Zero Knowledge Proofs

Recursive SNARKs

Instructors: Dan Boneh, Shafi Goldwasser, Dawn Song, Justin Thaler, Yupeng Zhang
A preprocessing SNARK is a triple \((S, P, V)\):

- \(S(C) \rightarrow\) public parameters \((pp, vp)\) for prover and verifier
- \(P(pp, x, w) \rightarrow\) proof \(\pi\)
- \(V(vp, x, \pi) \rightarrow\) 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

SNARK prover $P$

\[\text{public: } x\]
\[\text{witness: } w\]

proves $P$ knows $w$ s.t. $C(x, w) = 0$

SNARK prover $P'$

\[\text{public: } \nu p, x\]
\[\text{witness: } \pi\]

proves $P'$ knows $\pi$ s.t. $V(\nu p, x, \pi) = \text{yes}$

Use $V'(\nu p', x, \pi')$ to verify final proof $\pi'$
Application 1: proof compression

public: \( x \)
witness: \( w \)

SNARK prover \( P \)

\( (S, P, V) \)

\( \pi \)

\( x \)

\( \pi' \)

public: \( v_P, x \)
witness: \( \pi \)

SNARK prover \( P' \)

\( (S', P', V') \)

\( \Rightarrow \) fast overall prover, and final proof is short
(used to prove complex statements)

fast prover & verifier, but outputs a 100KB proof

slower prover, 1KB final proof
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 $y \in \mathbb{F}^n$:
    $$\Pr[C(y, w) = 0 : w \leftarrow E(pp, y)] \geq \Pr[V(vp, y, A(pp, y)) = \text{yes}] - \varepsilon$$

  - Knowledge error
  - Prob. $E$ extract a valid witness for $y$
  - Prob. adv. $A$ outputs a convincing proof for $y$
  - $\varepsilon$ negligible
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 \( C'((vp,x), \pi) \) be the circuit [ \( V(vp, x, \pi) == \text{`yes`} \) ]
- 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:

- **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. $C(x, w) = 0$

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

Success probability: let $w$ be the extracted witness

$$\Pr[C(x, w) = 0] \geq \Pr[\pi' \leftarrow A(pp, x) \text{ is a convincing proof}] - \varepsilon' - \varepsilon$$

Prob. $E'$ outputs a valid $\pi$

Prob. $E$ outputs a valid $w$

Caution:

Suppose $\text{time}(E') = 2 \times \text{time}(A)$, $\text{time}(E) = 2 \times \text{time}(E') \Rightarrow \text{time}(E) = 4 \times \text{time}(A)$

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

$\Rightarrow$ can only prove security of recursion of depth $\log(\text{security parameter } \lambda)$
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.

- start of batch of 100 Tx
- Tx1, ..., Tx10
- generate proof $\pi_1$
- Tx11, ..., Tx20
- generate proof $\pi_2$
- ... Tx91, ..., Tx100
- generate proof $\pi_{10}$
- generate proof of proofs
- $\pi$
- much shorter delay
- Post batch to L1
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

Layer-1 blockchain (L1)

Rollup state (L2)

Alice (on L2): 4 ETH, 1 DAI
Bob (on L2): 3 ETH, 2 DAI

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

Alice: 10 ETH
Bob: 17 DAI
Transfers inside Rollup are easy

Alice: 10 ETH
Bob: 17 DAI

[A→B: 2 ETH], $\text{sig}_A$
(along with hundreds of Tx)

Rollup state (L2)
Alice (on L2): 4 ETH, 1 DAI
Bob (on L2): 3 ETH, 2 DAI
Rollup contract: Merkle state root
assets: 7 ETH, 3 DAI, ...

Layer-1 blockchain (L1)
Transfers inside Rollup are easy

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

- Alice (on L2): 2 ETH, 1 DAI
- Bob (on L2): 5 ETH, 2 DAI

Rollup contract: **updated state root**

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

Note: no assets move on the L1

Layer-1 blockchain (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

⇒ Much more expensive than in-Rollup transfers
Running a dApps in a Rollup

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

- Alice: $\text{L2-state}_A$
- Bob: $\text{L2-state}_B$
- dApp: $\text{L2-state}$
- L2 Rollup (e.g., zkEVM)

exchange 4 ETH for DAI

proof $\pi$ of correct execution, new state root

- Alice: $\text{state}_A$
- Bob: $\text{state}_B$
- L2 Rollup contract: state root, assets
- Layer-1 blockchain (L1)
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

Alice: $\text{state}_A$  Bob: $\text{state}_B$  L2 Rollup contract: state root, assets  ...
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

L2 Rollup -- org [0]

Alice: \( \text{state}_A \)
Bob: \( \text{state}_B \)

L2 Rollup -- org [1]

root[0], root[1], root[2] holds assets of all Rollups

L2 Rollup contract

L2 Rollup -- org [2]

\[ \cdots \]

Layer-1 blockchain (L1)
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 → L1 settlement rate

What to do?
- Run an L3 on top of its L2

⇒ requires recursive state transition proofs
Layer-3 zk-Rollup

L3 Rollup (Game)

Alice (on L3): **L3state**  
Bob (on L3): **L3state**  

L2 Rollup (Gaming Co)

state root of L3, assets of L3  
**L3 Rollup contract (fancy EVM)**

Layer-1 blockchain (L1)

Alice: **state_A**  
Bob: **state_B**  

root[0], root[1], root[2] holds assets of all Rollups  
**L2 Rollup contract (EVM)**  

...
Layer-3 zk-Rollup

Alice on L3: [send an NFT to a dApp L3], $\text{sig}_A$ (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** $\pi_2$ 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
Consider a long computation done by iterating a function $F$:

\[
\omega_1 \xrightarrow{F} s_1 \xrightarrow{F} s_2 \xrightarrow{F} s_{n-1} \xrightarrow{F} s_n
\]

**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:

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

$\Rightarrow$ 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 $i$ ($i > 0$)

Prover $P$

$F$

$\omega_i$

$(vp, i-1, s_0, S_{i-1})$

proof $\pi_{i-1}$

Verifier

$(vp, n, s_0, S_n )$

statement

$\pi_n$

yes/no

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 $V(vp, (i-1, s_0, S_{i-1}), \pi_{i-1}) = yes$
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

\[
\begin{align*}
\text{init state } & \quad F \quad state \ s_0 \quad F \quad state \ s_1 \quad F \quad state \ s_n \quad F \\
\omega_1 & \quad \omega_2 & \quad \omega_n & \\
\text{final output } & \\
\end{align*}
\]
2. A succinct proof that current state of blockchain is correct

\[ s_0 : \text{initial state of chain, } s_n : \text{current state of chain} \]

\[ \omega_1, \ldots, \omega_n : \text{blocks of valid transactions} \]

Used in Mina blockchain \( \Rightarrow \) verify state of chain by checking one recursive proof

3. **Verifiable Delay Functions (VDF):** succinct proof that \( s_n \) is equal to \( H^{(n)}(s_0) \)
Anyone with a GPU will be paid to create ZK proofs
Choosing Curves to Support Recursion
Two level SNARK recursion: proving knowledge of a proof

**Public:** $x$

**Witness:** $w$

**SNARK prover $P$**

proves $P$ knows $w$ s.t. $C(x, w) = 0$

**Public:** $VP, x$

**Witness:** $\pi$

**SNARK prover $P'$**

proves $P'$ knows $\pi$ s.t. $V(\nu P, x, \pi) = yes$

**Inner proof system**

$(S, P, V)$

**Outer proof system**

$(S', P', V')$
Fix a circuit $C: \mathbb{F}_p^m \times \mathbb{F}_p^n \rightarrow \mathbb{F}_p$ 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?
We say that $\mathbb{G}$ is an **algebraic group defined over** $\mathbb{F}_q$ if $\mathbb{G} \subseteq \mathbb{F}_q^\ell$ and

- the group operation can be computed by polynomials over $\mathbb{F}_q$:
  
  there are polynomials $(f_1, \ldots, f_\ell) \in \mathbb{F}_q[X^\ell]$ such that
  
  for all $a, b \in \mathbb{G}$ we have that $a + b = (f_1(a, b), \ldots, f_\ell(a, b)) \in \mathbb{G}$

- 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 $G$ 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 $G$.

$\Rightarrow$ arithmetic in $\mathbb{F}_q$.

Verifier $V$
What to do?

**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 always easy in such groups

ex: a pairing circuit (as in KZG eval) is huge
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 $\mathbb{F}_r$

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

$|\mathbb{G}_2| = q \subseteq \mathbb{F}_r^\ell$
Solution: a chain of groups

Now, to do a two-level recursion:

- Inner proof system \((S, P, V)\) uses poly. commitments in \(\mathbb{G}_1\)
  \[\Rightarrow\quad \text{Prover } P \text{ supports circuits over } \mathbb{F}_p, \text{ Verifier needs arithmetic in } \mathbb{F}_q\]

- Outer proof system \((S', P', V')\) uses poly. commitments in \(\mathbb{G}_2\)
  \[\Rightarrow\quad \text{Prover } P' \text{ supports circuits over } \mathbb{F}_q, \text{ verifier needs arithmetic in } \mathbb{F}_r\]

A longer chain of groups supports more levels of recursion
Even better: a cycles of groups

Find groups $G_1$ and $G_2$ such that

- $G_1$ has order $p$, and is defined over $\mathbb{F}_q$
- $G_2$ has order $q$, and is defined over $\mathbb{F}_p$

$|G_1| = p \subseteq \mathbb{F}_q^\ell$
$\mathbb{F}_p^\ell \supseteq |G_2| = q$

enables longer recursion by jumping back and forth between two proof systems
Recursion using a cycle

\[ \text{public: } x\ \text{witness: } w \]
\[ \text{prover } P \]
\[ \text{prove: } \quad C(x, w) = 0 \]

\[ \text{public: } x, vk\ \text{witness: } \pi_1 \]
\[ \text{prover } P' \]
\[ \text{prove: } \quad V(vk, x, \pi_1) = 1 \]

\[ \text{public: } x, vk'\ \text{witness: } \pi_2 \]
\[ \text{prover } P \]
\[ \text{prove: } \quad V'(vk', x, \pi_2) = 1 \]

\[ \cdots \]

\( C \) uses arithmetic in \( \mathbb{F}_p \)
\( \text{Prover } P \) uses \( \mathbb{G}_1 \subseteq \mathbb{F}_q \)

\( V \) uses arithmetic in \( \mathbb{F}_q \)
\( \text{Prover } P' \) uses \( \mathbb{G}_2 \subseteq \mathbb{F}_p \)

\( V' \) uses arithmetic in \( \mathbb{F}_p \)
\( \text{Prover } P \) uses \( \mathbb{G}_1 \subseteq \mathbb{F}_q \)
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

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

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 )
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:** \( \forall P^* \exists E \text{ s.t. } \forall x_1, x_2: P^* \text{ outputs valid } w \text{ for } x \implies E \text{ outputs valid } w_1, w_2 \)
A folding scheme: compress two instances into one

Let \( C : \mathbb{F}_p^n \times \mathbb{F}_p^m \rightarrow \mathbb{F}_p \) be a circuit.

A folding scheme for \( C \) is a protocol between two parties:

\[
(x_1, w_1) \quad (x_2, w_2) \quad \text{Folding Prover} \quad T
\]

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)

\[ C: \mathbb{F}_p^n \times \mathbb{F}_p^m \rightarrow \mathbb{F}_p \]

(circuit \( C \))

\[ A, B, D \in \mathbb{F}_p^{u \times v} \]

(R1CS program)

\[ (x, w') \in \mathbb{F}_p^{n+m} \]

s.t. \( C(x, w') = 0 \)

(valid statement, witness pair)

\[ z = (x, w) \in \mathbb{F}_p^u \]

s.t. \( (Az) \circ (Bz) = Dz \)

\[ (x_1, x_2) \circ (y_1, y_2) = (x_1 y_1, x_2 y_2) \]
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 \( (A z_i) \circ (B z_i) = D z_i \) for \( i = 1, 2 \)
Folding the two instances into one

**Attempt 1:** verifier chooses $r \leftarrow \mathbb{F}_p$ 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:

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

$$(Az_1) \circ (Bz_1) + r^2 (Az_2) \circ (Bz_2) + r(Az_2) \circ (Bz_1) + r(Az_1) \circ (Bz_2)$$

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

$\Rightarrow$ 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:

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

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

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

**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

\[(Az) \circ (Bz) = \]
\[= (Az_1) \circ (Bz_1) + r^2 (Az_2) \circ (Bz_2) + \underline{r(Az_2) \circ (Bz_1) + r(Az_1) \circ (Bz_2)} \]
\[= \underline{c_1(Dz_1)} + E_1 + \underline{r^2 c_2(Dz_2)} + \underline{r^2 E_2} + r\left[(Az_2) \circ (Bz_1) + (Az_1) \circ (Bz_2)\right] \]
\[= (c_1 + rc_2)(Dz_1 + rDz_2) + E_1 + r^2 E_2 + rT \]
\[= c(Dz) + E \]

\[\Rightarrow \text{ So, } w \text{ is a valid witness for the relaxed R1CS instance } (x, c, E) \]
Why is this knowledge sound? (informal)

For every folding prover $P^*$, there is an extractor $E$ 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, $E$ extracts from $P^*$ valid witnesses $w_1, w_2$ for the two instances

[note: also need to commit to $w$ in the instance]
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 \((x, c, \text{commit}(E, r_E))\); prover has \((z, E, r_E)\)
  - short commitment to \(E\)
- Commitment needs to be “additive” to enable folding
Recall: homomorphic commitment scheme

Two algorithms:

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

Properties:  (informal)

- **binding**: cannot produce $\text{com}$ and two valid openings for $\text{com}$
- **hiding**: $\text{com}$ 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 $m_1, m_2, r_1, r_2$:

  $$\text{commit}(m_1, r_1) + \text{commit}(m_2, r_2) = \text{commit}(m_1 + m_2, r_1 + r_2)$$

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

Many examples: Pedersen, lattice-based, ...
Folding scheme for committed relaxed R1CS

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

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

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 \( \text{com}_T \leftarrow \text{commit}(T, r_T) \) to \( V \).

- **Verifier chooses** \( r \leftarrow \mathbb{F}_p \), sends \( r \) to \( P \), and sets
  \[ x \leftarrow x_1 + r x_2, \quad c \leftarrow c_1 + r c_2, \quad \text{com}_E \leftarrow \text{com}_{E_1} + r \cdot \text{com}_T + r^2 \cdot \text{com}_{E_2} \]

- **Prover sets**
  \[ z \leftarrow z_1 + r z_2, \quad E \leftarrow E_1 + rT + r^2 E_2, \quad r_E \leftarrow r_{E_1} + r \cdot r_T + r^2 \cdot r_{E_2} \]

homomorphic commitment
Folding scheme for committed relaxed R1CS

- Prover computes
  \[ T \leftarrow (A z_2)^\circ B z_1 + (A z_1)^\circ B z_2 - c_1 (D z_2) - c_2 (D z_1) \]
  sends \[ \text{commit} (T, r_T) \] to V.

- Verifier chooses \[ r \leftarrow \mathcal{R} \], sends \[ r \] to P, and sets \[ x \leftarrow x_1 + r x_2 \], \[ c \leftarrow c_1 + r c_2 \], \[ \text{commit} E \leftarrow c_1 \text{commit} E_1 + r \cdot \text{commit} E_2 \]

- Prover sets
  \[ z \leftarrow z_1 + r z_2, \quad E \leftarrow E_1 + r T + r^2 E_2, \quad r_E \leftarrow r_{E_1} + r \cdot r_T + r^2 \cdot r_{E_2} \]

This is complete and knowledge sound
Putting folding to use ...
Let's see how to build a very efficient IVC:

\[
\begin{align*}
\omega_1 & \quad F \quad \omega_1 \\
\omega_2 & \quad F \quad \omega_2 \\
\omega_n & \quad F \quad \omega_n
\end{align*}
\]

\[
\begin{align*}
\text{init state} & \quad s_0 \\
\text{final output} & \quad s_n \\
\text{final output} & \quad s_{n-1}
\end{align*}
\]

Goal: proof that prover knows $\omega_1, \ldots, \omega_n$ such that $s_n$ is correct.

Benefit of folding over SNARK recursion:
no need to run the verifier’s circuit in SNARK prover.
Putting folding to use ...

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

\[ x = (i, s_0, s, s', \omega) \]

\[ A, B, D \]

<table>
<thead>
<tr>
<th>check:</th>
<th>[ s' = F(s, \omega) ] if ( i = 0 ) then ( s = s_0 )</th>
</tr>
</thead>
</table>

\[ w \]

**The committed R1CS instance:**

**Instance:** \( (x, c, \text{com}_E) \)

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

s.t. \( z = (x, w) \) satisfies \[ (Az) \circ (Bz) = c(Dz) + E \]

\[ \text{com}_E = \text{commit}(E, r_E) \]

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

\[ (0, s_0, s_0, s_1, \omega_1), c_1, \text{com}_{E_1} \]
\[ \text{w}_1, E_1, r_{E_1} \]

\[ (1, s_0, s_1, s_2, \omega_2), c_2, \text{com}_{E_2} \]
\[ \text{w}_2, E_2, r_{E_2} \]

\[ (2, s_0, s_2, s_3, \omega_3), c_3, \text{com}_{E_3} \]
\[ \text{w}_3, E_3, r_{E_3} \]

\[ (3, s_0, s_3, s_4, \omega_4), c_4, \text{com}_{E_4} \]
\[ \text{w}_4, E_4, r_{E_4} \]

**final output**
Putting folding to use ...

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

\[ x = (i, s_0, s, s', \omega) \]

Check:

\[ s' = F(s, \omega) \]

if \( i = 0 \) then \( s = s_0 \)

The committed R1CS instance:

**Instance:** \( (x, c, \text{com}_E) \)

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

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

\[ (Az) \circ (Bz) = c(Dz) + E \]

\[ \text{com}_E = \text{commit}(E, r_E) \]

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

\[ \begin{align*}
    &x_1, c_1, \text{com}_{E_1} \\
    &w_1, E_1, r_{E_1} \\
    &x_2, c_2, \text{com}_{E_2} \\
    &w_2, E_2, r_{E_2} \\
    &x_3, c_3, \text{com}_{E_3} \\
    &w_3, E_3, r_{E_3} \\
    &x_4, c_4, \text{com}_{E_4} \\
    &w_4, E_4, r_{E_4}
\end{align*} \]
Putting folding to use ...

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

\[ x = (i, s_0, s, s', \omega) \]

Check:

\[ s' = F(s, \omega) \]

if \( i = 0 \) then \( s = s_0 \)

\[ w \checkmark \]

The committed R1CS instance:

- **Instance:** \((x, c, \text{com}_E)\)
- **Witness:** \((w, E, r_E)\)

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

\[
(Az) \circ (Bz) = c(Dz) + E
\]

\[ \text{com}_E = \text{commit}(E, r_E) \]

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

- Fold 1st and 2nd instances

\[
\begin{align*}
x_{12}, c_{12}, \text{com}_{E_{12}} \\
w_{12}, E_{12}, r_{E_{12}}
\end{align*}
\]

\[
\begin{align*}
x_3, c_3, \text{com}_{E_3} \\
w_3, E_3, r_{E_3}
\end{align*}
\]

\[
\begin{align*}
x_4, c_4, \text{com}_{E_4} \\
w_4, E_4, r_{E_4}
\end{align*}
\]
Putting folding to use ...

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

\[ x = (i, s_0, s, s', \omega) \]

check:

\[ s' = F(s, \omega) \]

if \( i = 0 \) then \( s = s_0 \)

\[ w \]

The committed R1CS instance:

**Instance:** \((x, c, \text{com}_E)\)

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

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

\[
(Az) \circ (Bz) = c(Dz) + E \\
\text{com}_E = \text{commit}(E, r_E)
\]

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

- Fold 3\(^{rd}\) instance into first two
  - \(x_{13}, c_{13}, \text{com}_{E_{13}}\)
  - \(w_{13}, E_{13}, r_{E_{13}}\)

- \(x_{4}, c_{4}, \text{com}_{E_{4}}\)
  - \(w_{4}, E_{4}, r_{E_{4}}\)
Putting folding to use ...

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

\[ x = (i, s_0, s, s', \omega) \]

\[ w \]

check:

\[ s' = F(s, \omega) \]

if \( i = 0 \) then \( s = s_0 \)

The committed R1CS instance:

Instance: \( (x, c, \text{com}_E) \)

Witness: \( (w, E, r_E) \)

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

\[
\begin{align*}
(Az) \circ (Bz) &= c(Dz) + E \\
\text{com}_E &= \text{commit}(E, r_E)
\end{align*}
\]

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

fold 4\textsuperscript{th} instance into first three

\[ x_{14}, c_{14}, \text{com}_{E_{14}} \]

\[ w_{14}, E_{14}, r_{E_{14}} \]
The key point ...

After all the superfast folding is done:

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

Use whatever proof system to prove that this single 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 $r_{13} \leftarrow H(x_{12}, x_3, \text{com}_{T_{13}}, ...)$ and

$$
x_{13} \leftarrow x_{12} + r_{13} x_3,
\ c_{13} \leftarrow c_{12} + r_{13},
\ \text{com}_{E_{13}} \leftarrow \text{com}_{E_{12}} + r_{13} \text{com}_{T_{13}}
$$

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

- Needs to prove that it used the correct $r_{13} \in \mathbb{F}_p$ (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 \(x_i\), accumulated instance \(x_{1\rightarrow i}\), folded instance \(x_{1\rightarrow i+1}\)
- Verify that given witness is valid for instance \(x_i\) with respect to \((A,B,D)\)
- Run folding alg. to verify that \(x_{1\rightarrow i+1}\) is the correct folding of \(x_{1\rightarrow i}\) and \(x_i\) two multiplications in \(\mathbb{G}\)
Prover’s work at each step

At each folding step:

- prover manipulates a witness for an R1CS program that does
  
  (i) evaluate $F$, (ii) do two multiplications in $\mathbb{G}$
  (iii) do some simple hashing.

⇒ 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 $F_1, \ldots, F_k$ 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

⇒ an efficient IVC using Plonk arithmetization

![Diagram](image_url)
END OF LECTURE

This completes the part of the course on efficient SNARK constructions