recursive proof composition

ABCDE ZK Hacker Camp
2 Sep 2023

agenda

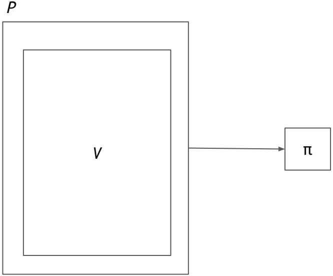
1. **overview**
    a) motivation
    b) constructions
2. **comparison**
    a) recursion threshold
    b) zero-knowledgeness
    c) security and cryptographic assumptions
3. **focus: CycleFold**

agenda

1. **overview**
    a) motivation
    b) constructions

2. **comparison**
    a) recursion threshold
    b) zero-knowledgeness
    c) security and cryptographic assumptions

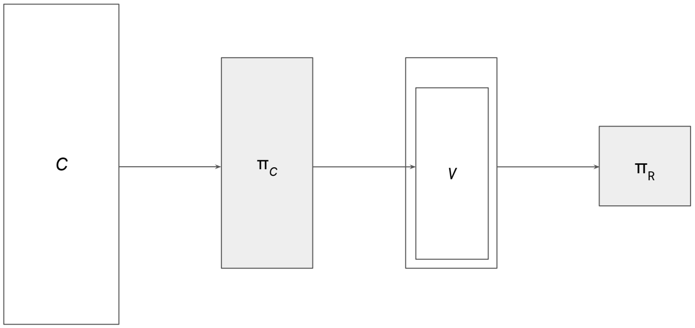
3. **focus: CycleFold**



*a **recursive proof** is a proof that enforces the accepting computation of the **proof system's own verifier***

overview: *motivation*

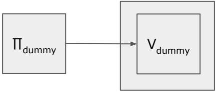
shrinking proof size



overview: *motivation*

shrinking proof size

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



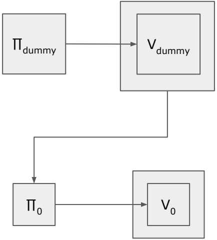
https://github.com/mir-protocol/plonky2/blob/main/plonky2/examples/bench_recursion.rs#L183

overview: *motivation*

shrinking proof size

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

```
// 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

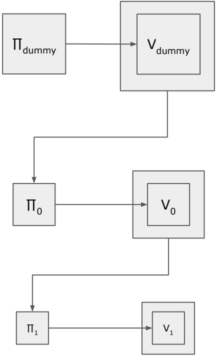
overview: *motivation*

shrinking proof size

```
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
```

```rust
// 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

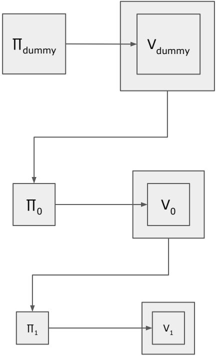
overview: *motivation*

shrinking proof size

```
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

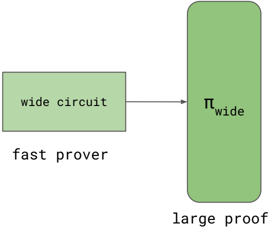
overview: *motivation*

shrinking proof size

overview: *motivation*

shrinking proof size

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



overview: *motivation*

shrinking proof size

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

wide circuit

π_{wide}

fast prover

large proof

narrow circuit

π_{narrow}

tiny proof

slow prover

overview: *motivation*

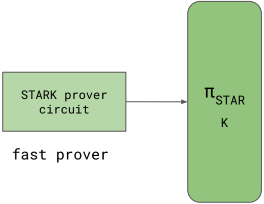
shrinking proof size

	fast prover	small proof / fast verifier
"wide" proof	✔	❌
"narrow" proof	❌	✔

overview: *motivation*

shrinking proof size

	fast prover	small proof / fast verifier
STARK	✓	❌



STARK prover circuit

π_{STARK}

fast prover

large proof

overview: *motivation*

shrinking proof size

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

STARK prover circuit

fast prover

π_{STARK}

large proof

Groth16 prover circuit

slow prover

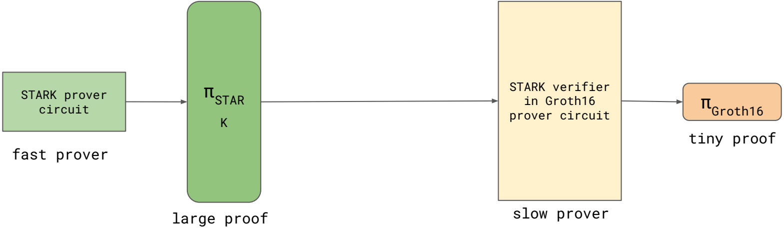
$\pi_{Groth16}$

tiny proof

overview: *motivation*

shrinking proof size

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



STARK prover circuit

fast prover

π_{STARK}

large proof

STARK verifier in Groth16 prover circuit

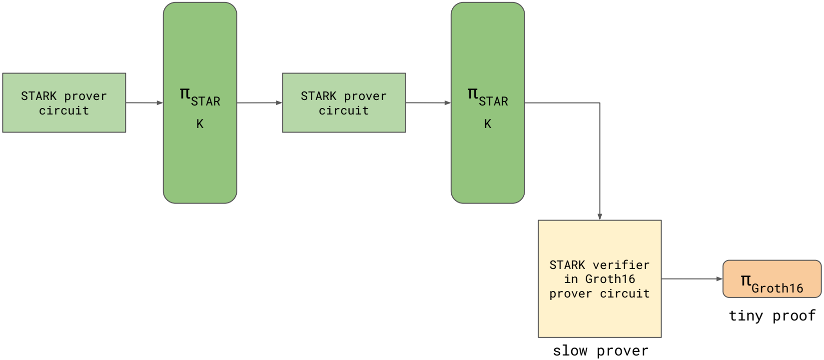
slow prover

π_{Groth16}

tiny proof

overview: *motivation*

shrinking proof size



STARK prover circuit

π_{STARK}

STARK prover circuit

π_{STARK}

STARK verifier in Groth16 prover circuit

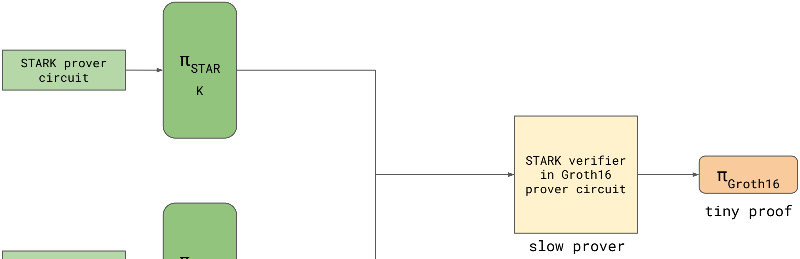
slow prover

π_{Groth16}

tiny proof

overview: *motivation*

shrinking proof size



STARK prover circuit

π_{STARK}

STARK prover circuit

π_{STARK}

STARK verifier in Groth16 prover circuit

slow prover

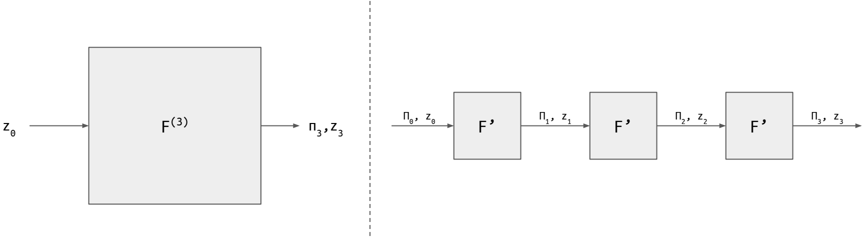
π_{Groth16}

tiny proof

overview: *motivation*

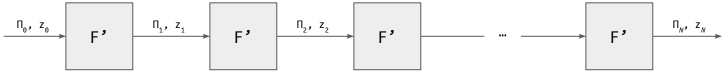
incrementally verifiable computation

break large circuit into N *repetitions of smaller circuit: reduces prover space complexity*



overview: *motivation*

incrementally verifiable computation



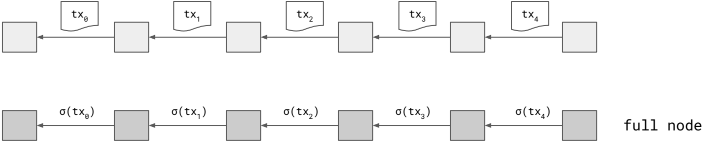
applications:

- verify chain of *N* blocks with a single proof (e.g. Mina Protocol)
- verify *N* steps of program in virtual machine (e.g. RISC Zero R0)
- verify inference of an *N*-layer neural network (e.g. Zator 🐊)

overview: *motivation*

e.g. succinct blockchain

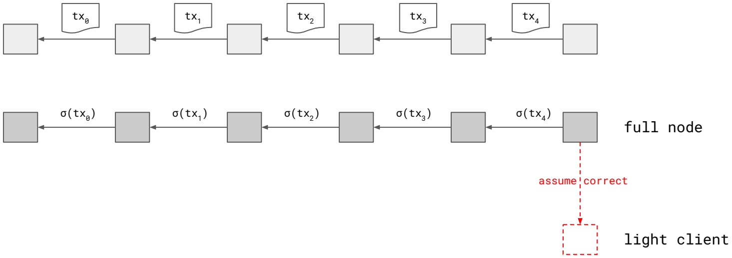
a blockchain in which each block can be verified in **constant time** regardless of the number of prior blocks in the history



overview: *motivation*

e.g. succinct blockchain

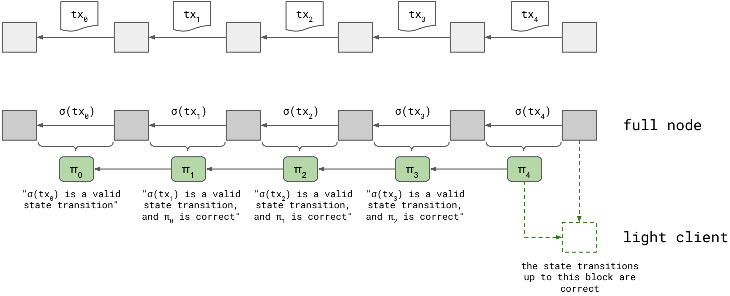
a blockchain in which each block can be verified in **constant time** regardless of the number of prior blocks in the history



overview: *motivation*

e.g. succinct blockchain

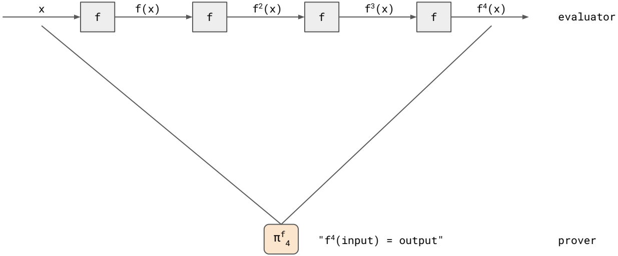
a blockchain in which each block can be verified in **constant time** regardless of the number of prior blocks in the history



overview: *motivation*

e.g. parallelising the VDF prover

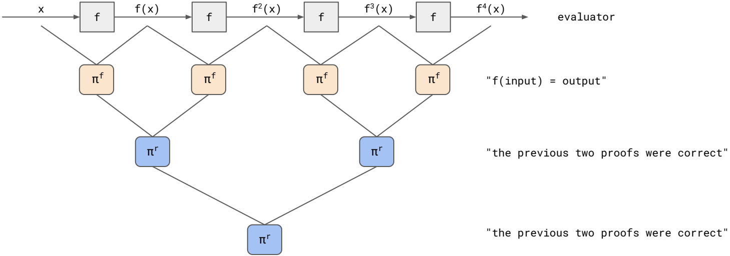
verifiable delay function [BBBF18]: a sequential computation that is slow to compute but efficient to verify



overview: *motivation*

e.g. parallelising the VDF prover

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



overview: *motivation*

proof-carrying data

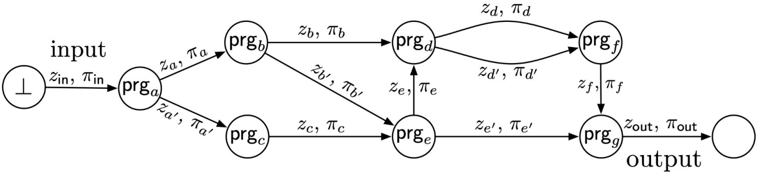
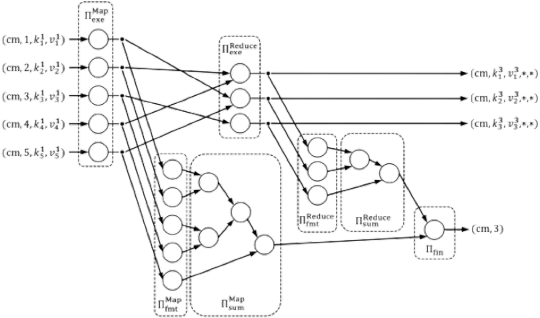


Figure 5: Example of an *augmented* distributed computation transcript. Programs are denoted by prg's, data by z's, and proof strings by π's. The corresponding (non-augmented) distributed computation transcript is with the proof strings omitted.

image from https://people.eecs.berkeley.edu/~alexch/docs/CT10.pdf

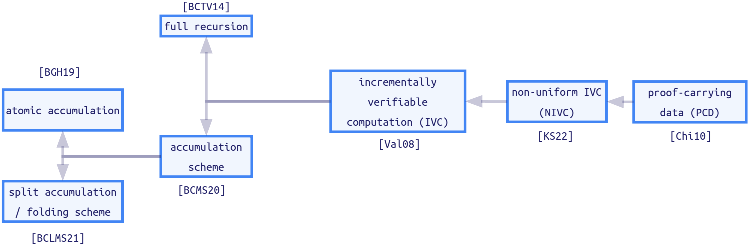
overview: *motivation*

proof-carrying data



e.g. MapReduce [CTV15]

overview: *constructions*



[BCTV14]

full recursion

[BGH19]

atomic accumulation

accumulation scheme

[BCMS20]

split accumulation / folding scheme

[BCLMS21]

incrementally verifiable computation (IVC)

[Val08]

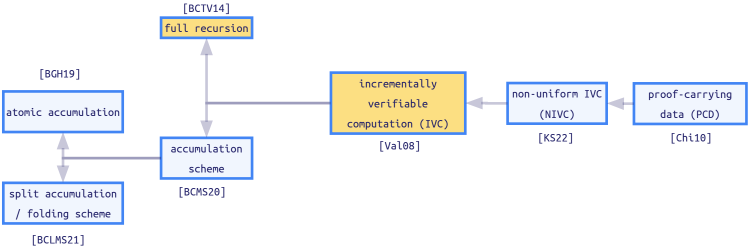
non-uniform IVC (NIVC)

[KS22]

proof-carrying data (PCD)

[Chi10]

overview: *constructions*



[BCTV14]
full recursion

[BGH19]
atomic accumulation

accumulation scheme
[BCMS20]

split accumulation / folding scheme
[BCLMS21]

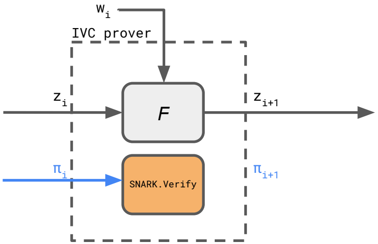
incrementally verifiable computation (IVC)
[Val08]

non-uniform IVC (NIVC)
[KS22]

proof-carrying data (PCD)
[Chi10]

overview: *constructions*

full recursion

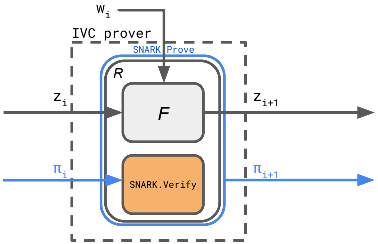


$z_{i+1} = F(z_i; w_i)$

$V(\pi_i) = 1$

overview: *constructions*

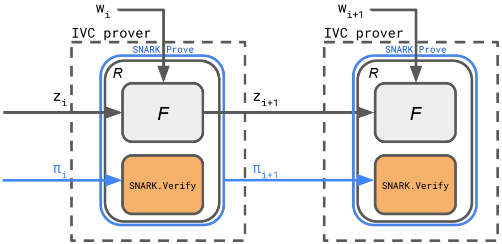
full recursion



$$\left.\begin{aligned} z_{i+1} &= F(z_i;\ w_i) \\ V(\pi_i) &= 1 \end{aligned}\right\}\ \pi_{i+1}$$

overview: *constructions*

full recursion

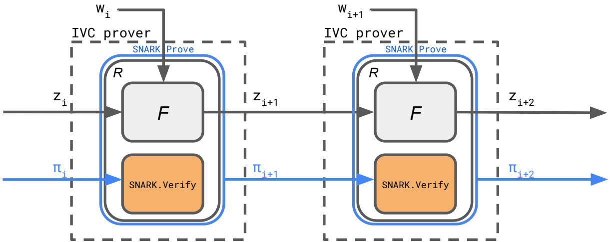


$$\left.\begin{array}{l} z_{i+1} = F(z_i;\ w_i) \\ V(\pi_i) = 1 \end{array}\right\} \pi_{i+1}$$

$$z_{i+2} = F(z_{i+1};\ w_{i+1})$$
$$V(\pi_{i+1}) = 1$$

overview: *constructions*

full recursion

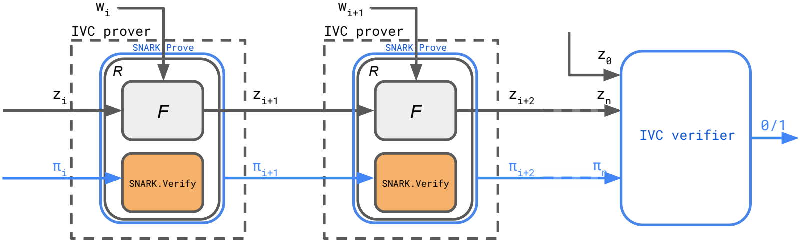


$$\left.\begin{aligned} z_{i+1} &= F(z_i;\ w_i)\\ V(\pi_i) &= 1\end{aligned}\right\}\ \pi_{i+1}$$

$$\left.\begin{aligned} z_{i+2} &= F(z_{i+1};\ w_{i+1})\\ V(\pi_{i+1}) &= 1\end{aligned}\right\}\ \pi_{i+2}$$

overview: *constructions*

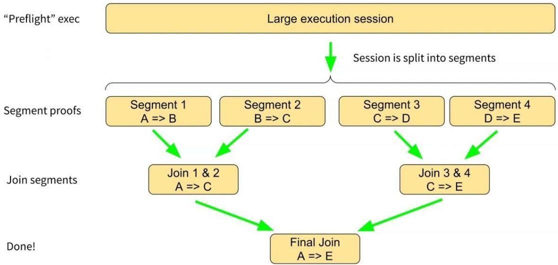
full recursion



$$\left.\begin{array}{l} z_{i+1} = F(z_i;\ w_i) \\ V(\pi_i) = 1 \end{array}\right\} \pi_{i+1} \qquad \left.\begin{array}{l} z_{i+2} = F(z_{i+1};\ w_{i+1}) \\ V(\pi_{i+1}) = 1 \end{array}\right\} \pi_{i+2}$$

overview: *constructions*

full recursion: small-field FRI

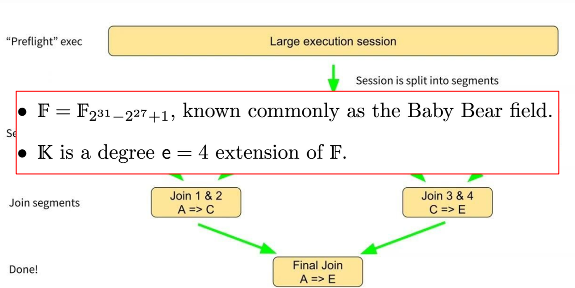


Segmented execution and proving

image from https://www.risczero.com/news/continuations

overview: *constructions*

full recursion: small-field FRI



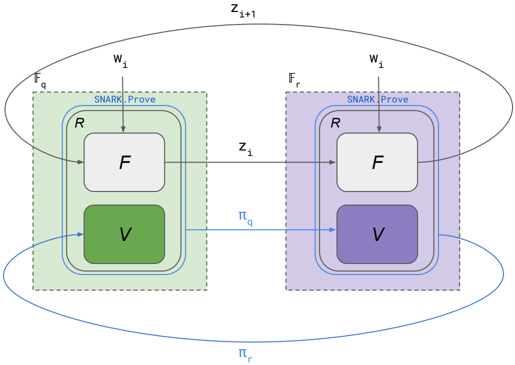
- $\mathbb{F} = \mathbb{F}_{2^{31}-2^{27}+1}$, known commonly as the Baby Bear field.
- $\mathbb{K}$ is a degree $\mathsf{e} = 4$ extension of $\mathbb{F}$.

Segmented execution and proving

image from https://www.risczero.com/news/continuations

overview: *constructions*

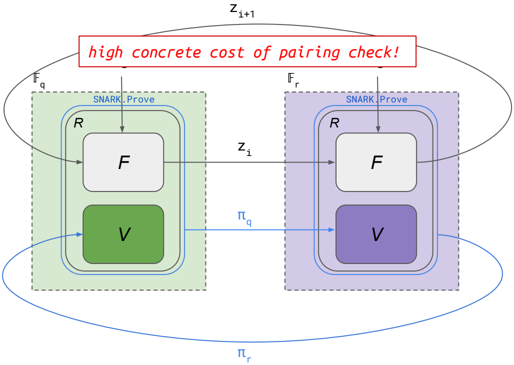
full recursion: pairings over a cycle of elliptic curves [BCTV14]



e.g. MNT4/6 curves

overview: *constructions*

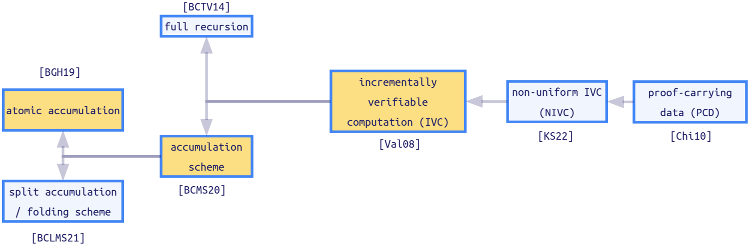
full recursion: pairings over a cycle of elliptic curves [BCTV14]



high concrete cost of pairing check!

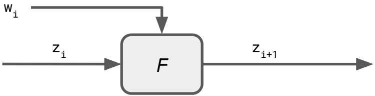
e.g. MNT4/6 curves

overview: *constructions*



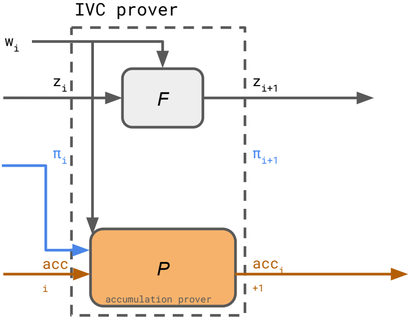
overview: *constructions*

atomic accumulation



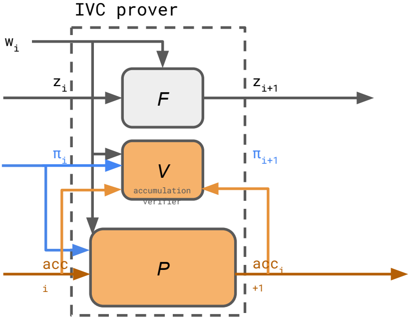
overview: *constructions*

atomic accumulation



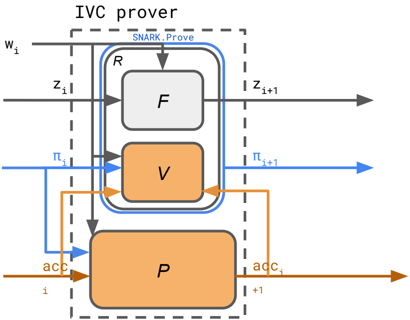
overview: *constructions*

atomic accumulation



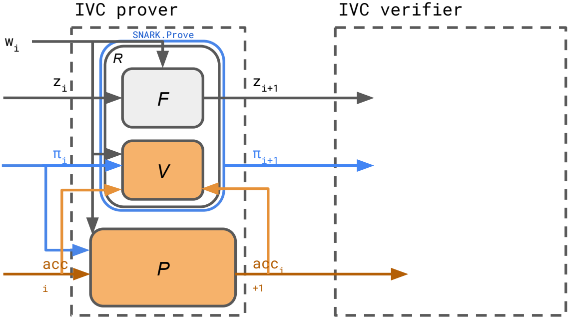
overview: *constructions*

atomic accumulation



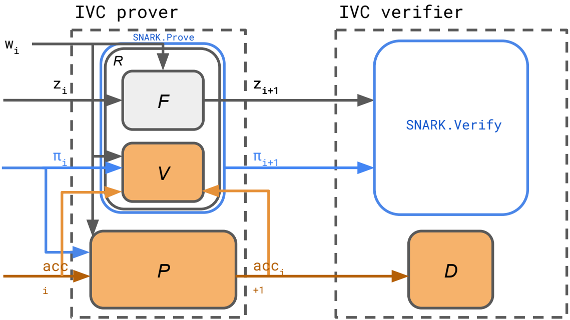
overview: *constructions*

atomic accumulation

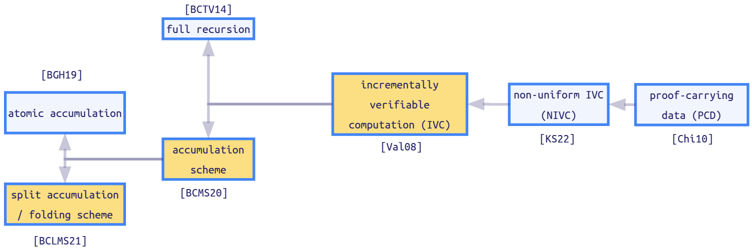


overview: *constructions*

atomic accumulation

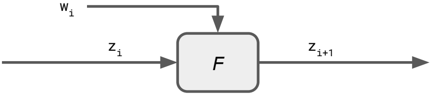


overview: *constructions*



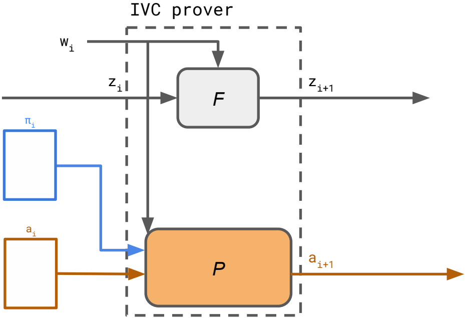
overview: *constructions*

split accumulation / folding



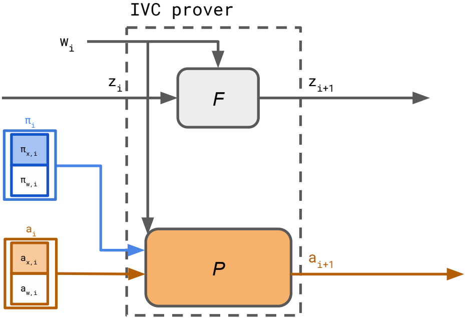
overview: *constructions*

split accumulation / folding



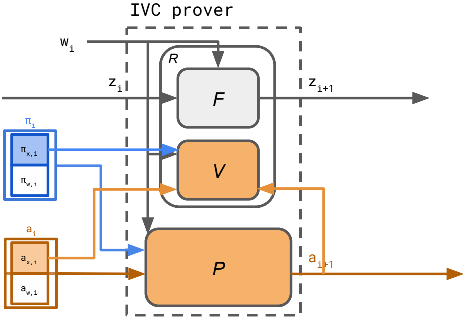
overview: *constructions*

split accumulation / folding



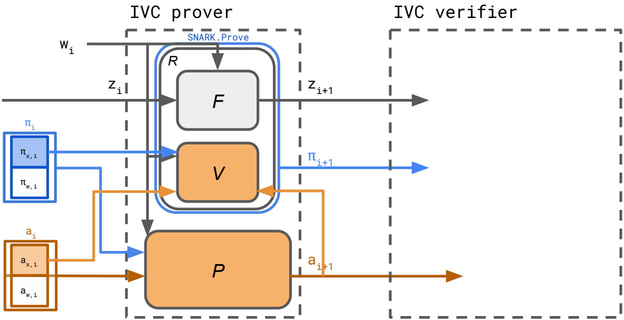
overview: *constructions*

split accumulation / folding



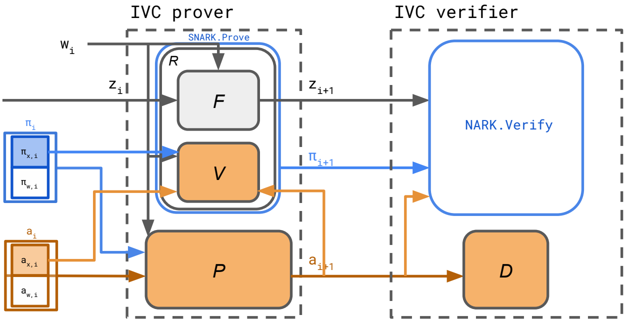
overview: *constructions*

split accumulation / folding



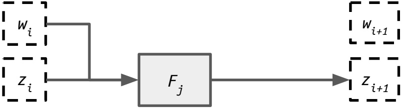
overview: *constructions*

split accumulation / folding



overview: *constructions*

non-uniform IVC (NIVC)



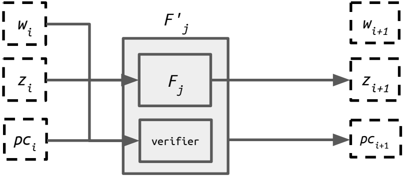
$F_j(w_i, z_i) = z_{i+1}$

overview: *constructions*

non-uniform IVC (NIVC)

verifier*:*

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

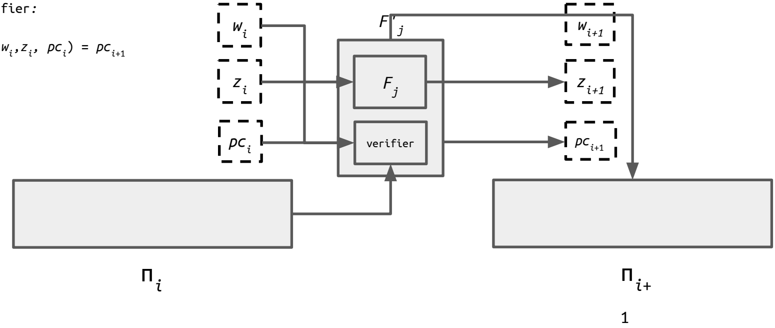


overview: *constructions*

non-uniform IVC (NIVC)

verifier*:*

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

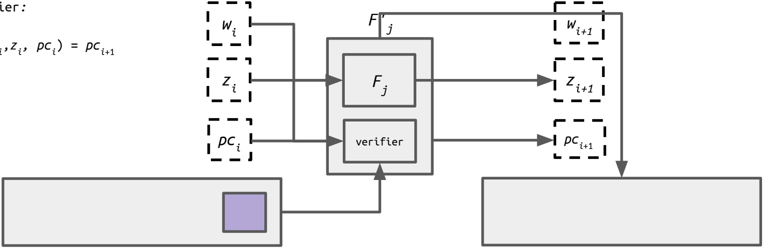


overview: *constructions*

non-uniform IVC (NIVC)

verifier*:*

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



Π_i

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

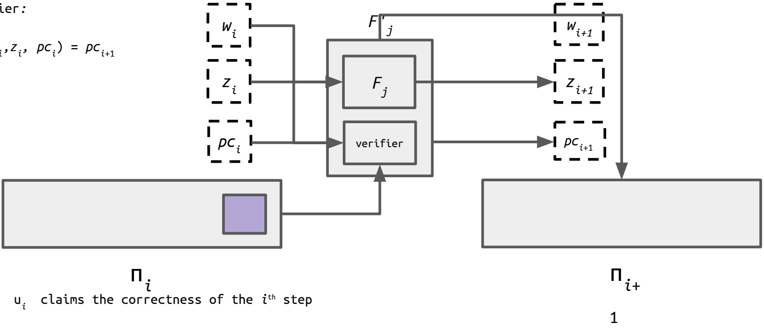
Π_{i+1}

overview: *constructions*

non-uniform IVC (NIVC)

verifier:

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



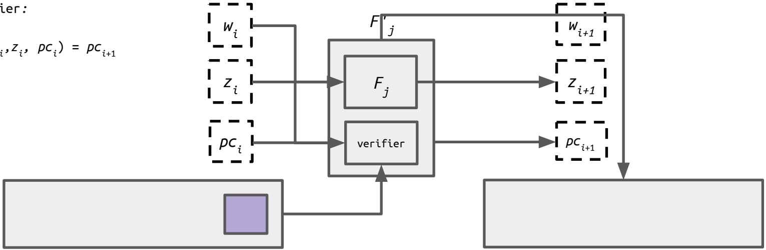
u_i claims the correctness of the i^{th} step

overview: *constructions*

non-uniform IVC (NIVC)

verifier:

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



π_i

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

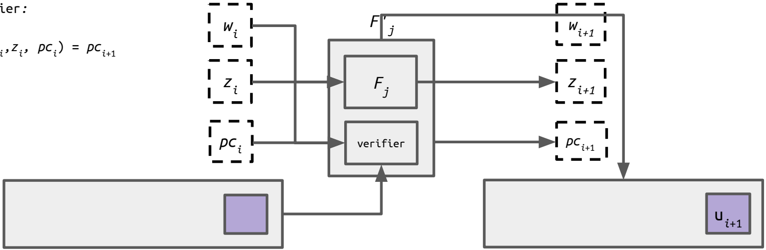
π_{i+1}

overview: *constructions*

non-uniform IVC (NIVC)

verifier*:*

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



Π_i

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

Π_{i+}

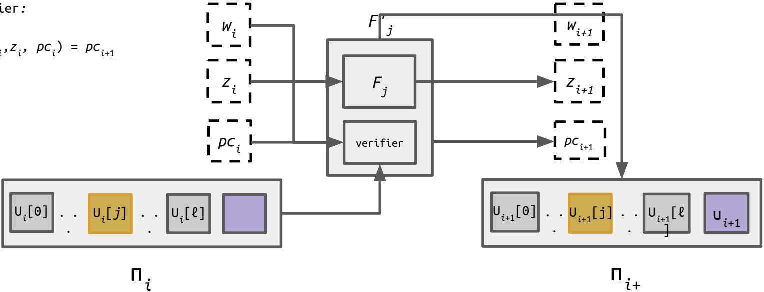
1

overview: *constructions*

non-uniform IVC (NIVC)

verifier:

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



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

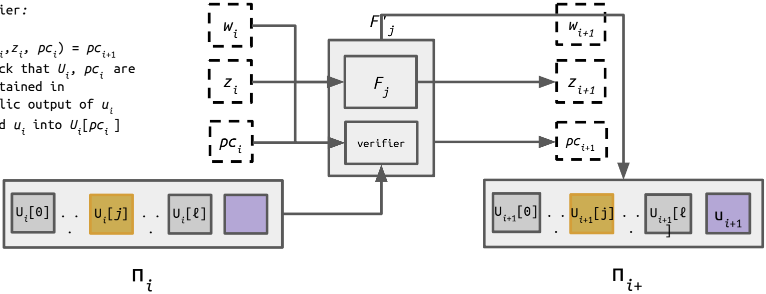
$U_i[j]$ attests to all prior i -1 iterations of F'_j

overview: *constructions*

non-uniform IVC (NIVC)

verifier*:*

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

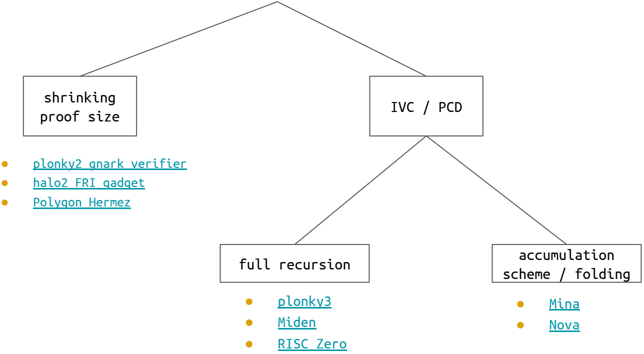
agenda

1. **overview**
    a) motivation
    b) constructions

2. **comparison**
    a) recursion threshold
    b) zero-knowledgeness
    c) security and cryptographic assumptions

3. **focus: CycleFold**

comparison: *implementations*



shrinking proof size

- plonky2 gnark verifier
- halo2 FRI gadget
- Polygon Hermez

IVC / PCD

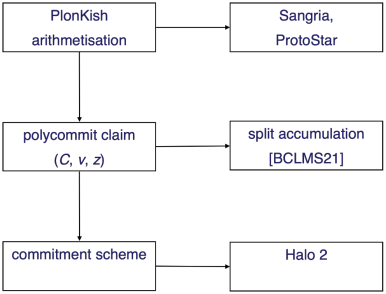
full recursion

- plonky3
- Miden
- RISC Zero

accumulation scheme / folding

- Mina
- Nova

comparison: *recursion threshold*



PlonKish arithmetisation

Sangria, ProtoStar

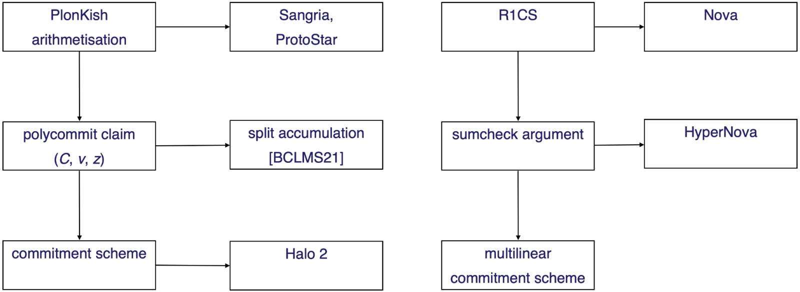
polycommit claim (C, v, z)

split accumulation [BCLMS21]

commitment scheme

Halo 2

comparison: *recursion threshold*



comparison: *recursion threshold*

protocol	**relation**	**accumulator**	**“reduce”**	**“combine”**
halo2-IPA	PlonKish	IPA polycommit opening proofs	*P*: vanishing argument, multiopen argument, IPA	*P*: random linear combination and opening proof
			V: produce challenges, check multiopen argument, check logarithmic part of IPA	*V*: random linear combination and partial opening proof
BCLMS21	R1CS	Hadamard product vector commitment claims	*P*: commit to matrix-vector product	*P*: commit to error term
			V: none	*V*: add commitments w/ error
Nova	R1CS	committed relaxed R1CS	*P*: commit to witness	*P*: commit to error term
			V: none	*V*: add commitments w/ error
Sangria	PlonK	committed relaxed PlonK	*P*: commit to witness	*P*: commit to error term
			V: none	*V*: add commitments w/ error

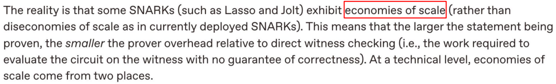
taken from Nico Mohnblatt’s presentation

comparison: *recursion threshold*

protocol	relation	accumulator	"reduce"	"combine"
Nova	R1CS	committed relaxed R1CS	*P*: commit to witness	*P*: commit to error term
			V: none	*V*: add commitments w/ error
HyperNova	CCS	linearised committed CCS	*P*: commit to witness	*P*: random linear combination
			P and *V*: run the sumcheck protocol	*V*: random linear combination
ProtoStar	any relation w/ algebraic verifier	commitments to all messages and compressed verifier check	*P*: commit to each message	*P*: compute the compressed cross terms
			V: produce random challenges	*V*: add commitments and compressed cross terms

taken from Nico Mohnblatt's presentation

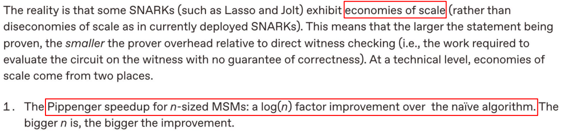
comparison: *recursion threshold*



The reality is that some SNARKs (such as Lasso and Jolt) exhibit economies of scale (rather than diseconomies of scale as in currently deployed SNARKs). This means that the larger the statement being proven, the *smaller* the prover overhead relative to direct witness checking (i.e., the work required to evaluate the circuit on the witness with no guarantee of correctness). At a technical level, economies of scale come from two places.

from https://a16zcrypto.com/posts/article/introducing-lasso-and-jolt

comparison: *recursion threshold*

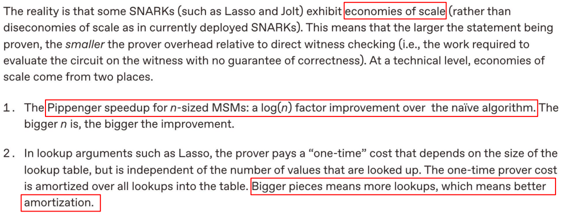


The reality is that some SNARKs (such as Lasso and Jolt) exhibit economies of scale (rather than diseconomies of scale as in currently deployed SNARKs). This means that the larger the statement being proven, the *smaller* the prover overhead relative to direct witness checking (i.e., the work required to evaluate the circuit on the witness with no guarantee of correctness). At a technical level, economies of scale come from two places.

1. The Pippenger speedup for n-sized MSMs: a $\log(n)$ factor improvement over the naïve algorithm. The bigger n is, the bigger the improvement.

from https://a16zcrypto.com/posts/article/introducing-lasso-and-jolt

comparison: *recursion threshold*

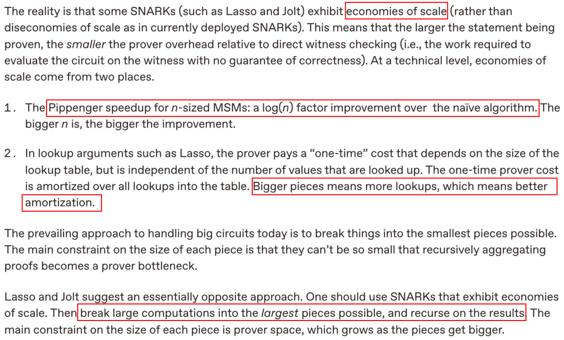


The reality is that some SNARKs (such as Lasso and Jolt) exhibit economies of scale (rather than diseconomies of scale as in currently deployed SNARKs). This means that the larger the statement being proven, the *smaller* the prover overhead relative to direct witness checking (i.e., the work required to evaluate the circuit on the witness with no guarantee of correctness). At a technical level, economies of scale come from two places.

1. The Pippenger speedup for n-sized MSMs: a $\log(n)$ factor improvement over the naïve algorithm. The bigger n is, the bigger the improvement.
2. In lookup arguments such as Lasso, the prover pays a "one-time" cost that depends on the size of the lookup table, but is independent of the number of values that are looked up. The one-time prover cost is amortized over all lookups into the table. Bigger pieces means more lookups, which means better amortization.

from https://a16zcrypto.com/posts/article/introducing-lasso-and-jolt

comparison: *recursion threshold*



The reality is that some SNARKs (such as Lasso and Jolt) exhibit economies of scale (rather than diseconomies of scale as in currently deployed SNARKs). This means that the larger the statement being proven, the *smaller* the prover overhead relative to direct witness checking (i.e., the work required to evaluate the circuit on the witness with no guarantee of correctness). At a technical level, economies of scale come from two places.

1. The Pippenger speedup for *n*-sized MSMs: a log(*n*) factor improvement over the naïve algorithm. The bigger *n* is, the bigger the improvement.
2. In lookup arguments such as Lasso, the prover pays a “one-time” cost that depends on the size of the lookup table, but is independent of the number of values that are looked up. The one-time prover cost is amortized over all lookups into the table. Bigger pieces means more lookups, which means better amortization.

The prevailing approach to handling big circuits today is to break things into the smallest pieces possible. The main constraint on the size of each piece is that they can’t be so small that recursively aggregating proofs becomes a prover bottleneck.

Lasso and Jolt suggest an essentially opposite approach. One should use SNARKs that exhibit economies of scale. Then break large computations into the *largest* pieces possible, and recurse on the results. The main constraint on the size of each piece is prover space, which grows as the pieces get bigger.

from [*https://a16zcrypto.com/posts/article/introducing-lasso-and-jolt*](https://a16zcrypto.com/posts/article/introducing-lasso-and-jolt)

comparison: *cryptographic assumptions*

protocol	**relation**	**accumulator**	**“reduce”**	**“combine”**
Nova	R1CS	committed relaxed R1CS	P: commit to witness	P: commit to error term
			V: none	V: add commitments w/ error
HyperNova	CCS	linearised committed CCS	P: commit to witness	P: random linear combination
			P and V: run the sumcheck protocol	V: random linear combination
ProtoStar	any relation w/ algebraic verifier	commitments to all messages and compressed verifier check	P: commit to each message	P: compute the compressed cross terms
			V: produce random challenges	V: add commitments and compressed cross terms

need additively homomorphic commitments!

taken from Nico Mohnblatt’s presentation

comparison: *cryptographic assumptions*

protocol	**relation**	**accumulator**	**“reduce”**	**“combine”**
Nova	R1CS	committed relaxed R1CS	*P*: commit to witness	*P*: commit to error term
			V: none	*V*: add commitments w/ error
HyperNova	CCS	linearised committed CCS	*P*: commit to witness	*P*: random linear combination
			P and *V*: run the sumcheck protocol	*V*: random linear combination
ProtoStar	any relation w/ algebraic verifier	commitments to all messages and compressed verifier check	*P*: commit to each message	*P*: compute the compressed cross terms
			V: produce random challenges	*V*: add commitments and compressed cross terms

need additively homomorphic commitments!
typically uses ***cryptographic group***

taken from Nico Mohnblatt’s presentation

comparison: *cryptographic assumptions*

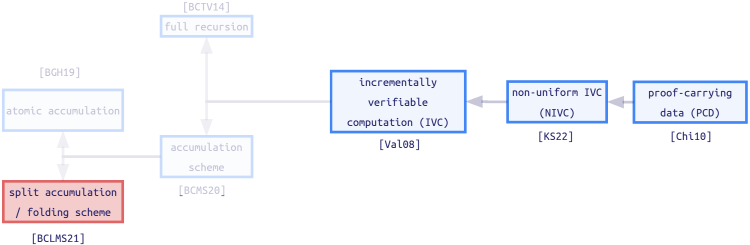
FRI-based
full recursion

assumes existence of one-way function
for non-interactivity (BCS transform)

folding schemes

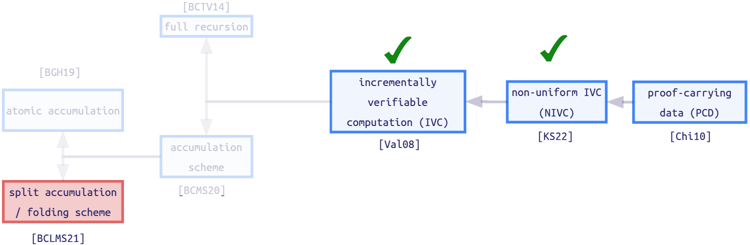
need additively homomorphic commitments!
typically uses ***cryptographic group (DLOG hardness)***

comparison: *zero-knowledgeness of recursive step*



proving step is not zero-knowledge in split accumulation

comparison: *zero-knowledgeness of recursive step*



[BCTV14]
full recursion

[BGH19]
atomic accumulation

accumulation scheme
[BCMS20]

split accumulation / folding scheme
[BCLMS21]

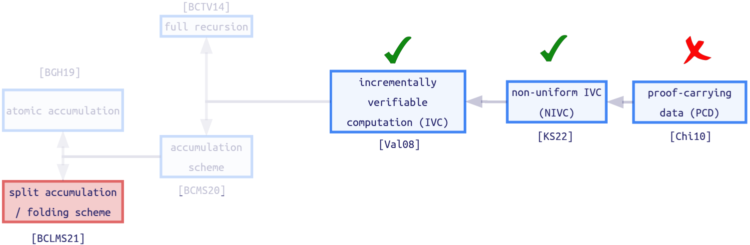
✓
incrementally verifiable computation (IVC)
[Val08]

✓
non-uniform IVC (NIVC)
[KS22]

proof-carrying data (PCD)
[Chi10]

proving step is not zero-knowledge in split accumulation; this is fine for IVC, where a single witness is split into incremental chunks

comparison: *zero-knowledgeness of recursive step*



[BCTV14]
full recursion

[BGH19]
atomic accumulation

accumulation scheme
[BCMS20]

split accumulation / folding scheme
[BCLMS21]

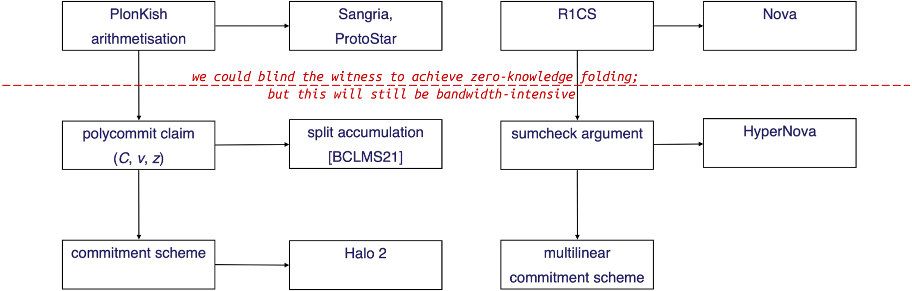
✓
incrementally verifiable computation (IVC)
[Val08]

✓
non-uniform IVC (NIVC)
[KS22]

✗
proof-carrying data (PCD)
[Chi10]

proving step is not zero-knowledge in split accumulation; this is fine for IVC, where a single witness is split into incremental chunks; but less suitable for PCD, where each prover has its own witness

comparison: *zero-knowledgeness of recursive step*



PlonKish arithmetisation

Sangria, ProtoStar

R1CS

Nova

we could blind the witness to achieve zero-knowledge folding; but this will still be bandwidth-intensive

polycommit claim (C, v, z)

split accumulation [BCLMS21]

sumcheck argument

HyperNova

commitment scheme

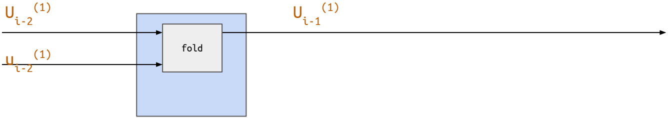
Halo 2

multilinear commitment scheme

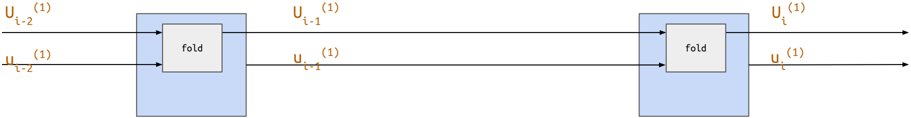
agenda

1. **overview**
   a) motivation
   b) constructions
2. **comparison**
   a) recursion threshold
   b) zero-knowledgeness
   c) security and cryptographic assumptions
3. **focus: CycleFold**

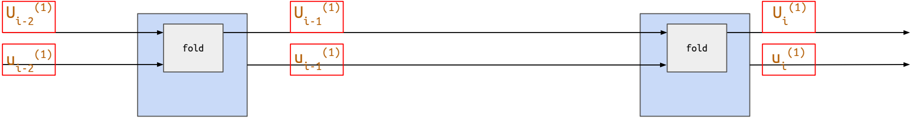
CycleFold



CycleFold

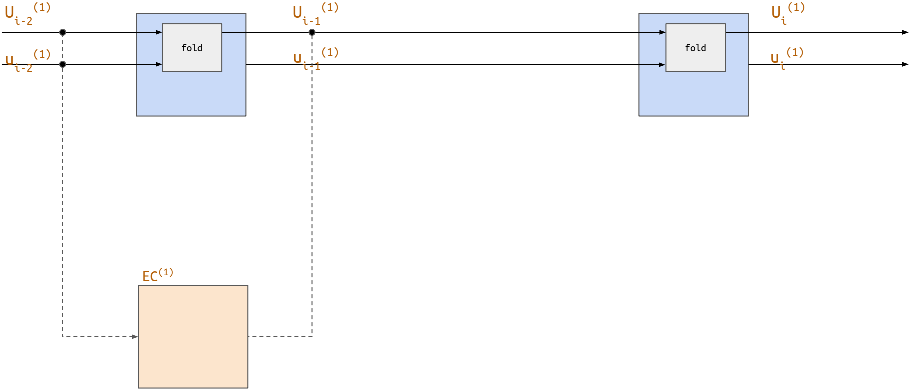


CycleFold



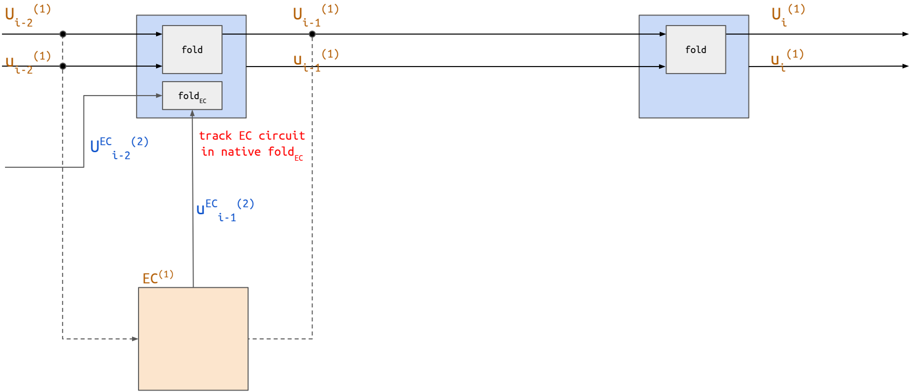
these values are in the wrong field!

CycleFold



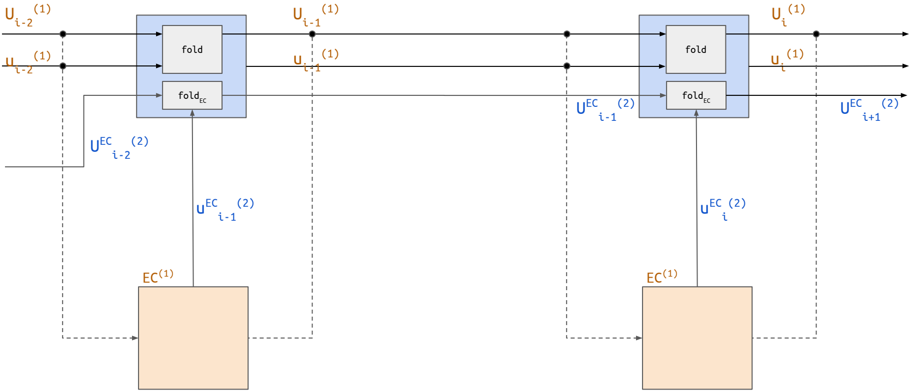
defer to EC circuit
on second curve

CycleFold



track EC circuit in native fold_{EC}

CycleFold


$U_{i-2}^{(1)}$
$u_{i-2}^{(1)}$
fold
fold_{EC}
$U^{EC}_{i-2}{}^{(2)}$
$U_{i-1}^{(1)}$
$u_{i-1}^{(1)}$
$u^{EC}_{i-1}{}^{(2)}$
$EC^{(1)}$
$U^{EC}_{i-1}{}^{(2)}$
fold
fold_{EC}
$U_{i}^{(1)}$
$u_{i}^{(1)}$
$U^{EC}_{i+1}{}^{(2)}$
$u^{EC}_{i}{}^{(2)}$
$EC^{(1)}$