Blendy 🍹

A Time-Space Tradeoff for the Sumcheck Prover

Alessandro Chiesa
EPFL

Elisabetta Fedele
ETH Zürich

Giacomo Fenzi
EPFL

Andrew Zitek-Estrada
EPFL

Motivation

Where do we see sumcheck?

- zkVMs, lookup arguments, and GKR
- Jolt, Lasso, Gemini, Spartan, Stwo

📈 **recent works aim to minimize prover time!**

… and more

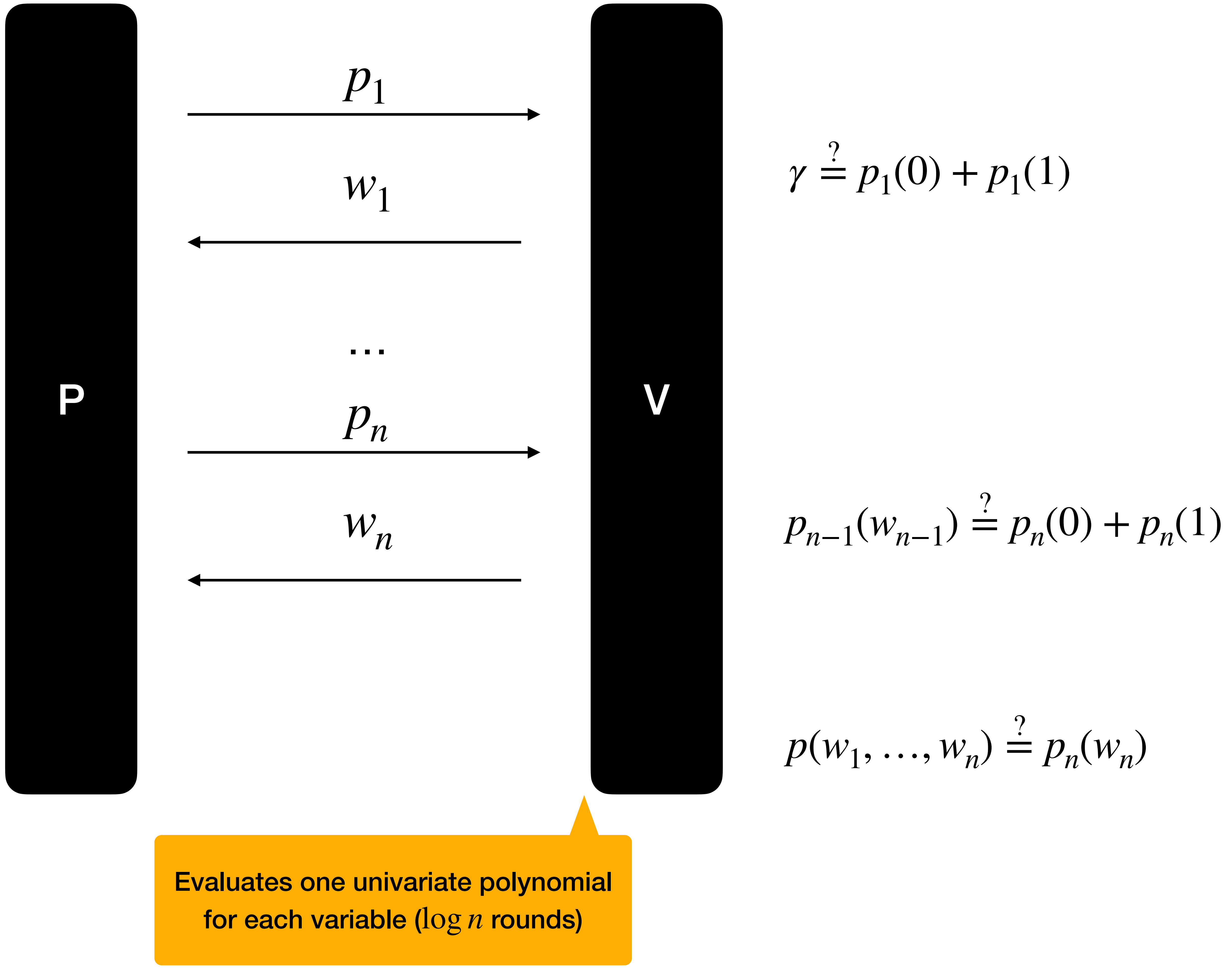
Background

What is Sumcheck?

Ingredients: $p, \mathbb{F}, \gamma, n, H$

Multilinear Polynomial　Field　Claimed Sum　Num Vars　Hypercube

$$\sum_{\boldsymbol{b} \in H^n} p(\boldsymbol{b}) = \gamma$$



LogSpaceSC

Time: $O(N \log N)$ **Space:** $O(\log N)$

Represent p as a stream of evaluations and interpolate

$$p(w_1, w_2, w_3, x_4, x_5, x_6, x_7, x_8)$$

fixed variables

open variables

subsets "fixed" and "open" provide structure to sum using two loops

for all $b_{fixed} \in \{0,1\}^{num\ fixed\ variables}$

$lag\_poly := lagrange\_polynomial(b_{fixed},\ fixed\ variables)$

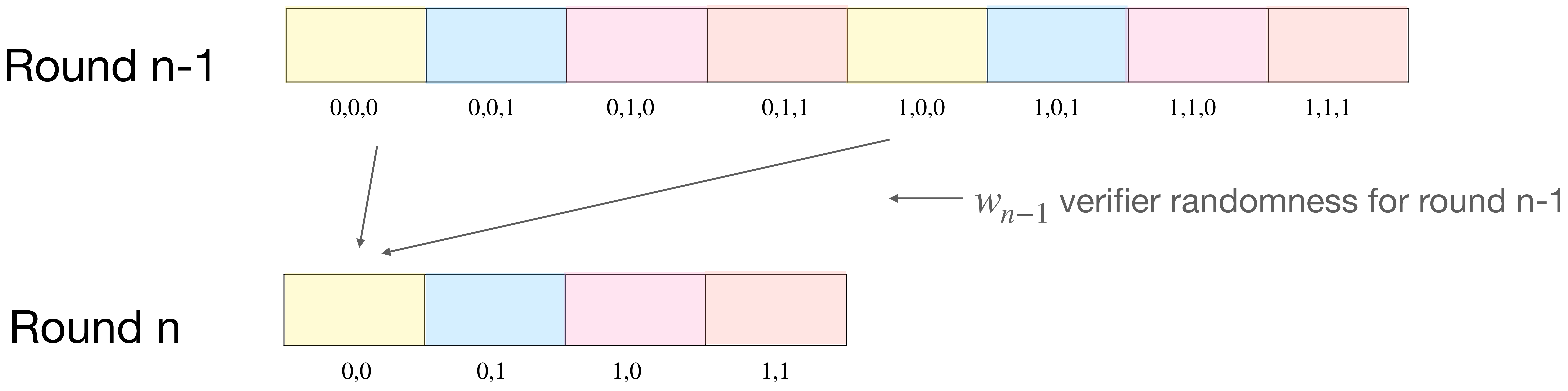
for all $b_{free} \in \{0,1\}^{num\ free\ variables}$

$sum += lag\_poly \cdot p(b_{fixed} \ldots b_{free})$

LinearTimeSC

Time: $O(N)$ **Space:** $O(N)$

Keep a lookup table and absorb randomness in each round



$$eval(0,0) := eval(1,0,0) \cdot w_{n-1} + eval(0,0,0) \cdot \hat{w}_{n-1}$$

Our results

Blendy 🍹

A multilinear Sumcheck Prover with

Time: $O(k \cdot N)$ **Space:** $O(N^{1/k})$

Tradeoffs: The value k regulates time and space efficiency

- $k = 1$ recovers asymptotics of LinearTimeSC
- $k = n$ recovers LogSpaceSC

Other choices of k enable previously unknown tradeoffs

Implementation Notes

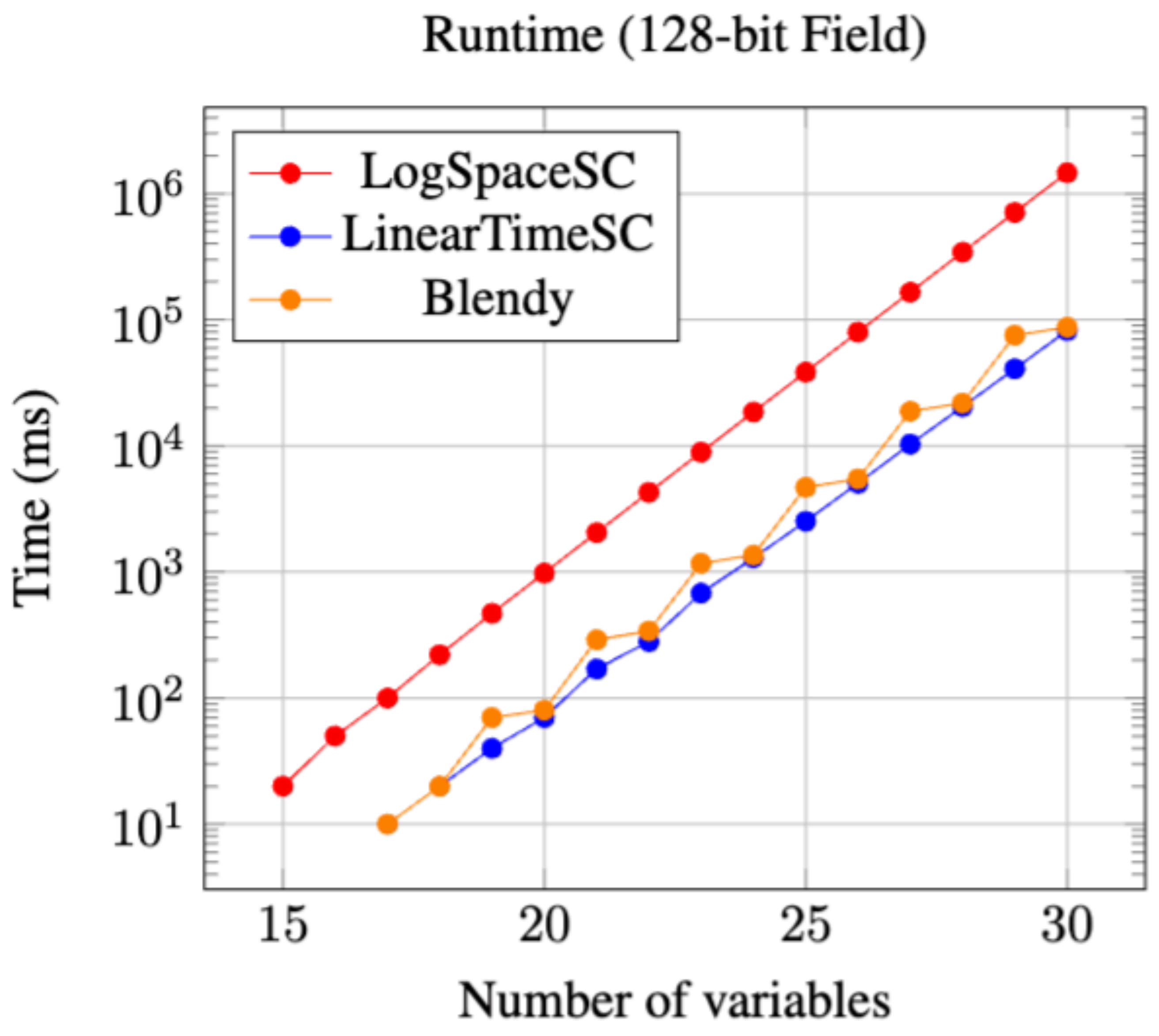
- Rust 🦀 implementation, available at compsec-epfl/space-efficient-sumcheck
- Arkworks ecosystem for underlying finite field arithmetic
- Implemented both LogSpaceSC and LinearTimeSC
- Modular choice of:
  - Field
  - Prover algorithm
  - Input stream

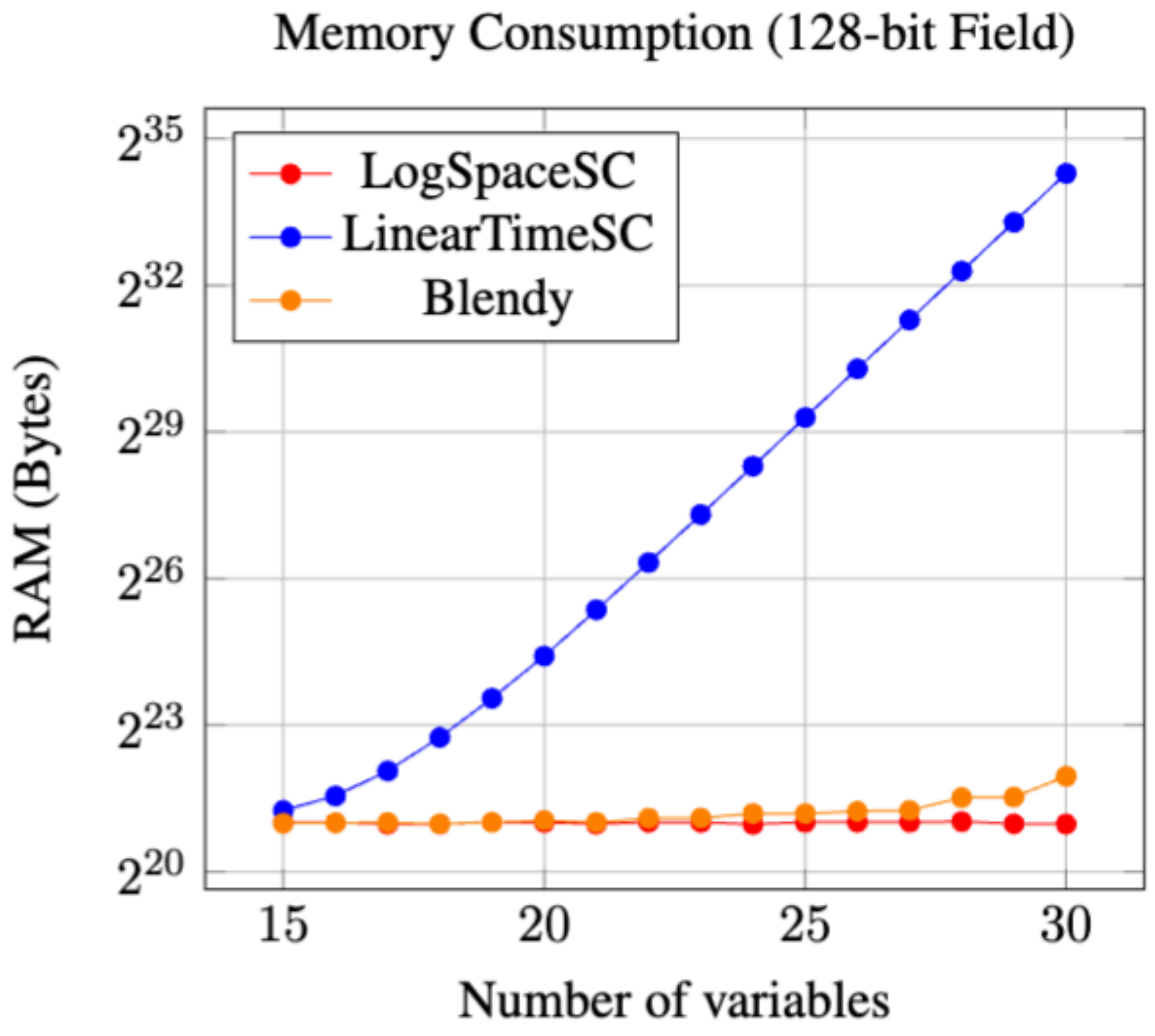
```
impl<'a, F: Field, S: EvaluationStream<F>> Sumcheck<F, S> {
    pub fn prove<P: Prover<'a, F, S>, R: Rng>(prover: &mut P, rng: &mut R) -> Self {
        // Initialize vectors to store prover and verifier messages
        let mut prover_messages: Vec<(F, F)> = Vec::with_capacity(prover.total_rounds());
        let mut verifier_messages: Vec<F> = Vec::with_capacity(prover.total_rounds());
        let mut is_accepted = true;

        // Run the protocol
        let mut verifier_message: Option<F> = None;
        while let Some(message) = prover.next_message(verifier_message) {
            let round_sum = message.0 + message.1;
            let is_round_accepted = match verifier_message {
                // If first round, compare to claimed_sum
                None => round_sum == prover.claimed_sum(),
                // Else compute f(prev_verifier_msg) = prev_sum_0 - (prev_sum_0 - prev_su
                Some(prev_verifier_message) => {
                    verifier_messages.push(prev_verifier_message);
                    let prev_prover_message = prover_messages.last().unwrap();
                    round_sum
                        == prev_prover_message.0
                            - (prev_prover_message.0 - prev_prover_message.1)
                                * prev_verifier_message
```

Results

- Compared to LinearTimeSC: significantly lower **memory consumption** all input sizes
- Compared to LogSpaceSC: orders of magnitude faster

n = 28	LogSpaceSC	Blendy	LinearTimeSC
Time (Seconds)	342.9	21.8	20.4
Memory (MiB)	0.1	1.0	5242.0

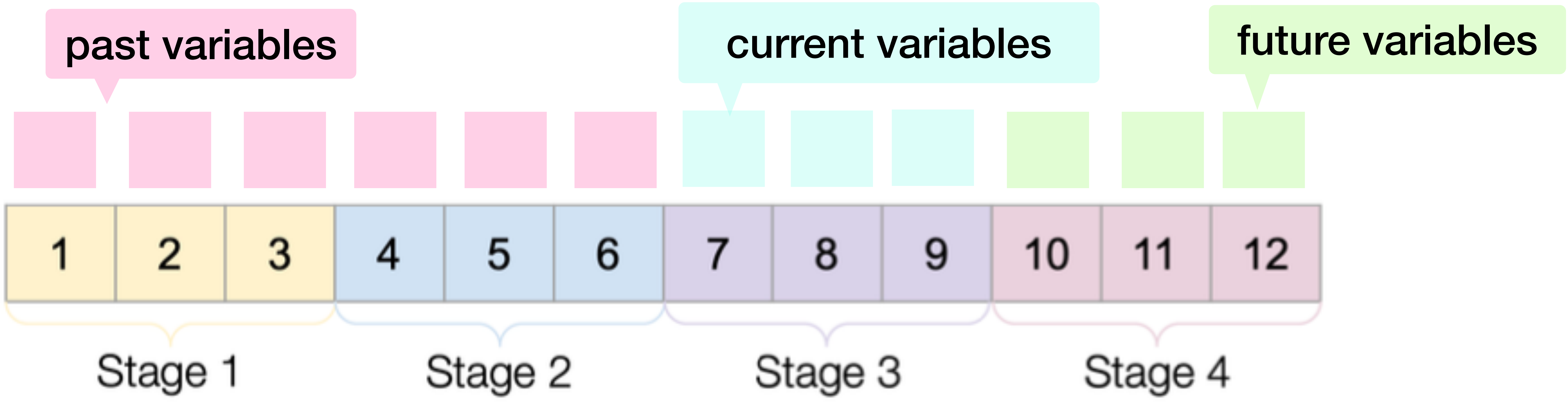




650x improvement in memory usage traded for 0.93x slowdown

Techniques

Main Idea: Divide rounds into k stages



Like in LogSpaceSC we create subsets

for all $b_{past} \in \{0,1\}^{num\ past\ variables}$

$$lag\_poly := lagrange\_polynomial(\,b_{past},\ verifier\ randomness)$$

for all $b_{current} \in \{0,1\}^{num\ current\ variables}$

for all $b_{future} \in \{0,1\}^{num\ future\ variables}$

$$AUX[b_{current}] += lag\_poly \cdot p(b_{past} \ldots b_{current} \ldots b_{future})$$

Like in LinearSpaceSC we fold into a lookup

Partial Sums and Round Evaluation

Partial sums preprocessing

$ps :=$	$AUX[0]$	$AUX[1] + ps[0]$	$AUX[2] + ps[1]$	$AUX[3] + ps[2]$

Round evaluation

$$p_j(\mathsf{X}) = \sum_{\boldsymbol{b}_2^{(\mathrm{s})} \in \{0,1\}^{j'}} \chi_{\boldsymbol{b}_2^{(\mathrm{s})}}\left(\boldsymbol{r}_2^{(\mathrm{s})}, \mathsf{X}\right) \underbrace{\sum_{\boldsymbol{b}_2^{(\mathrm{e})} \in \{0,1\}^{l-j'}} \mathsf{AUX}_{(s)}[\boldsymbol{b}_2^{(\mathrm{s})}, \boldsymbol{b}_2^{(\mathrm{e})}]}_{\mathsf{PS}_{(s)}[\boldsymbol{b}_2^{(\mathrm{s})}, \mathbf{1}] - \mathsf{PS}_{(s)}[\boldsymbol{b}_2^{(\mathrm{s})}, \mathbf{0}]}$$

Sequential Lagrange Polynomial

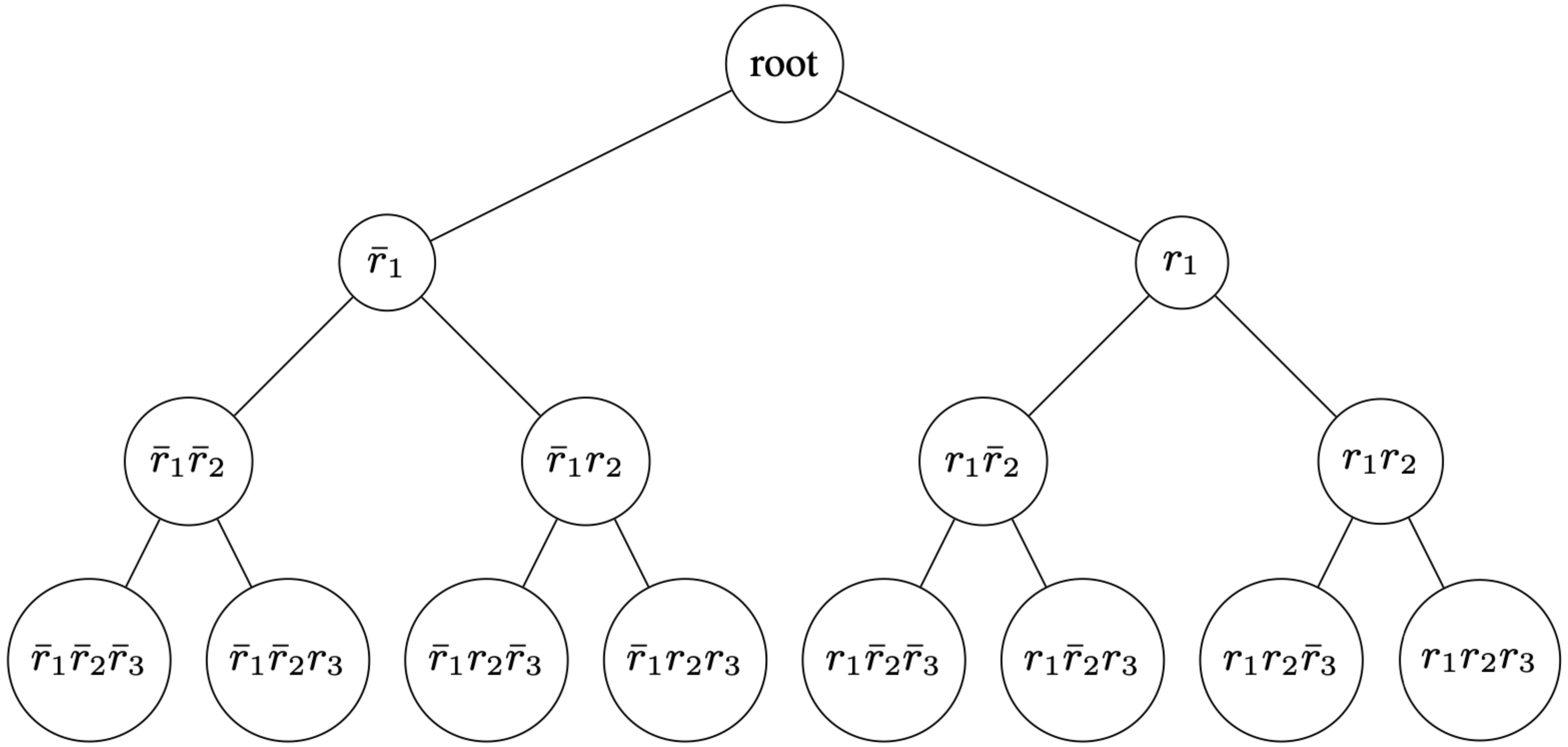
Compute as a path in a DFT

- Left is $(1 - r_i)$, Right r_i
- If traversing to a child: multiply
- If traversing to a parent: divide
- Reach leaf: this is a lag poly

Time complexity reduced:
$O(\ell 2^\ell)$ to $O(2^\ell)$

Multivariate Lagrange polynomial in $\{0,1\}^n$

$$\{\chi_{\boldsymbol{b}}(\mathbf{X}) = \prod_{i\in[n]} \chi_{b_i}(\mathsf{X}_i)\}_{\boldsymbol{b}\in\{0,1\}^n}$$



Siblings are computed with one with 1div, 1 mult

Conclusion

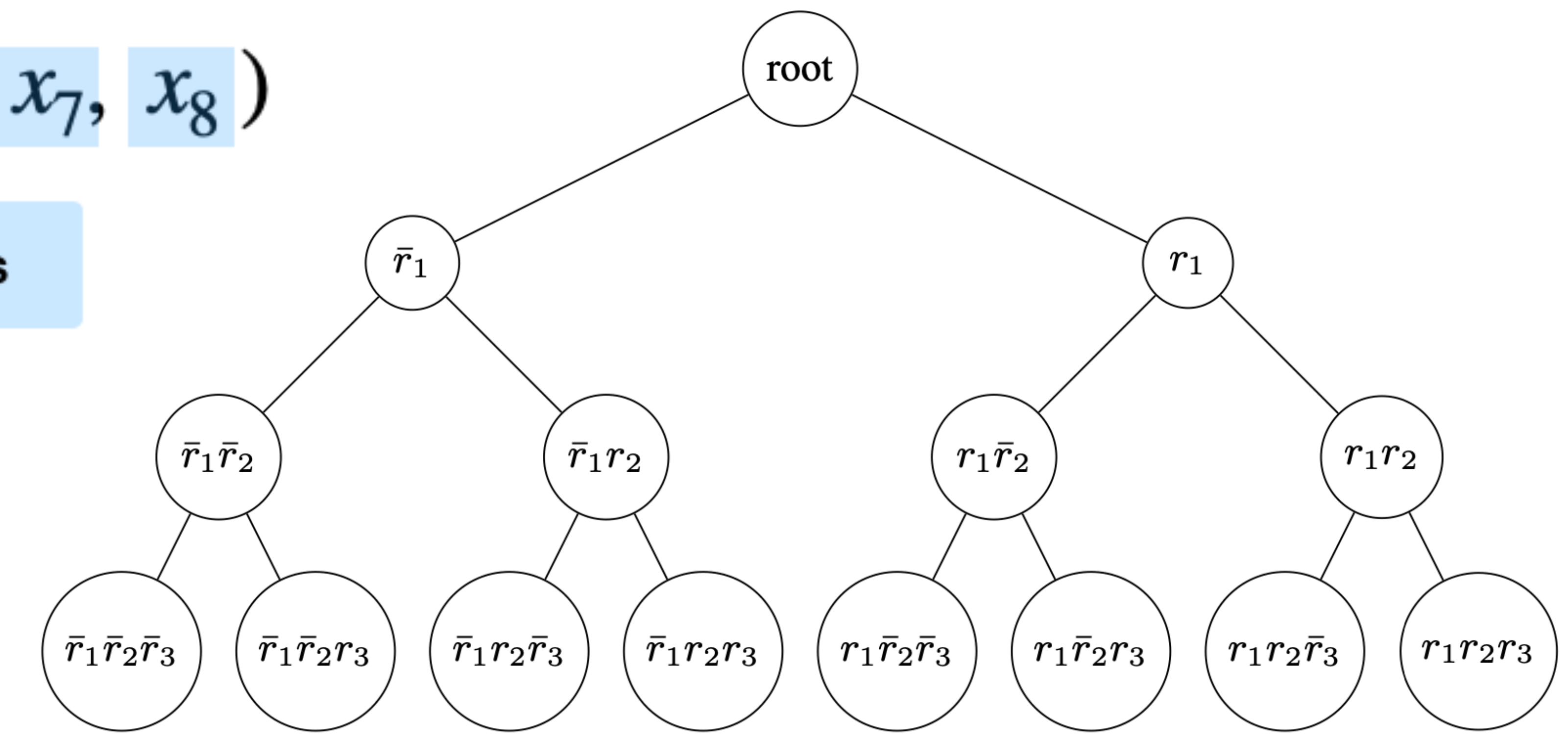
Recap Blendy 🍹

What we showed

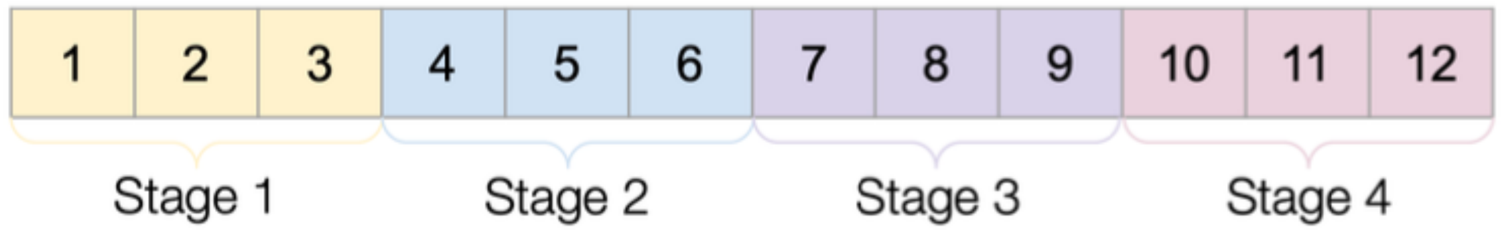
$p(w_1, w_2, w_3, x_4, x_5, x_6, x_7, x_8)$

fixed variables

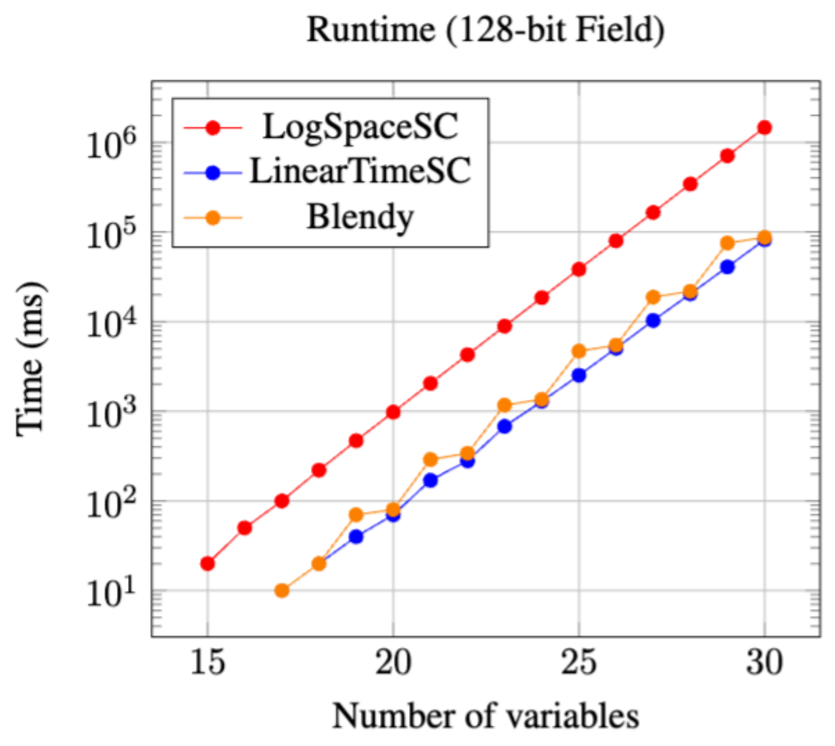
open variables

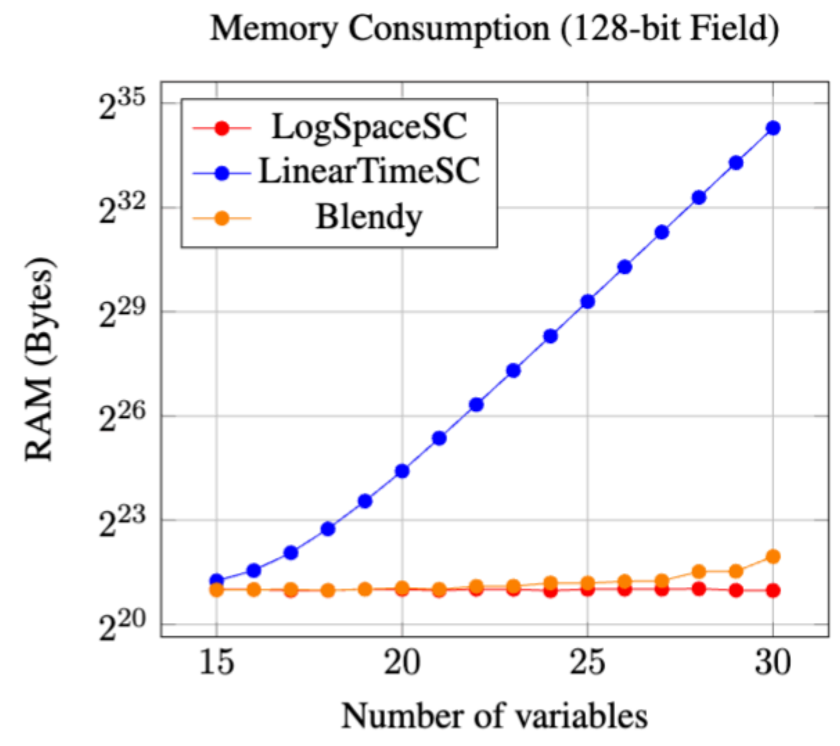


- Divide rounds into k stages



- Computes AUX lookup equal to stage size
- Partial sums technique
- Calculates Lagrange Polynomials sequentially





Time: $O(k \cdot N)$ **Space:** $O(N^{1/k})$

```
impl<'a, F: Field, S: EvaluationStream<F>> Sumcheck<F, S> {
    pub fn prove<P: Prover<'a, F, S>, R: Rng>(prover: &mut P, rng: &mut R) -> Self {
        // Initialize vectors to store prover and verifier messages
        let mut prover_messages: Vec<(F, F)> = Vec::with_capacity(prover.total_rounds());
        let mut verifier_messages: Vec<F> = Vec::with_capacity(prover.total_rounds());
        let mut is_accepted = true;

        // Run the protocol
        let mut verifier_message: Option<F> = None;
        while let Some(message) = prover.next_message(verifier_message) {
            let round_sum = message.0 + message.1;
            let is_round_accepted = match verifier_message {
                // If first round, compare to claimed_sum
                None => round_sum == prover.claimed_sum(),
                // Else compute f(prev_verifier_msg) = prev_sum_0 - (prev_sum_0 - prev_su
                Some(prev_verifier_message) => {
                    verifier_messages.push(prev_verifier_message);
                    let prev_prover_message = prover_messages.last().unwrap();
                    round_sum
                        == prev_prover_message.0
                            - (prev_prover_message.0 - prev_prover_message.1)
                                * prev_verifier_message
```

p_n

*See Lasso [STW23b] appendices F & G for similar techniques

Thank you!

See paper:

ia.cr/2024/524

And blog post:

gfenzi.io/papers/blendy-sumcheck