Constructions and trends in modern zero knowledge proofs

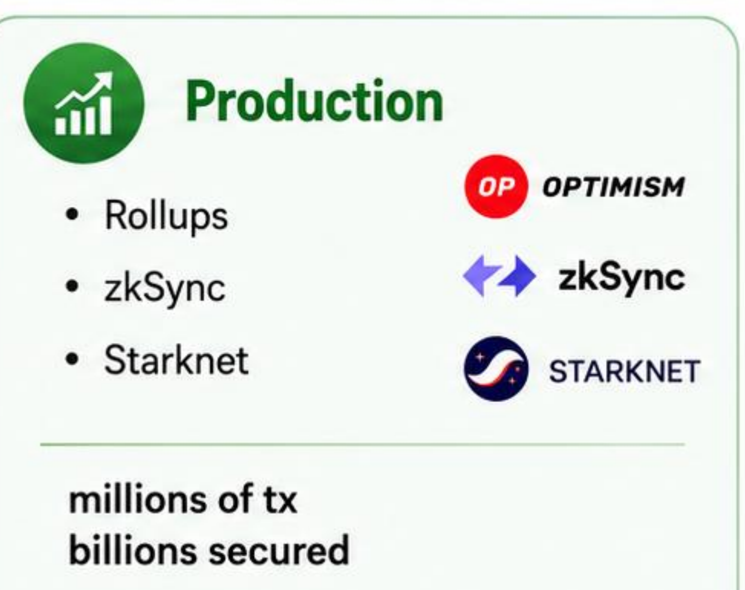
Marek Sýs, syso@mail.muni.cz

Centre for Research on Cryptography and Security, Masaryk University

Where ZK Works Today

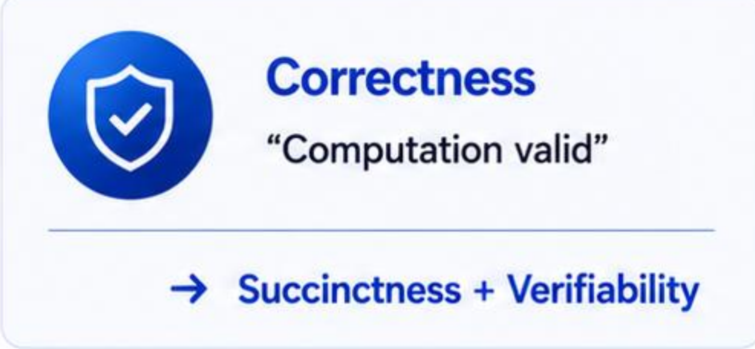
SUCCINCTNESS
verify cheaper than execution

VERIFIABILITY
anyone can verify



ZK succeeds when privacy or verification savings justify the proving overhead.

What ZK Proves



Private prover data

+

Public claim

+

Rules

→

Proof

→

Anyone can verify

EXAMPLE: AGE VERIFICATION

birth date (private)

+

age ≥ 18 (public)

+

correct age calc (rules)

→

proof

→

verifier checks claim

Different computations produce very different proving costs.

ZK Proves Math — Not Reality

Real World Data (untrusted input)

- fake sensor
- dishonest issuer
- manipulated oracle

↓

Rules / Circuit

↓

Proof

- ✓ computation correct
- ✗ real-world truth

Still requires trust anchors:

issuers • oracles • attestations • sensors

Valid proof ≠ true real-world claim.

Why Building ZK Systems Is Hard

Silent Bugs

proof verifies, but wrong claim was encoded

Example: off-by-one error in constraint

Under-Constrained Circuits

missing rule → fake witness can still pass

Example: forgot range check on balance

Expensive Audits

crypto + software expertise needed

subtle mistakes expensive reviews

A valid proof only means:

"the encoded rules were satisfied."

ZK correctness is an engineering problem, not only a math problem.

Core Asymmetry

Succinctness = verification cheaper than re-execution

Prover

- Heavy computation
- roughly O(n log n) work
- pays the cost

→ proof →

Verifier

- Small probabilistic check
- often O(1) or O(log n)
- checks, not recomputes

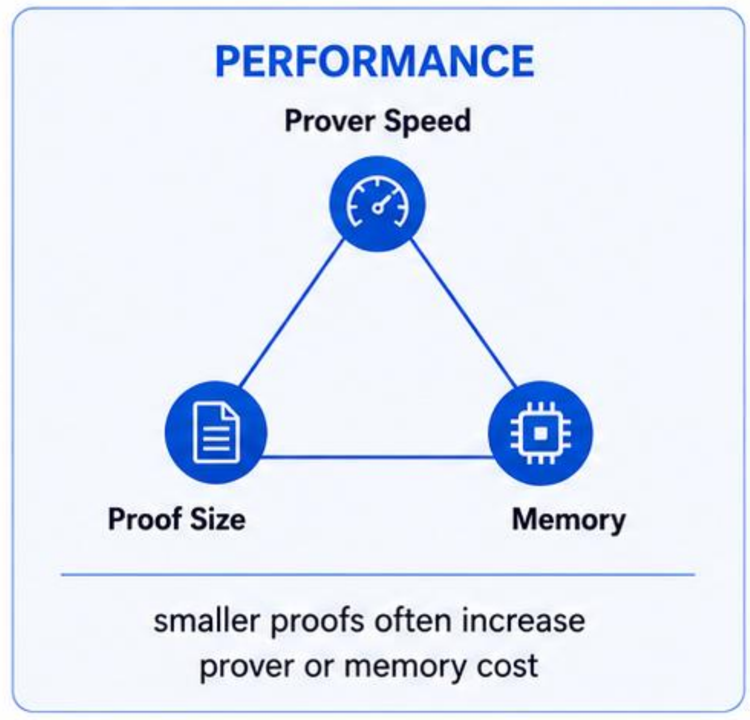
Direct execution:
1M-gate computation
seconds

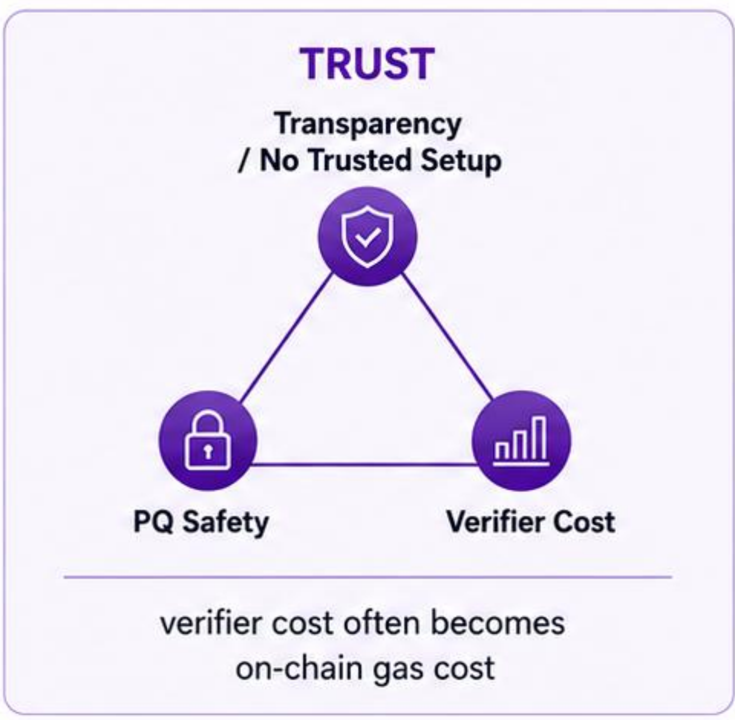
ZK verification:
same proof
milliseconds

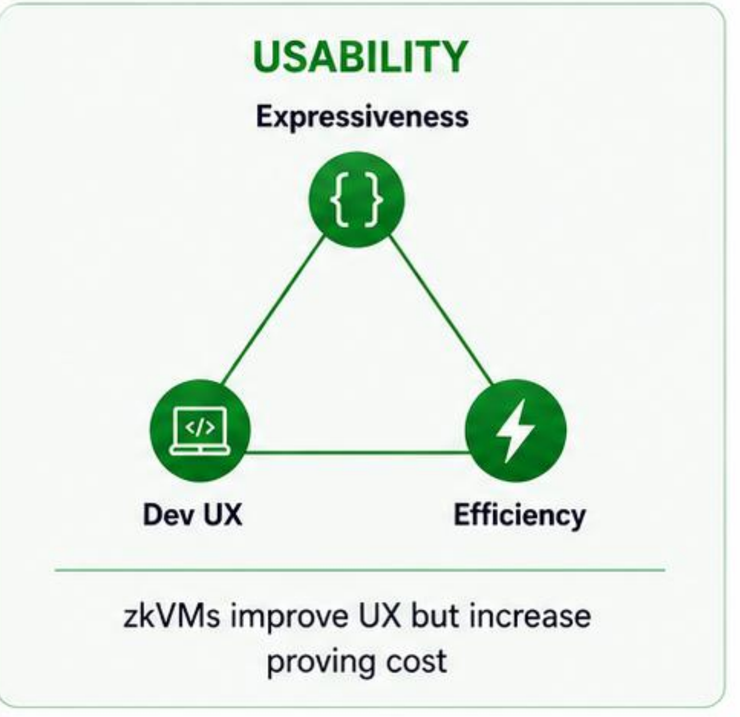
Verifier speedup:
often
100×–1000×

Cheap verification requires expensive proving.

You Cannot Optimize Everything







Note: Recursion / aggregation cost varies widely across systems.

SNARKs
→ tiny proofs, fast verification

STARKs
→ transparency + post-quantum friendliness

zkVMs
→ better developer UX, higher proving cost

Different priorities lead to different proof systems.

Digital Signature as a ZK Pattern

PROVER (has witness: private key, k)

VERIFIER (has public key)

Prover	Message	Verifier
Pick random k $R = g^k$	R →	$r = R \bmod q$
Solve equation $s \cdot k = H(m) + r \cdot \text{private}$	← m (challenge)	Source of message m (e.g., user / application)
Signature $= (r, s)$	(r, s) →	Verify lifted: $g^{H(m)} \cdot \text{public}^r = R^s$

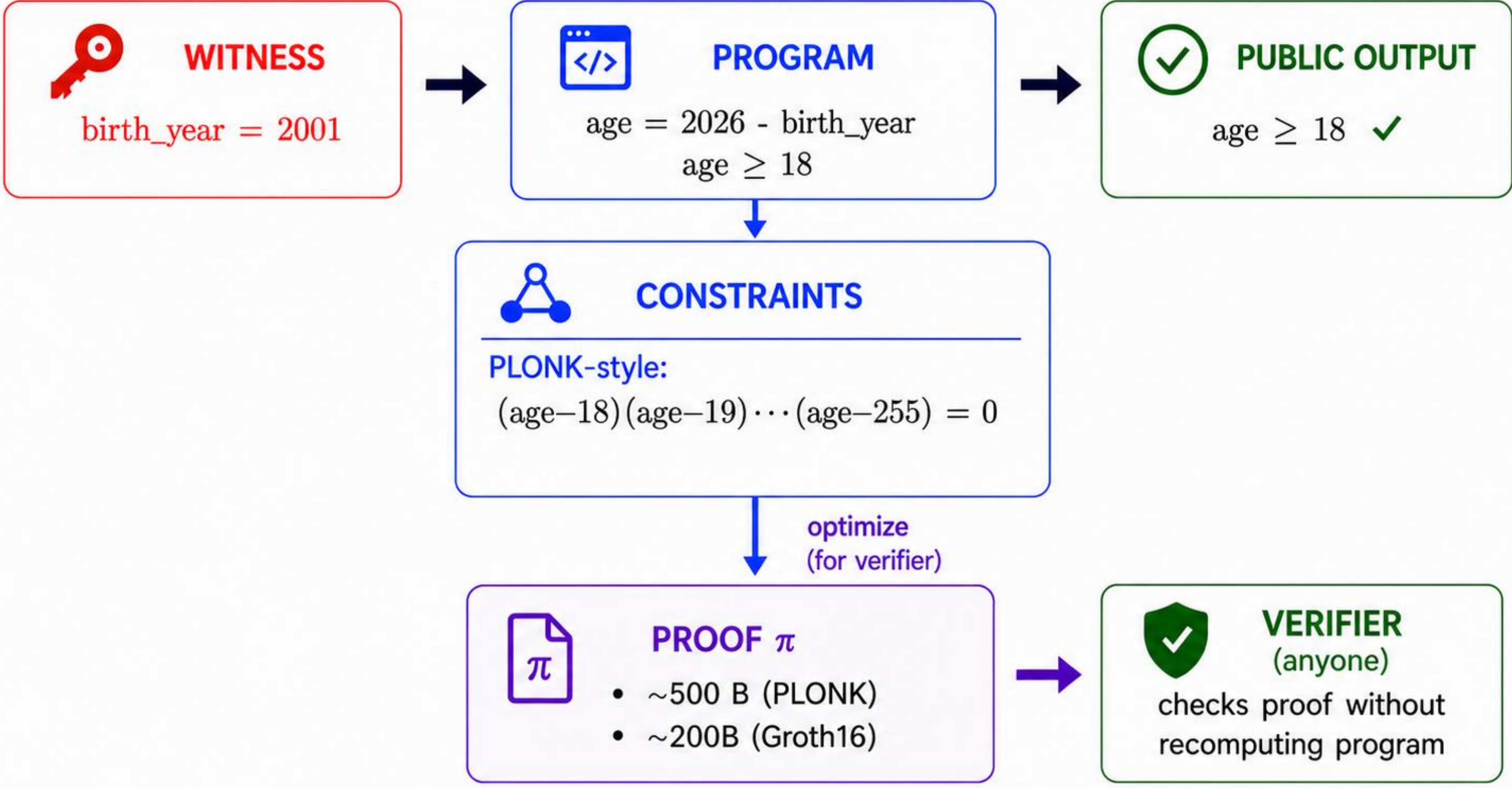
Embed in hard problem
(computational hardness)
$$g^{s \cdot k} = g^{H(m) + r \cdot p}$$

Completeness
honest prover verifies

Soundness
without witness, fake proof fails

Zero-knowledge
witness stays hidden

Generalizing ZK to Arbitrary Programs



From Program to Proof

Program expands into **many constraints.** Each constraint **embedded into hard problem.**
Prover creates **compressed algebraic certificate.** **Verifier** checks quickly without recomputing.

1. Program

Simple code

(public)

→ EXPANDS

2. Symbolic Circuit

Many constraints

(public)

→

3. Assign Witness

Fill private values

birth_year = 2001

(private)

EMBED IN HARD PROBLEM →

4. Commitment + Challenge

Commit C, challenge r

(public)

→ OPTIMIZATION

5. Proof π

Response

~100B–~100KB

(public)

→

6. Verify (anyone)

Check few evaluations

NOT all constraints

Public Result

age $\geq$ 18:

TRUE

(public)

Details: Commitment C binds to witness values. Challenge r = Hash(C) makes it non-interactive (Fiat–Shamir).

Every constraint embedded into hard problem → proving expensive.

SNARKs vs STARKs *(Old / Simplified Understanding)*

SNARKs

Succinct **N**on–interactive
ARgument of **K**nowledge

(pairing based)

- ✓ tiny proofs
- ✓ fast verification
- ✓ low on–chain cost
- ✗ often setup (trusted setup)
- ✗ not post–quantum resistant

Optimized for:
minimal proof size

STARKs

Scalable **T**ransparent
ARgument of **K**nowledge

- ✓ transparent setup (no trusted setup)
- ✓ post–quantum friendly
- ✓ scalable proving (high parallelism)
- ✗ larger proofs (typically O(log n))

Optimized for:
transparency + scalability

SNARK vs STARK vs HYBRID

Proof families, not single primitives

SNARK

Succinct proofs, fast verification

- ✓ Very small proofs
- ✓ Cheap verification
- ✓ Low on-chain cost
- ✗ Often setup assumptions

Constructions:

- Pairing / KZG (e.g., Groth16, PLONK)
- IPA (e.g., Halo2)
- Hash / FRI-based SNARKs (e.g., Plonky2)

STARK

Transparent proofs, scalable proving

- ✓ No trusted setup
- ✓ Post-quantum friendly
- ✓ Strong prover parallelism
- ✗ Larger proofs (typically O(log n))

Constructions:

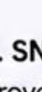
- FRI (Fast Reed–Solomon IOP)
- Merkle trees
- Hash-based commitments

HYBRID

Best of both worlds

1. Large computation
Executed off-chain

2. STARK proving
Scalable, transparent proof



3. SNARK compression
Proves "STARK verifier accepted"

4. Tiny final proof
Cheap to verify on-chain

- ✓ Fast proving (STARK)
- ✓ Tiny final proof (SNARK)
- ✓ Cheap L1 verification
- ✓ Common in rollups and production systems

HYBRID ARCHITECTURE (Typical Flow)

1. Computation
Program / VM executes

2. STARK Proof
Prove execution with STARK

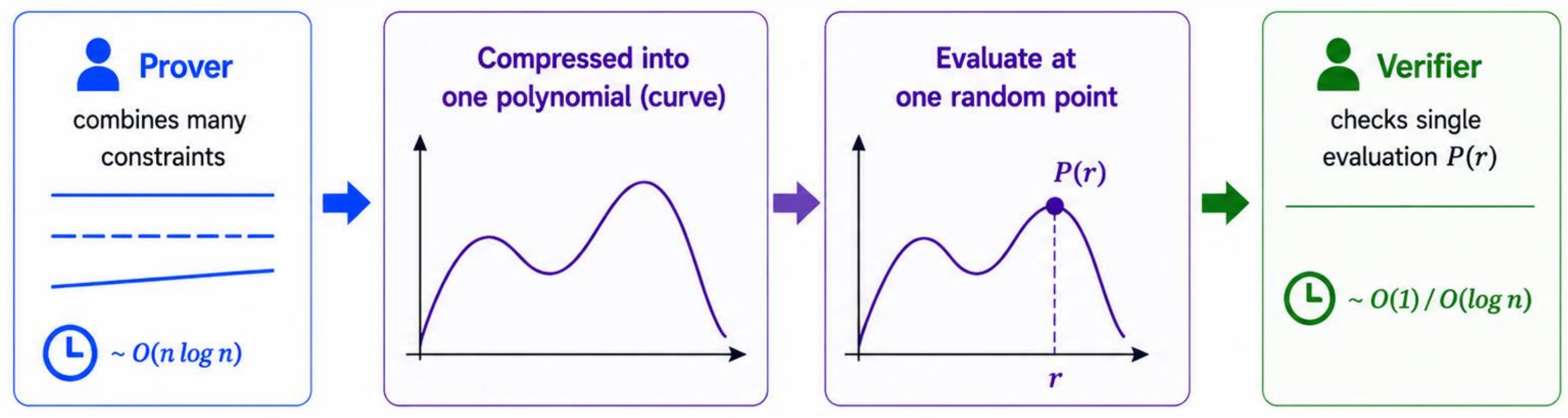
3. SNARK Proof
Prove STARK verifier accepted

4. Final Proof
Very small proof on-chain

Goal:
Scalable proving first, compression last.

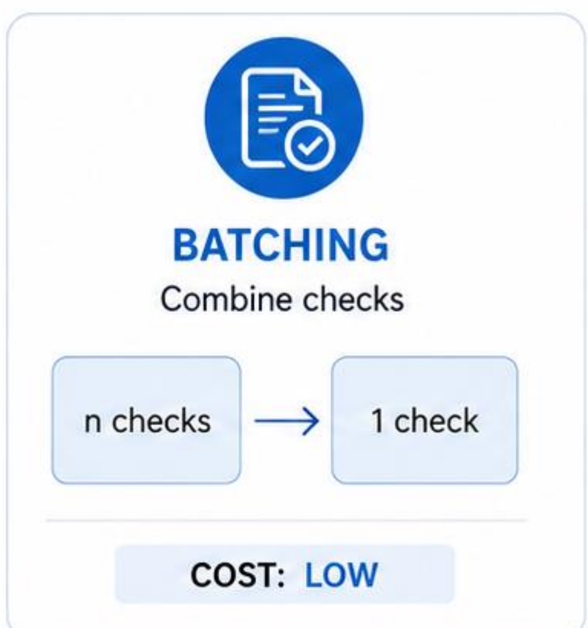
The Succinctness Trick: Constraint Compression

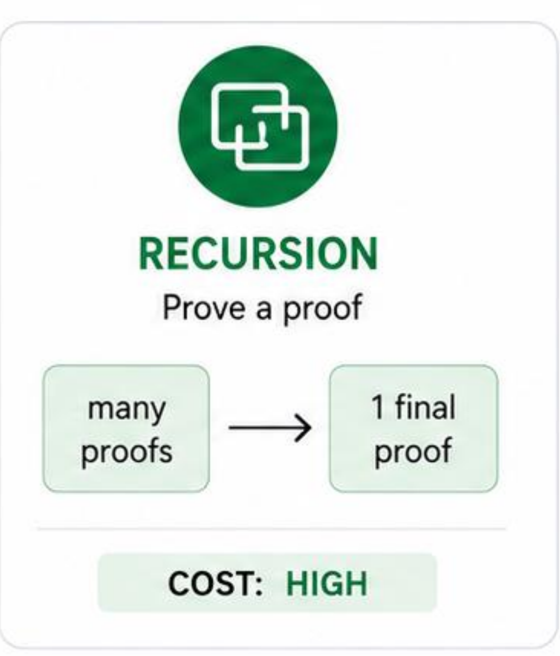
This is where the prover–verifier asymmetry comes from.

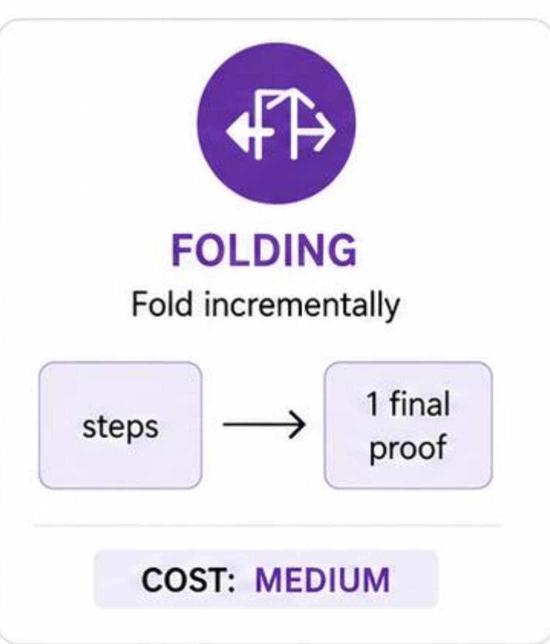


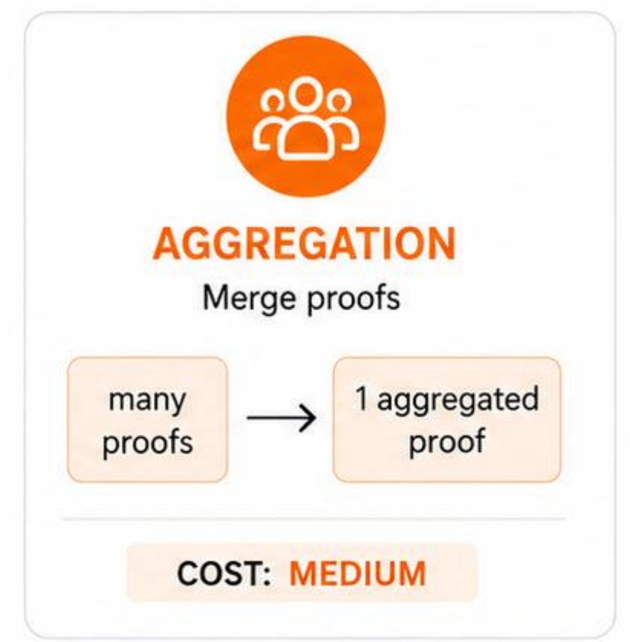
SUCCINCTNESS = many constraints compressed into a proof that is much faster to verify than re-execute.

ADVANCED COMPOSITION TECHNIQUES









TRADEOFF

Batching: cheap, specific use	**Recursion:** powerful but expensive	**Folding:** middle ground, enables IVC	**Aggregation:** parallel-friendly

Modern stacks combine all: batch, fold, aggregate, and compress.

Batching Trick: n Checks → 1 Check

NAIVE

$s_1 \cdot R_1 = H(m_1) \cdot G + r_1 \cdot Public$

$s_2 \cdot R_2 = H(m_2) \cdot G + r_2 \cdot Public$

$\vdots$

$s_n \cdot R_n = H(m_n) \cdot G + r_n \cdot Public$

n checks

$n \cdot$ expensive

BATCHING

$$s_1 \cdot R_1 + s_2 \cdot R_2 + \ldots + s_n \cdot R_n$$
$$= [H(m_1) + H(m_2) + \ldots + H(m_n)] \cdot G$$
$$+ [r_1 + r_2 + \ldots + r_n] \cdot Public$$

1 check

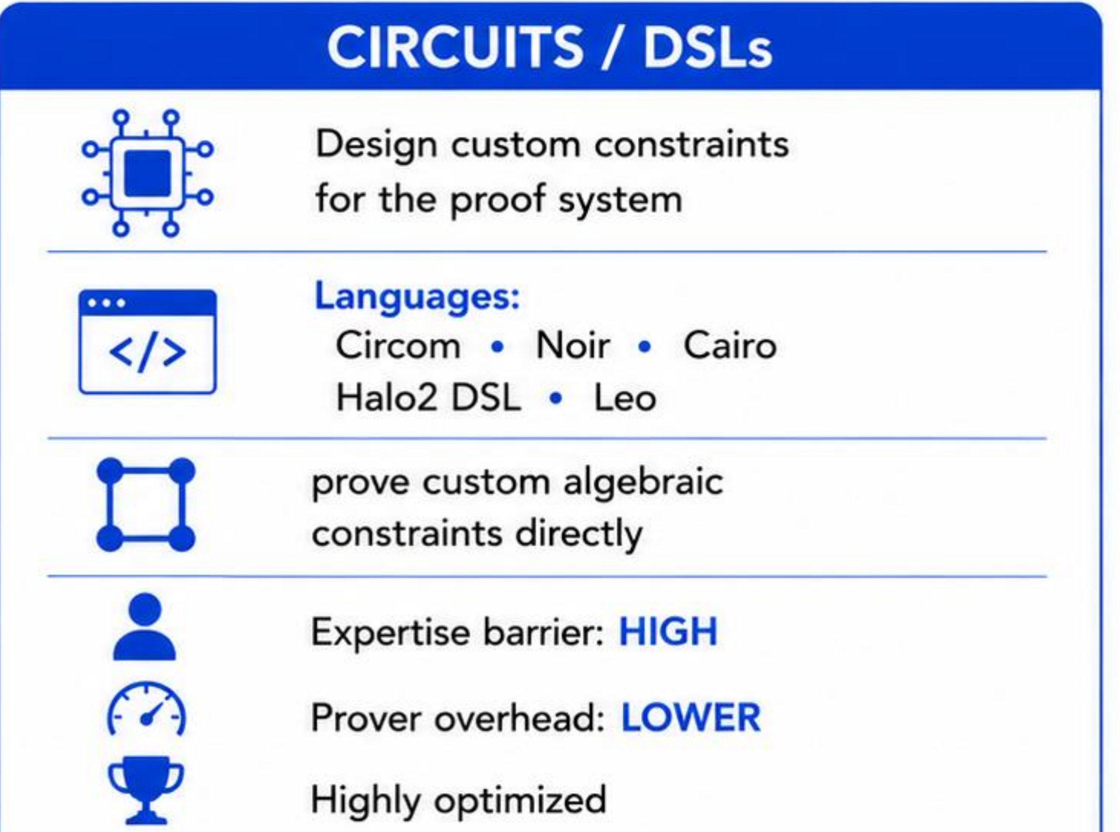
$1 \cdot$ expensive $+ n \cdot$ cheap

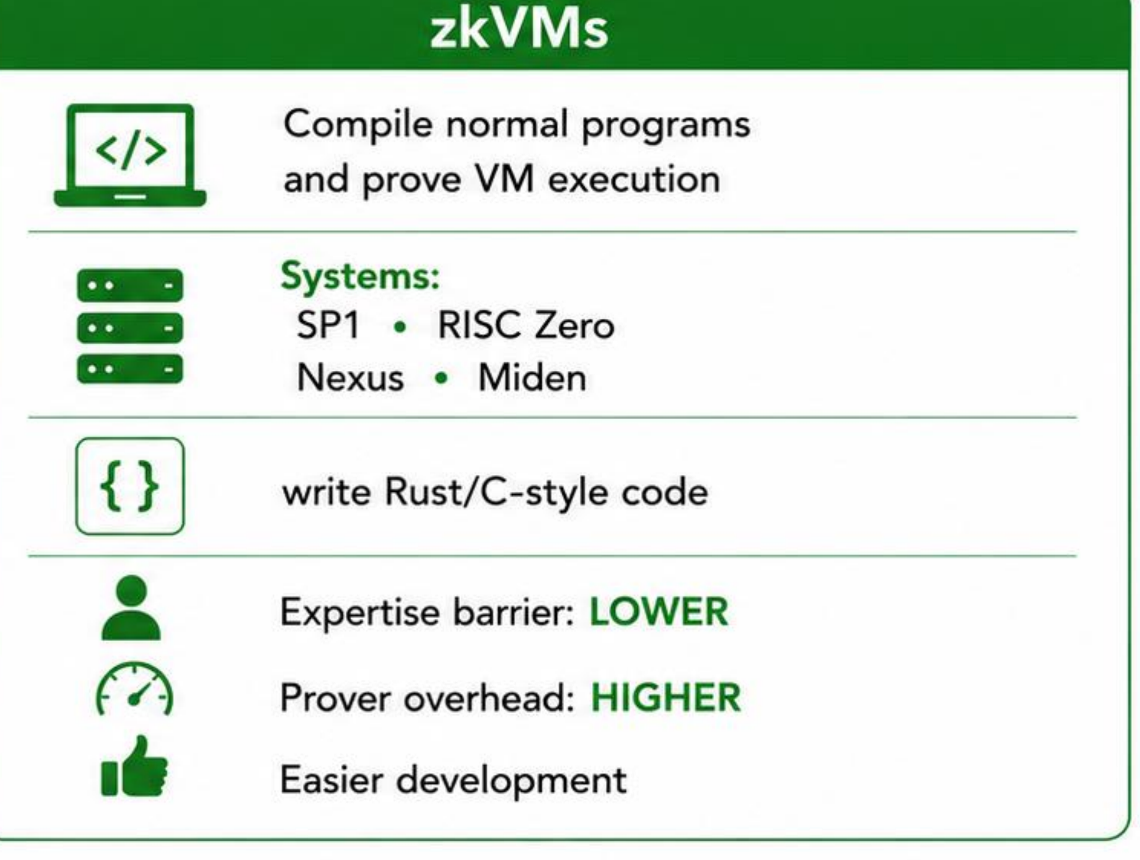
Random weights prevent error cancellation

- Weights chosen after commitment
- Cancellation probability: $\approx 1/|\text{field}|$ (negligible)

Circuits vs zkVMs — Two Ways to Build ZK

Choose your tradeoff: **maximum efficiency** vs **developer simplicity.**





APPLIED IN: ROLLUPS / L2s

prove batches of blockchain transactions

zkSync • Scroll
Polygon zkEVM • Starknet

zkVMs help reuse large existing software stacks

Improving Fast

Near-term progress comes from engineering:

faster hardware, **better tooling**, and **better arithmetization**.

GPUs → FPGAs → ASICs

speeds up repeated algebra

MSMs • FFTs/NTTs • hashing

SP1 • RISC Zero • Jolt

prove normal programs

easier developer workflow

tradeoff: more overhead

- lookups
- custom gates
- folding / recursion

fewer constraints

lower prover overhead

ZK-friendly signatures

cheaper circuits

Fast progress = optimize the prover and simplify development.

Improving Slowly vs. Hard Limits

Some bottlenecks improve with engineering. Others are structural.

IMPROVING SLOWLY

PQ Systems

Post-quantum systems and hash functions are getting better.

Memory Requirements

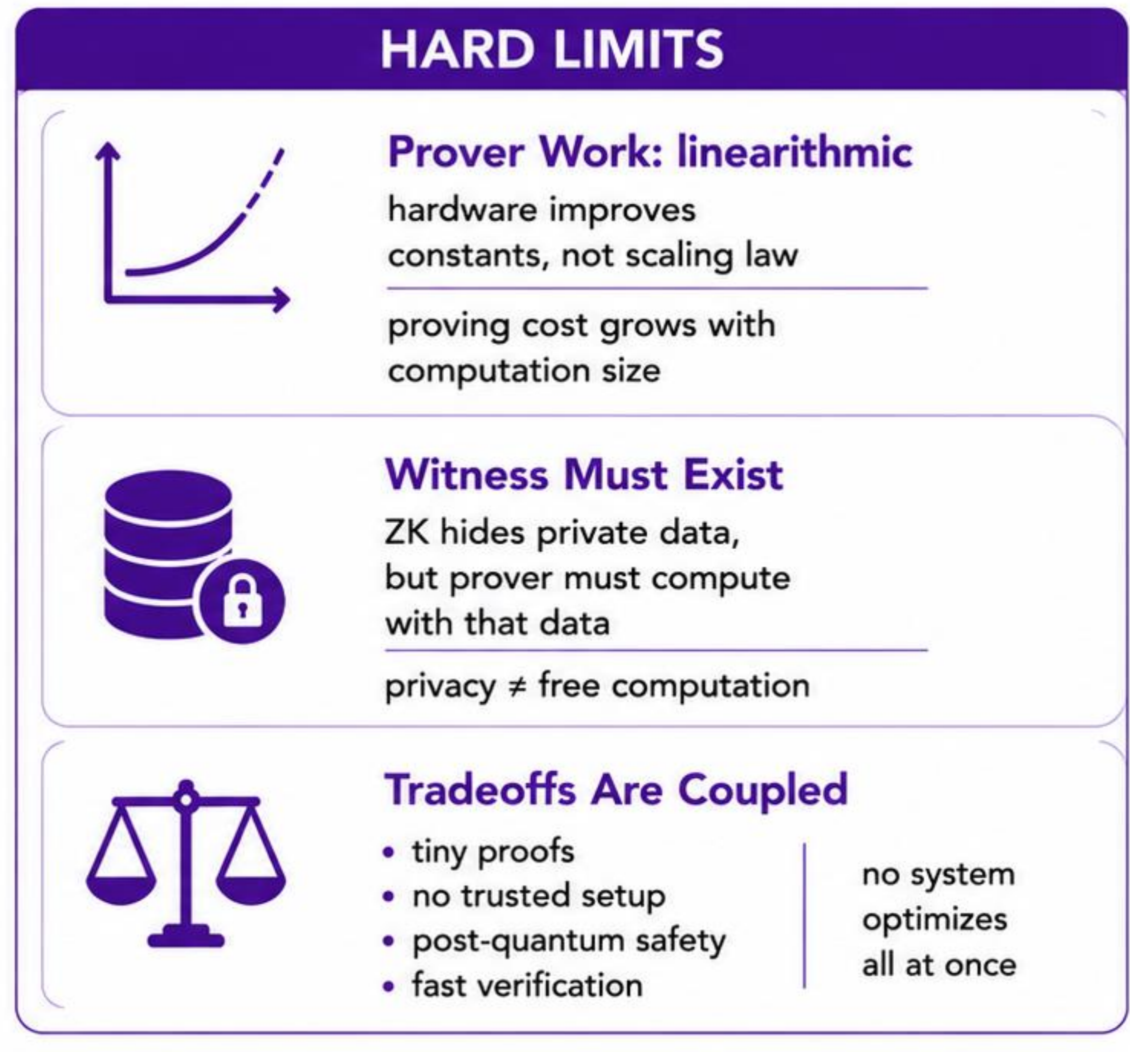
Better hardware and algorithms reduce memory needs gradually.

Developer Tooling

Tooling, debuggers, and SDKs are improving steadily.

Audits & Correctness

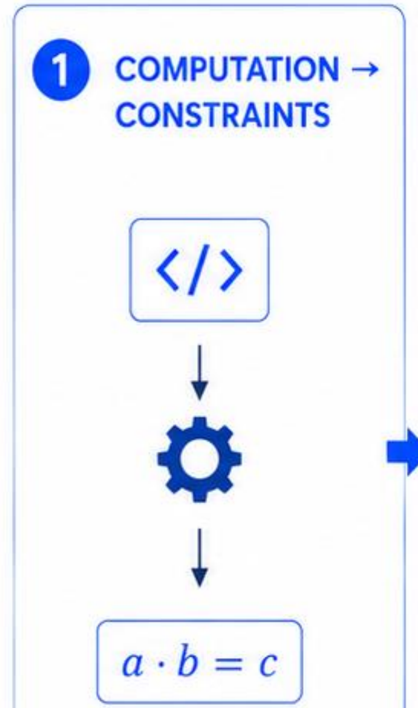
Audits, formal methods, and test coverage take time to mature.



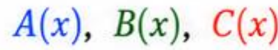
Some improve slowly. Some do not disappear.

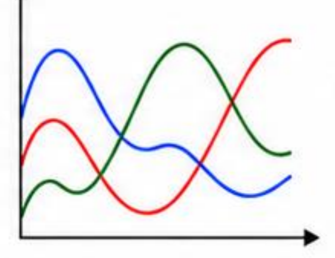
THANK YOU.

SNARK: Complete Process Summary



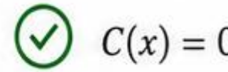
Step	a	b	c
0	a_0	b_0	c_0
1	a_1	b_1	c_1
...	...	...	...
N	a_n	b_n	c_n

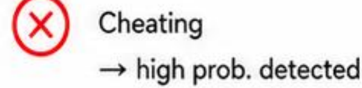


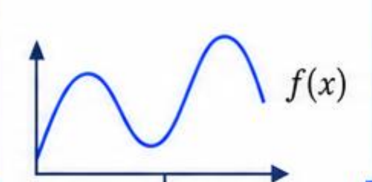


$$A(x) \cdot B(x) = C(x)$$

$$C(x) = a_1 \cdot P_1(x) + a_2 \cdot P_2(x) + \ldots + a_k \cdot P_k(x)$$

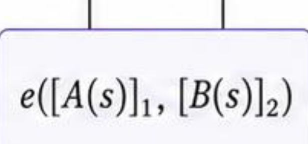






$$[f(s)]_1$$

= 48 bytes



$$e([A(s)]_1, [B(s)]_2) \stackrel{?}{=} e([C(s)]_1, [1]_2)$$

RESULT:

~ 200 byte proof

~ 5ms verification

Independent of computation size

STARKs: From Execution Trace to Succinct Proof

1. EXECUTION TRACE TABLE → HASH ROWS → MERKLE TREE

Step	Reg A	Reg B	Reg C
0	3	5	0
1	3	5	8
2	8	5	13
3	13	5	18
4	18	5	23
5	23	5	28
6	28	5	33
7	33	5	38

hash row 0 → Leaf 0
hash row 1 → Leaf 1
hash row 2 → Leaf 2
hash row 3 → Leaf 3
hash row 4 → Leaf 4
hash row 5 → Leaf 5
hash row 6 → Leaf 6
hash row 7 → Leaf 7

Leaf 0, Leaf 1 → h01
Leaf 2, Leaf 3 → h23
Leaf 4, Leaf 5 → h45
Leaf 6, Leaf 7 → h67
h01, h23 → h0123
h45, h67 → h4567
h0123, h4567 → Merkle Root (32 bytes)

interpolated by polynomial $P_A(x)$

interpolated by polynomial $P_B(x)$

interpolated by polynomial $P_C(x)$

What STARKs Optimize

- Transparent setup
- Hash-based security
- Post-quantum friendly
- Scalable / parallel proving

Tradeoffs

- Larger proofs
- Higher bandwidth
- Slower verification than SNARKs

STARKs replace trusted setup with transparent hash-based commitments.

Different Problems Need Different ZK

Use case	Priority	Natural direction
Rollups	tiny proof + cheap verify	SNARK / aggregation
Identity	privacy + auditability	credentials / SNARKs
Voting	public verifiability	transparent systems
Privacy payments	hidden user data	privacy SNARKs
Long-term infra	PQ safety + no setup	STARK / hash-based
Large systems	compression of many proofs	recursion / folding

Use case determines the **proof-system priorities.**