recursion & proof composition

JAN I 23

1. recursion enables **parallelised proving** of complex computational graphs.

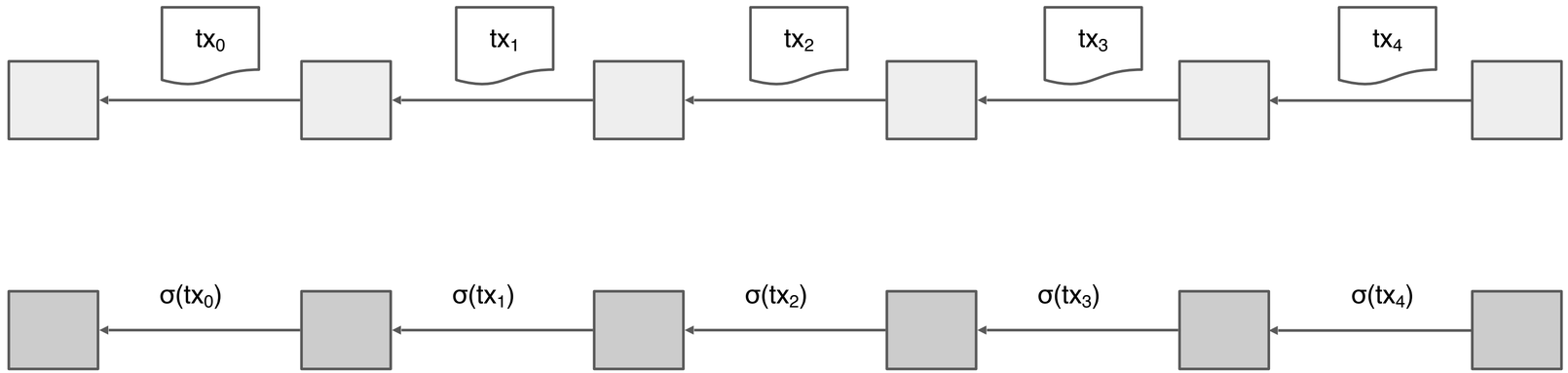
2. proof composition **embeds subprotocols** from different proof systems.

3. recursion + proof composition = 🔥

contents

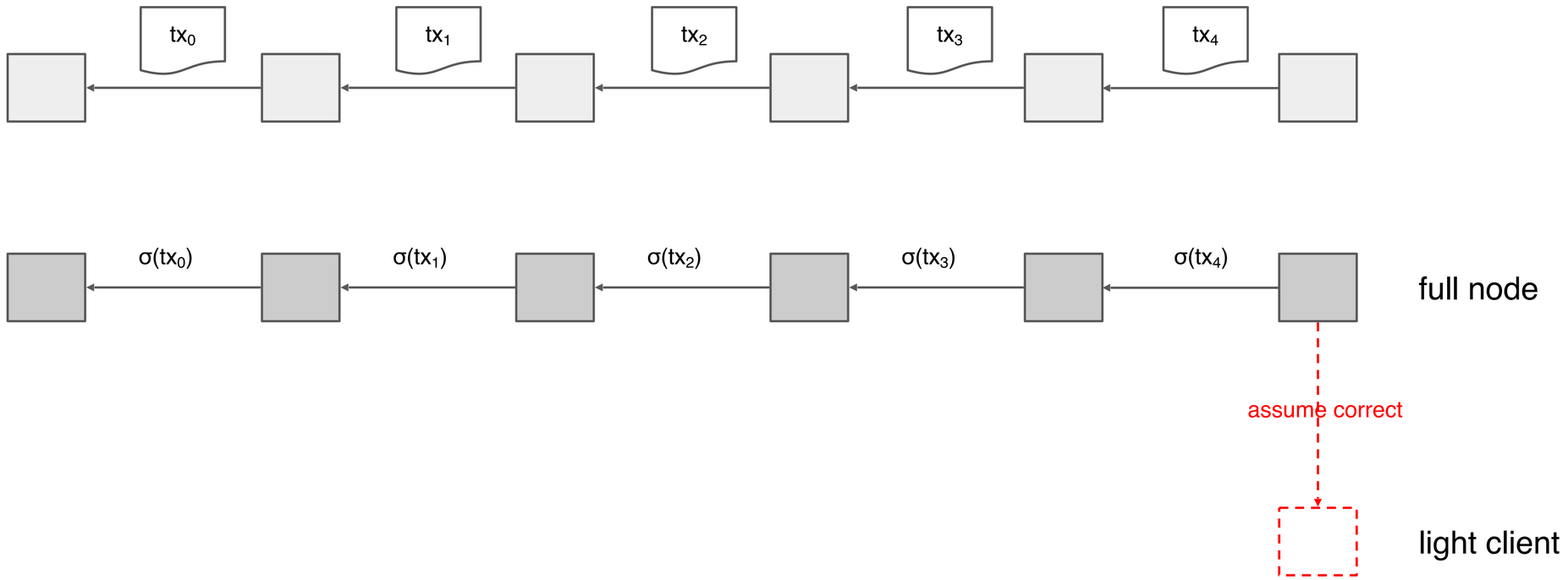
contents

a blockchain in which each block can be verified in **constant time** regardless of the number of prior blocks in the history

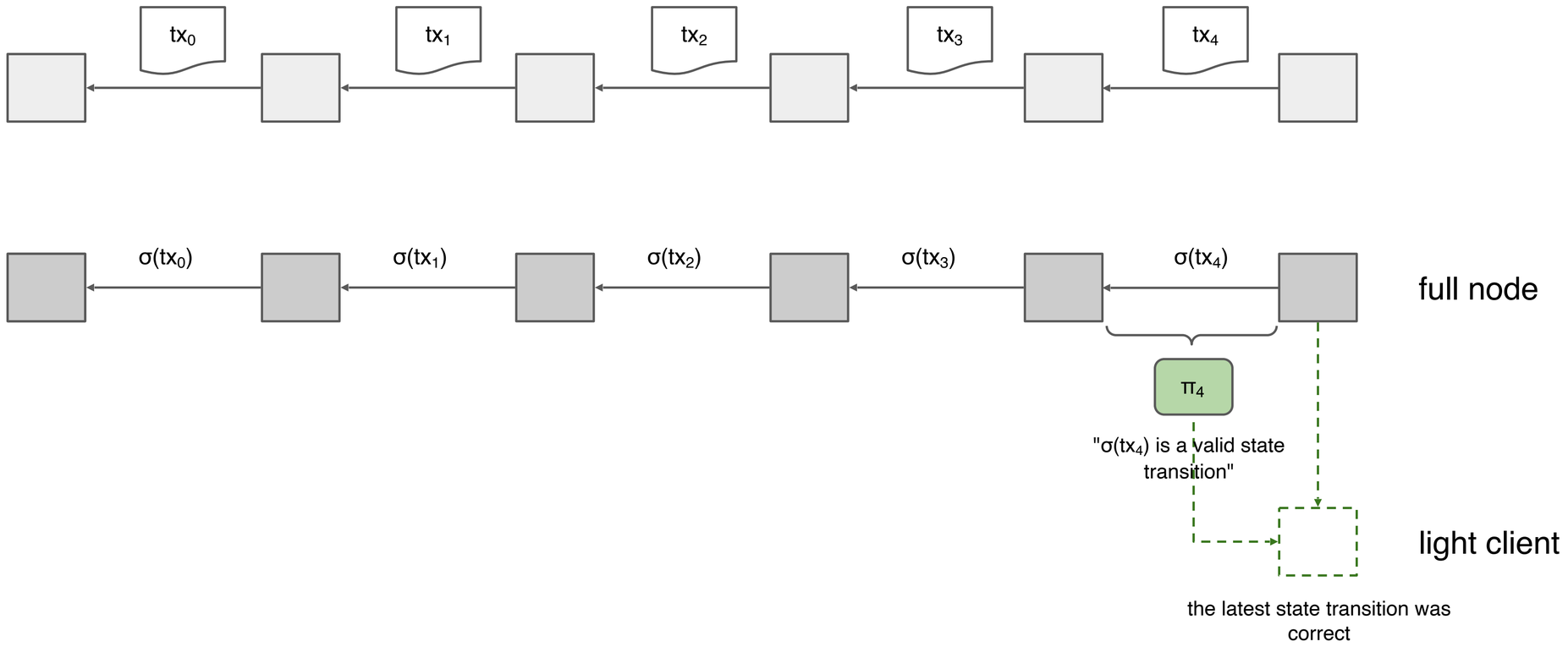


full node

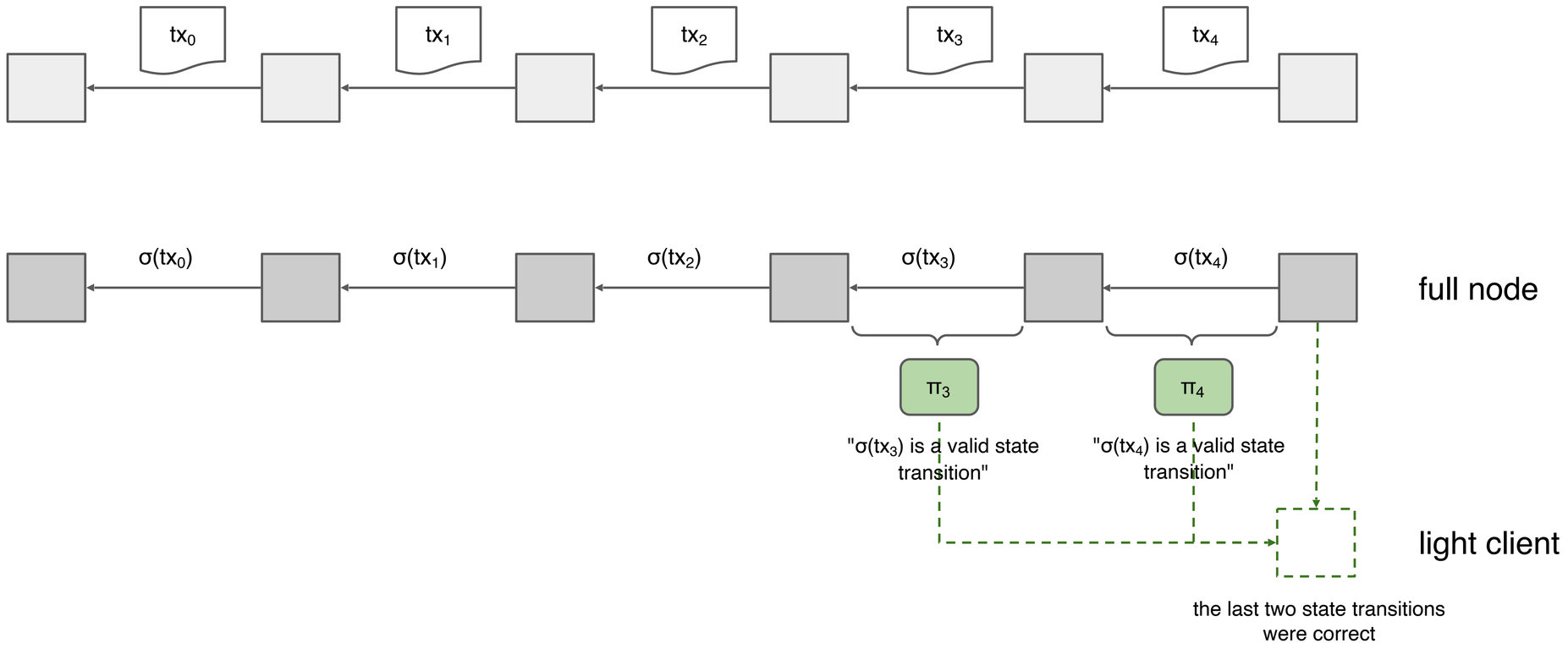
a blockchain in which each block can be verified in **constant time** regardless of the number of prior blocks in the history



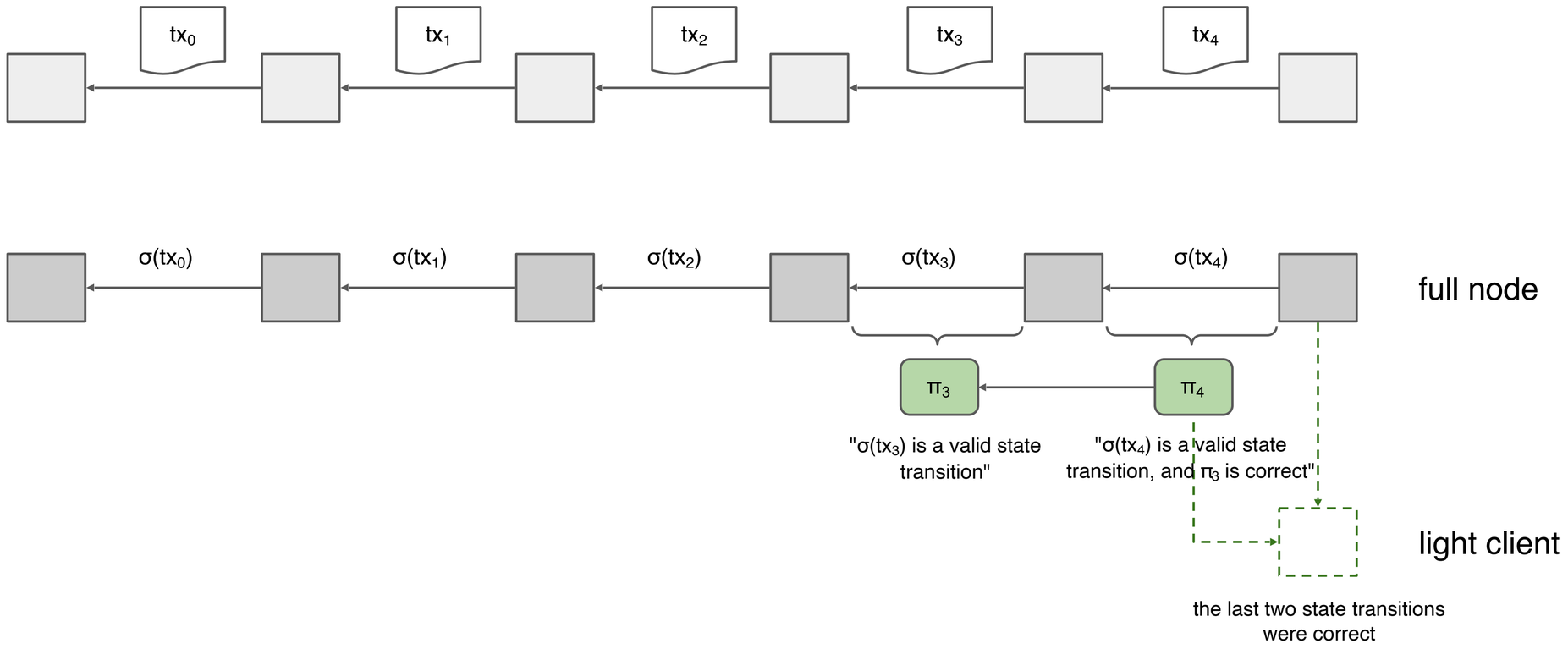
a blockchain in which each block can be verified in **constant time** regardless of the number of prior blocks in the history



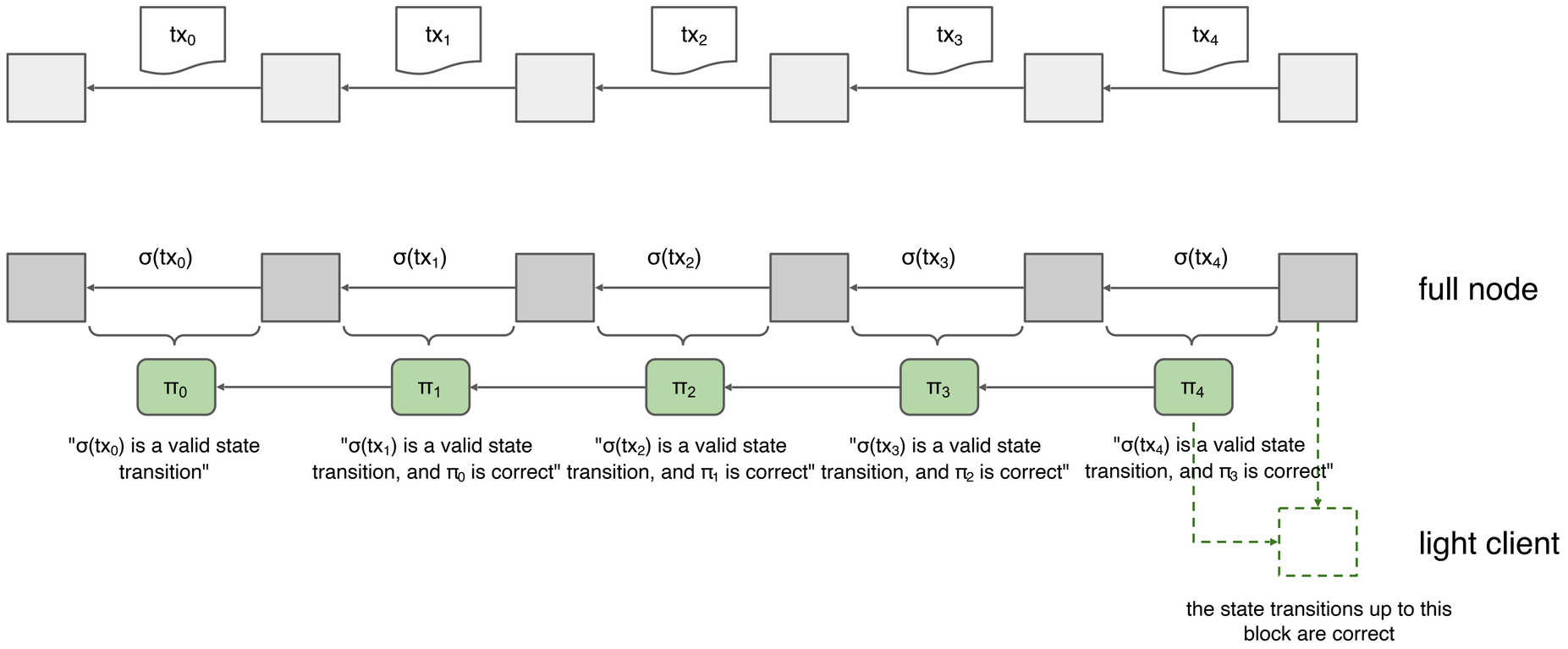
a blockchain in which each block can be verified in **constant time** regardless of the number of prior blocks in the history



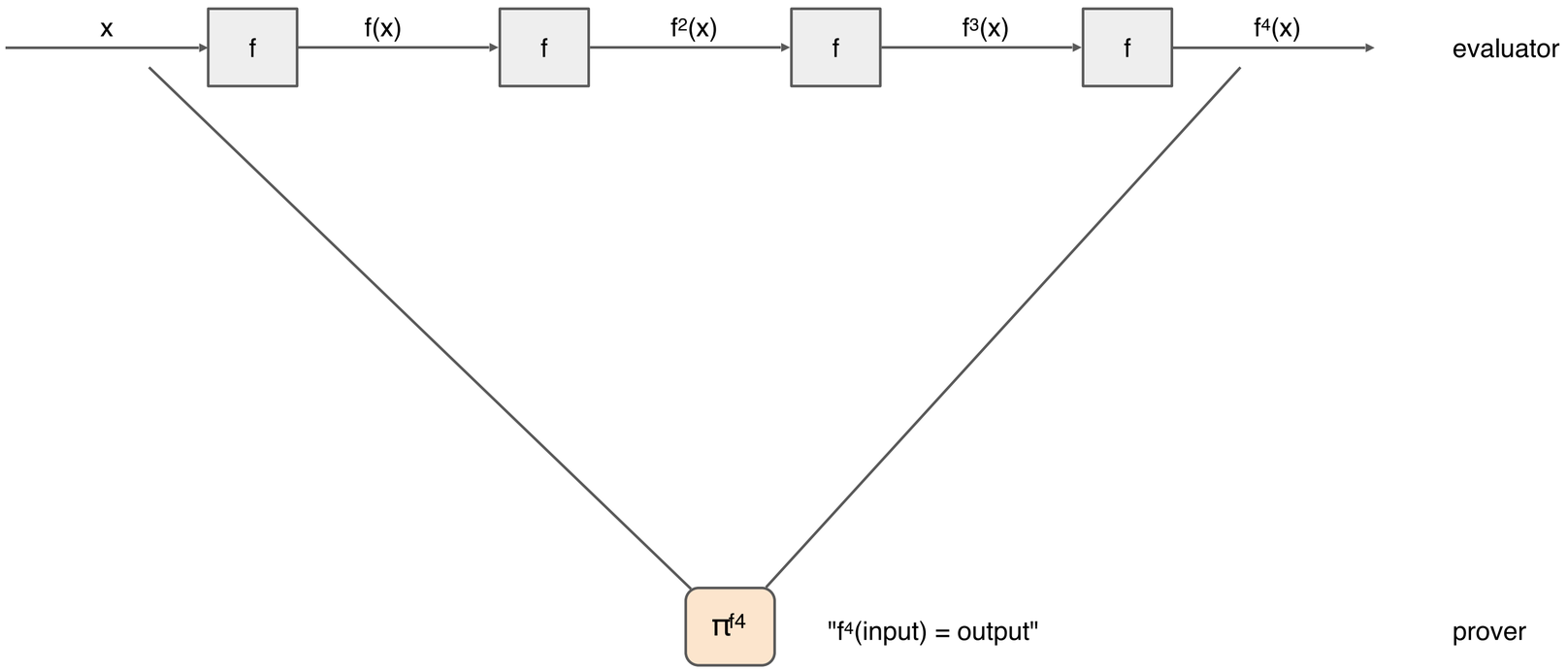
a blockchain in which each block can be verified in **constant time** regardless of the number of prior blocks in the history



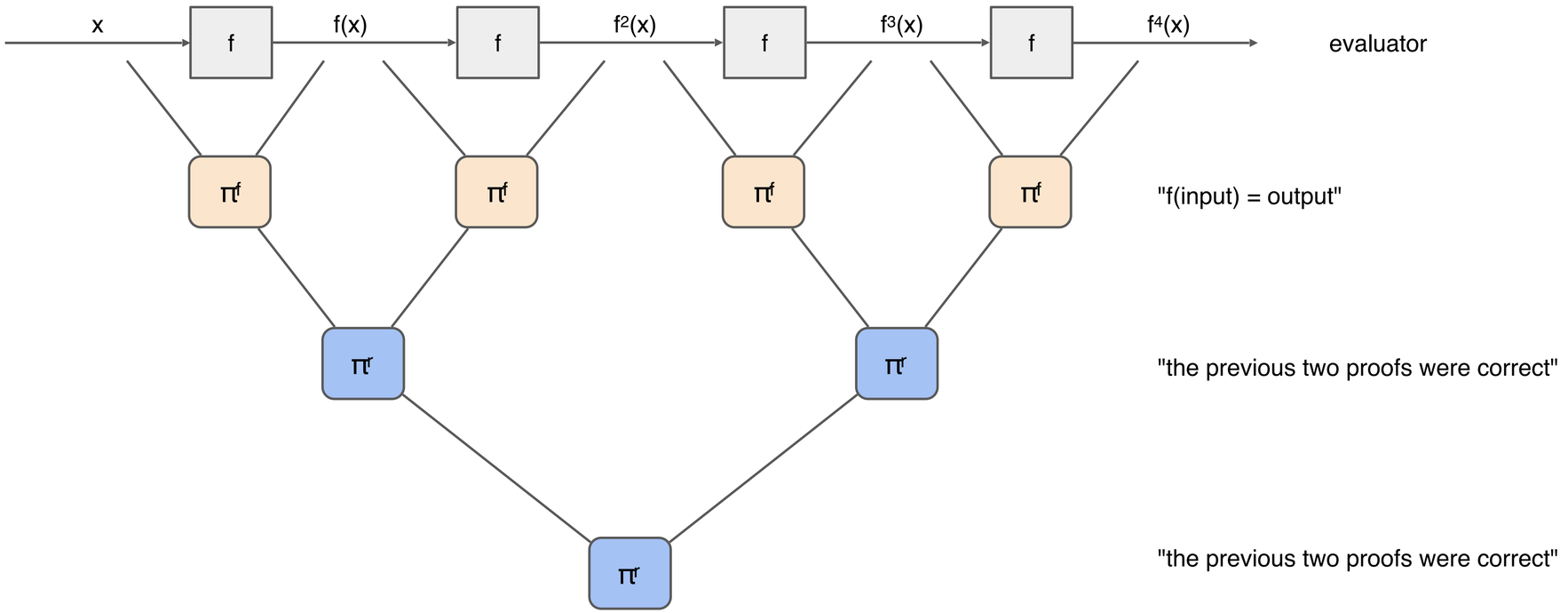
a blockchain in which each block can be verified in **constant time** regardless of the number of prior blocks in the history

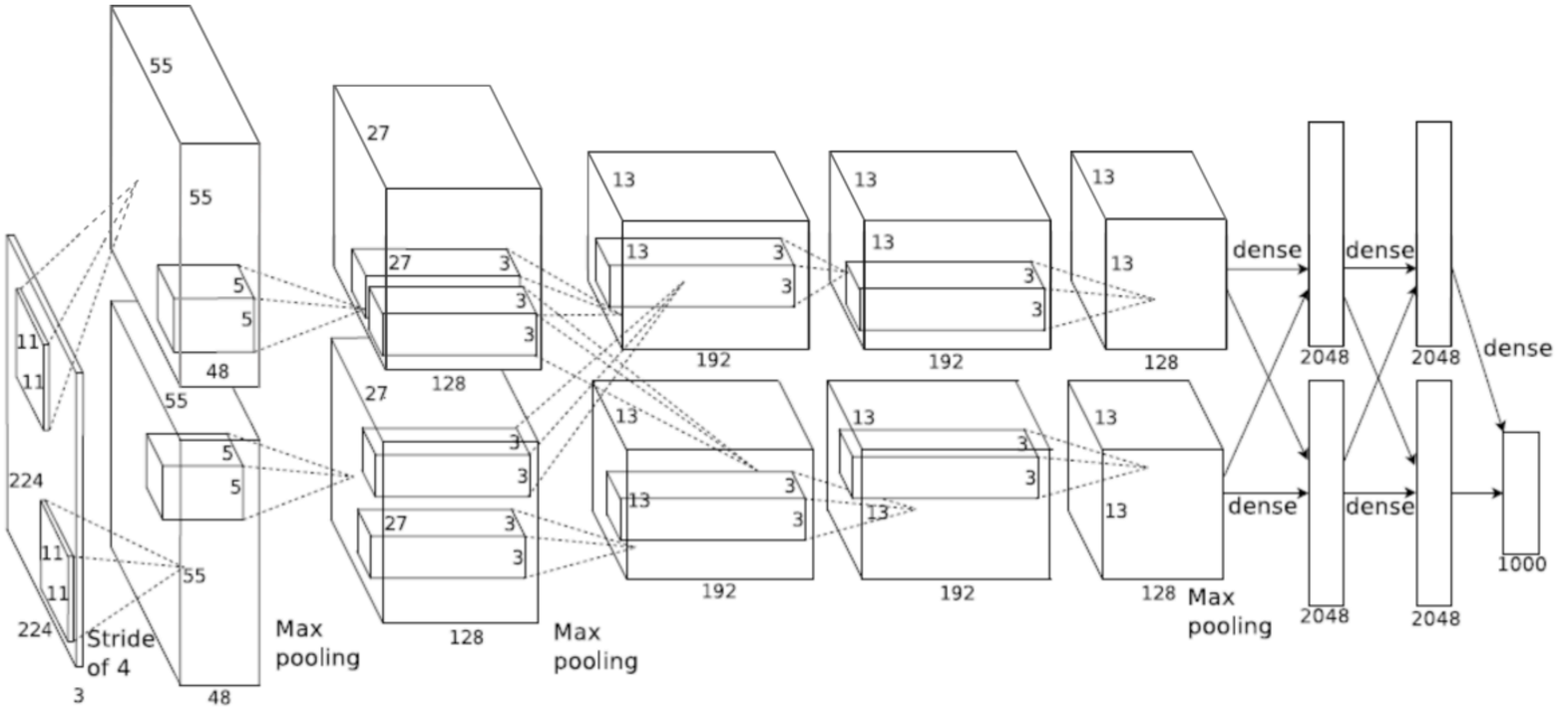


verifiable delay function [BBBF18]: a sequential computation that is slow to compute but efficient to verify



verifiable delay function [BBBF18]: a sequential computation that is slow to compute but efficient to verify



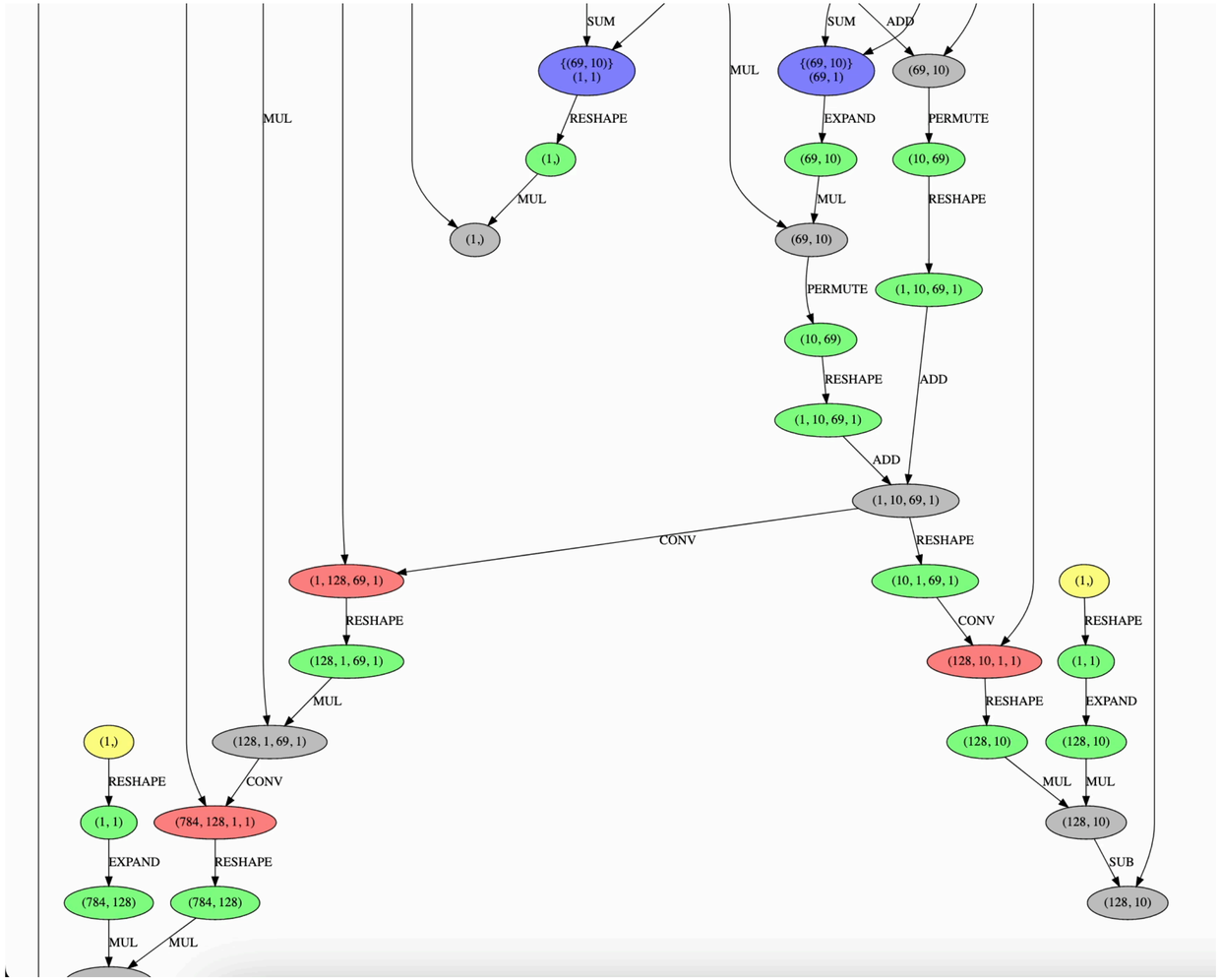


ImageNet Convolutional Neural Network Architecture

```
tinygrad % GRAPH=1 python3 test/
test_mnist.py TestMNIST.test_sgd_onestep

loss 2.35 accuracy 0.06: 100%|
                            | 1/1 [00:00<00:00,
138.37it/s]
.
----------------------------------------
----------------------------
Ran 1 test in 0.028s

OK
<enum 'LoadOps'> 8
<enum 'MovementOps'> 36
<enum 'ProcessingOps'> 5
<enum 'BinaryOps'> 21
<enum 'ReduceOps'> 6
saving DiGraph with 76 nodes and 94 edges
```

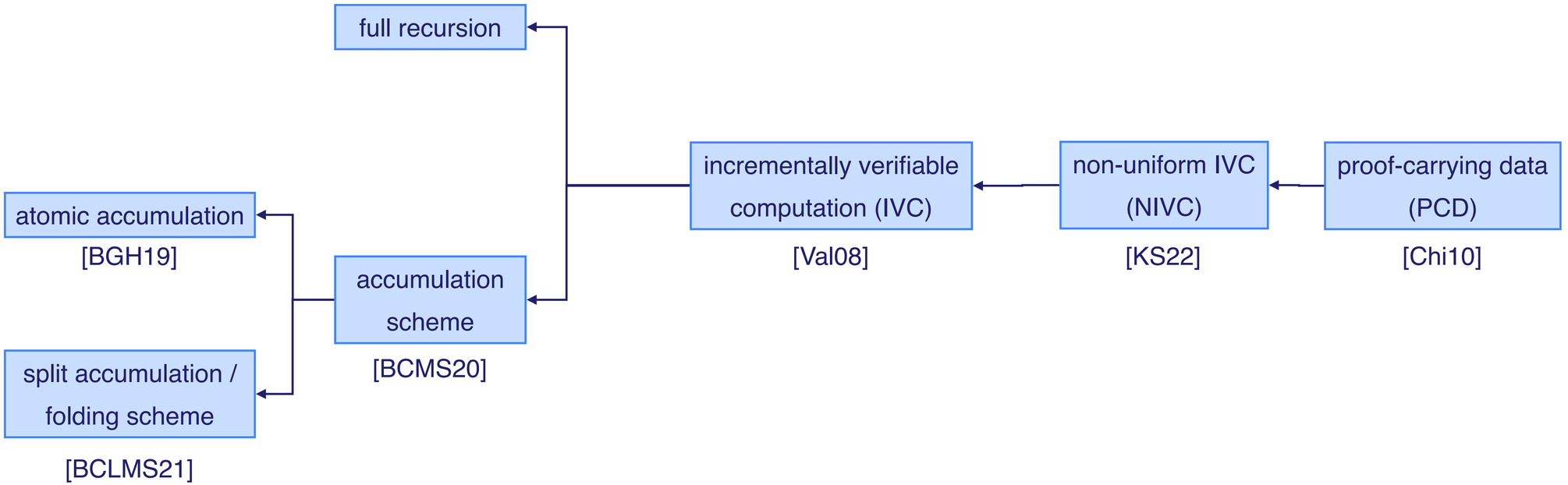


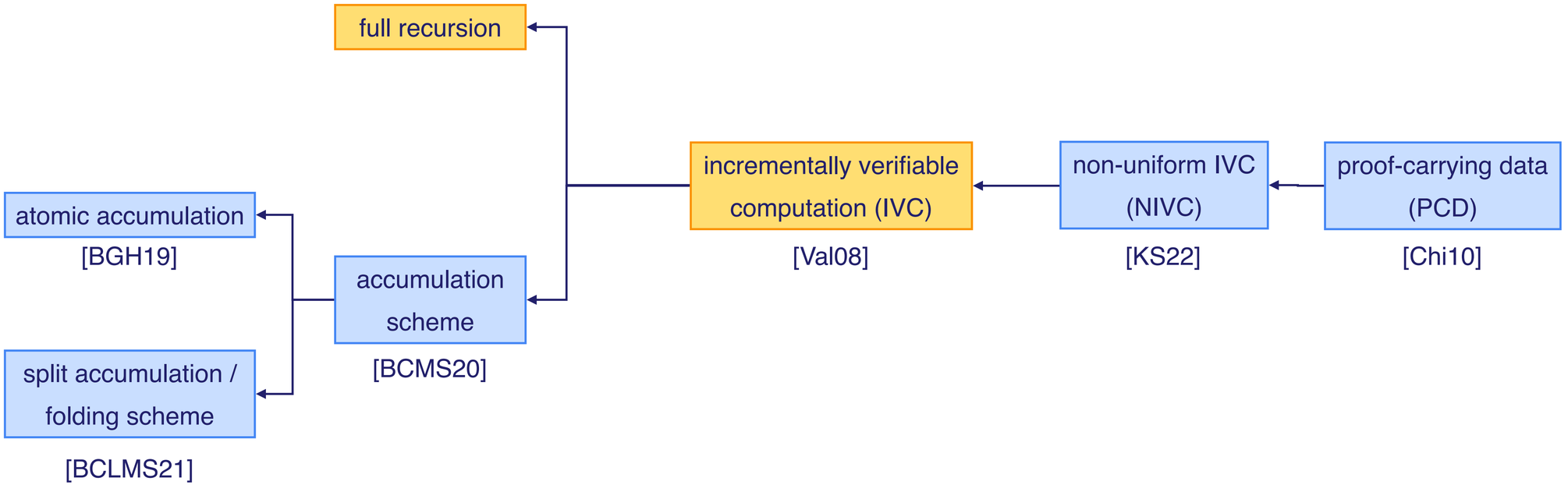
1. **applications:**
   history compression; reduction of prover space complexity; smaller proofs for large circuits

2. **recursive techniques:**
   full recursion; atomic accumulation; split accumulation;
   incrementally verifiable computation (IVC); non-uniform IVC; proof-carrying data (PCD)

3. **composition techniques:**
   recursive verification; linkable commit-and-prove

4. **future directions:**
   unifying frameworks; benchmarking and security; libraries and dev tooling

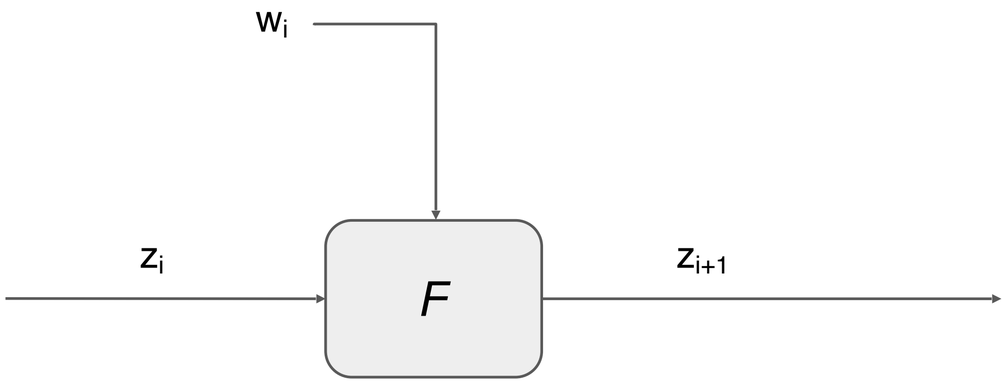

full recursion
incrementally verifiable computation (IVC)
[Val08]
non-uniform IVC (NIVC)
[KS22]
proof-carrying data (PCD)
[Chi10]
atomic accumulation
[BGH19]
accumulation scheme
[BCMS20]
split accumulation / folding scheme
[BCLMS21]


full recursion
incrementally verifiable computation (IVC)
[Val08]
non-uniform IVC (NIVC)
[KS22]
proof-carrying data (PCD)
[Chi10]
atomic accumulation
[BGH19]
accumulation scheme
[BCMS20]
split accumulation / folding scheme
[BCLMS21]

strategy: break $z_n = F^{(n)}(z_0; w_0,\ldots,w_{n-1})$ into recursive applications of F

recursive techniques: IVC from full recursion

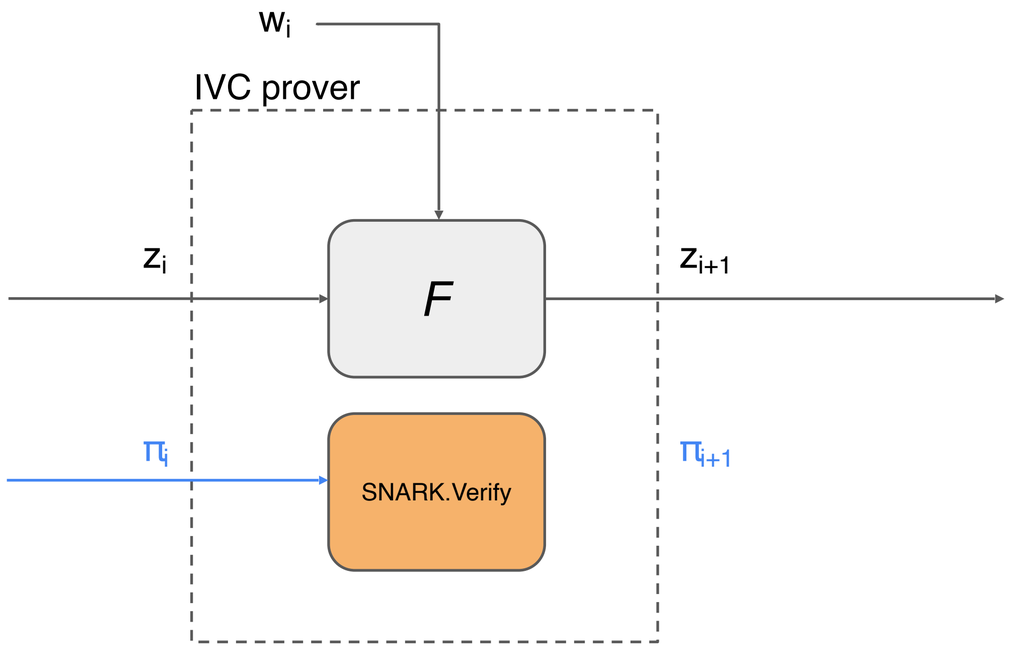
strategy: break $z_n = F^{(n)}(z_0; w_0,\ldots,w_{n-1})$ into recursive applications of F



$z_{i+1} = F(z_i; w_i)$

recursive techniques: IVC from full recursion

strategy: break $z_n = F^{(n)}(z_0; w_0,\ldots,w_{n-1})$ into recursive applications of F

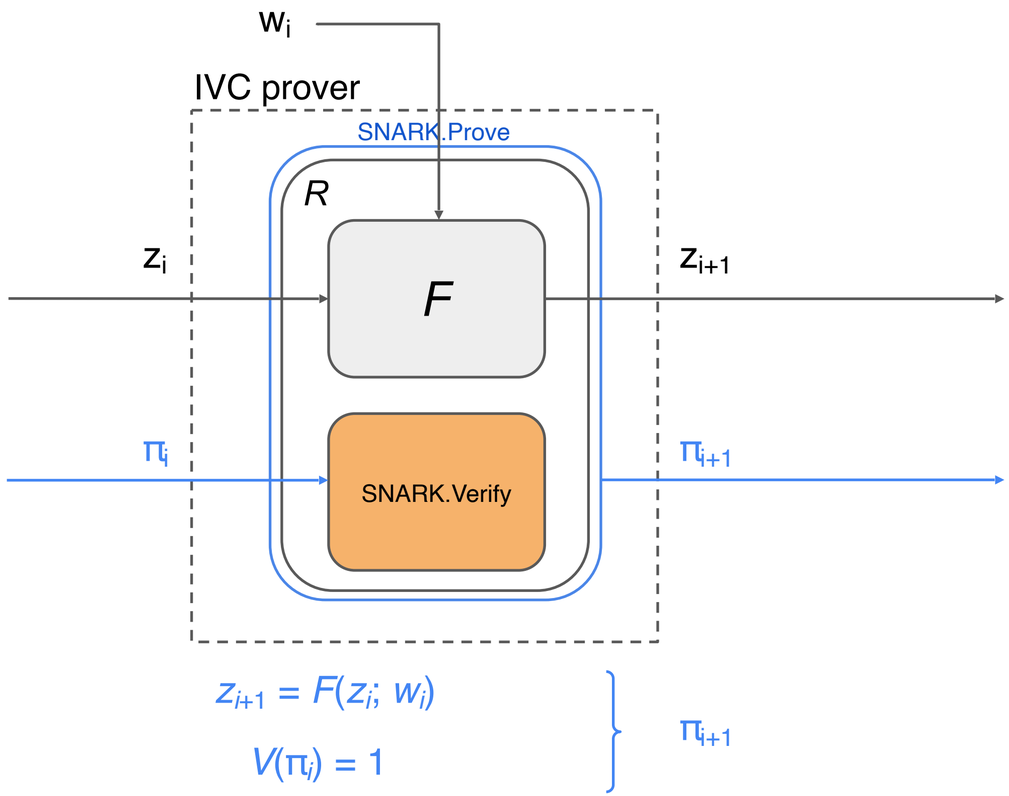


$z_{i+1} = F(z_i; w_i)$

$V(\pi_i) = 1$

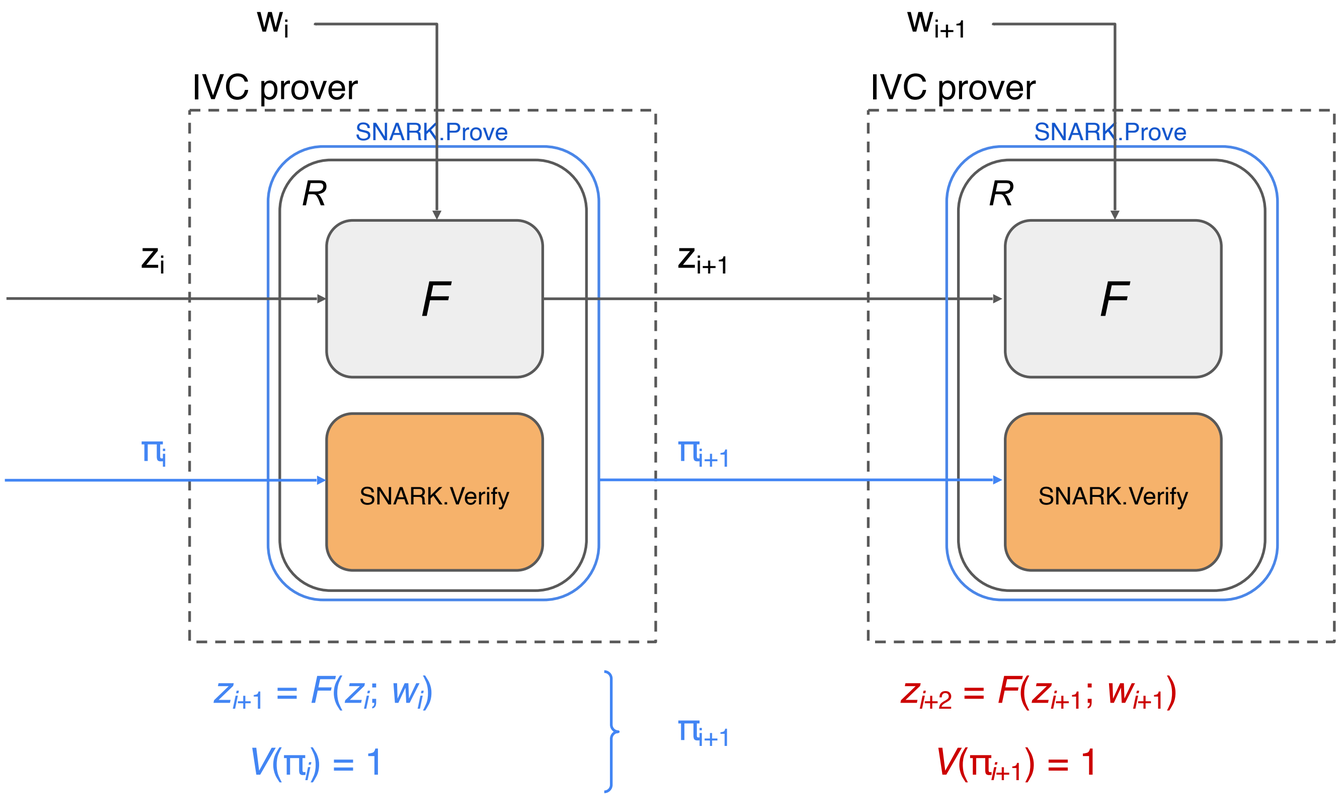
recursive techniques: IVC from full recursion

strategy: break $z_n = F^{(n)}(z_0; w_0,\ldots,w_{n-1})$ into recursive applications of F



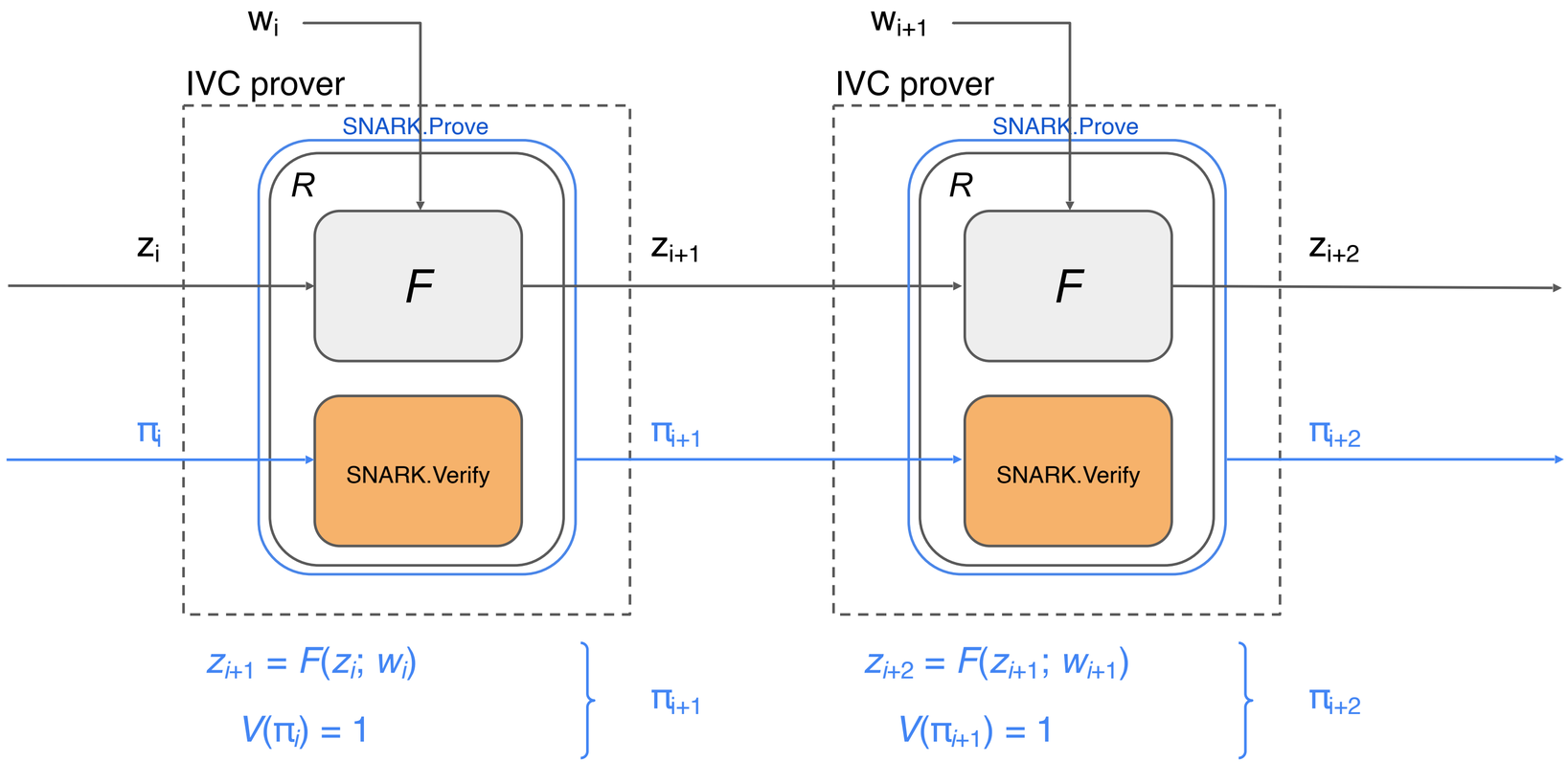
recursive techniques: IVC from full recursion

strategy: break $z_n = F^{(n)}(z_0; w_0,\ldots,w_{n-1})$ into recursive applications of F



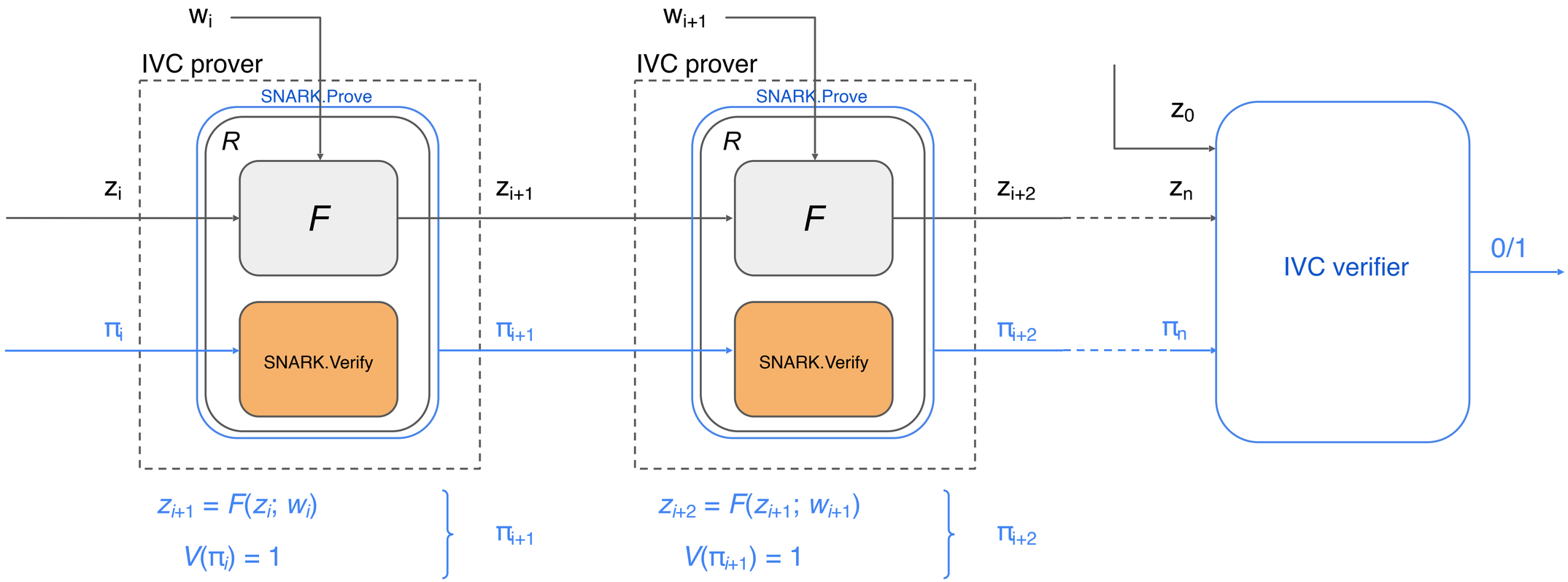
recursive techniques: IVC from full recursion

strategy: break $z_n = F^{(n)}(z_0; w_0,\ldots,w_{n-1})$ into recursive applications of F



recursive techniques: IVC from full recursion

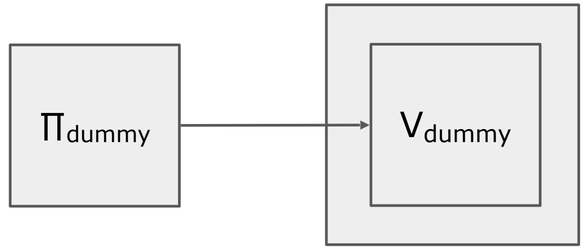
strategy: break $z_n = F^{(n)}(z_0; w_0,\ldots,w_{n-1})$ into recursive applications of F



recursive techniques: IVC from full recursion

example: plonky2

```
// Start with a dummy proof of specified size
let inner = dummy_proof::<F, C, D>(config, log2_inner_size)?;
let (_, _, cd) = &inner;
```



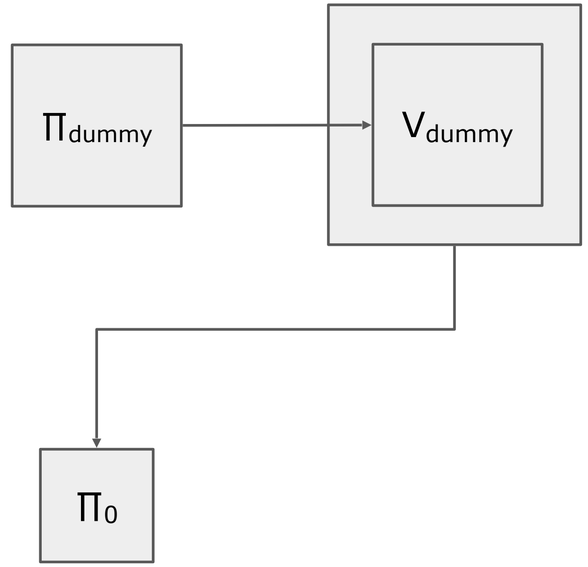
https://github.com/mir-protocol/plonky2/blob/main/plonky2/examples/bench_recursion.rs#L183

recursive techniques: IVC from full recursion

example: plonky2

```
Initial proof degree 16384 = 2^14
Degree before blinding & padding: 4028
Degree after blinding & padding: 4096
```

```rust
// Recursively verify the proof
let middle = recursive_proof::<F, C, C, D>(&inner, config, None)?;
let (_, _, cd) = &middle;
```



https://github.com/mir-protocol/plonky2/blob/main/plonky2/examples/bench_recursion.rs#L183

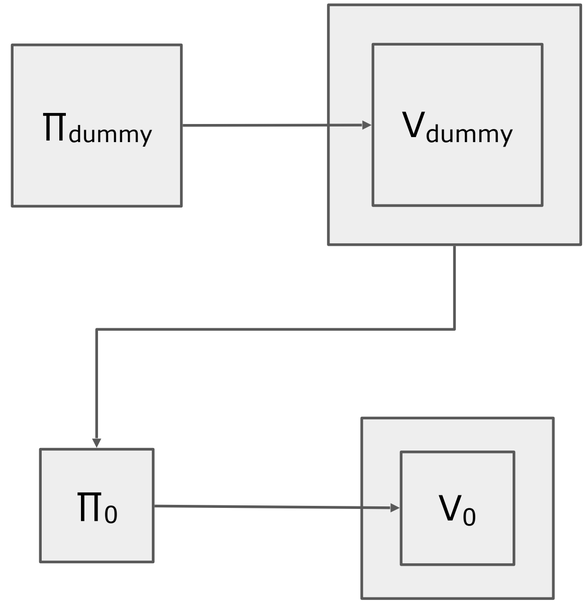
recursive techniques: IVC from full recursion

example: plonky2

```
Initial proof degree 16384 = 2^14
Degree before blinding & padding: 4028
Degree after blinding & padding: 4096
```

```
Single recursion proof degree 4096 = 2^12
Degree before blinding & padding: 3849
Degree after blinding & padding: 4096
```

```
// Add a second layer of recursion to shrink the proof size further
let outer = recursive_proof::<F, C, C, D>(&middle, config, None)?;
let (proof, vd, cd) = &outer;
```



https://github.com/mir-protocol/plonky2/blob/main/plonky2/examples/bench_recursion.rs#L183

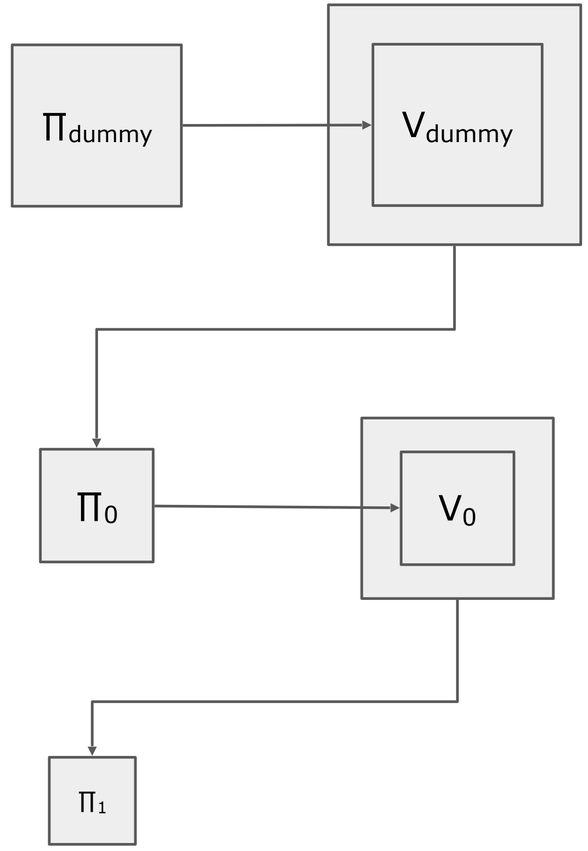
recursive techniques: IVC from full recursion

example: plonky2

```
Initial proof degree 16384 = 2^14
Degree before blinding & padding: 4028
Degree after blinding & padding: 4096
```

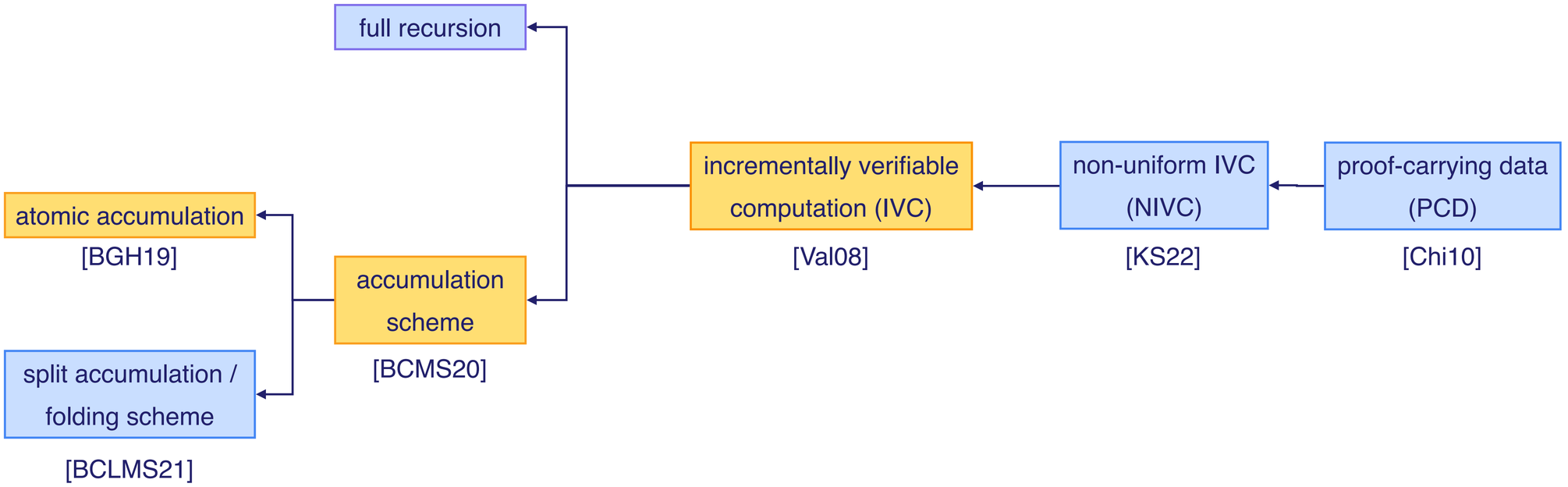
```
Single recursion proof degree 4096 = 2^12
Degree before blinding & padding: 3849
Degree after blinding & padding: 4096
```

```
Double recursion proof degree 4096 = 2^12
Proof length: 127184 bytes
0.2511s to compress proof
Compressed proof length: 115708 bytes
```

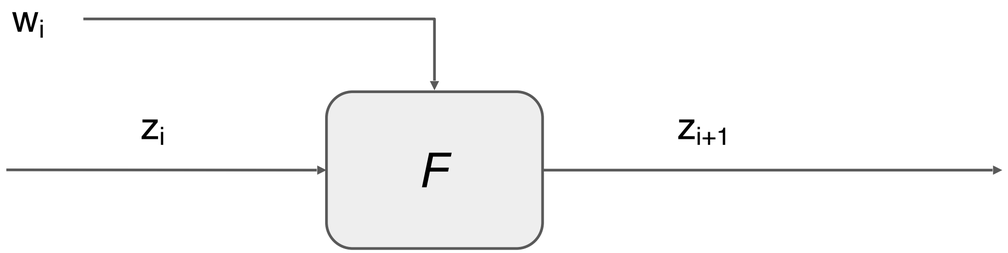


https://github.com/mir-protocol/plonky2/blob/main/plonky2/examples/bench_recursion.rs#L183

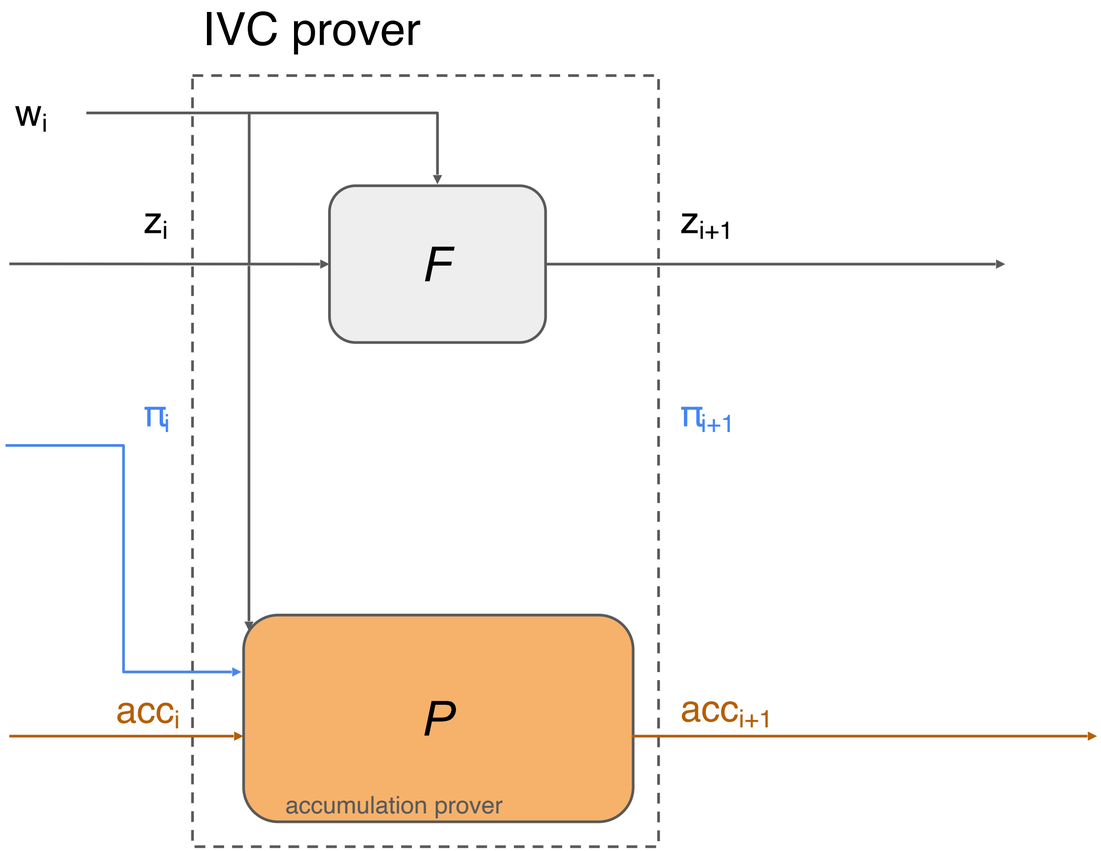
recursive techniques



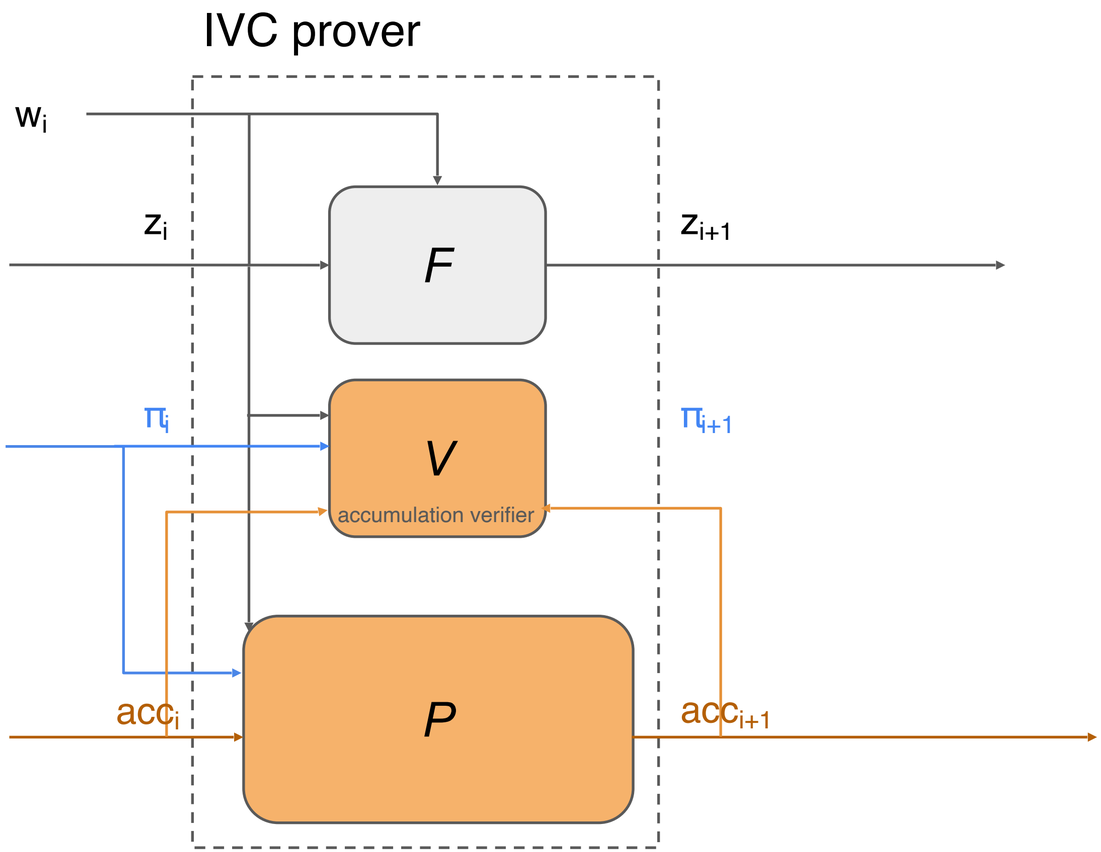
recursive techniques: IVC from atomic accumulation



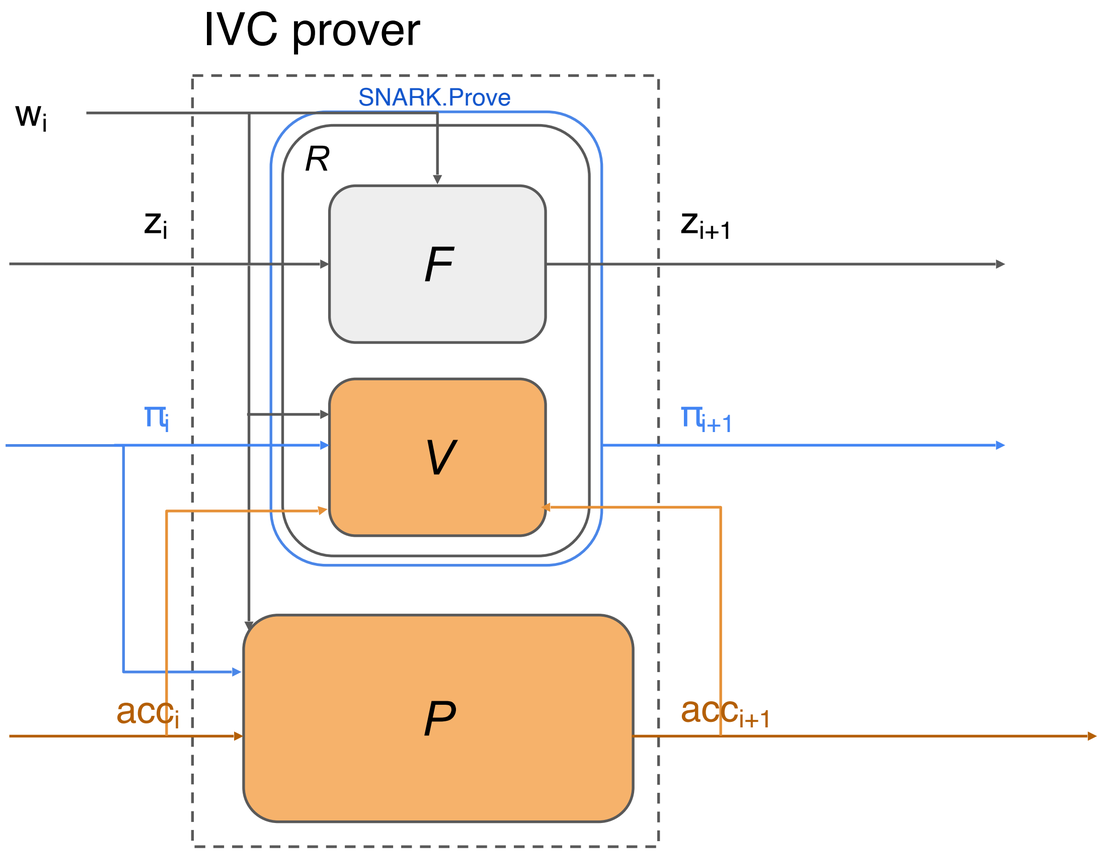
recursive techniques: IVC from atomic accumulation



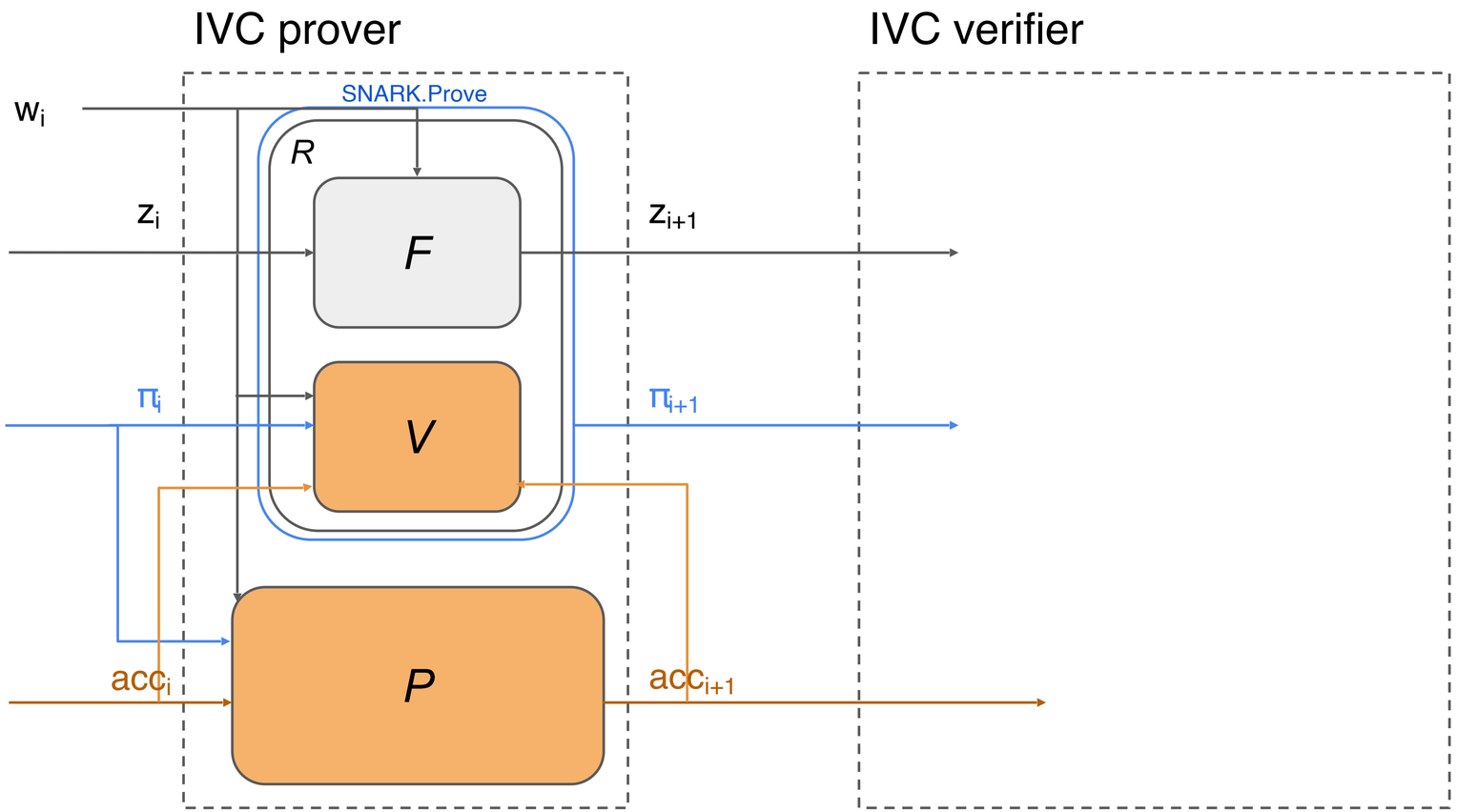
recursive techniques: IVC from atomic accumulation



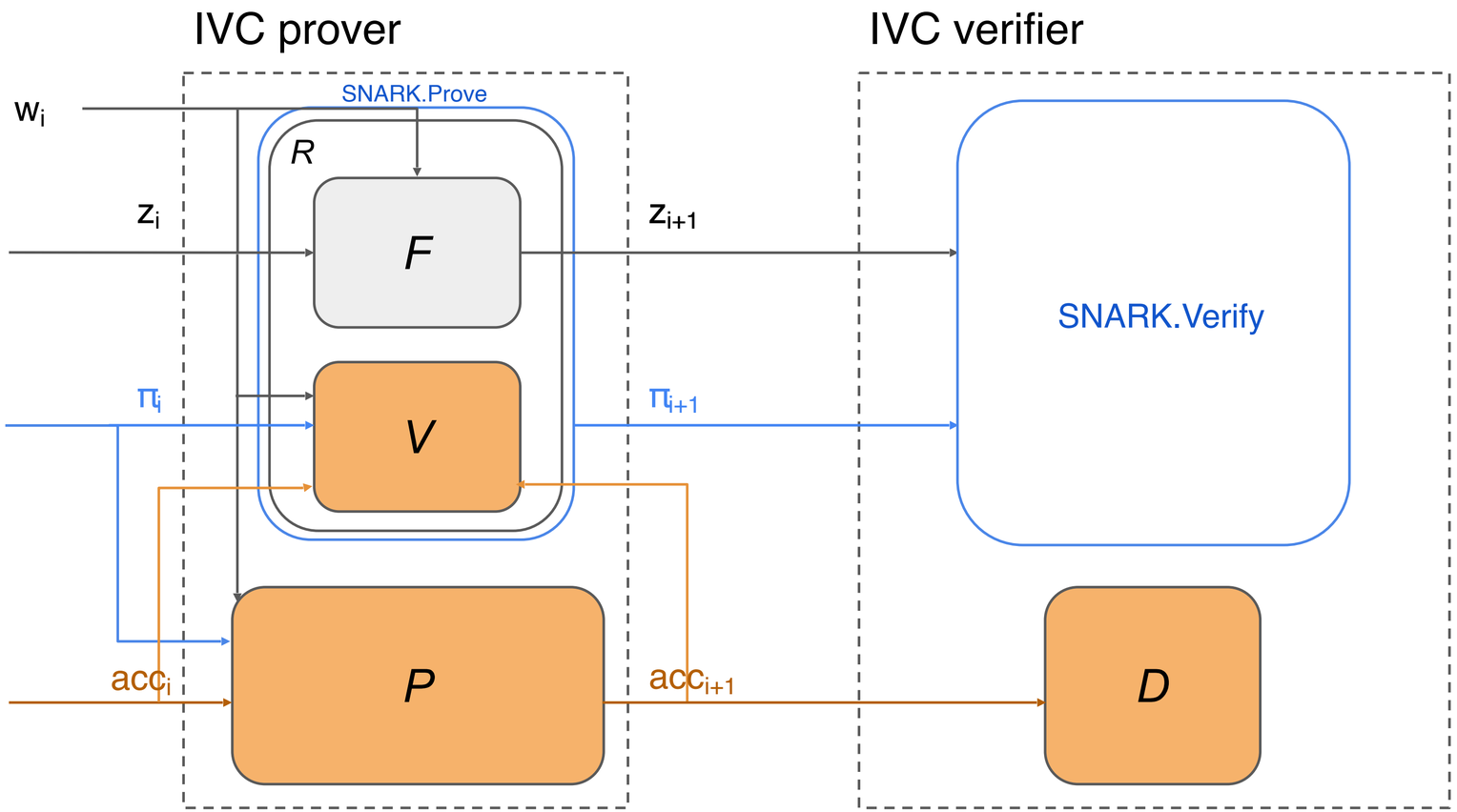
recursive techniques: IVC from atomic accumulation

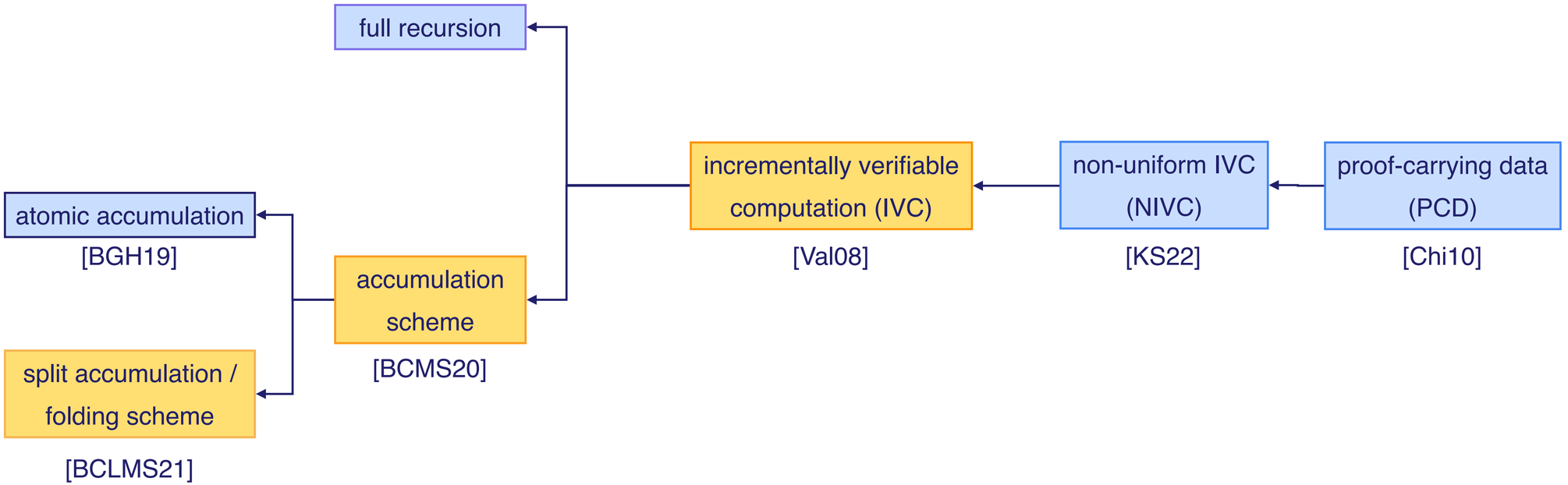


recursive techniques: IVC from atomic accumulation

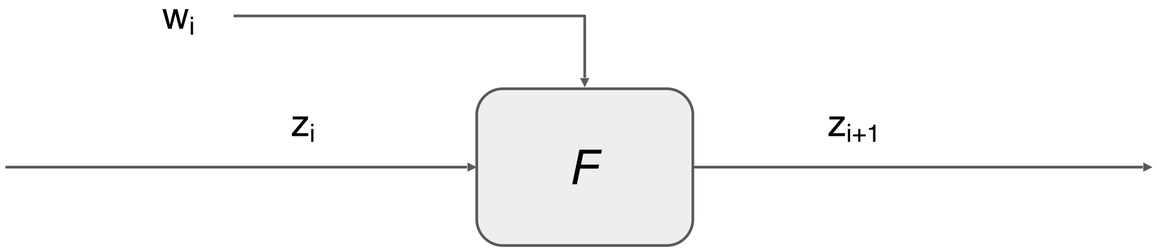


recursive techniques: IVC from atomic accumulation

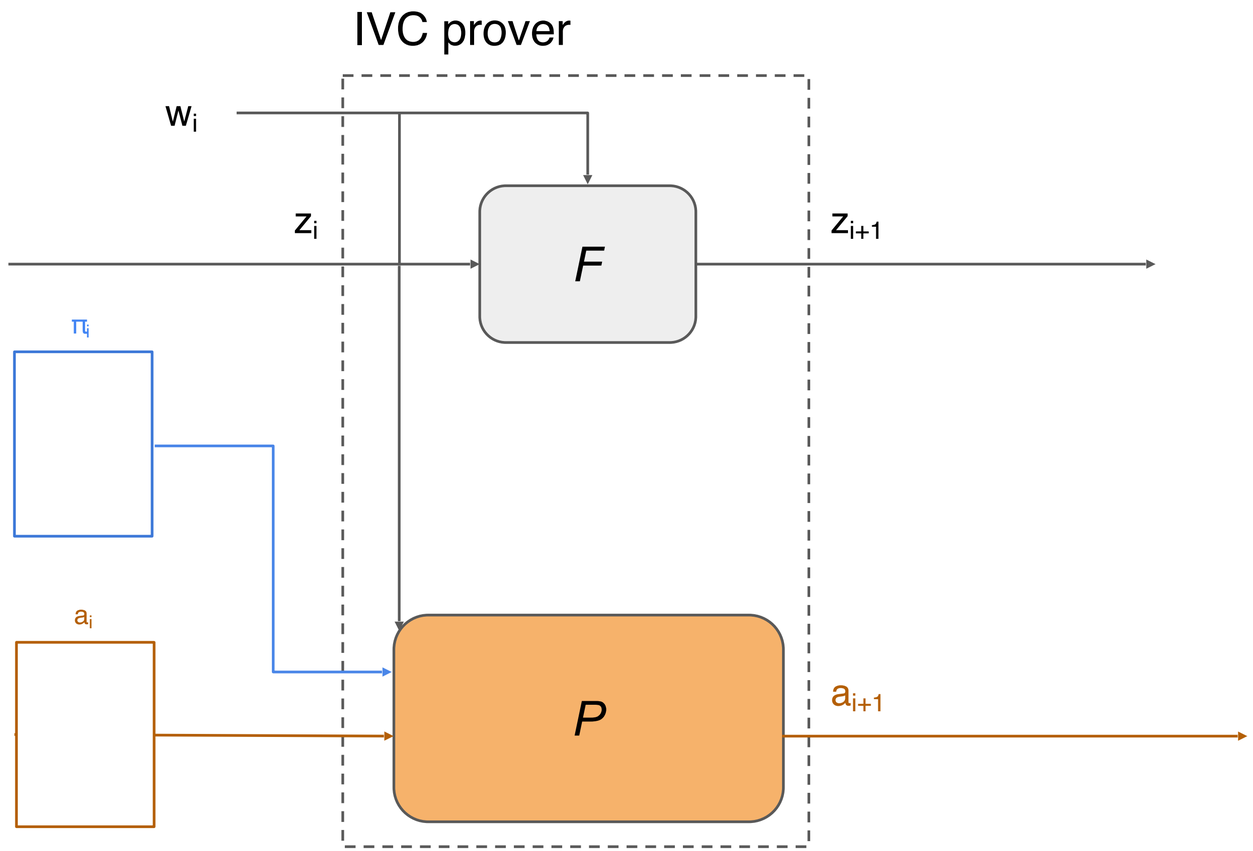



full recursion
incrementally verifiable computation (IVC)
[Val08]
non-uniform IVC (NIVC)
[KS22]
proof-carrying data (PCD)
[Chi10]
atomic accumulation
[BGH19]
accumulation scheme
[BCMS20]
split accumulation / folding scheme
[BCLMS21]

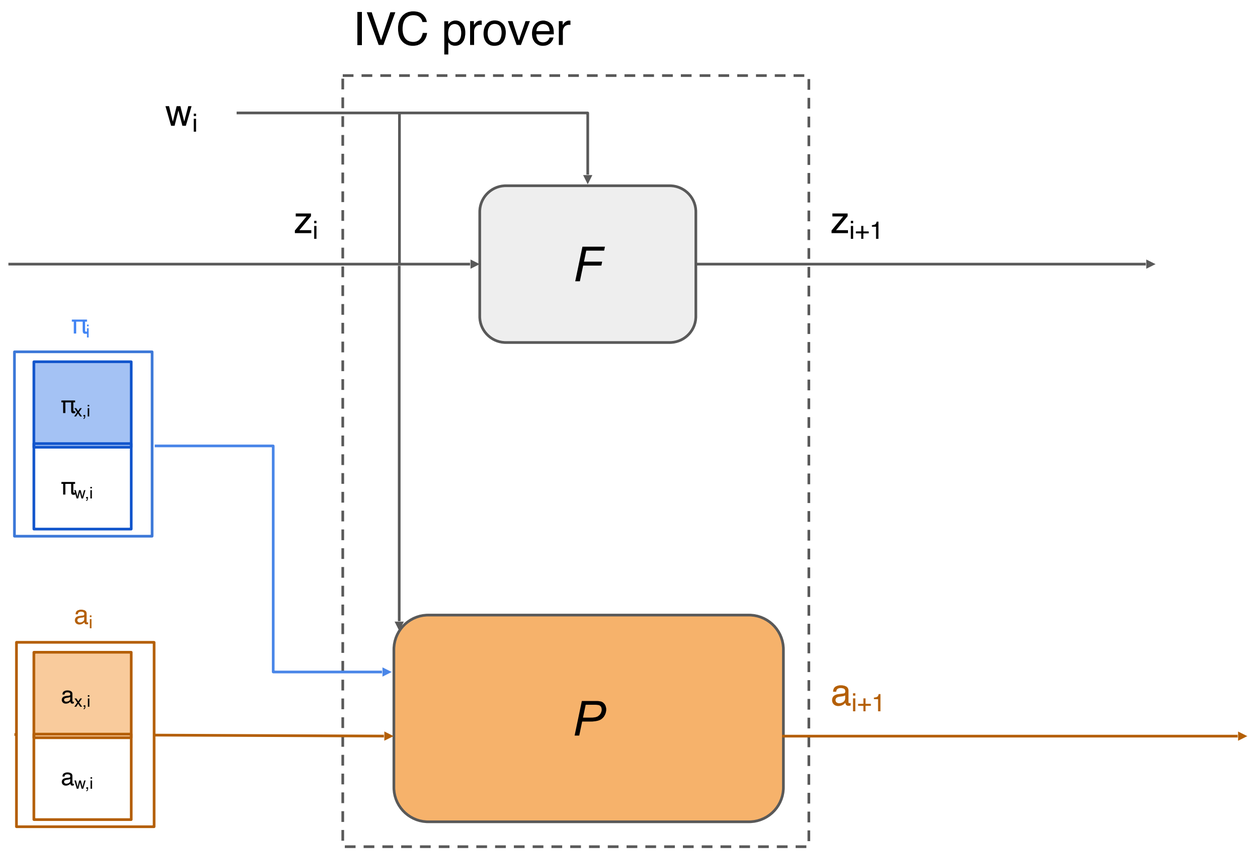
recursive techniques: IVC from split accumulation



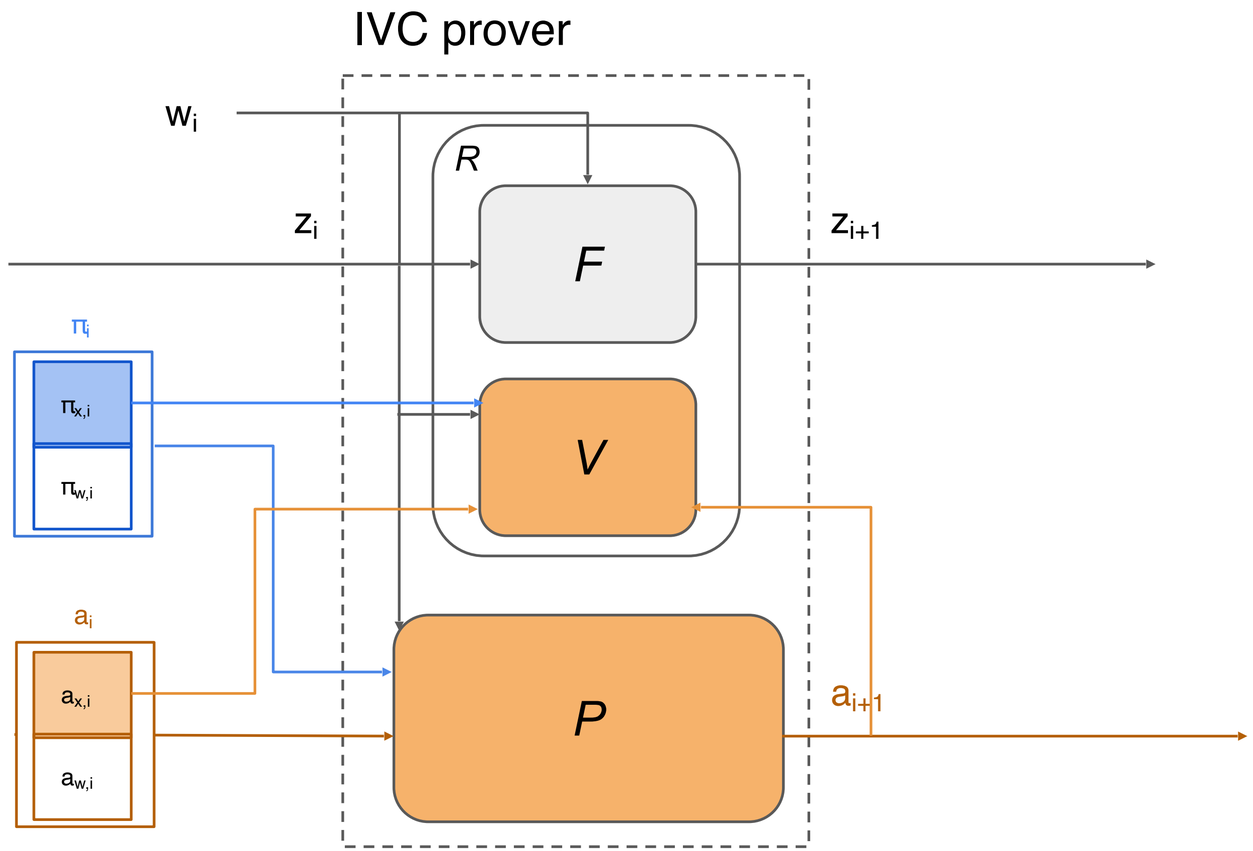
recursive techniques: IVC from split accumulation



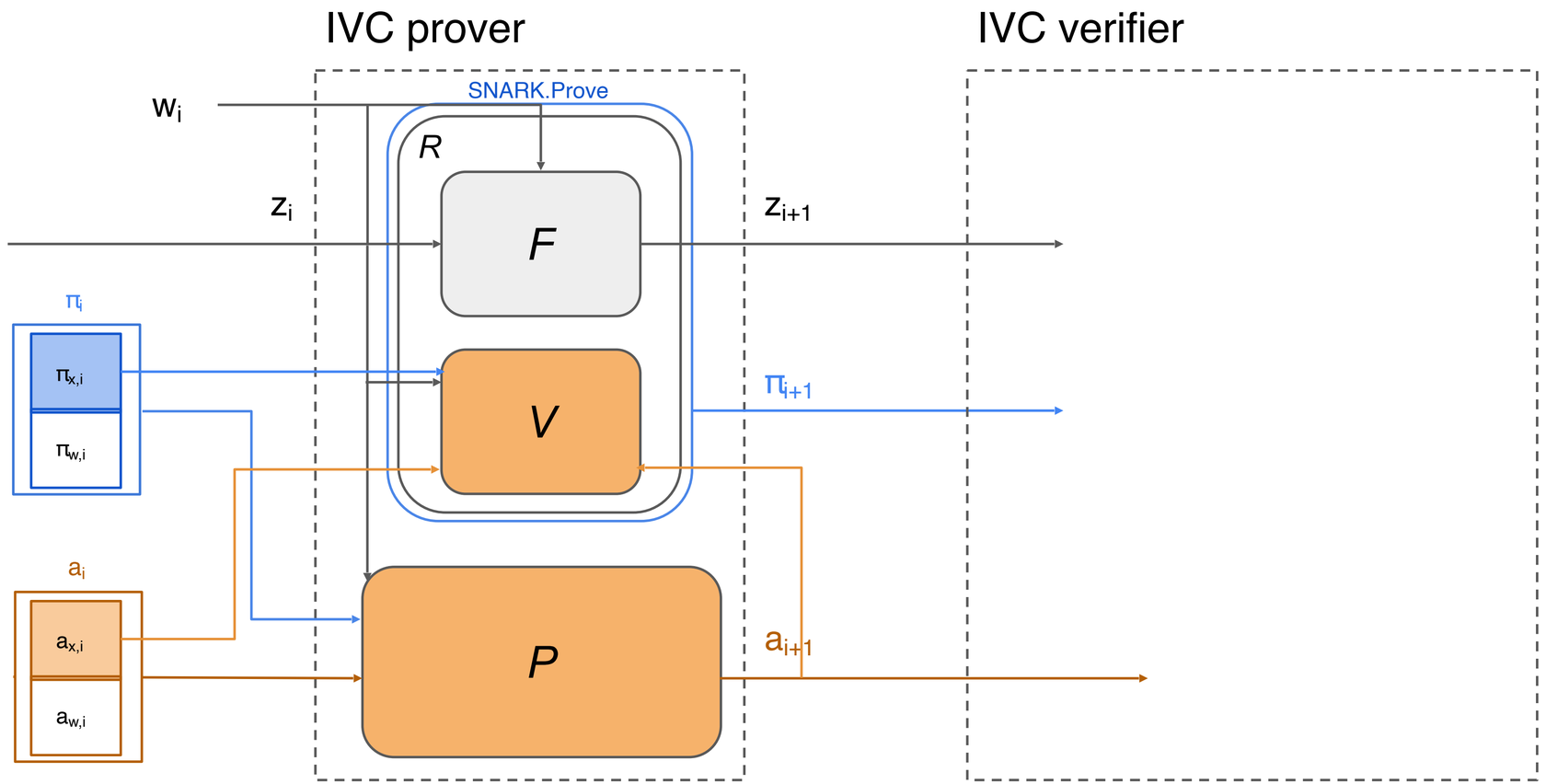
recursive techniques: IVC from split accumulation



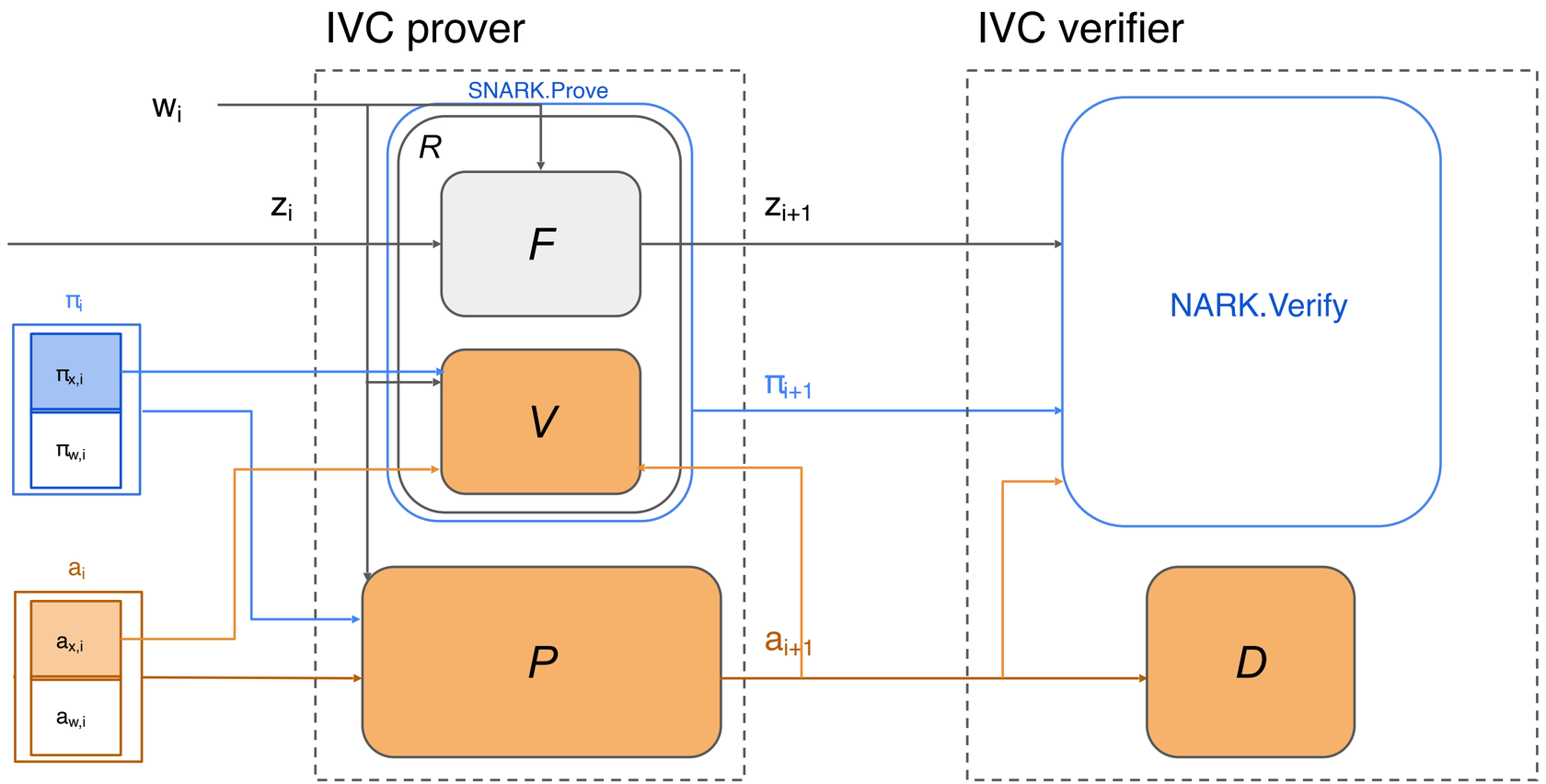
recursive techniques: IVC from split accumulation



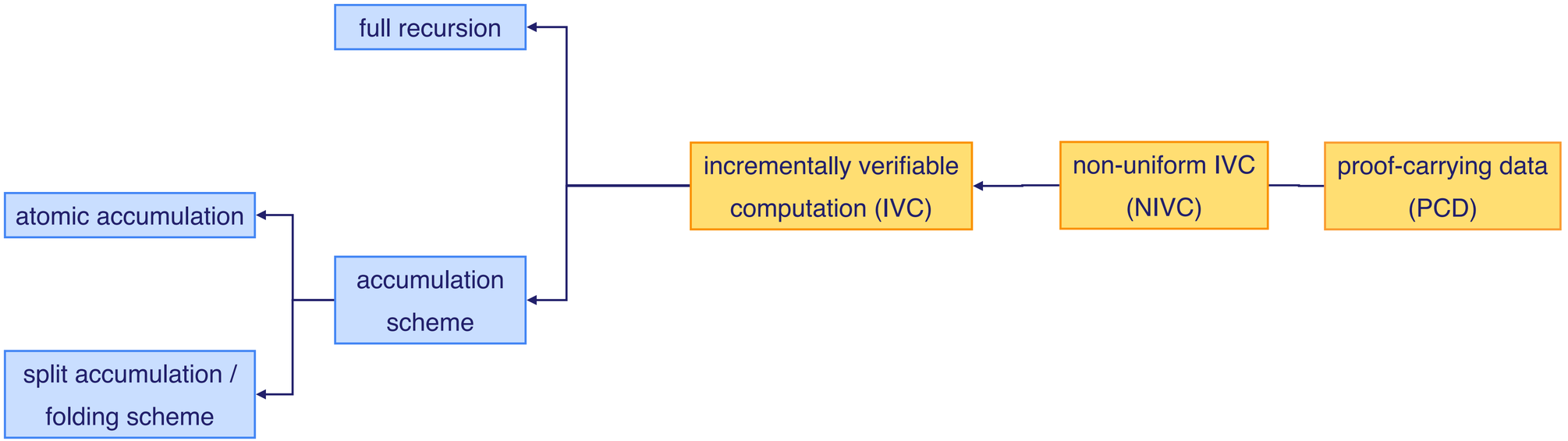
recursive techniques: IVC from split accumulation



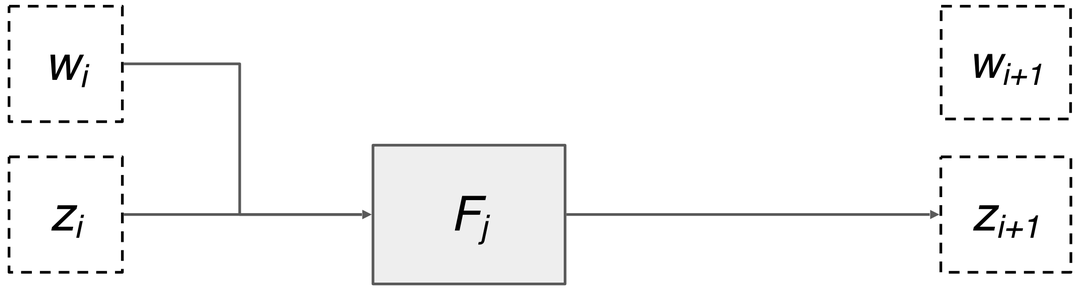
recursive techniques: IVC from split accumulation



recursive techniques



recursive techniques: non-uniform IVC (NIVC)

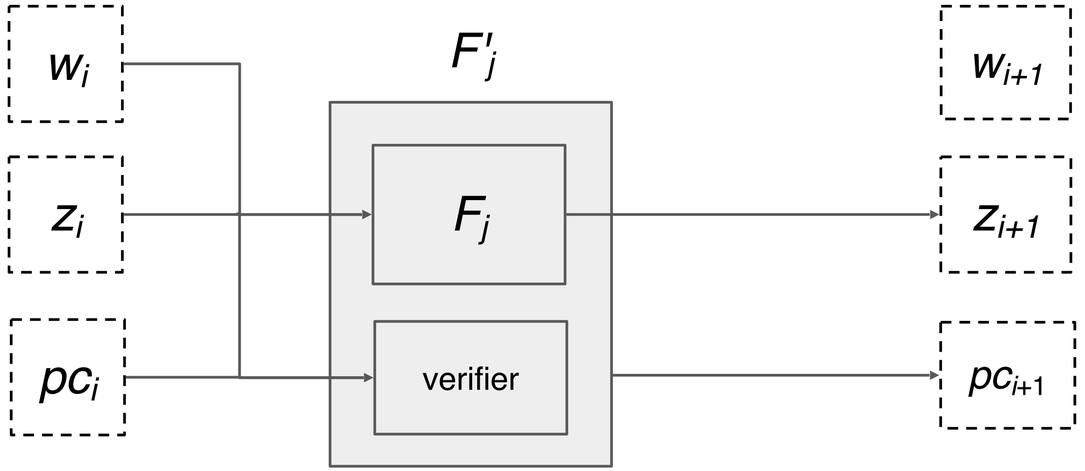


$F_j(w_i, z_i) = z_{i+1}$

recursive techniques: non-uniform IVC (NIVC)

verifier*:*

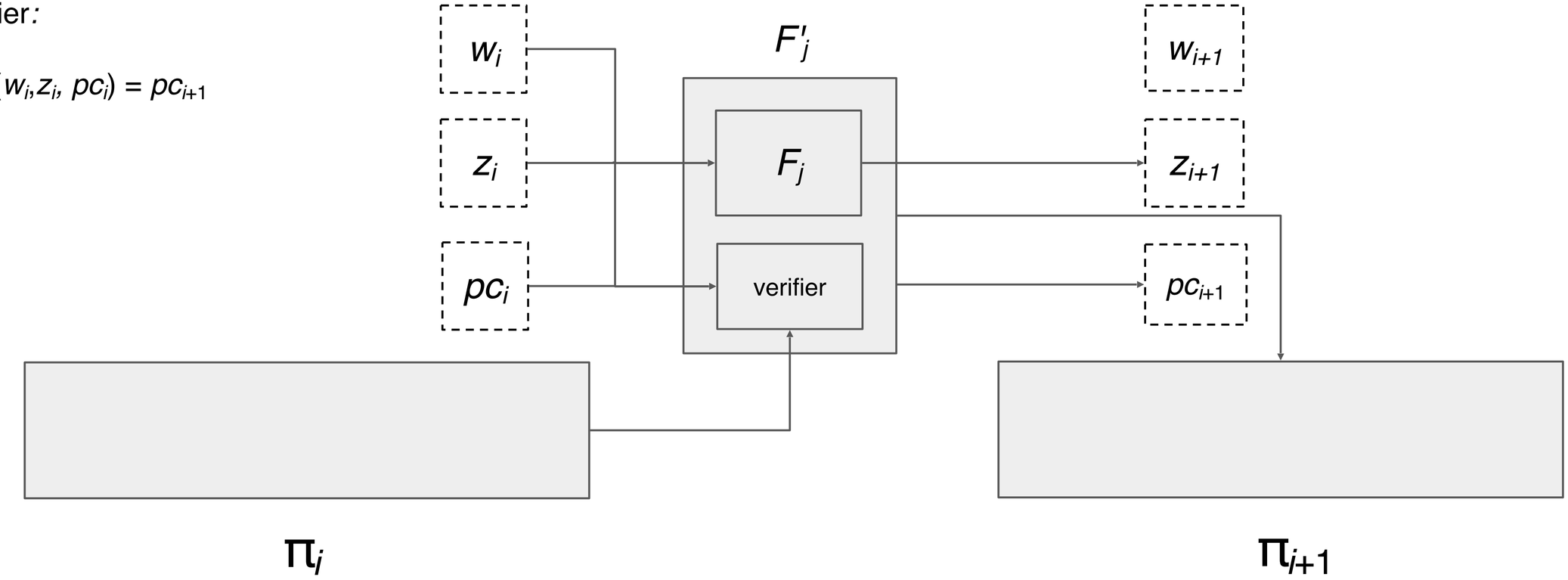
- $\phi(w_i, z_i, pc_i) = pc_{i+1}$



recursive techniques: non-uniform IVC (NIVC)

verifier*:*

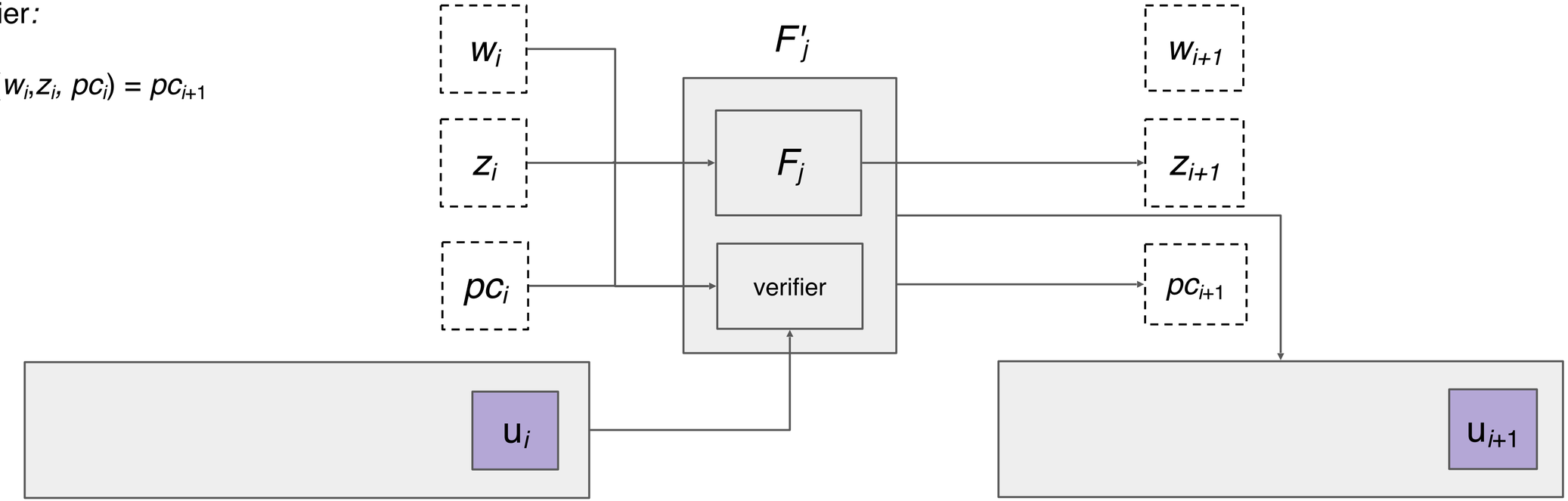
- $\phi(w_i, z_i, pc_i) = pc_{i+1}$



recursive techniques: non-uniform IVC (NIVC)

verifier*:*

- $\phi(w_i, z_i, pc_i) = pc_{i+1}$

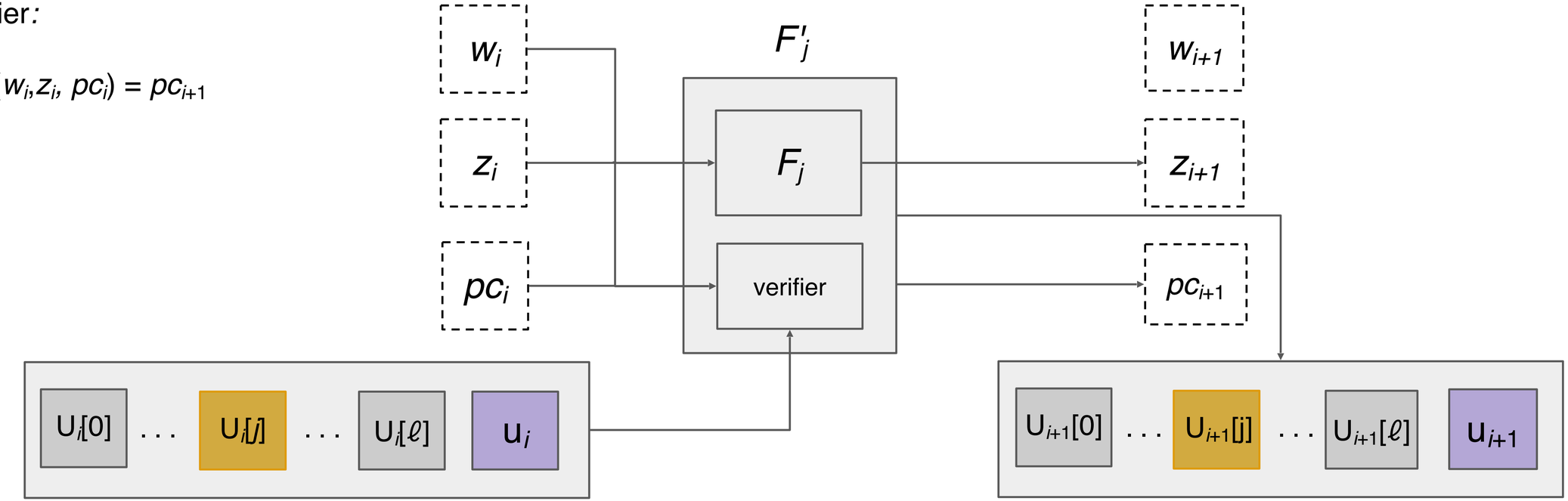


u_i claims the correctness of the i^{th} step

recursive techniques: non-uniform IVC (NIVC)

verifier*:*

- $\phi(w_i, z_i, pc_i) = pc_{i+1}$



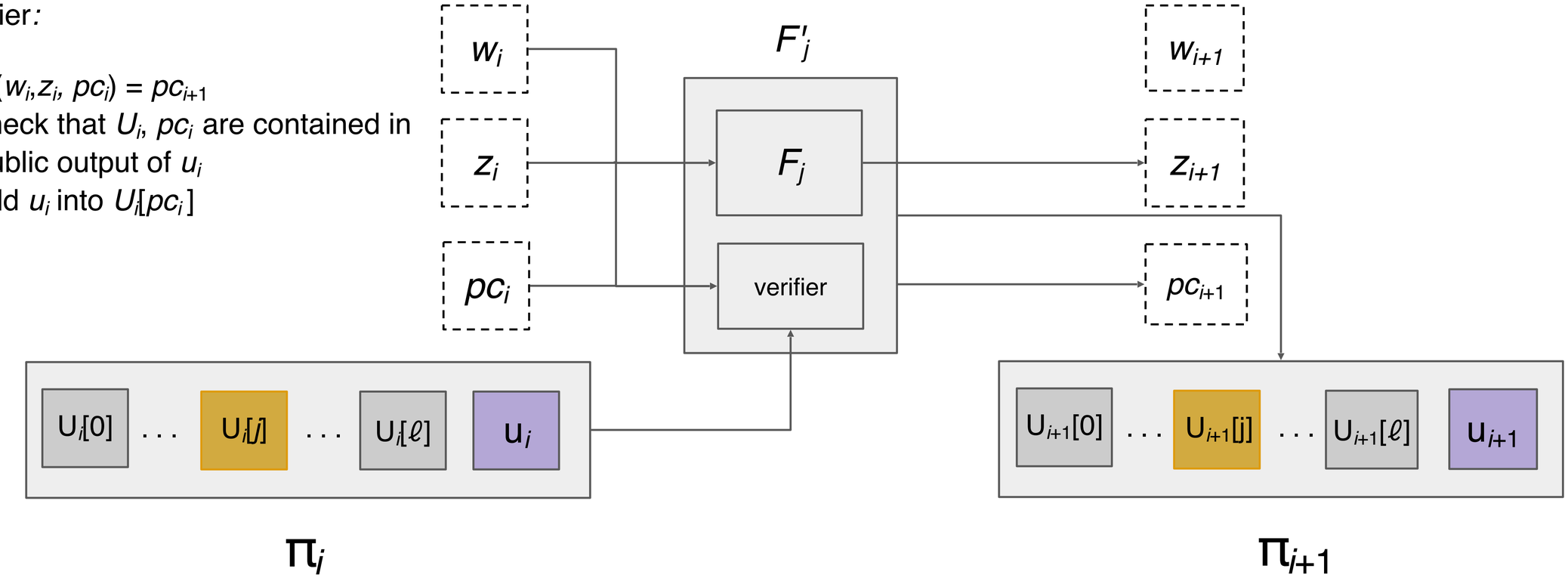
u_i claims the correctness of the i^{th} step

$U_i[j]$ attests to all prior i -1 iterations of F'_j

recursive techniques: non-uniform IVC (NIVC)

verifier*:*

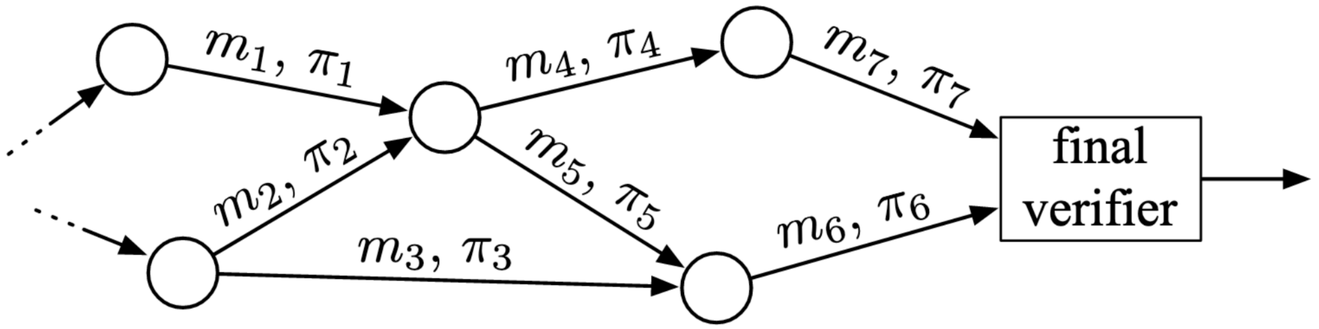
- $\phi(w_i, z_i, pc_i) = pc_{i+1}$
- check that U_i, pc_i are contained in public output of u_i
- fold u_i into $U_i[pc_i]$



u_i claims the correctness of the i^{th} step

$U_i[j]$ attests to all prior i -1 iterations of F'_j

recursive techniques: proof-carrying data (PCD)



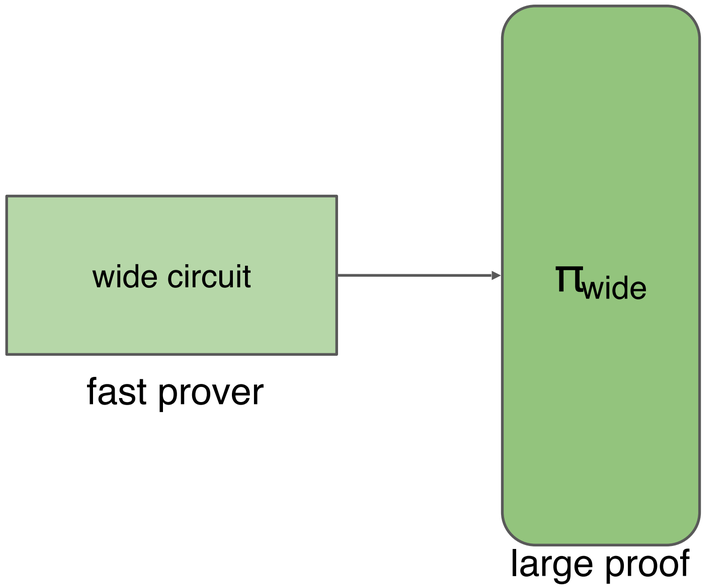
- allows for multiple parties
- allows for arbitrary communication graphs
- each party does not have to commit in advance to what computation it will have to perform

diagram from [Chi10]

contents

composition techniques: recursive verification

	fast prover	small proof / fast verifier
"wide" proof	✓	✗



composition techniques: recursive verification

	fast prover	small proof / fast verifier
"wide" proof	✓	✗
"narrow" proof	✗	✓

composition techniques: recursive verification

	fast prover	small proof / fast verifier
"wide" proof	✓	✗
"narrow" proof	✗	✓

composition techniques: recursive verification

	fast prover	small proof / fast verifier
STARK	✓	✗
Groth16	✗	✓

composition techniques: recursive verification

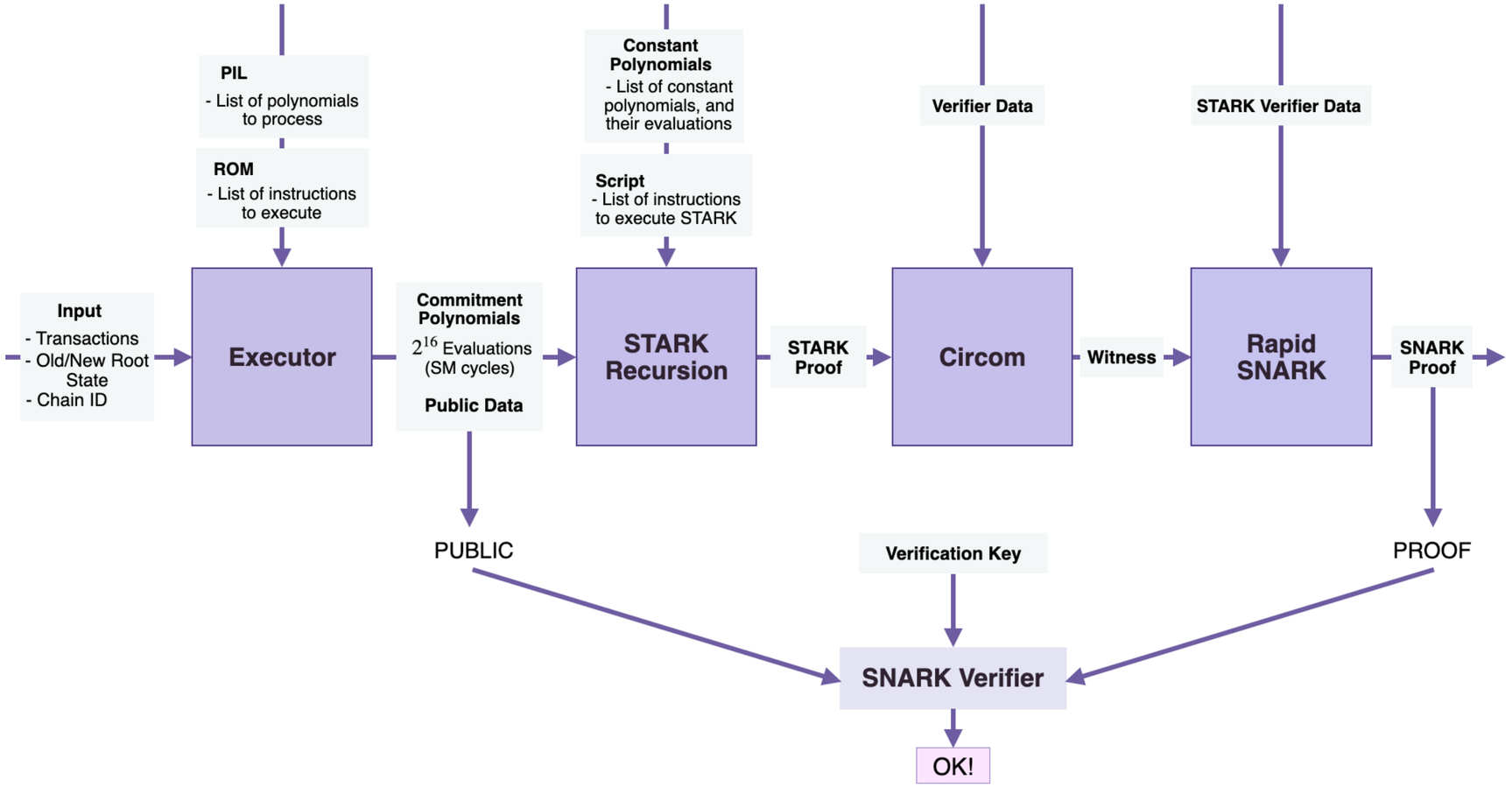
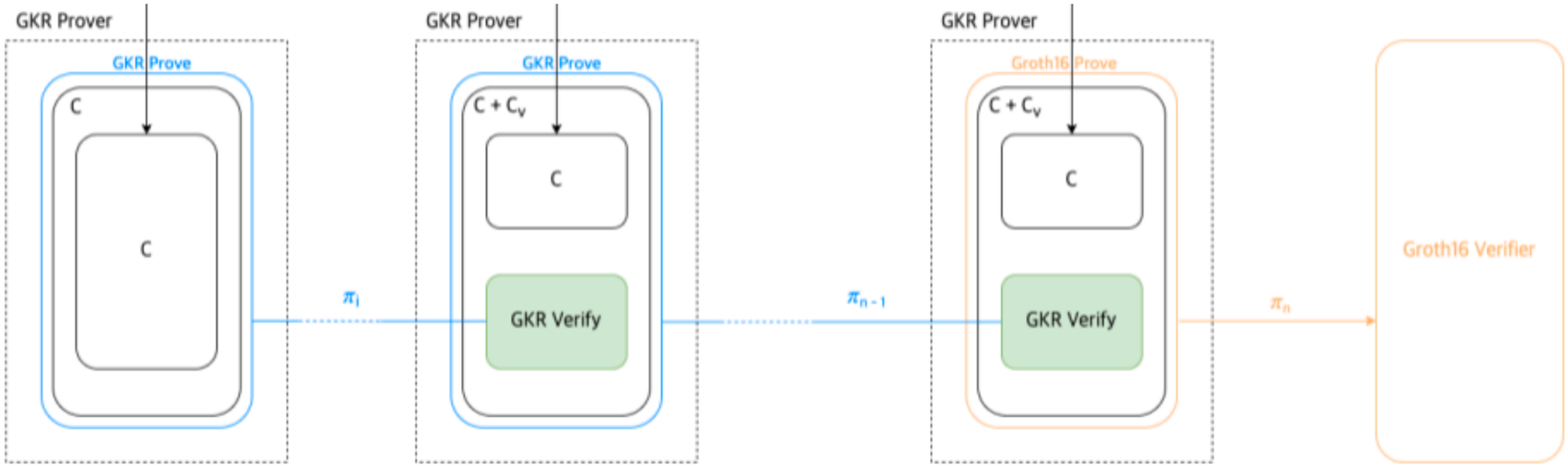


diagram from https://docs.hermez.io/zkEVM/zkProver/State-Machines/Overview/zkProver-State-Machines/#the-zk-snark-prover

composition techniques: recursive verification

example: [BSB22]

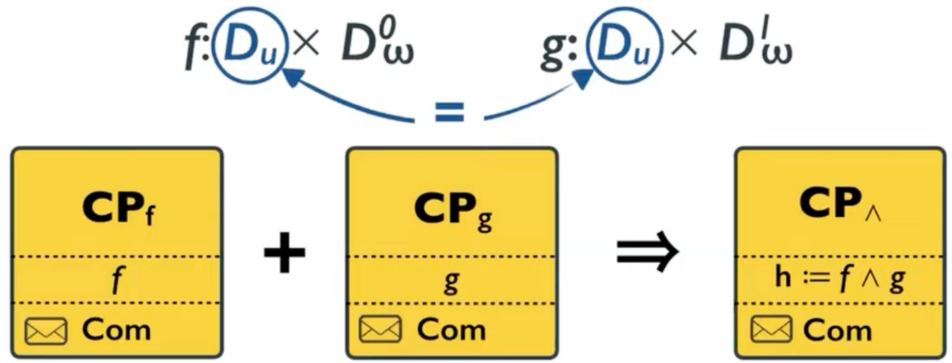


- the **GKR** protocol [GKR08]:
  - practical prover runtime is comparable to the **runtime of the alleged computation itself**
  - suitable for layered circuits with **smaller width, larger depth, and lower-degree** (at each layer): e.g. S-Box hash functions (e.g. Poseidon, MiMC)

- **recursion** using GKR [BSB22]:
  - compress input to Fiat-Shamir: hash a **commitment** of the GKR input-output, instead of the strings themselves
  - **externalise** proof of correctness for commitment

composition techniques: linkable commit-and-prove

example: **LegoSNARK [CFQ19]**



$h(\, u, (\omega_0, \omega_1)\,) := f(u, \omega_0) \wedge g(u, \omega_1)$

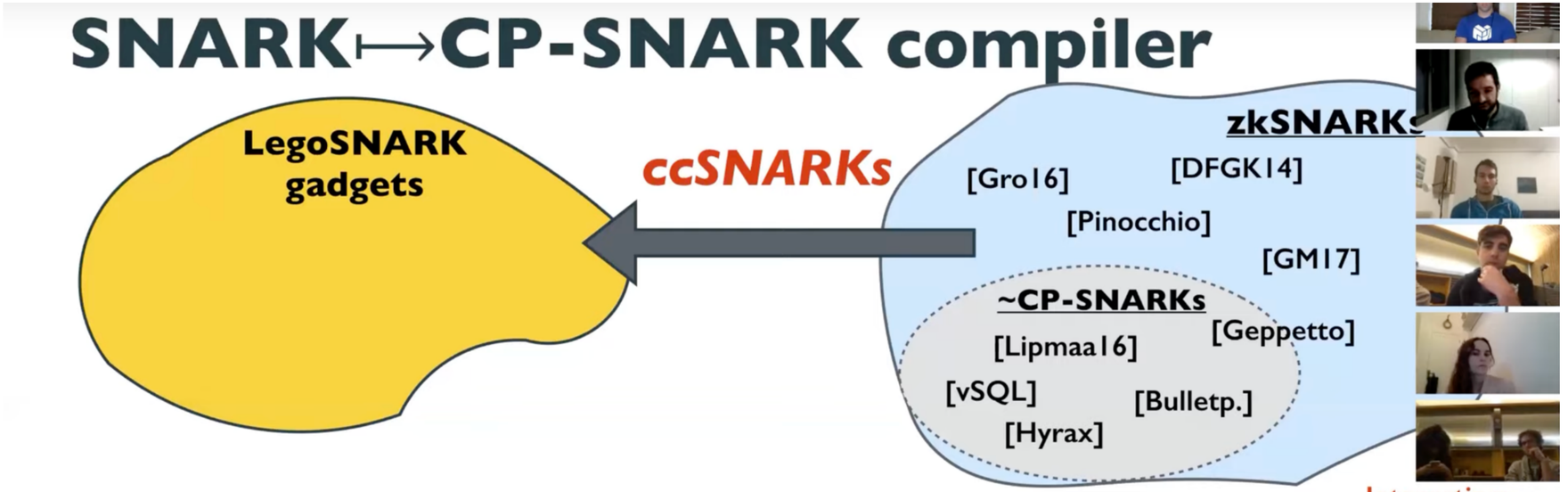
simple idea. $\pi_\wedge = (Com_{ck}(u), \pi_f, \pi_g)$, $\pi_f \leftarrow CP_f$, $\pi_g \leftarrow CP_g$

other compositions. disjunction, sequential composition, >2 relations

main message. focus on constructing proof gadgets, security is proven once for all

composition techniques: linkable commit-and-prove

example: **LegoSNARK [CFQ19]**



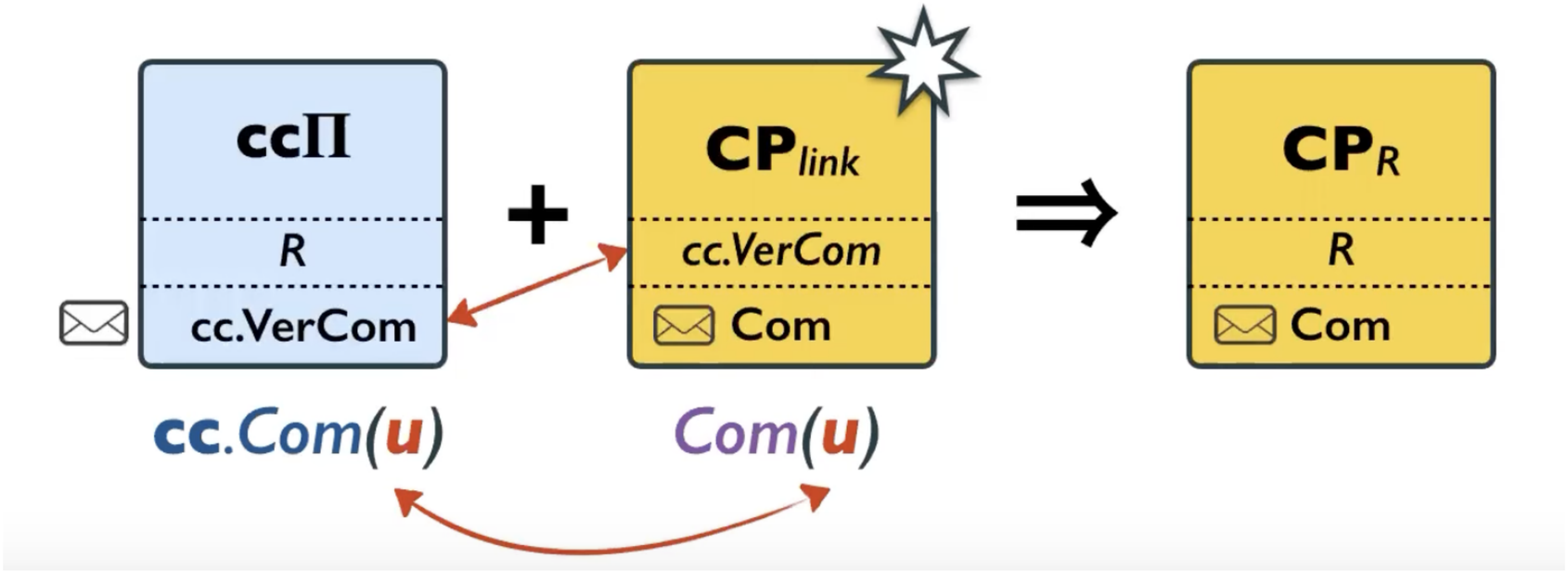
1. formalize the notion of **commit-carrying SNARKs** (ccSNARKs)

   $\mathrm{cc}\Pi.\mathrm{KeyGen}(R) \to (ck, crs)$ $\mathrm{cc}\Pi.\mathrm{Prove}(crs, x, w) \to (com_{ck}(w), \pi)$

2. for many existing schemes we prove they are ccSNARKs
3. **cc-SNARK-lifting compiler**

composition techniques: linkable commit-and-prove

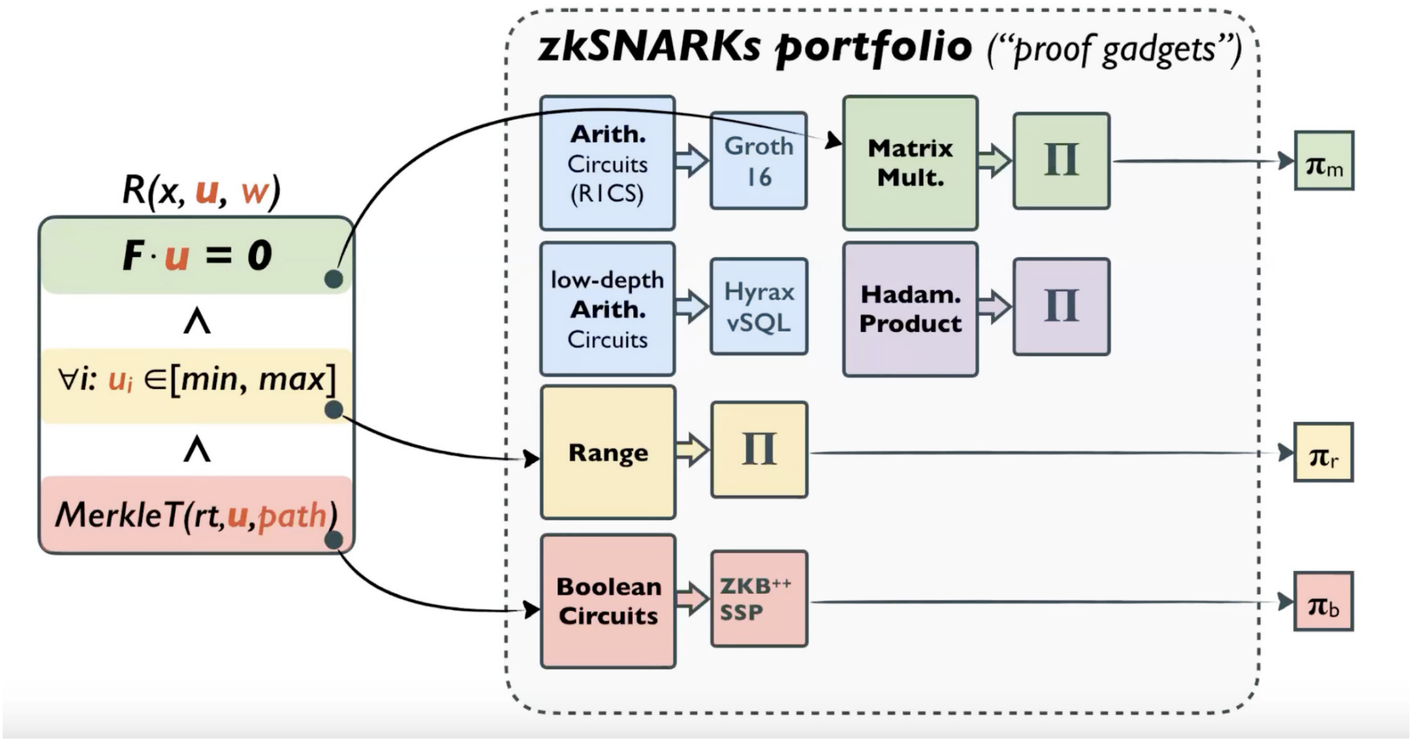
example: **LegoSNARK [CFQ19]**



CP$_{link}$ proves that **cc**.*Com*(***u***) and *Com*(***u***) open to the same data

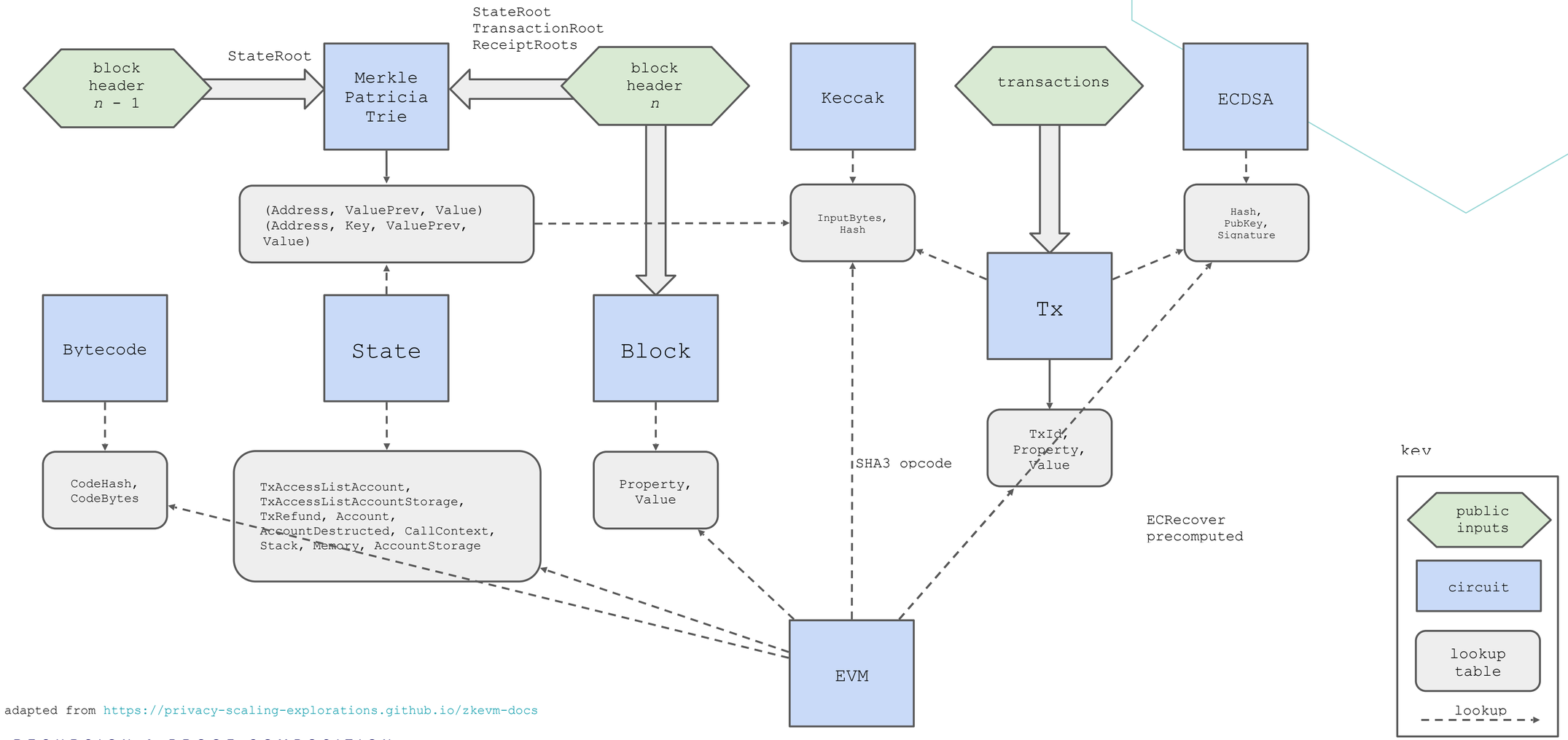
composition techniques: linkable commit-and-prove

example: **LegoSNARK [CFQ19]**



composition techniques: linkable commit-and-prove

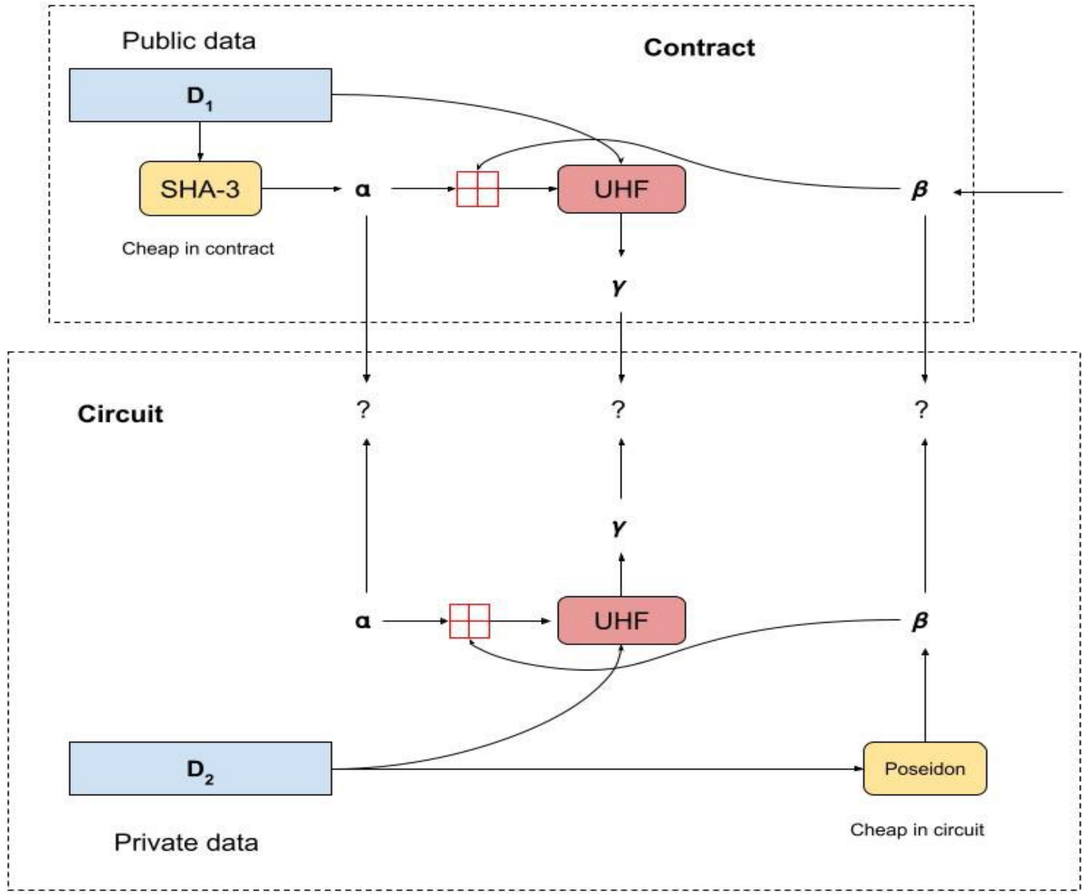
example: **cross-circuit lookups**



adapted from https://privacy-scaling-explorations.github.io/zkevm-docs

composition techniques: linkable commit-and-prove

example: efficient compression and verification of public input in circuits (Dmitry Khovratovich, 2022)



$$UHF(\mathbf{D}, a) = \sum_i a^i \cdot D_i \bmod p$$

Checks pass if

$$\bar{\delta} \circ \mathbf{D}_1 = \bar{\delta} \circ \mathbf{D}_2;$$

where

$$\bar{\delta} = (1, \delta, \delta^2, \ldots)$$

$$\delta = H_{SHA}(\mathbf{D}_1) + H_{Poseidon}(\mathbf{D}_2).$$

composition techniques: linkable commit-and-prove

example: proof of equivalence between multiple polynomial commitment schemes to the same data

vbuterin Nov '20

Suppose you have multiple polynomial commitments $C_1 \ldots C_k$, under k different commitment schemes (eg. Kate, FRI, something bulletproof-based, DARK...), and you want to prove that they all commit to the same polynomial P . We can prove this easily:

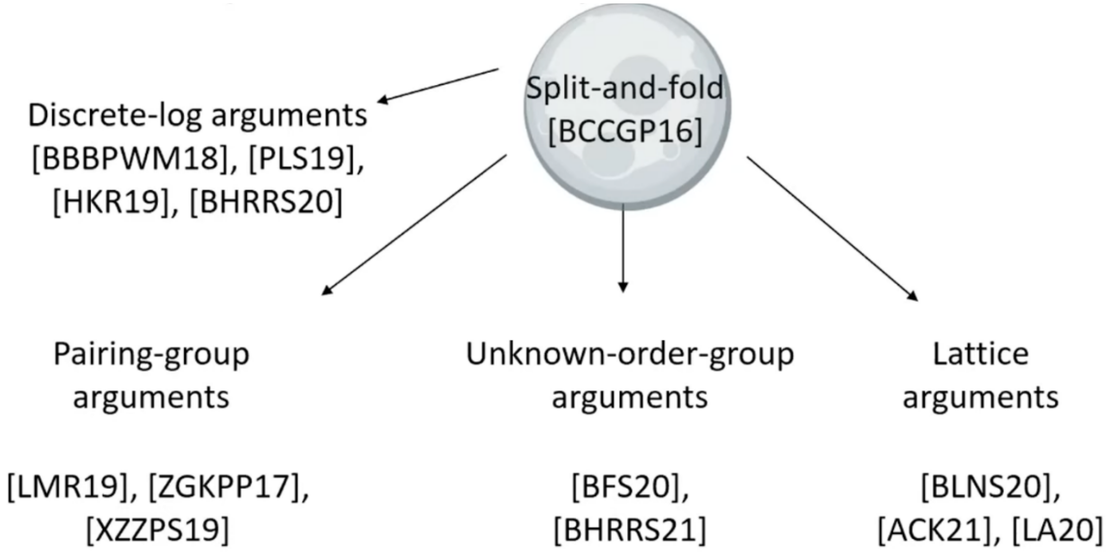
Let $z = hash(C_1 \ldots C_k)$, where we interpret z as an evaluation point at which P can be evaluated.

Publish openings $O_1 \ldots O_k$, where O_i is a proof that $C_i(z) = a$ under the i'th commitment scheme. Verify that a is the same number in all cases.

contents

future directions: unifying frameworks

example: **split-and-fold techniques**



Useful properties:

- Linear-time prover
- Streaming prover [BHRRS20], [BHRRS21] (can be implemented in small space)

some unifying abstractions:

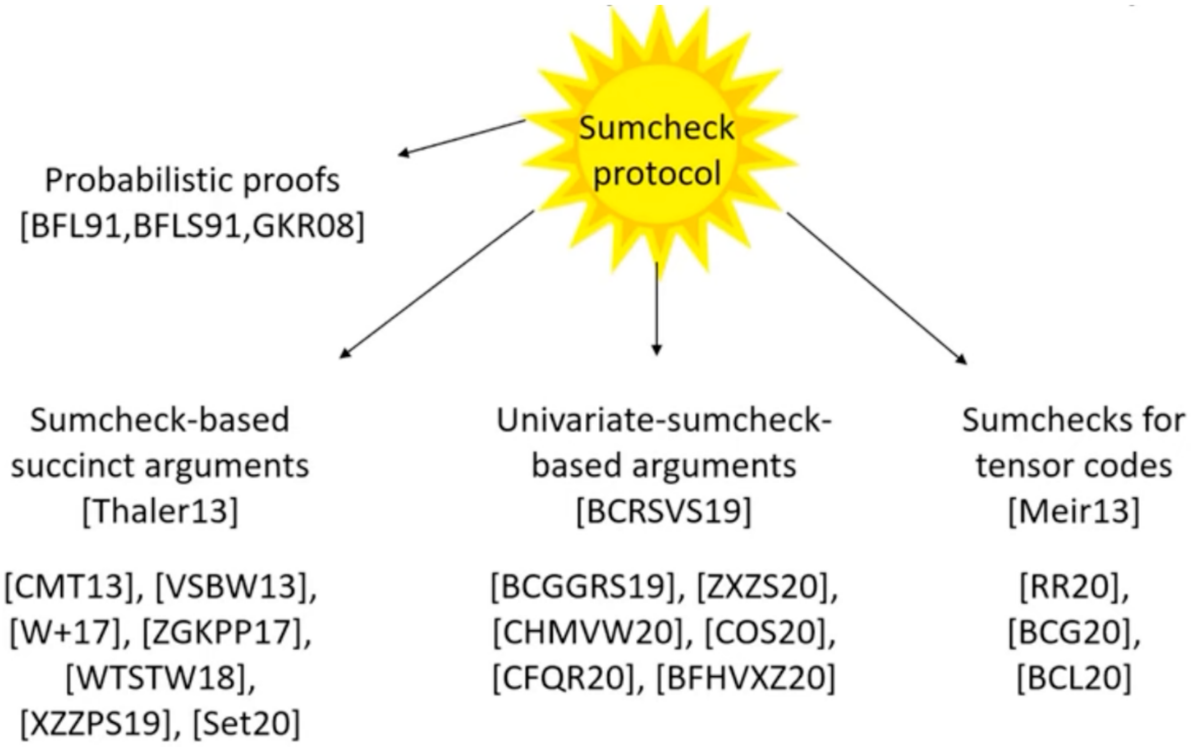
[BMMTV19] *Proofs for Inner Pairing Products and Applications*
[ACF20] *Compressing Proofs of k-Out-Of-n-Partial Knowledge*
[BDFG20] *Halo Infinite: Proof-Carrying Data from Additive Polynomial Commitments*
[KP22] *Algebraic Reductions of Knowledge*

future directions: unifying frameworks

example: **sumcheck arguments**

future directions: unifying frameworks

example: **sumcheck arguments**



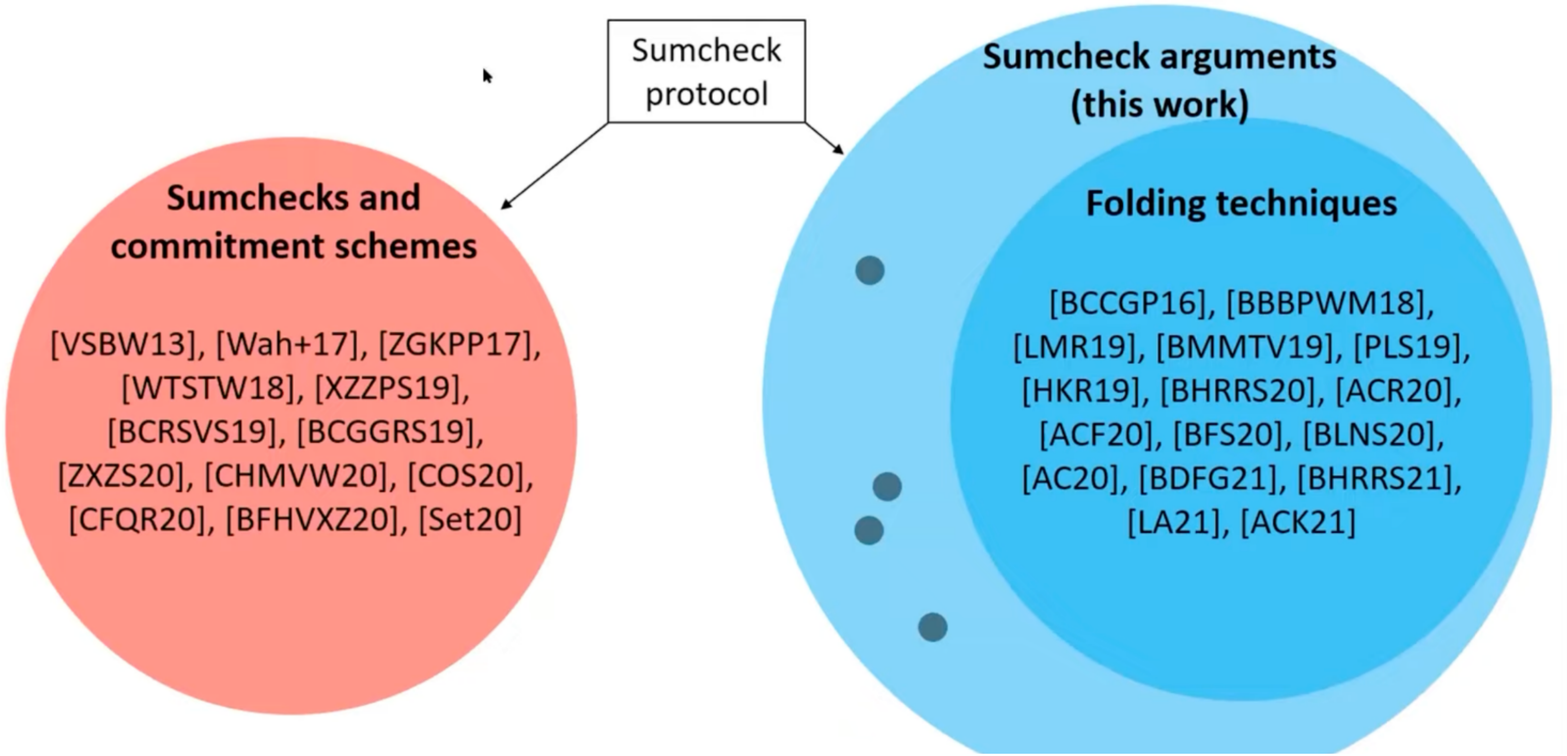
Useful properties:

- Linear-time prover [Thaler13,ZXZS20]
- Small space [CMT13] (can be implemented with streaming access)
- Strong soundness properties [CCHLRR18] (can make non-interactive without random oracles)

ZKStudyClub - Sumcheck Arguments and Their Applications w/ Jonathan Bootle and Katerina Sotiraki

future directions: unifying frameworks

example: **sumcheck arguments**



future directions: project ideas

- **benchmarks**
  - polynomial commitment benchmarks (2π.com/23/pc-bench/)
  - ZKP compiler shootout (github.com/anoma/zkp-compiler-shootout)
  - what is an ideal "**inner**" proof? and an ideal "**outer**" proof
- **libraries, standards, dev tooling**
  - LegoSNARK: github.com/imdea-software/legosnark
  - arkworks: arkworks-rs/accumulation, arkworks-rs/pcd
  - can we define **high-level APIs** for recursion/composition strategies?
- **security analysis**
  - minimal example that **breaks soundness in recursive plonky2**?
  - security in recursive composition: **stronger knowledge extractor** needed (must be successful in every recursive step)
  - make use of **unifying frameworks** to simplify analysis