3.5kB	Fast	Linear	Yes
BN128 PLONK	Trusted, universal	Full	0.5kB	Fast	No	Yes
MNT Groth16	Trusted, per-circuit	Full	1.5kB	Medium	Arbitrary	Yes

Curves used

Mina uses the curves Pallas and Vesta (collectively Pasta).

Resources

Extropy Medium Article

A great introduction to ZKPs and Mina is the Mina Book
Zero Knowledge Podcast ZKP Intro
Zero Knowledge Podcast Mina Episode

O1JS

See Docs

zkApps are written in TypeScript using o1.js.

o1.js is a TypeScript library for writing smart contracts based on zero-knowledge proofs for the Mina Protocol. It is included automatically when creating a new project using the Mina zkApp CLI.

Primitive data types

Field elements are the basic unit of data in zero-knowledge proof programming. Each field element can store a number up to almost 256 bits in size.

```
new Bool(x); // accepts true or false

new Field(x); // accepts an integer,or a numeric string

new UInt64(x); // accepts a Field - useful for constraining numbers to 64 bits

new UInt32(x); // accepts a Field - useful for constraining numbers to 32 bits
```

```
PrivateKey, PublicKey, Signature; //
useful for accounts and signing

new Group(x, y); // a point on our
elliptic curve, accepts two
Fields/numbers/strings

Scalar; // the corresponding scalar field
(different than Field)
```

In typical programming, you might use:

```
const sum = 1 + 3;
```

In o1.js, you would write this as:

```
const sum = new Field(1).add(new
Field(3));
```

This can be simplified as:

```
const sum = new Field(1).add(3);
```

Conditionals (Important !)

Traditional conditional statements are not supported by

o1.js:

```
// this will NOT work
if (foo) {
x.assertEquals(y);
}
```

Instead, use o1.js' built-in Circuit.if() method, which is a ternary operator:

```
const x =
Circuit.if(y.equals(Field.zero)), a, b);
// behaves like `foo ? a : b`
```

Common methods

```
let x = new Field(4); // x = 4

x = x.add(3); // x = 7

let hash = Poseidon.hash([x]); // takes
array of Fields, returns Field

x = x.sub(1); // x = 6

let privKey = PrivateKey.random(); //
create a private key

x = x.mul(3); // x = 18

let pubKey =
PublicKey.fromPrivateKey(privKey); //
derive public key

x = x.div(2); // x = 9

let msg = [hash];

x = x.square(); // x = 81

let sig = Signature.create(privKey, msg);
```

```
// sign a message

x = x.sqrt(); // x = 9

let b = x.equals(8); // b = Bool(false)

b = x.greaterThan(8); // b = Bool(true)

b = b.not().or(b).and(b); // b =
Bool(true)

b.toBoolean(); // true

sig.verify(pubKey, msg); // Bool(true)
```

Setup

Dependencies:

NodeJS 16+ (or 14 using node --experimental-wasm-threads)
NPM 6+
Git 2+
IDE (VS Code recommended)

zkapp-cli

```
npm install -g zkapp-cli

zk project my-proj-name

npm run build && node ./build/src/index.js
```

Useful links for o1.js

Crowdcast

Resource Kit