# Zero Knowledge Proofs

# Recursive SNARKs


Instructors: **Dan Boneh**, Shafi Goldwasser, Dawn Song, Justin Thaler, Yupeng Zhang





Stanford University | Berkeley UNIVERSITY OF CALIFORNIA | GEORGETOWN UNIVERSITY | TEXAS A&M UNIVERSITY

# Recall: SNARK algorithms

A **preprocessing SNARK** is a triple (S, P, V):

- **S**$(C)$ → public parameters $(pp, vp)$ for prover and verifier
- **P**$(pp, \boldsymbol{x}, \boldsymbol{w})$ → proof $\pi$
- **V**$(vp, \boldsymbol{x}, \boldsymbol{\pi})$ → accept or reject

# SNARK types

In the last few lectures, we saw several SNARKs:

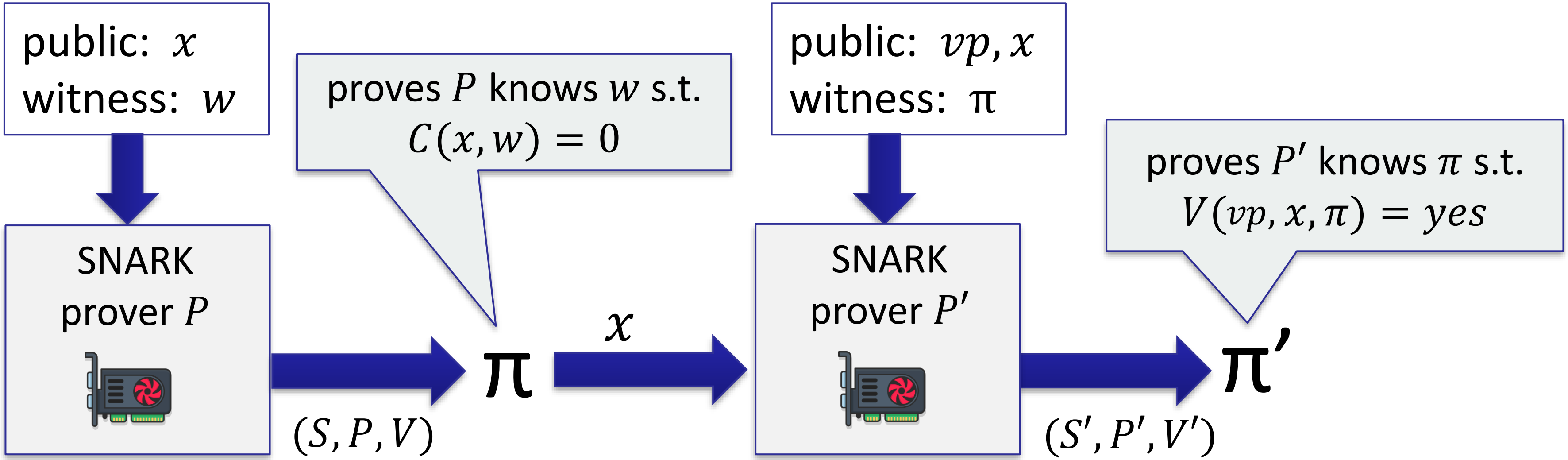
- Groth16, Plonk-KZG:

  $\Rightarrow$ short proofs, but prover time is $O(n \log n)$

- FRI-based proofs (as well as Breakdown, Orion, Orion+, …):

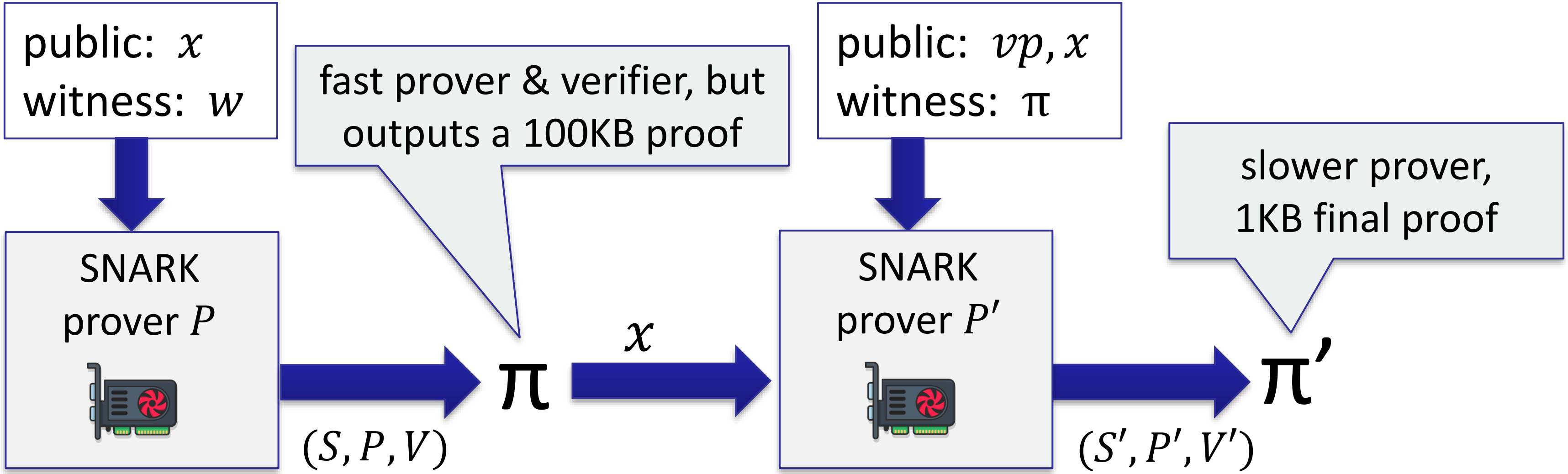
  $\Rightarrow$ faster prover, but longer proofs

# Two level SNARK recursion: proving knowledge of a proof




Use V'($vp'$, $x$, $\pi$') to verify final proof $\pi$'

# Application 1: proof compression




$\Rightarrow$ fast overall prover, and final proof is short

(used to prove complex statements)

# Why is this sound?

fix a circuit $C: \mathbb{F}^n \times \mathbb{F}^m \to \mathbb{F}$

**Recall**: a SNARK (S,P,V) is **knowledge sound** for $C$ if (simplified):

for every poly-time prover $A$ there is a poly-time extractor $E$ s.t.

for all statements y ∈ $\mathbb{F}^n$ :

knowledge error

$$\Pr[\underbrace{C(y,w) = 0 : w \leftarrow E(pp,y)}_{\text{Prob. } E \text{ extract a valid witness for } y}] \geq \Pr[\underbrace{V(vp,y,A(pp,y)) = \text{yes}}_{\text{Prob. adv. } A \text{ outputs a convincing proof for } y}] - \underbrace{\varepsilon}_{\text{negl}}$$

# Why is this sound?

fix a circuit $C: \mathbb{F}^n \times \mathbb{F}^m \to \mathbb{F}$ and let $(pp, vp) \leftarrow S(C)$.

**Goal**: prove that a 2-level recursive SNARK is knowledge sound for $C$

- Let $C'\big((vp, x), \pi\big)$ be the circuit [ V($vp$, $x$, $\pi$) == 'yes' ]
- Let $A$ be a convincing prover for (S',P',V') with respect to $C'$

We need to build an extractor that outputs $w \in \mathbb{F}^m$ s.t. $C(x, w) = 0$

# Why is this sound?

- Let $C'((vp,x),\pi)$ be the circuit [ V($vp$, $x$, $\pi$) == 'yes' ]
- Let $A$ be a convincing prover for (S',P',V') with respect to $C'$

For a given $x \in \mathbb{F}^n$ and $vp$, our extractor does:

$E'$ is a convincing prover for (S,P,V) for $C$.

- **step 1**: (S',P',V') is knowledge sound for $C'$ $\Rightarrow$
  there is an extractor $E'$ that extracts a witness $\pi$ from $A$ s.t. V($vp$, $x$, $\pi$) = 'yes'
- **step 2**: (S,P,V) is knowledge sound for $C$ $\Rightarrow$
  there is an extractor $E$ that extracts a witness $w$ from $E'$ s.t. C($x$, $w$) = 0

# Why is this sound?

Success probability: let $w$ be the extracted witness

$$\Pr[C(x,w)=0] \;\geq\; \underbrace{\underbrace{\Pr[\pi' \leftarrow A(pp,x) \text{ is a convincing proof}] - \varepsilon'}_{\text{Prob. } E' \text{ outputs a valid } \pi} - \;\varepsilon}_{\text{Prob. } E \text{ outputs a valid w}}$$

Caution:

Suppose time($E$') = 2 × time($A$) , time($E$) = 2 × time($E$') ⇒ time($E$) = 4 × time($A$)

⇒ for n-level recursion $\text{time}(E^{(n)}) = 2^{\text{n}} \times \text{ time(A)}$ , not poly-time!

⇒ can only prove security of recursion of depth log(security parameter $\lambda$)

# Another difficulty: random oracles

Recall: the Fiat-Shamir transform results in a SNARK (*S*,*P*,*V*) where the *P* and *V* circuits query a **random oracle** (RO).

During recursion, how does prover process the verifier's RO gates?

Answer:

- Instantiate the verifier's RO with a concrete hash function H
- Then assume that the resulting ($S^H$,$P^H$,$V^H$) is still secure
- Now we can recurse (but security proof requires an ugly assumption)

# Application 2: streaming proof generation

A typical prover (e.g., for zk-Rollup):

- Collect statements $(x_1, w_1), \ldots, (x_n, w_n)$ from the public (e.g., Tx)
- Prove a conjunction: $C(x_1, w_1) = \cdots = C(x_n, w_n) = 0$

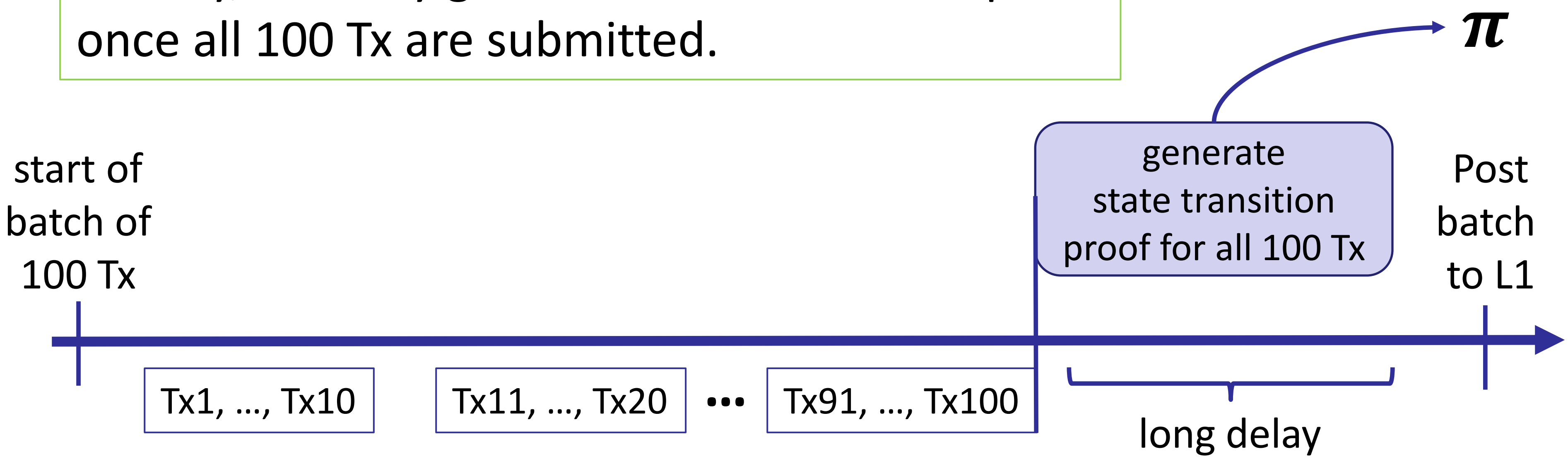
**The problem**: need all $n$ statements before can begin to build proof

Can we generate the proof in a streaming fashion?

- **Goal**: begin to generate proof as soon as $(x_1, w_1)$ is available

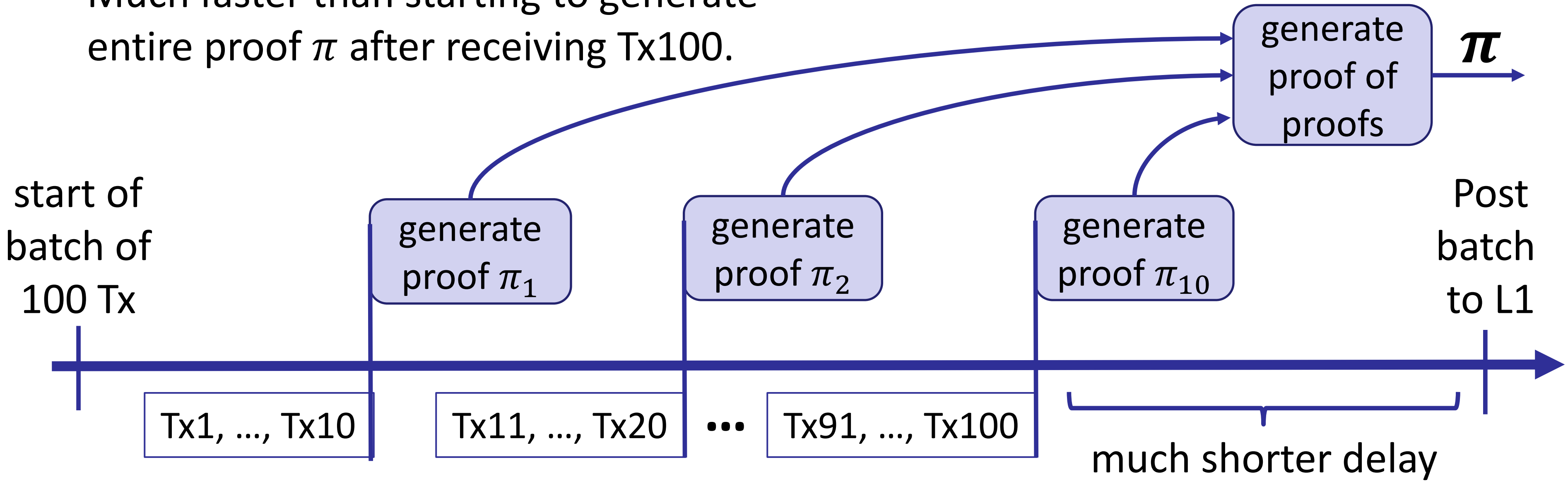
# Streaming proof generation: zk-Rollups

Naively, can only generate state transition proof once all 100 Tx are submitted.



# Streaming proof generation:  zk-Rollups

Much faster than starting to generate entire proof $\pi$ after receiving Tx100.



# Application 3: Layer-3 zk-Rollups

## First, a very brief review of Rollups

Rollup contract on layer-1 holds assets of all Rollup users, and Merkle root of layer-2

Rollup state (L2)

Alice (on L2):
**4 ETH, 1 DAI**

Bob (on L2):
**3 ETH, 2 DAI**

…

Alice:
10 ETH

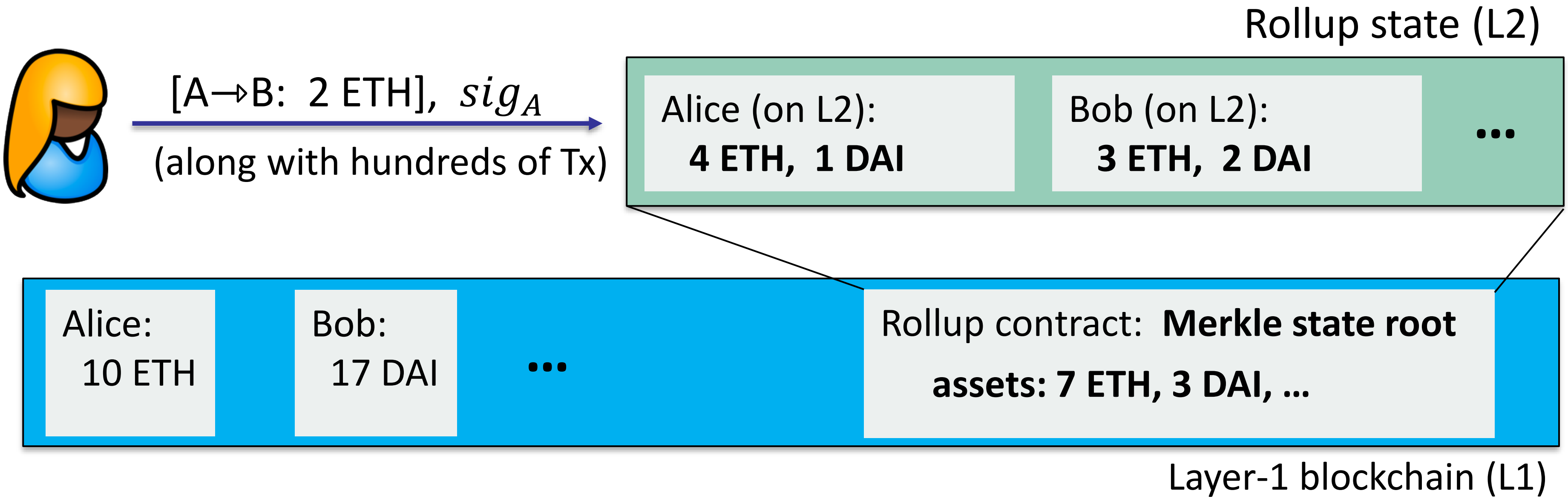
Bob:
17 DAI

…

Rollup contract: **Merkle state root**
**assets: 7 ETH, 3 DAI, …**

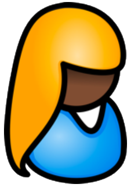
Layer-1 blockchain (L1)

# Transfers inside Rollup are easy



# Transfers inside Rollup are easy

**State transition proof $\pi$:** proves that Tx batch is valid and that new root is correct



$[A \rightarrow B{:}\ 2\ \text{ETH}],\ sig_A$

(along with hundreds of Tx)

Alice (on L2):
**2 ETH, 1 DAI**

Bob (on L2):
**5 ETH, 2 DAI**

...

updated state root, **<u>proof $\pi$</u>**

Alice:
10 ETH

Bob:
17 DAI

...

Rollup contract: **updated state root**

**assets: 7 ETH, 3 DAI, ...**

Layer-1 blockchain (L1)

note: no assets move on the L1

# Transferring funds to and from Rollup

## Alice sends funds to Rollup:

- Alice sends funds from her L1 address to the Rollup contract
- Rollup coordinator sends updated state root to L1 contract to record Alice's new balance

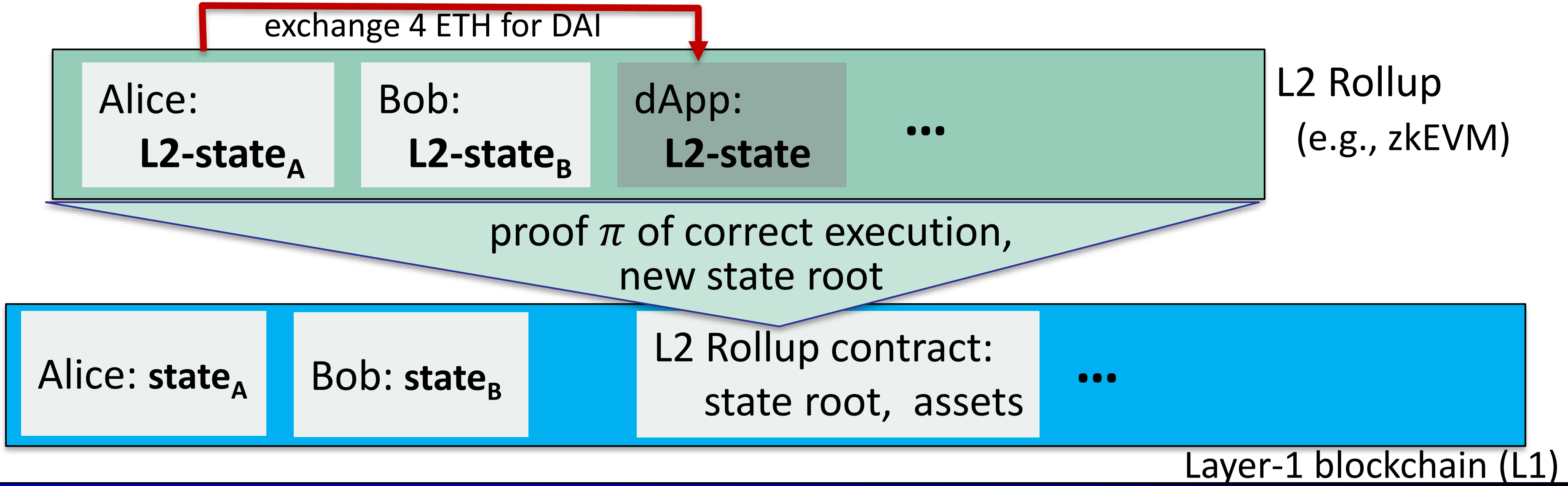
## Alice withdraws funds from Rollup:

- Alice requests L1 Rollup contract to send her funds to an L1 address
- Rollup coordinator sends updated state root to L1 contract

$\Rightarrow$ Much more expensive than in-Rollup transfers

# Running a dApps in a Rollup

Rollup coordinator computes updated state root, and state transition proof $\pi$



# Running a dApps in a Rollup

Rollup coordinator computes updated state root, and proof $\pi$

State transition proof $\pi$ proves that:

- Tx from Alice is valid (properly signed and she has sufficient balance)
- new state root reflects the correct execution of dApp EVM code on L2

… a complex statement to prove

new state root

Alice: **state**$_A$

Bob: **state**$_B$

L2 Rollup contract:
state root, assets

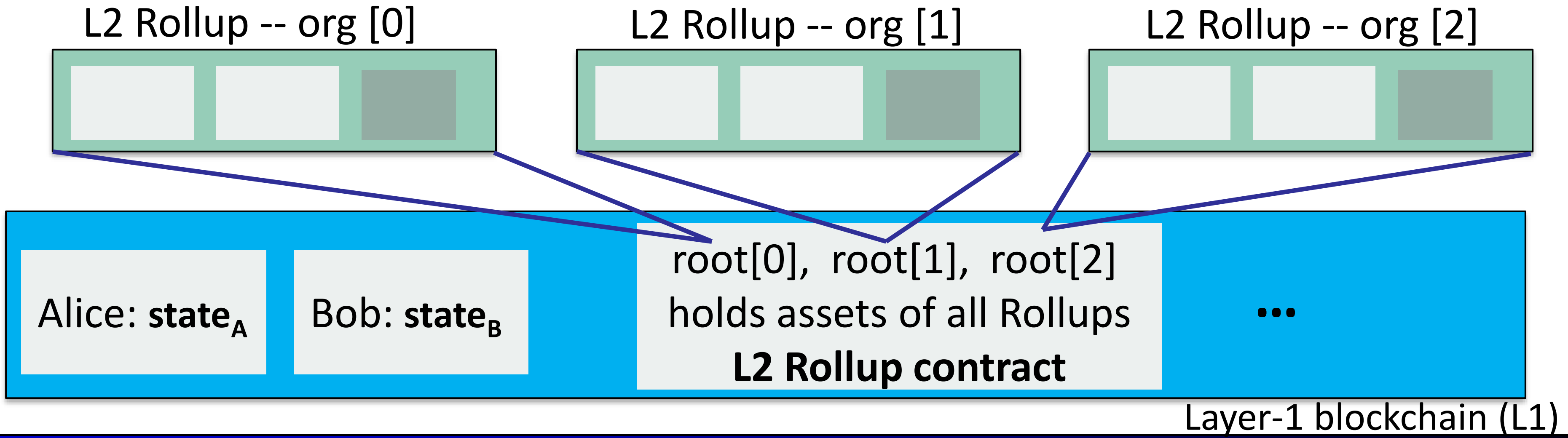
…

Layer-1 blockchain (L1)

# One Rollup contract can support many L2's

Rollups run by different orgs: all must use the same rules for updating state root

⇒ same execution engine (e.g., EVM) for all L2 dApps



# Layer-3 zk-Rollup

A gaming company runs an L2 Rollup:

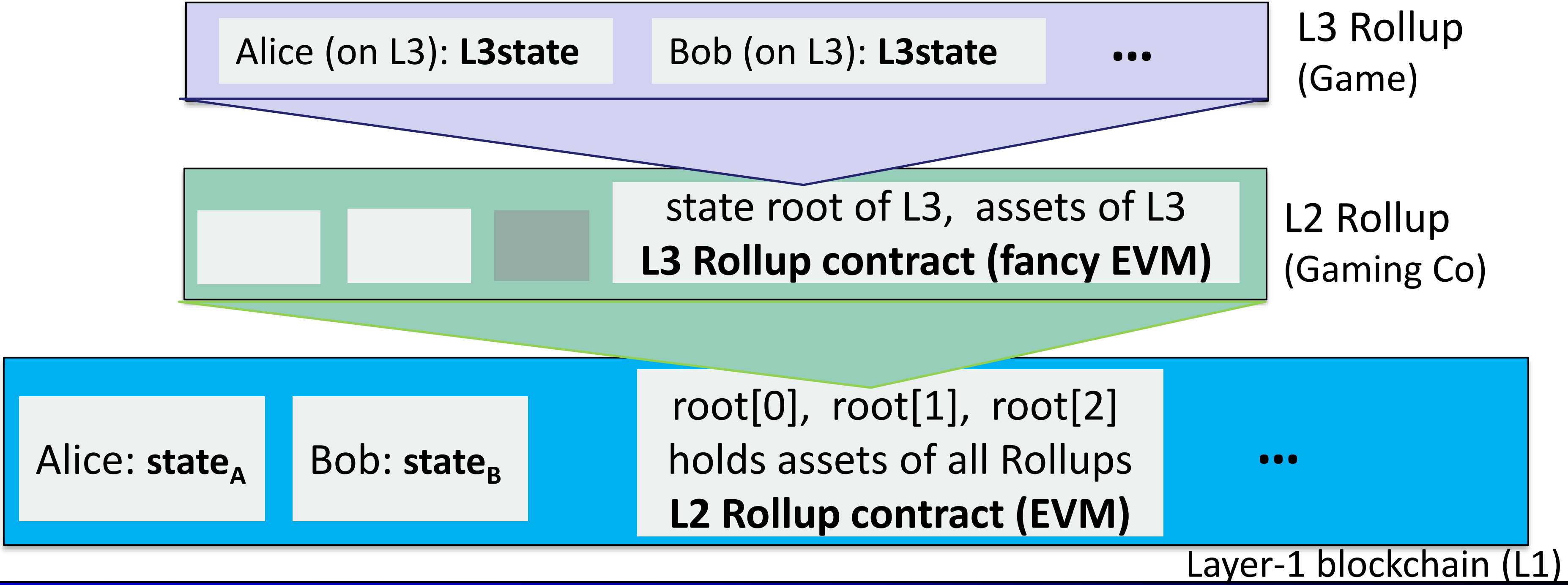
- Wants a custom execution engine optimized for its games
- Wants a faster settlement rate than L2 $\rightarrow$ L1 settlement rate

What to do?

- Run an L3 on top of its L2

> $\Rightarrow$ requires <u>recursive</u> state transition proofs

# Layer-3 zk-Rollup



# Layer-3 zk-Rollup

Alice on L3: [send an NFT to a dApp L3], $sig_A$ (dApp uses fancy EVM code)

---

Every second: L3 coordinator $\rightarrow$ L2 coordinator:

new L3 state root and state transition proof $\pi_3$

- L2 Rollup contract: check proof and record updated L3 state root

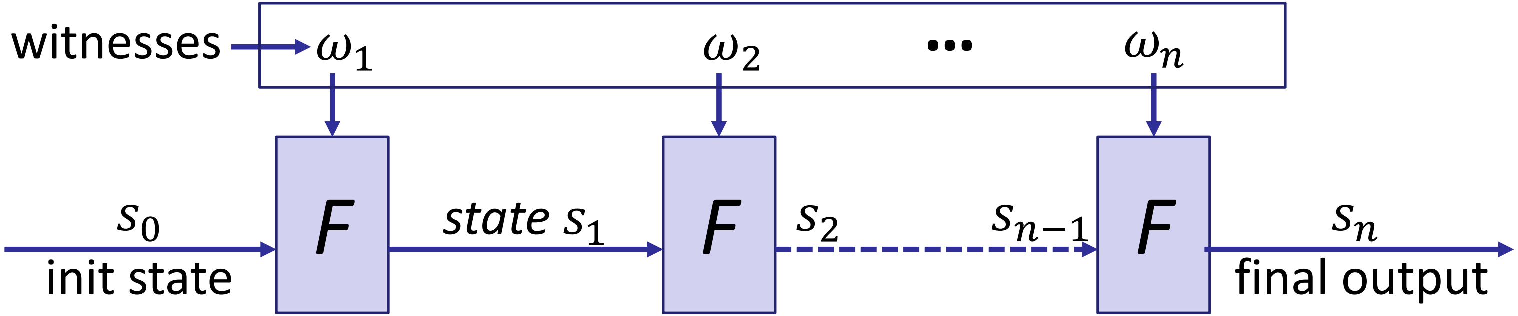
---

Every minute: L2 coordinator has 60 proofs from L3 coordinator

- Construct a **single recursive proof $\boldsymbol{\pi_2}$** that all 60 proofs from L3 are valid
- L2 coordinator $\rightarrow$ L1 contract: latest L2 state root and the proof $\pi_2$
- L1 contract: check proof $\pi_2$ and record updated L2 state root

# Application 4: Incrementally Verifiable Computation (IVC) [Valiant'08]

Consider a long computation done by iterating a function *F*:




Deep Thought

**Goal**: succinct proof that prover has $\omega_1, \ldots, \omega_n$ s.t. final output $s_n$ is correct

The verifier has $F$ and public values: $n, s_0, s_n$

# IVC: construction [Valiant'08]

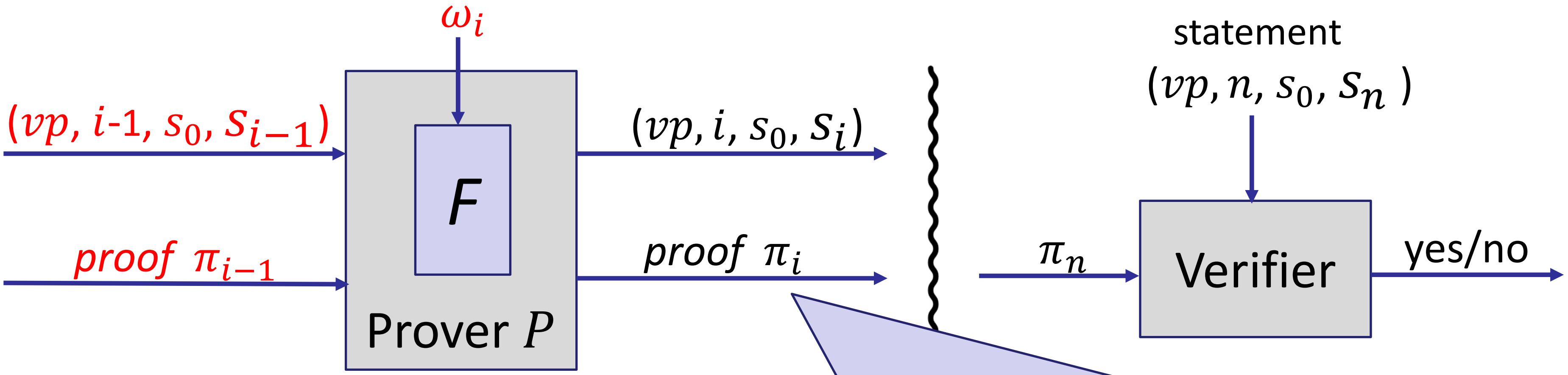
High level idea (informal):

- every step outputs a proof that the computation is correct to this point
- Specifically, for $i = 1, \ldots, n$, at step $i$ prover outputs $s_i$ and a proof $\pi_i$ that proves prover has a witness $(s_{i-1}, \omega_i,\ \pi_{i-1})$ such that:

$$F(s_{i-1}, \omega_i) = s_i \quad \text{and} \quad V(vp, (i\text{-}1, s_0, s_{i-1}), \pi_{i-1}) = \mathit{yes}$$

$\Rightarrow$ final proof $\pi_n$ is a succinct proof that
prover has $\omega_1, \ldots, \omega_n$ s.t. final output $s_n$ is correct

# The statement at step number $i$ $(i > 0)$




I know a witness $(s_{i-1}, \omega_i, \pi_{i-1})$ for the statement $(vp, i, s_0, s_i)$ such that:

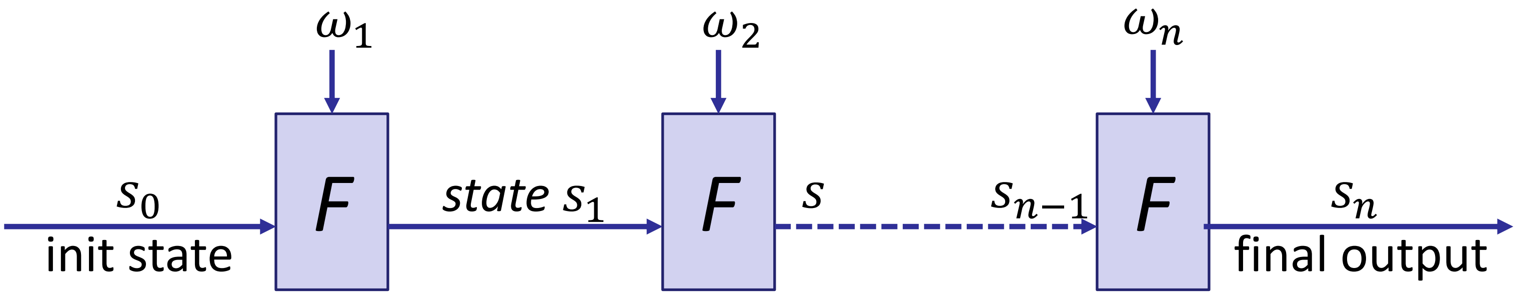
$F(s_{i-1}, \omega_i) = s_i$ and $V(vp, (i\text{-}1, s_0, s_{i-1}), \pi_{i-1}) = yes$

# Applications of IVC

1. Break a long computation into a sequence of small steps

$F$: one microprocessor step (Risc5, EVM, …)

Prover needs far less memory per step compared to a monolithic proof



# Applications of IVC

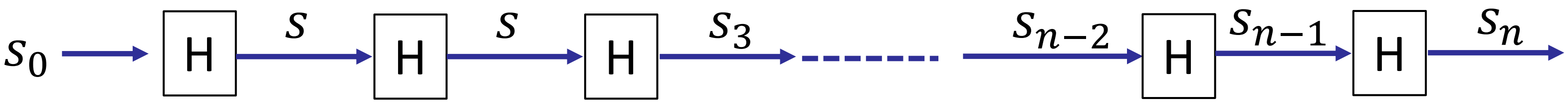
2. A succinct proof that current state of blockchain is correct

$s_0$: initial state of chain, $s_n$: current state of chain

$\omega_1, \ldots, \omega_n$: blocks of valid transactions

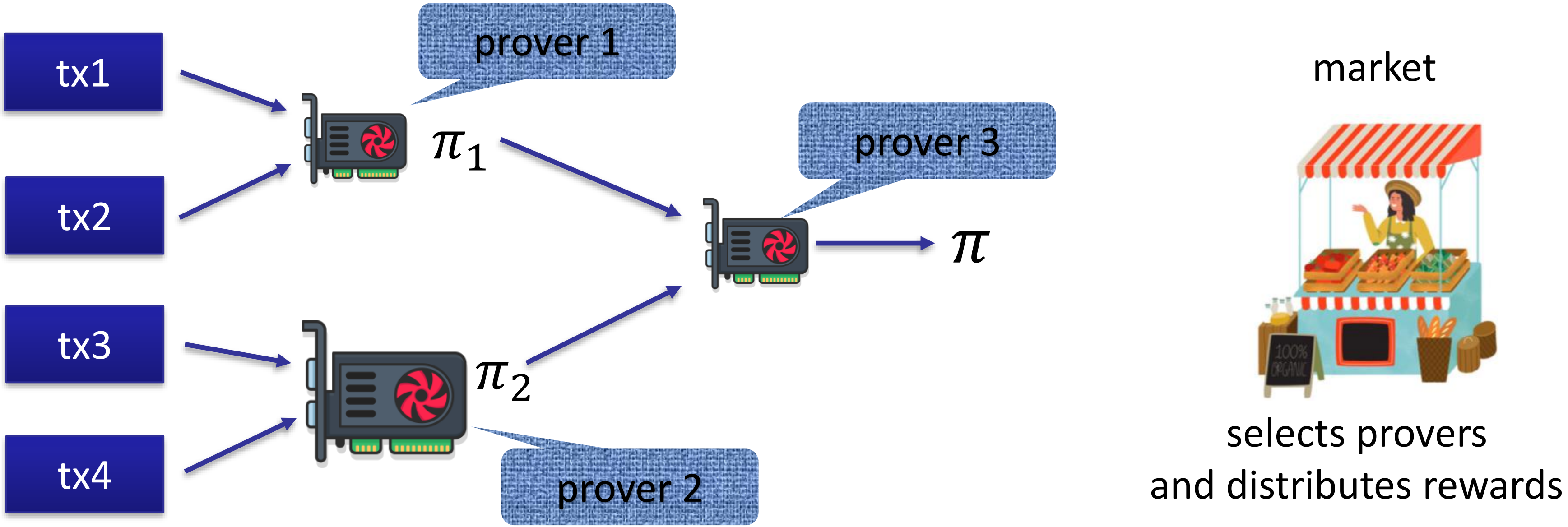
Used in Mina blockchain $\Rightarrow$ verify state of chain by checking <u>one</u> recursive proof

3. **Verifiable Delay Functions** (VDF): succinct proof that $s_n$ is equal to $H^{(n)}(s_0)$

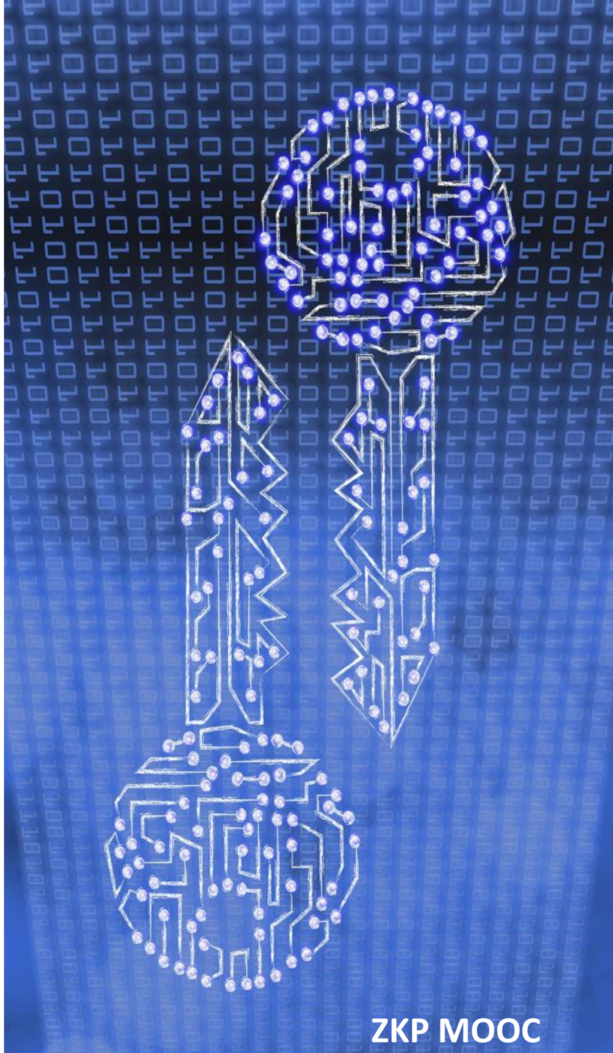


# Application 5: a market for ZK provers

## Anyone with a GPU will be paid to create ZK proofs

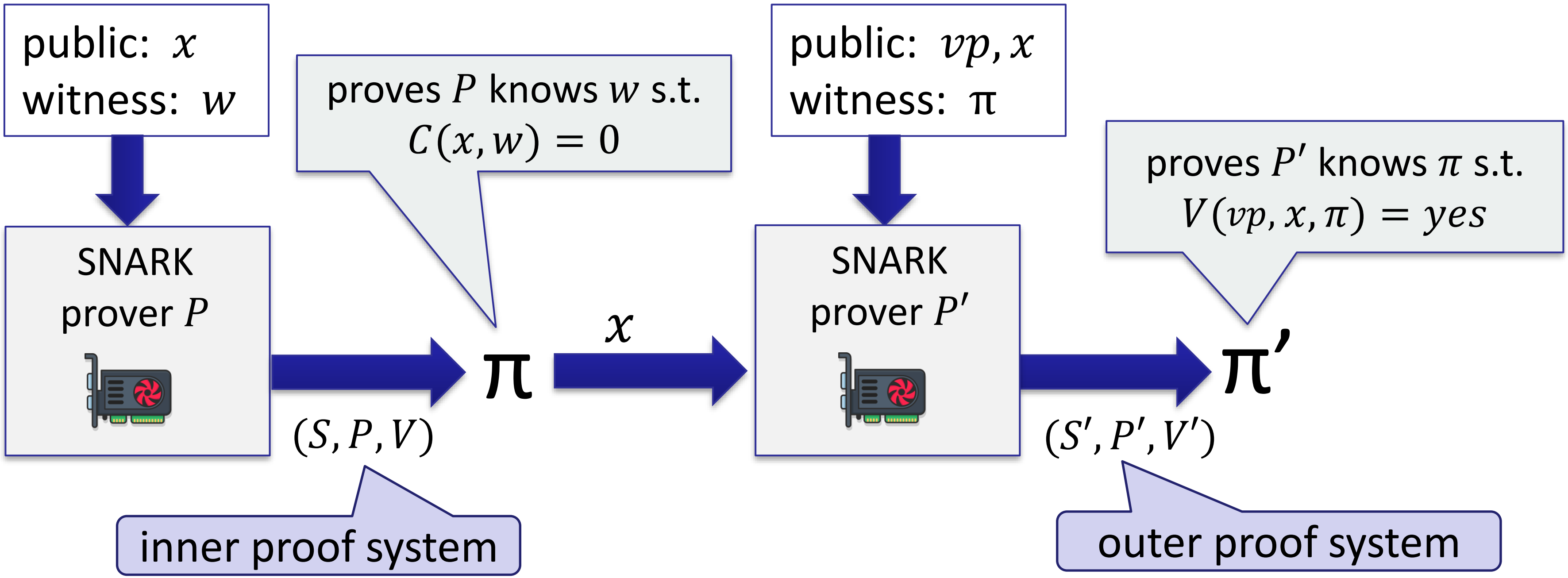


# Choosing Curves to Support Recursion




Credit: Faithie/Shutterstock

# Two level SNARK recursion: proving knowledge of a proof



# Review

Fix a circuit $C: \mathbb{F}_p^n \times \mathbb{F}_p^m \to \mathbb{F}_p$ and a statement $x \in \mathbb{F}_p^n$.

- To prove "I know $w$ s.t. $C(x, w) = 0$" we use commitments to polys. in $\mathbb{F}_p[X]$
    - Prover commits to a polynomial that encodes the computation trace

- To commit to a polynomial $f \in \mathbb{F}_p[X]$ using KZG:
    - need a group $\mathbb{G}$ of order $p$ ; a KZG commitment is a single element in $\mathbb{G}$

How is the group $\mathbb{G}$ represented?

# Algebraic Groups

We say that $\mathbb{G}$ is an **<u>algebraic group defined over</u>** $\mathbb{F}_q$ if $\mathbb{G} \subseteq \mathbb{F}_q^{\ell}$ and

- the group operation can be computed by polynomials over $\mathbb{F}_q$:

  there are polynomials $(f_1, \dots, f_\ell) \in \mathbb{F}_q[X^\ell]$ such that

  for all $a, b \in \mathbb{G}$ we have that $a + b = \big(f_1(a,b), \dots, f_\ell(a,b)\big) \in \mathbb{G}$

- there is an efficient algorithm that test if $a, b \in \mathbb{G}$ satisfy $a = b$.

Example: $\mathbb{G}$ is the group of points of an elliptic curve defined over $\mathbb{F}_q$ $(\mathbb{G} \subseteq \mathbb{F}_q^3)$

- The group has some order $p$ (which is close to $q$)

# Recursive proofs: the arithmetic problem

Let $\mathbb{G}$ be an algebraic group of order $p$, defined over $\mathbb{F}_q$

$\Rightarrow$ The prover supports circuits over $\mathbb{F}_p$, but verifier needs $\mathbb{F}_q$ for group ops.

I can do proofs for circuits over $\mathbb{F}_p$

That's great, but I need arithmetic over $\mathbb{F}_q$

Prover P'

Verifier circuit does group operations in $\mathbb{G}$ $\Rightarrow$ arithmetic in $\mathbb{F}_q$

Verifier V

# What to do?

ex: a pairing circuit (as in KZG eval) is huge

**Option 1**: field emulation

- Implement arithmetic in $\mathbb{F}_q$ as a circuit over $\mathbb{F}_p$
- The problem: blows up the size of verifier circuit $\Rightarrow$ slow prover.

**Option 2**: find an algebraic group $\mathbb{G}$ of order $p$, defined over $\mathbb{F}_p$

- Now both the prover and verifier use arithmetic in $\mathbb{F}_p$ ✔
- The bad news: the universe doesn't want us to have that …

$\Rightarrow$ the discrete log problem is <u>always</u> easy in such groups

# Solution: a chain of groups

Idea: find groups $\mathbb{G}_1$ and $\mathbb{G}_2$ such that

- $\mathbb{G}_1$ has order $p$, and is defined over $\mathbb{F}_q$
- $\mathbb{G}_2$ has order $q$, and is defined over $\mathbb{F}_r$

$$\boxed{|\mathbb{G}_1| = p} \subseteq \boxed{\mathbb{F}_q^{\ell}}$$

$$\boxed{|\mathbb{G}_2| = q} \subseteq \boxed{\mathbb{F}_r^{\ell}}$$

# Solution: a chain of groups

$|\mathbb{G}_1| = p \subseteq \mathbb{F}_q$

$|\mathbb{G}_2| = q \subseteq \mathbb{F}_r$

Now, to do a <u>two-level recursion</u>:

- Inner proof system (S, P, V) uses poly. commitments in $\mathbb{G}_1$

  $\Rightarrow$ Prover P supports circuits over $\mathbb{F}_p$, Verifier needs arithmetic in $\mathbb{F}_q$

- Outer proof system (S', P', V') uses poly. commitments in $\mathbb{G}_2$

  $\Rightarrow$ Prover P' supports circuits over $\mathbb{F}_q$, verifier needs arithmetic in $\mathbb{F}_r$

A longer chain of groups supports more levels of recursion

# Even better: a cycles of groups [BCTV'14]

Find groups $\mathbb{G}_1$ and $\mathbb{G}_2$ such that

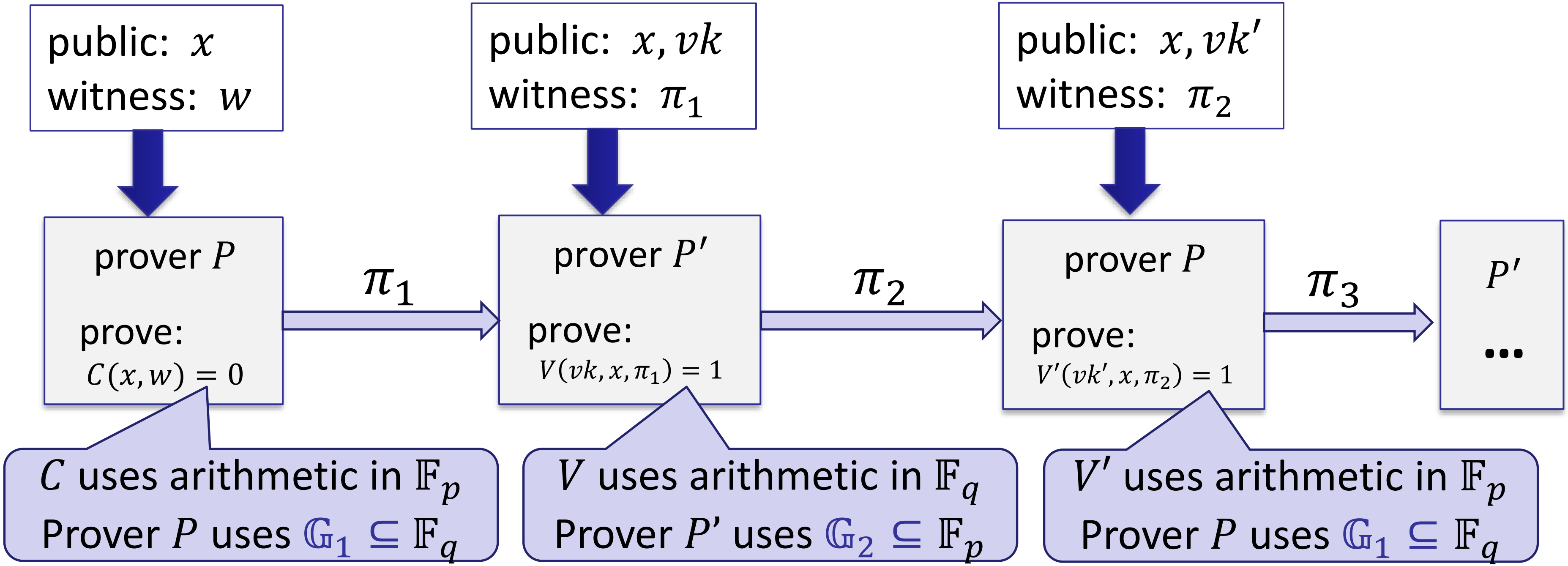
- $\mathbb{G}_1$ has order $p$, and is defined over $\mathbb{F}_q$
- $\mathbb{G}_2$ has order $q$, and is defined over $\mathbb{F}_p$

$$\boxed{|\mathbb{G}_1| = p} \subseteq \boxed{\mathbb{F}_q^{\ell}}$$

$$\boxed{\mathbb{F}_p^{\ell}} \supseteq \boxed{|\mathbb{G}_2| = q}$$

enables longer recursion by jumping back and forth between two proof systems

# Recursion using a cycle [BCTV'14]



# Three types of cycles of length two

Both $\mathbb{G}_1$ and $\mathbb{G}_2$ are "pairing" groups (both support KZG)

- the bad news: best constructions result in inefficient groups

$\mathbb{G}_1$ is a pairing group, but $\mathbb{G}_2$ is a regular group

- use KZG in $\mathbb{G}_1$ and a non-pairing PCS in $\mathbb{G}_2$ (e.g., bulletproofs)

Neither group is a pairing group: use a non-pairing PCS in both

- The pasta curves: pallas and vesta (next slide)

# A large family of cycles of type-3

Let $E/\mathbb{Q}$ be the elliptic curve: $y^2 = x^3 + d$ (for some $d$)

For "many" primes $q$, if $p = |E(\mathbb{F}_q)|$ is a prime then

$$\boxed{|E(\mathbb{F}_p)| = q \quad \text{and} \quad |E(\mathbb{F}_q)| = p}$$

Pasta uses $d = 5$ and both curves are convenient for recursion

- Developed for Halo2

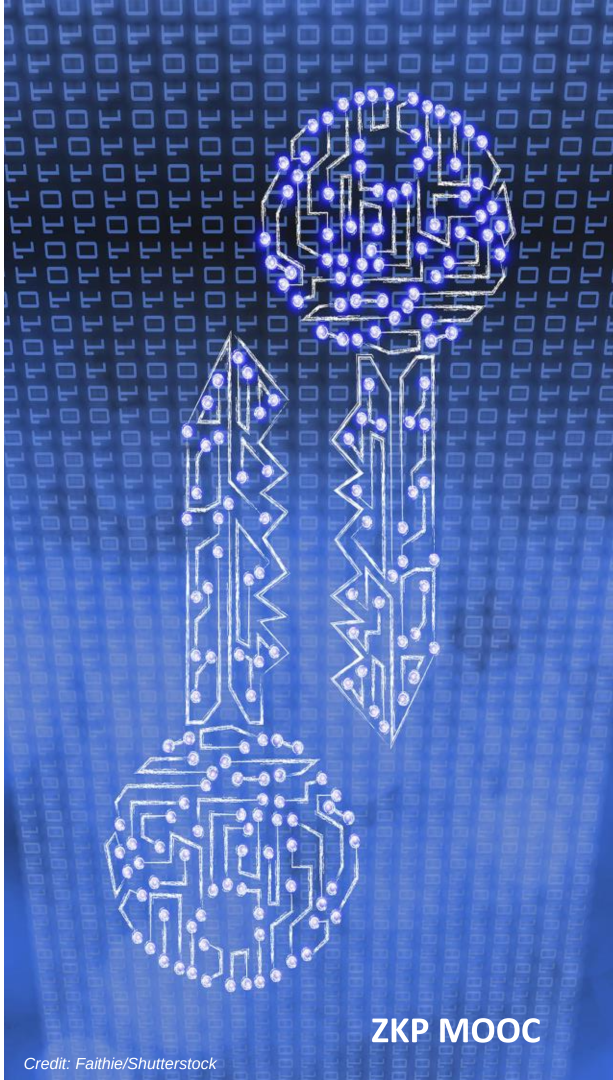
Silverman, Stange 2009: Corollary 22

# Efficient Recursion via Statement Folding:

## **Nova**, **Supernova**, and generalizations

eprint.iacr.org/2021/370.pdf

( see also  eprint.iacr.org/2020/1618.pdf )




*Credit: Faithie/Shutterstock*

# The difficulty with full recursion

- Prover $P$ needs to build a proof for a circuit $C$ that runs the entire verification algorithm $V(vk, x, \pi)$.
  - Expensive: $V$ needs to verify eval. proofs for a poly. commitment

- Halo: takes eval proof verification out of $C$ $\Rightarrow$ simpler $C$

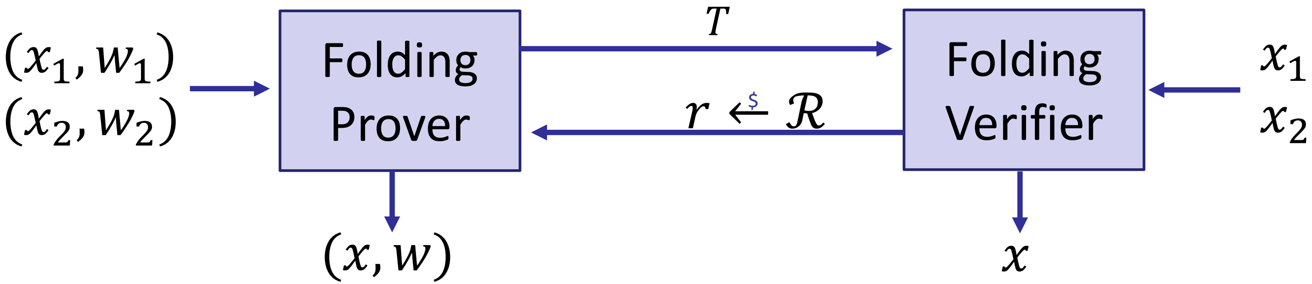
- Nova: takes (almost) all verification checks out of $C$

  $\Rightarrow$ even simpler $C$

# A folding scheme: compress two instances into one

Let $C: \mathbb{F}_p^n \times \mathbb{F}_p^m \to \mathbb{F}_p$ be a circuit

A folding scheme for $C$ is a protocol between two parties:




**Complete**: if $C(x_1, w_1) = C(x_2, w_2) = 0$ then $C(x, w) = 0$

**Knowledge sound**: $\forall P^* \ \exists E$ s.t. $\forall\ x_1, x_2$: $P^*$ outputs valid $w$ for $x \Rightarrow E$ outputs valid $w_1, w_2$

# A folding scheme: compress two instances into one

Let $C: \mathbb{F}_p^n \times \mathbb{F}_p^m \to \mathbb{F}_p$ be a circuit

A folding scheme for $C$ is a protocol between two parties:

$(x_1, w_1)$
$(x_2, w_2)$ $\rightarrow$ Folding Prover $\xrightarrow{T}$

$\downarrow$
$(x, w)$

Verifier of $(x, w)$ must be convinced that $r$ was computed correctly

To make Folding Prover non-interactive, use Fiat-Shamir:

(i) $r \leftarrow H(x_1, x_2, T)$ , (ii) output $(x, w)$

# Recall: every circuit can be represented as a rank-1 constraint system (R1CS)

$C: \mathbb{F}_p^n \times \mathbb{F}_p^m \to \mathbb{F}_p$ $\Longleftrightarrow$ $A, B, D \in \mathbb{F}_p^{u \times v}$

(circuit $C$) simple translation (R1CS program)

$(x, w') \in \mathbb{F}_p^{n+m}$ $\Longleftrightarrow$ $z = (x, w) \in \mathbb{F}_p^u$

s.t. $C(x, w') = 0$ simple translation s.t. $(Az) \circ (Bz) = Dz$

(valid statement,witness pair)

$(x_1, x_2) \circ (y_1, y_2) = (x_1 y_1, x_2 y_2)$

# A folding scheme for R1CS

A folding scheme: compress two instances into one

Example: fix an R1CS program $A, B, D \in \mathbb{F}_p^{u \times v}$

- instance 1: public $x_1 \in \mathbb{F}_p^n$, witness $z_1 = (x_1, w_1) \in \mathbb{F}_p^v$
- instance 2: public $x_2 \in \mathbb{F}_p^n$, witness $z_2 = (x_2, w_2) \in \mathbb{F}_p^v$

We know $(Az_i) \circ (Bz_i) = Dz_i$ for $i = 1,2$

# Folding the two instances into one

**Attempt 1**: verifier chooses $r \xleftarrow{\$} \mathbb{F}_p$ and sets $x \leftarrow x_1 + r\,x_2$ .

prover sets $z \leftarrow z_1 + r\,z_2 = (x_1 + r\,x_2\,, w_1 + r\,w_2)$

Then:

$$(Az) \circ (Bz) = A(z_1 + r\,z_2) \circ B(z_1 + r\,z_2)$$

$E \in \mathbb{F}_p^u$

$$= (Az_1) \circ (Bz_1) + \mathrm{r}^2\,(Az_2) \circ (Bz_2) + \boxed{r(Az_2) \circ (Bz_1) + r(Az_1) \circ (Bz_2)}$$

$$= \quad Dz_1 \quad + \quad r^2 Dz_2 \quad + \quad E$$

$\Rightarrow$ not quite an R1CS witness: we want $(Az) \circ (Bz) = Dz$

# Let's try again: relaxed R1CS

**Relaxed R1CS instance**: $A, B, D \in \mathbb{F}_p^{u\times v}, \ (x \in \mathbb{F}_p^n, \ \ c \in \mathbb{F}_p, \ \ E \in \mathbb{F}_p^u)$

**Witness**: $z = (x, w) \in \mathbb{F}_p^v$ s.t. $(Az) \circ (Bz) = c(Dz) + E$

Now, again, fix a relaxed R1CS program $A, B, D \in \mathbb{F}_p^{u\times v}$

- instance 1: public $(x_1, c_1, E_1)$, witness $z_1 = (x_1, w_1) \in \mathbb{F}_p^v$
- instance 2: public $(x_2, c_2, E_2)$, witness $z_2 = (x_2, w_2) \in \mathbb{F}_p^v$

We know $(Az_i) \circ (Bz_i) = c_i(Dz_i) + E_i$ for $i = 1,2$

# Folding the two relaxed R1CS instances into one

**Attempt 2**: step 1: Prover computes and send to V:

$$T \leftarrow \underbrace{(Az_2) \circ (Bz_1) + (Az_1) \circ (Bz_2) - c_1(Dz_2) - c_2(Dz_1)}_{\text{(cross terms)}}$$

step 2: verifier chooses $r \xleftarrow{\$} \mathbb{F}_p$, sends $r$ to P, and sets

$$x \leftarrow x_1 + r\,x_2\,, \qquad c \leftarrow c_1 + r\,c_2\,, \qquad E \leftarrow E_1 + rT + r^2\,E_2$$

step 3: prover sets $z \leftarrow z_1 + r\,z_2 = (x_1 + r\,x_2\,, w_1 + r\,w_2)$

# Why this is correct

$$(Az) \circ (Bz) =$$

$$= (Az_1) \circ (Bz_1) + \mathrm{r}^2\,(Az_2) \circ (Bz_2) + \boxed{r(Az_2) \circ (Bz_1) + r(Az_1) \circ (Bz_2)}$$

$$= \overbrace{c_1(Dz_1) + E_1} + \overbrace{r^2 c_2(Dz_2) + r^2 E_2} + r\,\boxed{[(Az_2) \circ (Bz_1) + (Az_1) \circ (Bz_2)]}$$

$$= (c_1 + rc_2)(Dz_1 + rDz_2) + \underbrace{\mathrm{E}_1 + r^2\mathrm{E}_2 + rT}_{E}$$

$$= c(Dz) + E$$

$\Rightarrow$ So, $w$ is a valid witness for the relaxed R1CS instance $(x, c, E)$

# Why is this knowledge sound? (informal)

For every folding prover $P^*$, there is an extractor $\boldsymbol{E}$ s.t.

for all instances $(x_1, c_1, E_1)$ and $(x_2, c_2, E_2)$,

if folding verifier outputs $(x, c, E)$ and $P^*$ outputs a valid $w$,

$\Rightarrow$

w.h.p, $\boldsymbol{E}$ extracts from $P^*$ valid witnesses $w_1, w_2$ for the two instances

[note: also need to commit to $w$ in the instance]

eprint.iacr.org/2021/370.pdf (lemma 4)

# Not good enough

In a relaxed R1CS the verifier has $(x, c, E)$ ; prover has $z$.

- The problem: $E$ can be large (much larger than $x$)

Solution: **committed relaxed R1CS**

- Verifier has $\left(x,\ c,\ \underbrace{commit(E, r_E)}_{\text{short commitment to } E}\right)$ ; prover has $(z, E, r_E)$

- Commitment needs to be "additive" to enable folding

# Recall: homomorphic commitment scheme

Two algorithms:

- *commit*$(m, r_m) \rightarrow$ ***com*** $\qquad m \in \mathcal{M}, \quad r_m \xleftarrow{\$} \mathcal{R}, \quad$ ***com*** $\in \mathcal{C}$
- *verify*$(m,$ ***com***$, r_m) \rightarrow$ accept or reject

Properties: (informal)

- **binding**: cannot produce ***com*** and two valid openings for ***com***
- **hiding**: ***com*** reveals nothing about committed data

# Recall: homomorphic commitment scheme

Suppose $\mathcal{M} = \mathbb{F}^n$, $\mathcal{R} = \mathbb{F}$, and $\mathcal{C}$ is an additive group

- The commitment scheme is **homomorphic** if for all $m_1, m_2, r_1, r_2$:

$$commit(m_1, r_1) + commit(m_2, r_2) = commit(m_1 + m_2, r_1 + r_2)$$

- The commitment scheme is **succinct** if commitment size is $O_\lambda(1)$

Many examples: Pedersen, lattice-based, …

# Folding scheme for committed relaxed R1CS

**Instance**: $A, B, D \in \mathbb{F}_p^{u\times v}$, $(x \in \mathbb{F}_p^n$, $c \in \mathbb{F}_p$, $\mathrm{com}_E \in \mathbb{F}_p^u)$

**Witness**: $(z, E, r_E)$ s.t. $(Az) \circ (Bz) = c(Dz) + E$ and $\mathrm{com}_E = commit(E, r_E)$

As usual, fix an R1CS program $A, B, D \in \mathbb{F}_p^{u\times v}$

- instance 1: public $(x_1, c_1, com_{E_1})$, witness $(z_1, E_1, r_{E_1})$
- instance 2: public $(x_2, c_2, com_{E_2})$, witness $(z_2, E_2, r_{E_2})$

# Folding scheme for committed relaxed R1CS

$

- Prover computes

$$T \leftarrow (Az_2) \circ (Bz_1) + (Az_1) \circ (Bz_2) - c_1(Dz_2) - c_2(Dz_1)$$

sends $com_T \leftarrow commit(T, r_T)$ to V.

homomorphic commitment

- Verifier chooses $r \xleftarrow{\$} \mathbb{F}_p$, sends $r$ to P, and sets

$$x \leftarrow x_1 + r\, x_2\,, \quad c \leftarrow c_1 + r\, c_2\,, \quad \mathrm{com}_E \leftarrow com_{E_1} + r \cdot com_T + r^2 \cdot com_{E_2}$$

- Prover sets

$$z \leftarrow z_1 + r\, z_2, \quad E \leftarrow E_1 + rT + r^2 E_2\,, \quad r_E \leftarrow r_{E_1} + r \cdot r_T + r^2 \cdot r_{E_2}$$

# Folding scheme for committed relaxed R1CS

- Prover computes

sen

nitment

- Ver

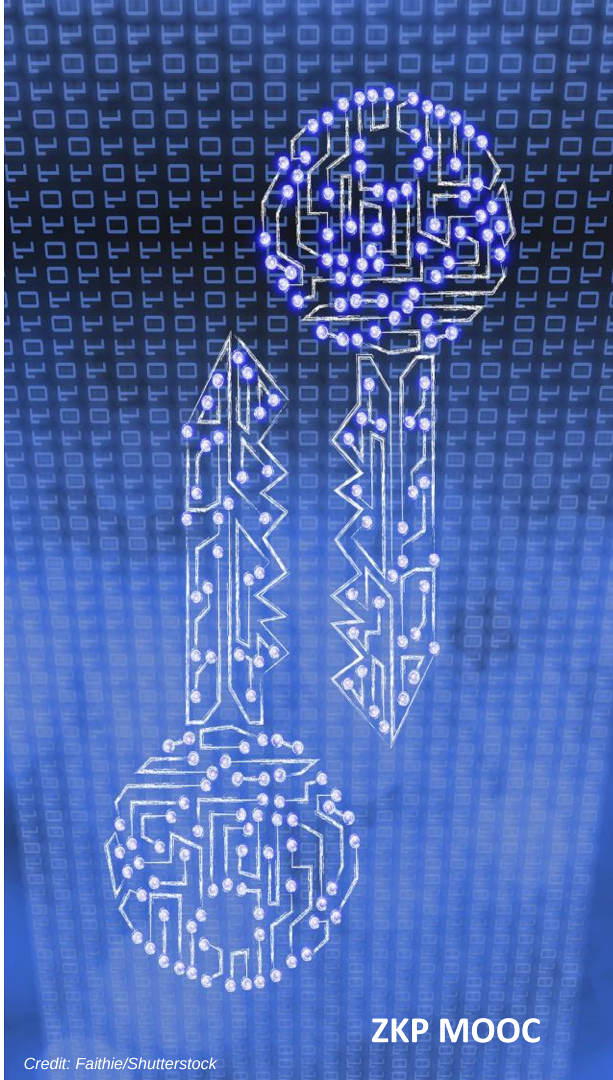
$x$ $om_{E_2}$

This is complete and knowledge sound

- Prover sets

$$z \leftarrow z_1 + r\, z_2, \quad E \leftarrow E_1 + rT + r^2 E_2\ , \quad r_E \leftarrow r_{E_1} + r \cdot r_T + r^2 \cdot r_{E_2}$$

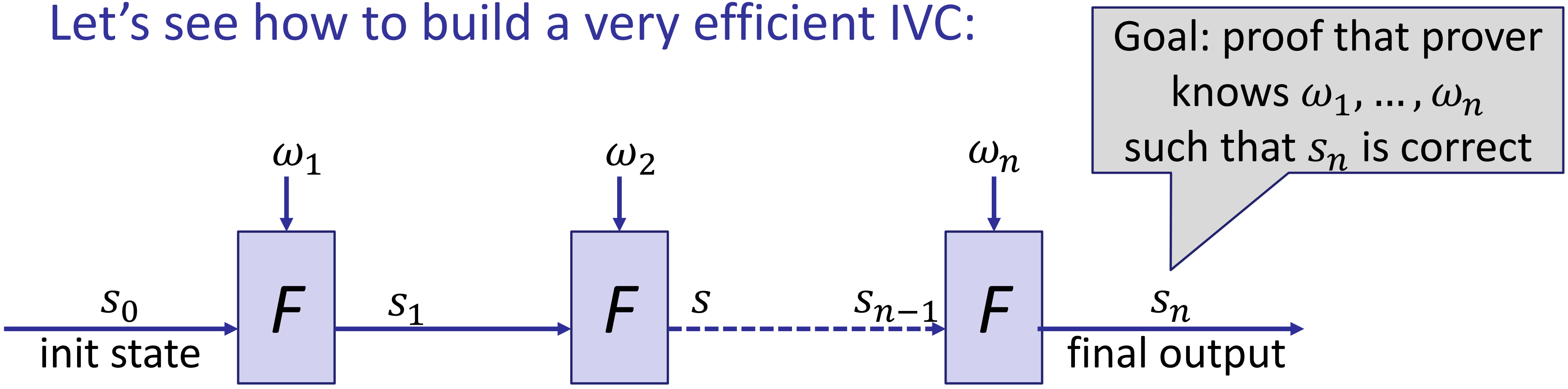
# Putting folding to use …




Credit: Faithie/Shutterstock

# Putting folding to use …

Let's see how to build a very efficient IVC:




Benefit of folding over SNARK recursion:

no need to run the verifier's circuit in SNARK prover

# Putting folding to use …

$A, B, D \in \mathbb{F}_p^{u \times v}$: an R1CS program

A,B,D

$\boldsymbol{x} = (i, s_0, s, s', \omega)$

$w$

check:

$s' = F(s, \omega)$

if $i = 0$ then $s = s_0$

The committed R1CS instance:

**Instance**: $(\boldsymbol{x},\ c,\ \mathrm{com}_E)$

**Witness**: $(w, E, r_E)$

s.t. $z = (x, w)$ satisfies

$$\begin{cases} (Az) \circ (Bz) = c(Dz) + E \\ \mathrm{com}_E = commit(E, r_E) \end{cases}$$

**IVC is a sequence of valid (instance-witness) pairs:**

final output

| | | | |
|---|---|---|---|
| $(0, s_0, s_0, s_1, \omega_1), c_1, com_{E_1}$ | $(1, s_0, s_1, s_2, \omega_2), c_2, com_{E_2}$ | $(2, s_0, s_2, s_3, \omega_3), c_3, com_{E_3}$ | $(3, s_0, s_3, s_4, \omega_4), c_4, com_{E_4}$ |
| $w_1, E_1, r_{E_1}$ | $w_2, E_2, r_{E_2}$ | $w_3, E_3, r_{E_3}$ | $w_4, E_4, r_{E_4}$ |

# Putting folding to use …

$A, B, D \in \mathbb{F}_p^{u \times v}$: an R1CS program

A,B,D

$\boldsymbol{x} = (i, s_0, s, s', \omega)$

$w$

check:

$s' = F(s, \omega)$

if $i = 0$ then $s = s_0$

The committed R1CS instance:

**Instance**: $(\boldsymbol{x},\ c,\ \mathrm{com}_E)$

**Witness**: $(w, E, r_E)$

s.t. $z = (x, w)$ satisfies

$$\begin{cases} (Az) \circ (Bz) = c(Dz) + E \\ \mathrm{com}_E = commit(E, r_E) \end{cases}$$

**IVC is a sequence of valid (instance-witness) pairs:**

| | | | |
|---|---|---|---|
| $\boldsymbol{x}_1, c_1, com_{E_1}$<br>$w_1, E_1, r_{E_1}$ | $\boldsymbol{x}_2, c_2, com_{E_2}$<br>$w_2, E_2, r_{E_2}$ | $\boldsymbol{x}_3, c_3, com_{E_3}$<br>$w_3, E_3, r_{E_3}$ | $\boldsymbol{x}_4, c_4, com_{E_4}$<br>$w_4, E_4, r_{E_4}$ |

# Putting folding to use …

$A, B, D \in \mathbb{F}_p^{u \times v}$: an R1CS program

A,B,D

$\boldsymbol{x} = (i, s_0, s, s', \omega)$

$w$

check:

$s' = F(s, \omega)$

if $i = 0$ then $s = s_0$

The committed R1CS instance:

**Instance**: $(\boldsymbol{x},\ c,\ \mathrm{com}_E)$

**Witness**: $(w, E, r_E)$

s.t. $z = (x, w)$ satisfies

$$\begin{cases} (Az) \circ (Bz) = c(Dz) + E \\ \mathrm{com}_E = commit(E, r_E) \end{cases}$$

**IVC is a sequence of valid (instance-witness) pairs:**

fold 1st and 2nd instances

$\boldsymbol{x}_{12}, c_{12}, com_{E_{12}}$
$w_{12}, E_{12}, r_{E_{12}}$

$\boldsymbol{x}_3, c_3, com_{E_3}$
$w_3, E_3, r_{E_3}$

$\boldsymbol{x}_4, c_4, com_{E_4}$
$w_4, E_4, r_{E_4}$

# Putting folding to use …

$A, B, D \in \mathbb{F}_p^{u \times v}$: an R1CS program

A,B,D

$\boldsymbol{x} = (i, s_0, s, s', \omega)$

$w$

check:

$s' = F(s, \omega)$

if $i = 0$ then $s = s_0$

The committed R1CS instance:

**Instance**: $(\boldsymbol{x},\ c,\ \mathrm{com}_E)$

**Witness**: $(w, E, r_E)$

s.t. $z = (x, w)$ satisfies

$$\begin{cases} (Az) \circ (Bz) = c(Dz) + E \\ \mathrm{com}_E = commit(E, r_E) \end{cases}$$

**IVC is a sequence of valid (instance-witness) pairs:**

fold 3rd instance into first two

$\boldsymbol{x}_{13}, c_{13}, com_{E_{13}}$
$w_{13}, E_{13}, r_{E_{13}}$

$\boldsymbol{x}_4, c_4, com_{E_4}$
$w_4, E_4, r_{E_4}$

# Putting folding to use …

$A, B, D \in \mathbb{F}_p^{u \times v}$: an R1CS program

$A, B, D$

$\boldsymbol{x} = (i, s_0, s, s', \omega)$

$w$

check:

$s' = F(s, \omega)$

if $i = 0$ then $s = s_0$

The committed R1CS instance:

**Instance**: $(\boldsymbol{x},\ c,\ \mathrm{com}_E)$

**Witness**: $(w, E, r_E)$

s.t. $z = (x, w)$ satisfies

$$\begin{cases} (Az) \circ (Bz) = c(Dz) + E \\ \mathrm{com}_E = commit(E, r_E) \end{cases}$$

**IVC is a sequence of valid (instance-witness) pairs:**

fold 4th instance into first three

$\boldsymbol{x}_{\mathbf{14}}, c_{14}, com_{E_{14}}$

$w_{14}, E_{14}, r_{E_{14}}$

# The key point …

After all the superfast folding is done:

$$\boldsymbol{x}_{\mathbf{14}}, c_{14}, com_{E_{14}}$$
$$w_{14}, E_{14}, r_{E_{14}}$$

- Verifier has instance $(\boldsymbol{x}_{\mathbf{14}}, c_{14}, com_{E_{14}})$.
- Prover needs to prove that $(w_{14}, E_{14}, r_{E_{14}})$ is a valid witness

Use whatever proof system to prove that this <u>single</u> pair is valid

Note: for a proving marketplace, fold in a tree structure so that folding can be carried out in parallel by different parties.

# Unfortunately … not so simple

To make this non-interactive: use Fiat-Shamir

- Folding the first pair: prover does $r_{13} \leftarrow H(\boldsymbol{x}_{12}, \boldsymbol{x}_3, \mathrm{com}_{T_{13}}, \ldots)$ and

$$\mathbf{x}_{13} \leftarrow \mathbf{x}_{12} + r_{13}\mathbf{x}_3\,, \quad c_{13} \leftarrow c_{12} + r_{13}, \quad \mathrm{com}_{E_{13}} \leftarrow \mathrm{com}_{E_{12}} + r_{13} com_{T_{13}}$$

$\Rightarrow$ prover needs to prove that folding was done correctly

- Needs to prove that it used the correct $r_{13} \in \mathbb{F}_{\mathrm{p}}$ (otherwise not sound)

Also need to link all instances: output of step $i$ is input of step $i+1$

# Unfortunately … not so simple

How?    Augment R1CS (A,B,D) to also check folding.

Augmented R1CS program to $(A', B', D')$:        [details omitted]

- takes a hash of three $(A', B', D')$ instances as input:  instance $\boldsymbol{x_i}$ ,
  accumulated instance $\boldsymbol{x_{1\to i}}$ ,  folded instance $\boldsymbol{x_{1\to i+1}}$
- Verify that given witness is valid for instance $\boldsymbol{x_i}$ with respect to $(A, B, D)$
- $\underbrace{\text{Run folding alg.}}_{\text{two multiplications in } \mathbb{G}}$ to verify that $\boldsymbol{x_{1\to i+1}}$ is the correct folding of $\boldsymbol{x_{1\to i}}$ and $\boldsymbol{x_i}$

# Prover's work at each step

At each folding step:

- prover manipulates a witness for an R1CS program that does

  (i) evaluate $F$, $\boxed{\text{(ii) do two multiplications in } \mathbb{G}}$

  (iii) do some simple hashing.

$\Rightarrow$ much faster than proving a SNARK verification circuit for $F$

# Supernova

**Nova**: repeated application of the same function $F$

(same relaxed R1CS program)

**Supernova**:

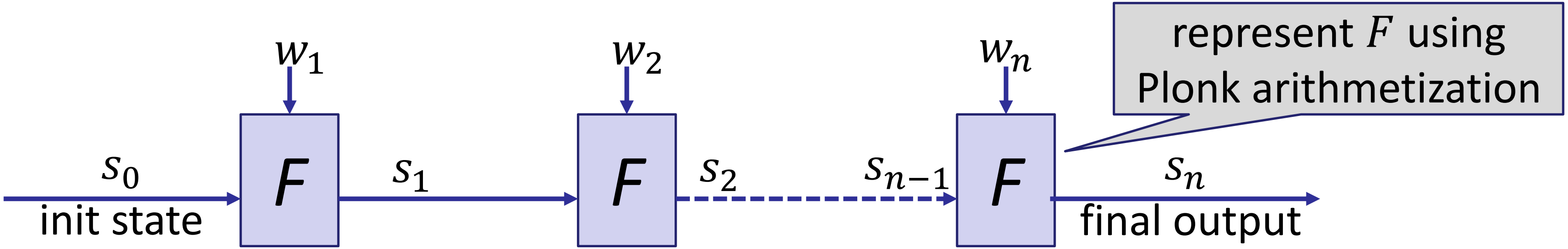
- supports $F_1, \dots, F_k$ in chain (each one may appear multiple times)
- How? apply Nova to each set of $F_i$ separately

# Generalizations: Sangria

Nova's folding scheme applies to any quadratic constraint system

**Sangria**: a folding technique for Plonk arithmetization

$\Rightarrow$ an efficient IVC using Plonk arithmetization



# END OF LECTURE

This completes the part of the course on efficient SNARK constructions