A Time-Space Tradeoff for the Sumcheck Prover

Alessandro Chiesa
alessandro.chiesa@epfl.ch
EPFL

Elisabetta Fedele
efedele@ethz.ch
ETH Zurich

Giacomo Fenzi
giacomo.fenzi@epfl.ch
EPFL

Andrew Zitek-Estrada
andrew.zitek@epfl.ch
EPFL

April 3, 2024

Abstract

The sumcheck protocol is an interactive protocol for verifying the sum of a low-degree polynomial over a hypercube. This protocol is widely used in practice, where an efficient implementation of the (honest) prover algorithm is paramount. Prior work contributes highly-efficient prover algorithms for the notable special case of multilinear polynomials (and related settings): [CTY11] uses logarithmic space but runs in superlinear time; in contrast, [VSBW13] runs in linear time but uses linear space.

In this short note, we present a family of prover algorithms for the multilinear sumcheck protocol that offer new time-space tradeoffs. In particular, we recover the aforementioned algorithms as special cases. Moreover, we provide an efficient implementation of the new algorithms, and our experiments show that the asymptotics translate into new concrete efficiency tradeoffs.

1 Introduction

The sumcheck protocol [LFKN92] enables a verifier to succinctly check that an n-variate polynomial p over a finite field $\mathbb{F}$ sums to a claimed value γ over the hypercube H^n, that is, to check claims of the form:

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

The sumcheck protocol facilitates central results in the theory of computation, such as the proof of $\mathsf{IP} = \mathsf{PSPACE}$ [Sha92]. Moreover, the sumcheck protocol can be used to construct *concretely efficient* succinct non-interactive arguments of knowledge (SNARKs) (see, e.g., [Set20; GLSTW21; BCHO22; XZS22; CBBZ23; STW23; DP23]). An efficient algorithm of the sumcheck protocol prover is an important ingredient of the aforementioned concretely-efficient SNARKs.

In this note we focus on the case of the *multilinear sumcheck protocol* (the summation polynomial p is multilinear and the summation domain is $\{0,1\}^n$). For this case there are two main prover algorithms:
- [CTY11] runs in quasilinear time $O(N \log N)$ and uses logarithmic space $O(\log N)$; and
- [VSBW13] runs in linear time $O(N)$ and uses linear space $O(N)$.

Above, $N := 2^n$ denotes the number of addends in the sum.

Our result. We present a family of prover algorithms for the multilinear sumcheck protocol that contributes new tradeoffs in time and space.

Theorem 1.1 (Informal)**.** *Let $1 \leq k \leq \log N$ be an integer. There is a prover algorithm for the multilinear sumcheck protocol with time complexity $O(kN)$ and space complexity $O(N^{1/k})$.*

Note that the parameter k regulates a tradeoff between time and space complexity.

We implement and evaluate our algorithm and compare it to the state-of-the-art. Our asymptotic improvements translate into *concrete* efficiency improvements, yielding fast prover algorithms that use much less memory than prior work.

Organization. In Section 2 we recall the sumcheck protocol. In Section 3 we describe the previous prover algorithms for sumcheck. In Section 4 we present our algorithm, which we then analyze in Sections 5 and 6. Finally, we evaluate concretely our algorithm in Section 7.

2 Sumcheck protocol

The sumcheck protocol is an interactive protocol between a prover and a verifier that enables the verifier to check claims of the form $\sum_{\boldsymbol{b} \in H^n} p(\boldsymbol{b}) = \gamma$, where $p \in \mathbb{F}[\mathsf{X}_1, \ldots, \mathsf{X}_n]$ is a polynomial of individual degree at most d. Below is a description of the sumcheck protocol.

Protocol 2.1. The sumcheck protocol to check the claim $\sum_{\boldsymbol{b} \in H^n} p(\boldsymbol{b}) = \gamma$ over a field $\mathbb{F}$ is an interactive protocol between a prover $\mathbf{P}$ and a verifier $\mathbf{V}$. The prover $\mathbf{P}$ receives as input the field $\mathbb{F}$, subset $H \subseteq \mathbb{F}$, number of variables n, and polynomial p. The verifier $\mathbf{V}$ receives as input the field $\mathbb{F}$, the subset H, number of variables n, individual degree d, and claimed sum $\gamma \in \mathbb{F}$; moreover, it receives oracle access to p. The prover $\mathbf{P}$ and verifier $\mathbf{V}$ interact over n rounds as follows.

1. In the first round, $\mathbf{P}$ sends a univariate polynomial $p_1 \in \mathbb{F}^{\leq \mathsf{d}}[\mathsf{X}]$. In the honest case:

$$p_1(\mathsf{X}) := \sum_{\boldsymbol{b} \in H^{n-1}} p(\mathsf{X}, \boldsymbol{b}) \ .$$

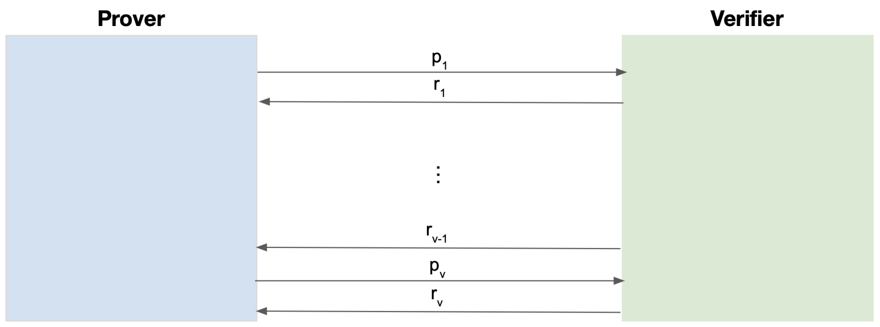
$\mathbf{V}$ checks that $\gamma = \sum_{b \in H} p_1(b)$. Then $\mathbf{V}$ samples and sends $r_1 \leftarrow \mathbb{F}$ to $\mathbf{P}$.

2. For $j \in \{2, \ldots, n-1\}$ in the j-th round $\mathbf{P}$ sends a univariate polynomial $p_j \in \mathbb{F}^{\leq \mathsf{d}}[\mathsf{X}]$. In the honest case:
$$p_j(\mathsf{X}) \coloneqq \sum_{\boldsymbol{b} \in H^{n-j}} p(r_1, \ldots, r_{j-1}, \mathsf{X}, \boldsymbol{b}) \enspace .$$
$\mathbf{V}$ checks that $p_{j-1}(r_{j-1}) = \sum_{b \in H} p_j(b)$. Then $\mathbf{V}$ samples and sends $r_j \leftarrow \mathbb{F}$ to $\mathbf{P}$.

3. In the n-th round, $\mathbf{P}$ sends to $\mathbf{V}$ a univariate polynomial p_n. In the honest case:
$$p_n(\mathsf{X}) = p(r_1, \ldots, r_{n-1}, \mathsf{X}) \enspace .$$
$\mathbf{V}$ checks that $p_{n-1}(r_{n-1}) = \sum_{b \in H} p_n(b)$. Then $\mathbf{V}$ samples a random element $r_n \leftarrow \mathbb{F}$, and checks that $p(r_1, \ldots, r_n) = p_n(r_n)$ using a single query to the polynomial p at $(r_1, \ldots, r_n)$.

The multilinear case. We consider the sumcheck protocol when p is a multilinear polynomial summed over the boolean hypercube: $p \in \mathbb{F}^{\leq 1}[\mathsf{X}_1, \ldots, \mathsf{X}_n]$ and $H = \{0, 1\}$. In this case the polynomial p is uniquely determined by its restriction $f \colon \{0,1\}^n \to \mathbb{F}$ on the boolean hypercube (i.e. $\forall\, \boldsymbol{b} \in \{0,1\}^n : \ f(b) = p(b)$). The prover algorithms that we consider receive f as an *input stream*, interact with the verifier for n rounds, and in round $j \in [n]$ send polynomials p_j and receive randomness r_j. The prover and verifier must agree on a representation of the (linear) polynomials p_j; in this work they are represented via their evaluations on $\{0, 1\}$.

When implemented naively, the running time of the sumcheck prover is $O(N \cdot |p|)$, where $|p|$ denotes the time needed to evaluate the polynomial p at a point. Since p is a multilinear polynomial $|p| = O(N)$, yielding a quadratic cost. The algorithms that we describe next improve on this naive prover time.



3 Previous algorithms

We review the (honest) prover algorithms in [CTY11; VSBW13] for the multilinear sumcheck protocol. Their efficiency is summarized in Table 1, alongside the efficiency of our algorithm.

Below we use Lagrange polynomials over boolean domains, which we recall: the univariate Lagrange polynomials over $\{0, 1\}$ are $\{\chi_b(\mathsf{X}) = b\mathsf{X} + (1-b)(1-\mathsf{X})\}_{b \in \{0,1\}}$ and the multivariate Lagrange polynomials over $\{0,1\}^n$ are $\{\chi_{\boldsymbol{b}}(\mathbf{X}) = \prod_{i \in [n]} \chi_{b_i}(\mathsf{X}_i)\}_{\boldsymbol{b} \in \{0,1\}^n}$.

Linear-time algorithm. The algorithm in [VSBW13] runs in linear time and uses linear space. The algorithm maintains a table during its execution. This table initially has size N and is obtained from a single pass over the input. At each round, the table is updated based on the received randomness, and its size halves.

$\mathsf{LinearTimeSC}^f$:

```
1. For $\boldsymbol{b} \in \{0,1\}^n$, initialize $A^{(0)}[\boldsymbol{b}] := f(\boldsymbol{b})$.
2. For each round $j = 1, 2, \ldots, n-1$:
   (a) Compute $p_j(0)$ and $p_j(1)$ as
         $p_j(0) := \sum_{\boldsymbol{b} \in \{0,1\}^{n-j}} A^{(j-1)}[0, \boldsymbol{b}]$ ,
         $p_j(1) := \sum_{\boldsymbol{b} \in \{0,1\}^{n-j}} A^{(j-1)}[1, \boldsymbol{b}]$ .
   (b) Send $p_j(0), p_j(1)$ to $\mathbf{V}$.
   (c) Receive $r_j$ from $\mathbf{V}$.
   (d) For $\boldsymbol{b} \in \{0,1\}^{n-j}$ compute $A^{(j)}[\boldsymbol{b}]$ as
         $A^{(j)}[\boldsymbol{b}] := A^{(j-1)}[0, \boldsymbol{b}] \cdot \chi_0(r_j) + A^{(j-1)}[1, \boldsymbol{b}] \cdot \chi_1(r_j)$ .
3. Compute $p_n(0)$ and $p_n(1)$ as
         $p_n(0) := A^{(n-1)}[0]$ ,
         $p_n(1) := A^{(n-1)}[1]$ .
4. Send $p_n(0)$ and $p_n(1)$ to $\mathbf{V}$.
5. Receive $r_n$ from $\mathbf{V}$.
```

Logarithmic-space algorithm. The algorithm in [CTY11] uses logarithmic space and runs in quasilinear time. At each round the algorithm performs a linear pass over its input. By leveraging the special structure of Lagrange polynomials on binary inputs, it achieves an improved running time over the naive prover algorithm.

```
LogSpaceSC$^f$:
1. Compute $p_1(0)$ and $p_1(1)$ as:
         $p_1(0) := \sum_{\boldsymbol{b}_3 \in \{0,1\}^{n-1}} f(0, \boldsymbol{b}_3)$.
         $p_1(1) := \sum_{\boldsymbol{b}_3 \in \{0,1\}^{n-1}} f(1, \boldsymbol{b}_3)$.
2. Send $p_1(0)$ and $p_1(1)$ to $\mathbf{V}$.
3. Receive $r_1$ from $\mathbf{V}$.
4. For each round $j = 2, 3, \ldots, n$:
   (a) Initialize $p_j(0) := 0$ and $p_j(1) := 0$.
   (b) For $\boldsymbol{b}_1 \in \{0,1\}^{j-1}$:
         i. Compute $\mathsf{LagPoly} := \chi_{\boldsymbol{b}_1}(r_1, \ldots, r_{j-1})$.
        ii. Update $p_j(0)$ and $p_j(1)$:
              $p_j(0) := p_j(0) + \mathsf{LagPoly} \cdot \sum_{\boldsymbol{b}_3 \in \{0,1\}^{j-1}} f(\boldsymbol{b}_1, 0, \boldsymbol{b}_3)$.
              $p_j(1) := p_j(1) + \mathsf{LagPoly} \cdot \sum_{\boldsymbol{b}_3 \in \{0,1\}^{j-1}} f(\boldsymbol{b}_1, 1, \boldsymbol{b}_3)$.
   (c) Send $p_j(0)$ and $p_j(1)$ to $\mathbf{V}$.
   (d) Receive $r_j$ from $\mathbf{V}$.
```

Algorithm	Time complexity	Space complexity	Additions	Multiplications
$\mathsf{LinearTimeSC}$	$O(N)$	$O(N)$	$3N$	$2N$
$\mathsf{LogSpaceSC}$	$O(N \log N)$	$O(\log N)$	$N \log N$	$N \log N$
$\mathsf{BlendySC}_k$	$O(kN)$	$O(N^{1/k})$	$(k+1)N + 4kN^{1/k}$	$kN + 4kN^{1/k} + 2N^{1-1/k}$

Table 1: Time and space complexities of prover algorithms for the multilinear sumcheck protocol. Field operations ignore low order terms.

4 Our algorithm

We propose a family of prover algorithms for the multilinear sumcheck protocol: $\{\mathsf{BlendySC}_k\}_{k \in [n]}$. The value k regulates the tradeoff between time and space efficiency. Increasing k reduces memory consumption while increasing running time. When $k = 1$ the algorithm recovers the asymptotics of [VSBW13], while when $k = n$ those of [CTY11]. Other choices yield new tradeoffs between time and space efficiency.

Outline. We partition the n rounds of the sumcheck protocol in k stages of length $l := \frac{n}{k}$.[1] At the start of each stage, the prover performs a precomputation that is then used for the rounds belonging to said stage.

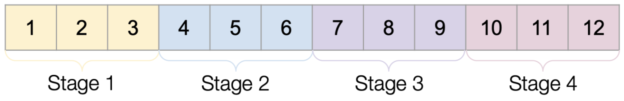


Figure 1: Example of the division of rounds into stages for $n = 12$ and $k = 4$.

Notation. We denote the empty string by ε. Given $\boldsymbol{v} \in \{0,1\}^\ell$ and $a, b \in [\ell]$ with $a \leq b$, we define $\boldsymbol{v}[a : b] := (v_a, \ldots, v_b)$ and $\boldsymbol{v}[: b] := \boldsymbol{v}[1 : b]$. Given $\boldsymbol{b} \in \{0,1\}^n$ and a stage $s \in [k]$ we parse $\boldsymbol{b}$ as $\boldsymbol{b} = (\boldsymbol{b}_1, \boldsymbol{b}_2, \boldsymbol{b}_3)$ where $\boldsymbol{b}_1 \in \{0,1\}^{(s-1)l}$, $\boldsymbol{b}_2 \in \{0,1\}^l$, and $\boldsymbol{b}_3 \in \{0,1\}^{(k-s)l}$. Intuitively, $\boldsymbol{b}_1$ contains the bits related to previous stages, $\boldsymbol{b}_2$ contains the bits related to the current stage, and $\boldsymbol{b}_3$ contains the bits related to future stages. We also divide the verifier randomness $\boldsymbol{r}$ in the same way so that $\boldsymbol{r} = (\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3)$.

Organization. In Section 4.1 we describe how to efficiently perform sequential evaluations of Lagrange polynomials. In Section 4.2 we describe the precomputation performed at the start of each stage. In Section 4.3 we describe the operations performed in each round. In Section 4.4 we present our algorithm.

4.1 Sequential evaluations of Lagrange polynomials in logarithmic space

The prover algorithm computes, as an intermediate step, the evaluations of the Lagrange polynomials at a given point: given $\boldsymbol{r} = (r_1, \ldots, r_\ell) \in \mathbb{F}^\ell$, it computes $\{\chi_{\boldsymbol{b}}(\boldsymbol{r})\}_{\boldsymbol{b} \in \{0,1\}^\ell}$, where $\chi_{\boldsymbol{b}}(\mathbf{X}) = \prod_{i \in [n]} \chi_{b_i}(\mathsf{X}_i)$.

Storing all evaluations requires storing 2^ℓ fields elements, which we wish to avoid. Instead, the algorithm produces the sequence of Lagrange evaluations *sequentially*. The naive approach for this requires time $O(\ell \cdot 2^\ell)$ and space $O(\ell)$. Instead, we describe a method that uses time $O(2^\ell)$ and space $O(\ell)$.

The method has two subroutines: $\mathsf{LagInit}$ receives the evaluation point $\boldsymbol{r}$ outputs an initial state st; and $\mathsf{LagNext}$ receives the state st and outputs an updated state and an evaluation of the Lagrange polynomial at a point of the hypercube.

1. $\mathsf{st} := \mathsf{LagInit}(\ell, \boldsymbol{r})$

[1] If n does not divide k, we instead partition n into k stages of length $l := \left\lfloor \frac{n}{k} \right\rfloor$, and a final stage of length $n - k \cdot l$. In this note, we focus on the case where k divides n, but our implementation also supports the case where k does not divide n.

2. For $\boldsymbol{b} \in \{0,1\}^{\ell}$:
   (a) $(v_{\boldsymbol{b}}, \mathsf{st}) \coloneqq \mathsf{LagNext}(\mathsf{st})$.

We ensure that the total running time is $O(2^{\ell})$, and also ensure that $v_{\boldsymbol{b}} = \chi_{\boldsymbol{b}}(\boldsymbol{r})$ and $|\mathsf{st}| = O(\ell)$.

Initialization. LagInit initializes the state st by computing the Lagrange polynomial $\chi_{0^{\ell}}(\boldsymbol{r})$. The state st consists of the current location in a DFT tree (see below), the evaluation point, and the intermediate values of this computation. Specifically, LagInit outputs

$$\mathsf{st} \coloneqq \left(0^{\ell}, \boldsymbol{r}, \left(\prod_{i \le j} \chi_0(r_i)\right)_{j \in [\ell]}\right) .$$

Update. The invocations of LagNext correspond to a Depth First Traversal (DFT) of a complete binary tree on $\ell + 1$ levels[2] where the nodes at level i contain the values of the evaluations of all the i-variate multilinear Lagrange polynomials at the point $(r_1, \ldots, r_i)$. A visualization of the tree for the case $\ell = 3$ is provided in Figure 2. Specifically, LagNext receives a state of the following form:

$$\mathsf{st} = \left(\boldsymbol{b}, \boldsymbol{r}, \left(\prod_{i \le j} \chi_{\boldsymbol{b}_i}(r_i)\right)_{j \in [\ell]}\right) .$$

Letting $\boldsymbol{b}'$ denote the (binary) labeling of the next leaf in the tree, LagNext returns $\prod_{i \le \ell} \chi_{\boldsymbol{b}_i}(r_i) = \chi_{\boldsymbol{b}}(r_1, \ldots, r_{\ell})$ (which is stored in the state st) and updates the state to

$$\mathsf{st} = \left(\boldsymbol{b}', \boldsymbol{r}, \left(\prod_{i \le j} \chi_{\boldsymbol{b}'_i}(r_i)\right)_{j \in [\ell]}\right) .$$

Since many intermediate values of the Lagrange computation are shared by neighboring nodes in the tree, this enables computing the Lagrange polynomials more efficiently than naively.

Complexity analysis. The method involves a DFT where visiting each new node requires a multiplication. Each node is visited at most once, so the total time complexity of the algorithm is $O(2^{\ell})$. The space complexity of the algorithm is $O(\ell)$ field elements as it requires storing the position of the current node, the cumulative product, and the randomness vector $(r_1, \ldots, r_{\ell})$.

4.2 Precomputation

The algorithm has k stages and, at the beginning of stage $s \in [k]$, the algorithm precomputes an array $\mathsf{PS}_{(s)}$ of size $2^{l} = N^{1/k}$ that is derived as the partial sums of an auxiliary array $\mathsf{AUX}_{(s)}$. Later, in Section 4.3, we show how this precomputation allows computing the evaluations of the sumcheck polynomials in stage s. Here we describe how to compute $\mathsf{AUX}_{(s)}$ and then how to derive $\mathsf{PS}_{(s)} \coloneqq \mathsf{partialSum}(\mathsf{AUX}_{(s)})$.

Computing $\mathsf{AUX}_{(s)}$. If $k = 1$, $\mathsf{AUX}_{(s)}[\boldsymbol{b}] \coloneqq f(\boldsymbol{b})$. Otherwise, $\mathsf{AUX}_{(s)}$ is defined as follows:

$$\forall\, \boldsymbol{b}_2 \in \{0,1\}^{l},\ \mathsf{AUX}_{(s)}[\boldsymbol{b}_2] \coloneqq \sum_{\boldsymbol{b}_1 \in \{0,1\}^{(s-1)l}} \chi_{\boldsymbol{b}_1}(\boldsymbol{r}_1) \sum_{\boldsymbol{b}_3 \in \{0,1\}^{(k-s)l}} f(\boldsymbol{b}_1, \boldsymbol{b}_2, \boldsymbol{b}_3) .$$

We efficiently compute this array with the following procedure.

[2]The levels of the binary tree are indexed from 0 (root) to ℓ (leaves).

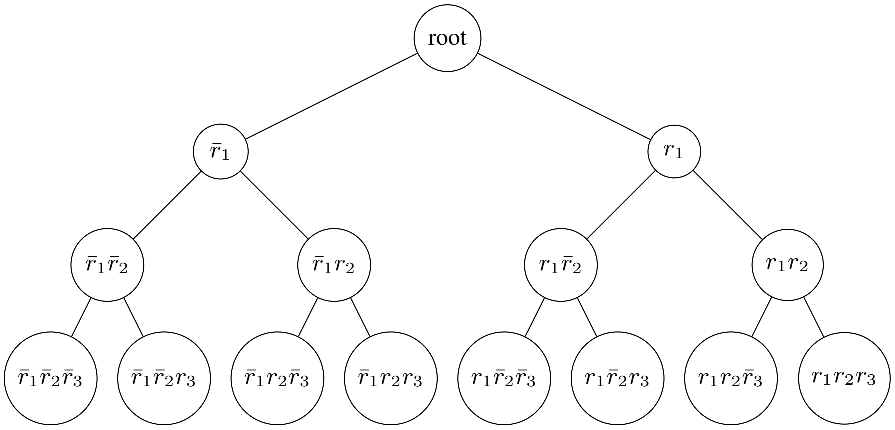


Figure 2: Visualization of the binary tree used for Lagrange polynomials evaluation at the 3rd round given randomness $\boldsymbol{r} = (r_1, r_2, r_3)$. Above, $\bar{r}_i := 1 - r_i$ for $i = 1, 2, 3$.

```
$\mathsf{Aux}^f(\boldsymbol{r}_1)$:
1. If $k = 1$:
    (a) For every $\boldsymbol{b} \in \{0,1\}^n$, set $\mathsf{AUX}_{(s)}[\boldsymbol{b}] := f(\boldsymbol{b})$.
    (b) Return $\mathsf{AUX}_{(s)}$.
2. For every $\boldsymbol{b}_2 \in \{0,1\}^l$, initialize $\mathsf{AUX}_{(s)}[\boldsymbol{b}_2] := 0$.
3. Initialize $\mathsf{st} := \mathsf{LagInit}((s-1)l, \boldsymbol{r}_1)$.
4. For every $\boldsymbol{b}_1 \in \{0,1\}^{(s-1)l}$:
    (a) Compute $(\mathsf{LagPoly}, \mathsf{st}) := \mathsf{LagNext}(\mathsf{st})$.
    (b) For every $\boldsymbol{b}_2 \in \{0,1\}^l$, update $\mathsf{AUX}_{(s)}[\boldsymbol{b}_2]$:
        $$\mathsf{AUX}_{(s)}[\boldsymbol{b}_2] := \mathsf{AUX}_{(s)}[\boldsymbol{b}_2] + \mathsf{LagPoly} \cdot \sum_{\boldsymbol{b}_3 \in \{0,1\}^{(k-s)l}} f(\boldsymbol{b}_1, \boldsymbol{b}_2, \boldsymbol{b}_3)$$
5. Return $\mathsf{AUX}_{(s)}$.
```

The terms $\mathsf{LagPoly}$ are computed efficiently using the method in Section 4.1.

Partial sums. Given an arbitrary array $\boldsymbol{x}$, we write $\boldsymbol{S} := \mathsf{partialSum}(\boldsymbol{x})$ for the array $\boldsymbol{S}$ with $\boldsymbol{S}[-1] = 0$ and $\boldsymbol{S}[i] = \sum_{i=0}^{|\boldsymbol{x}|-1} x_i$. Note that, given $\boldsymbol{S}$, we can express the sum of elements of $\boldsymbol{x}$ between two indices i and j as:

$$x_i + \cdots + x_j = \boldsymbol{S}[j] - \boldsymbol{S}[i-1]$$

Note that computing $\boldsymbol{S}$ from $\boldsymbol{x}$ can be done in a linear pass over $\boldsymbol{x}$ and $O(|\boldsymbol{x}|)$ additions. Once this precomputation is done, each sum of consecutive elements only requires a single subtraction.

4.3 Round computation

Recall that, in round j, the prover aims to compute the evaluations of the polynomial $p_j(\mathsf{X})$ on $\{0,1\}$. We show how, given $\mathsf{PS}_{(s)}$ precomputed at the beginning of the corresponding stage s as in Section 4.2, these evaluations can be computed efficiently *without making additional passes over the input stream*.

Let $j' := j - (s-1)l$ (thus, $j' \in [l]$ is the index of the j-th round in stage s). Write $\boldsymbol{b}_2 = (\boldsymbol{b}_2^{(\mathsf{s})}, \boldsymbol{b}_2^{(\mathsf{e})})$ with $\boldsymbol{b}_2^{(\mathsf{s})} \in \{0,1\}^{j'}$, $\boldsymbol{b}_2^{(\mathsf{e})} \in \{0,1\}^{l-j'}$. Accordingly, write $\boldsymbol{r}_2^{(\mathsf{s})} \in \mathbb{F}^{j'-1}$ for the current randomness. Then,

$$p_j(\mathsf{X}) = \sum_{\boldsymbol{b}_2 \in \{0,1\}^l} \chi_{\boldsymbol{b}_2^{(\mathrm{s})}}\left(\boldsymbol{r}_2^{(\mathrm{s})}, \mathsf{X}\right) \cdot \mathsf{AUX}_{(s)}[\boldsymbol{b}_2] \ .$$

Which can be rewritten as

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

where $\mathbf{1} := 1^{l-j'}$ and $\mathbf{0} := 0^{l-j'}$. The inner sum can be computed in constant time from $\mathsf{PS}_{(s)}$. Moreover, to efficiently compute the terms $\chi_{\boldsymbol{b}_2^{(\mathrm{s})}}\left(\boldsymbol{r}_2^{(\mathrm{s})}\right)$ at each round the prover stores in memory the tree containing all the Lagrange polynomials relative to the $\boldsymbol{r}_2^{(\mathrm{s})}$, updating its leaves round after round after having received new randomness from the verifier. This uses space $O(N^{1/k})$ and can be efficiently updated at round $j' \in [l]$ in $O(2^{j'})$ time. Note that this operation is distinct from that described in Section 4.1, as the size of the tree is small enough that we can afford to completely materialize it into memory. Alternatively, one can also use those same techniques, trading a slightly higher number of field operations for memory savings.

4.4 Blendy algorithm

```
$\mathsf{BlendySC}_k^f$:
1. Set $l := n/k$.
2. For every round $j \in [n]$:
   (a) If $(j-1) \bmod l = 0$:
       i. Set $s := 1 + (j-1)/l$.
       ii. Compute $\mathsf{AUX}_{(s)} := \mathsf{Aux}^f(\boldsymbol{r}_1)$.
       iii. Set $\mathsf{PS}_{(s)} := \mathsf{partialSum}(\mathsf{AUX}_{(s)})$.
       iv. Initialize $\mathsf{LagVector}^{(1)}[\varepsilon] = 1$.
   (b) Let $j' := j - (s-1)l$ and $\boldsymbol{r}_2^{(\mathrm{s})} := \boldsymbol{r}_2[0 : j'-1]$.
   (c) Compute $p_j(0)$ and $p_j(1)$ as
       $$p_j(0) := \sum_{\boldsymbol{b}_2^{(\mathrm{s})} \in \{0,1\}^{j'-1}} \mathsf{LagVector}^{(j')}[\boldsymbol{b}_2^{(\mathrm{s})}] \left(\mathsf{PS}_{(s)}[\boldsymbol{b}_2^{(\mathrm{s})}, 0, \mathbf{1}] - \mathsf{PS}_{(s)}[\boldsymbol{b}_2^{(\mathrm{s})}, 0, \mathbf{0}]\right)$$
       $$p_j(1) := \sum_{\boldsymbol{b}_2^{(\mathrm{s})} \in \{0,1\}^{j'-1}} \mathsf{LagVector}^{(j')}[\boldsymbol{b}_2^{(\mathrm{s})}] \left(\mathsf{PS}_{(s)}[\boldsymbol{b}_2^{(\mathrm{s})}, 1, \mathbf{1}] - \mathsf{PS}_{(s)}[\boldsymbol{b}_2^{(\mathrm{s})}, 1, \mathbf{0}]\right)$$
   (d) Send $p_j(0)$ and $p_j(1)$ to $\mathbf{V}$.
   (e) Receive $r_j$ from $\mathbf{V}$.
   (f) Update the tree of Lagrange polynomials. For each $\boldsymbol{b} \in \{0,1\}^{j'-1}$:
       $$\mathsf{LagVector}^{(j'+1)}[\boldsymbol{b}, 0] := \mathsf{LagVector}^{(j')}[\boldsymbol{b}] \cdot (1 - r_j)$$
       $$\mathsf{LagVector}^{(j'+1)}[\boldsymbol{b}, 1] := \mathsf{LagVector}^{(j')}[\boldsymbol{b}] \cdot r_j$$
```

Note that $\mathsf{BlendySC}$ makes a pass of the input stream f in Item 2(a)ii, and thus makes k input passes in total.

5 Asymptotic efficiency

We analyze the time and space complexity of $\mathsf{BlendySC}_k$. First, we discuss the complexity of computing $\mathsf{PS}_{(s)}$, then that of computing $p_j(\mathsf{X})$ (given $\mathsf{PS}_{(s)}$), and finally the overall complexity.

Computation of $\mathsf{PS}_{(s)}$. At the start of stage s, the prover computes the evaluations of the Lagrange polynomials in $(s-1)l$ variables using the method in Section 4.1, which requires time $O(2^{(s-1)l})$. Additionally, the prover populates the table $\mathsf{AUX}_{(s)}$, which requires additional time $O(N)$.

As this operation is repeated k times (once at the start of each stage), the total time complexity is

$$T(N) = \sum_{s=1}^{k} \Big(O(2^{(s-1)l}) + O(N)\Big) = k \cdot O(N) \ \ .$$

Turning to space complexity, at each step the algorithm stores the tables $\mathsf{AUX}_{(s)}$ (which can be deleted after the computation of $\mathsf{PS}_{(s)}$), and the partial sum table $\mathsf{PS}_{(s)}$ both of size $O(2^l)$, the current value of st, $\mathsf{LagPoly}$, and the randomness $\boldsymbol{r}_1$. The space complexity for this is

$$S(N) = O(2^l) = O(N^{1/k}) \ \ .$$

Computation of $p_j(\mathsf{X})$. Write $j' := j - (s-1) \cdot l$ for the index of the round j in stage s. Note first that the table $\mathsf{LagVector}$ can be updated in time $O(2^{j'})$. Assuming that the table has been computed, p_j can be computed as in Equation (1) in time $O(2^{j'})$, and thus the time complexity of the whole algorithm (excluding the precomputation steps) is:

$$T(N) = \sum_{s=1}^{k} \sum_{j'=1}^{l} O(2^{j'}) = k \cdot O(N^{1/k}) \ \ .$$

The only additional memory used in this portion of the computation is that used to store $\mathsf{LagVector}$, of size $O(2^l)$, thus

$$S(N) = O(2^l) = O(N^{1/k}) \ \ .$$

Overall complexity. The overall time complexity is

$$T(N) = k \cdot O(N) + k \cdot O(N^{1/k}) = k \cdot O(N) \ \ ,$$

and the overall space complexity is

$$S(N) = O(N^{1/k}) + O(N^{1/k}) = O(N^{1/k}) \ \ .$$

Further, the algorithm makes k passes over the input.

6 Number of field operations

We compute the number of field operations performed in $\mathsf{LinearTimeSC}$, $\mathsf{LogSpaceSC}$, and $\mathsf{BlendySC}_k$. We count additions and subtractions jointly, and multiplications separately.

Evaluation of Lagrange polynomials. In all described algorithms, the prover computes Lagrange polynomials, either in their univariate or multivariate form. For univariate polynomials, consider computing $\chi_b\,(r)$

for $b \in \{0,1\}, r \in \mathbb{F}$. As long as $1-r$ is computed previously, this requires no field operations. Thus, we assume that when the prover receives r from the verifier, it computes $1-r$ and stores it. This requires n additions across the whole algorithm. As long as this precomputation is done, computing a *multivariate* Lagrange polynomial of ℓ variables, requires only $\ell - 1$ multiplications.

LinearTimeSC.

- Computing $1 - r_j$ for $j \in [n]$ requires n additions.
- Initializing $A^{(0)}$ requires no field operations.
- Let $j \in [n-1]$. Computing $p_j(0), p_j(1)$ requires $2 \cdot (2^{n-j}-1)$ additions and no multiplications. Computing $A^{(j)}$ from $A^{(j-1)}$ requires 2^{n-j} additions and $2 \cdot 2^{n-j}$ multiplications.
- Computing $p_n(0), p_n(1)$ requires no additions and no multiplications.

In total, the number of additions is:

$$n + \sum_{j \in [n-1]} 2 \cdot (2^{n-j} - 1) + 2^{n-j} = 3N - 2n - 4 \ .$$

And the number of multiplications is:

$$\sum_{j \in [n-1]} 2 \cdot 2^{n-j} = 2N - 2 \ .$$

LogSpaceSC.

- Computing $1 - r_j$ for $j \in [n]$ requires n additions.
- Computing $p_1(0), p_1(1)$ requires $2 \cdot (2^{n-1} - 1)$ additions and no multiplications.
- For $j \in [2, n]$, we perform each following operations 2^{j-1} times:
  - Computing LagPoly, which takes no additions and $(j-1)$ multiplications.
  - Computing $p_j(0), p_j(1)$ which requires $2 \cdot (2^{n-j} - 1)$ additions and 2^{n-j+1} multiplications.

  In total, the number of additions is:

$$n + 2 \cdot (2^n - 1) + \sum_{j=2}^{n} 2^{j-1} \cdot 2 \cdot (2^{n-j} - 1) = (n-2)N + n + 2 \ .$$

  and the number of multiplications is:

$$\sum_{j=2}^{n} 2^{j-1} \cdot 2^{n-j+1} = (n-1)N \ .$$

BlendySC$_k$.

- Computing $1 - r_j$ for $j \in [n]$ requires n additions.
- Stage $s \in [k]$ requires:
  - Computing the Aux function once:
    - Computing $2^{(s-1)l}$ Lagrange polynomials using Section 4.1. This requires $2 \cdot 2^{(s-1)l}$ multiplications and no additions.
    - Updating the $\mathsf{AUX}_{(s)}$ table requires $2^{(s-1)l} \cdot (2^{(k-s+1)l} - 1) = 2^n - 2^{(s-1)l}$ additions and $2^{(s-1)l} \cdot 2^{(k-s+1)l} = 2^n$ multiplications.
    - Computing the partial sum $\mathsf{PS}_{(s)}$ requires a final 2^l additions.
  - For $j' \in [l]$, computing the $j = (s-1)l + j'$ polynomial requires:

- $*$ Update the Lagrange table with the new randomness. This requires at most $2^{j'}$ multiplications and no additions.
- $*$ Computing $p_j(0), p_j(1)$ requires $2 \cdot (2^{j'-1} - 1)$ additions (for the sum), $2^{j'}$ subtractions and $2^{j'}$ multiplications.

In total, the number of additions is

$$\begin{aligned}\sum_{s \in [k]} \left(N - 2^{(s-1)l} + 2^l + \sum_{j' \in [l]} 2 \cdot (2^{j'-1} - 1) + 2^{j'} \right) &\leq (k+1) \cdot N + k \cdot 2^{l+2} \\ &= (k+1)N + 4kN^{1/k} \quad .\end{aligned}$$

and the number of multiplications is

$$\begin{aligned}\sum_{s \in [k]} \left(2 \cdot 2^{(s-1)l} + N + \sum_{j \in [l]} 2 \cdot 2^{j'} \right) &= kN + k \cdot (4 \cdot 2^l - 4) + 2 \cdot \frac{N-1}{2^l - 1} \\ &\leq kN + 4kN^{1/k} + 2\frac{N-1}{N^{1/k} - 1}\end{aligned}$$

Note in particular that when $k = 2$ both $\mathsf{BlendySC}_2$ and $\mathsf{LinearTimeSC}$ have the same leading constant for both number of additions and multiplications. For $k = 1$, the terms $4kN^{1/k}$ would instead contribute a worse constant, and thus we expect that the best running time of $\mathsf{BlendySC}$ is achieved when $k = 2$.

7 Evaluation

We evaluate the performance of $\mathsf{BlendySC}$ compared to $\mathsf{LinearTimeSC}$ and $\mathsf{LogSpaceSC}$. We focus on two metrics: (i) prover time; and (ii) prover memory.

7.1 Implementation

We implemented the three prover algorithms in Rust, by leveraging the `arkworks` ecosystem for developing zkSNARKs [ark]. Our implementation is open sourced at `compsec-epfl/space-efficient-sumcheck` and we plan to upstream it to `arkworks`.

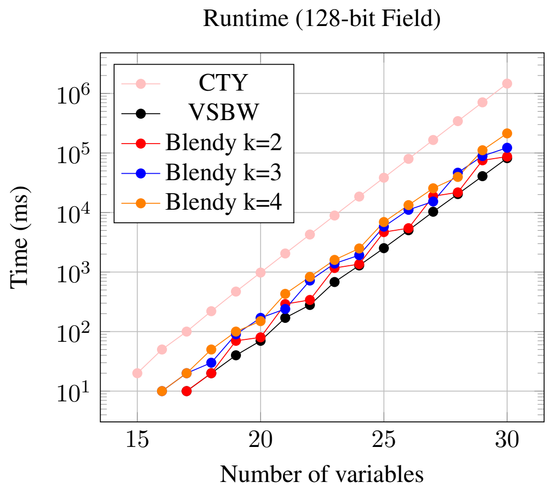
Organization. We expose a common interface for a generic prover algorithm for the multilinear sumcheck protocol, which is then implemented by the three prover algorithms. The prover interface receives as an input stream the evaluation table f of the polynomial, which allows us to accurately measure memory consumption. While $\mathsf{BlendySC}_k$ as described in Section 4.4 assumes that k divides n, our implementation removes this limitation.

Primitives. We use `arkworks` for the underlying finite field arithmetic (provided by the `ark-ff` crate).

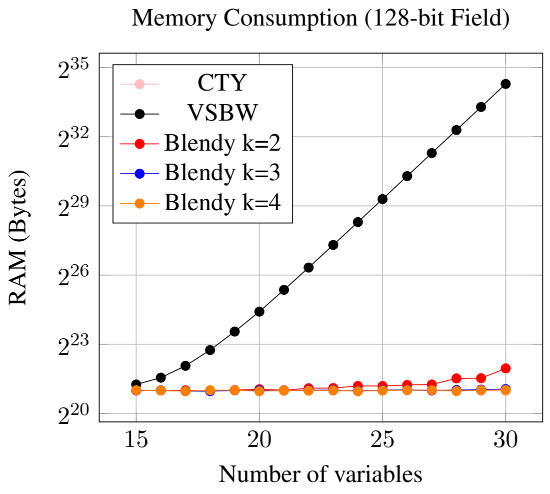
Optimizations. While our implementation has been optimized on a best-effort basis, it should be considered a reference implementation, rather than an optimized one.

7.2 Benchmarks

We run our experiments on an AWS-hosted machine with instance type *m5.8xlarge* with 32 vCPU and 128GiB of memory (Intel Xeon Platinum 8259CL CPU @ 2.50GHz). We measure (i) wall time; and (ii) maximum



(a) Running time of the prover.



(b) Memory consumption of the prover.

Figure 3: Comparison of running time and memory of the prover algorithms considered in this paper. Number of variables ranges from 15 to 30 variables. They y-axis is log scaled.

resident set size, using the *GNU-time* facility. We chose maximum resident set size as a proxy measure to estimate the space complexity of the algorithms we benchmark. Our methodology is as follows: we select an instance size by choosing the number of variables $n \in \{15, \ldots, 30\}$. Recall that then the instance size is $N = 2^n$. For each n, we collect both the wall time and the peak memory consumption from a single process that instantiates one prover of the chosen type. Since we observe that a Rust (1.74.1) binary requests a baseline amount of memory that is approximately 2 MiB, our results are then offset by this amount.

7.3 Results

In Figure 3, we compare running time and memory consumption across our implementations of prover algorithms. We also provide the raw data in Table 2.

Discussion.

- The asymptotic improvement in space of BlendySC translates in significantly lower memory consumption than LinearTimeSC across all instances that we tested. For $n = 24$, LinearTimeSC consumes 0.3 GiB of RAM and BlendySC 0.4 MiB. For $n = 28$, LinearTimeSC consumes 5.2 GiB of RAM and BlendySC 1 MiB.

- LinearTimeSC and BlendySC$_k$ have similar running times, and are order of magnitudes faster than LogSpaceSC. Especially when $k = 2$, the BlendySC algorithm performs similarly to LinearTimeSC, as it was suggested in Section 6. For $n = 24$, LinearTimeSC runs in 1.3s and BlendySC 1.4s. For $n = 28$, LinearTimeSC runs in 20.4s and BlendySC 21.8s.

	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Runtime (Seconds)																
LogSpaceSC	0.02	0.05	0.1	0.2	0.5	1.0	2.1	4.3	8.9	18.5	38.4	79.7	165.2	342.9	708.5	1464.1
LinearTimeSC	0.0	0.0	0.01	0.02	0.04	0.1	0.2	0.3	0.7	1.3	2.5	5.0	10.3	20.4	40.8	81.8
BlendySC$_2$	0.0	0.0	0.01	0.02	0.07	0.08	0.3	0.3	1.2	1.4	4.7	5.5	18.8	21.8	75.3	87.0
BlendySC$_3$	0.0	0.01	0.02	0.03	0.09	0.2	0.2	0.7	1.4	1.9	5.8	11.1	15.3	46.6	88.9	122.3
BlendySC$_4$	0.0	0.01	0.02	0.05	0.1	0.15	0.4	0.8	1.6	2.5	6.9	13.3	25.5	39.6	111.2	213.3
Memory Consumption (MiB)																
LogSpaceSC	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.0	0.1	0.1	0.1	0.1	0.1	0.1
LinearTimeSC	0.5	1.1	2.4	5.0	10.2	20.3	41	82	164	328	655	1310	2621	5242	10485	20971
BlendySC$_2$	0.1	0.1	0.1	0.1	0.1	0.2	0.1	0.2	0.2	0.4	0.4	0.5	0.5	1.0	1.0	2.0
BlendySC$_3$	0.1	0.1	0.0	0.0	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.2
BlendySC$_4$	0.1	0.1	0.0	0.1	0.1	0.0	0.1	0.1	0.1	0.0	0.1	0.1	0.1	0.0	0.1	0.1

Table 2: Comparison of runtime and memory consumption of prover algorithms using a 128-bit field for input sizes ranging from 15 to 30 variables.

References

[BCHO22] Jonathan Bootle, Alessandro Chiesa, Yuncong Hu, and Michele Orrù. “Gemini: Elastic SNARKs for Diverse Environments”. In: *Proceedings of the 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT ’22. 2022, pp. 427–457.

[CBBZ23] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. “HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates”. In: *Proceedings of the 42nd Annual International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT ’23. 2023, pp. 499–530.

[CTY11] Graham Cormode, Justin Thaler, and Ke Yi. “Verifying computations with streaming interactive proofs”. In: *Proceedings of the VLDB Endowment* 5.1 (2011), pp. 25–36.

[DP23] Benjamin Diamond and Jim Posen. “Succinct Arguments over Towers of Binary Fields”. In: *IACR Cryptol. ePrint Arch.* (2023), p. 1784. URL: `https://eprint.iacr.org/2023/1784`.

[GLSTW21] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad Wahby. *Brakedown: Linear-time and post-quantum SNARKs for R1CS*. Cryptology ePrint Archive, Report 2021/1043. 2021.

[LFKN92] Carsten Lund, Lance Fortnow, Howard J. Karloff, and Noam Nisan. “Algebraic Methods for Interactive Proof Systems”. In: *Journal of the ACM* 39.4 (1992), pp. 859–868.

[STW23] Srinath Setty, Justin Thaler, and Riad Wahby. “Unlocking the lookup singularity with Lasso”. In: *IACR Cryptol. ePrint Arch.* (2023), p. 1216. URL: `https://eprint.iacr.org/2023/1216`.

[Set20] Srinath Setty. “Spartan: Efficient and general-purpose zkSNARKs without trusted setup”. In: *Proceedings of the 40th Annual International Cryptology Conference*. CRYPTO ’20. Referencing Cryptology ePrint Archive, Report 2019/550, revision from 2020.02.28. 2020, pp. 704–737.

[Sha92] Adi Shamir. “IP = PSPACE”. In: *Journal of the ACM* 39.4 (1992), pp. 869–877.

[VSBW13] Victor Vu, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. “A hybrid architecture for interactive verifiable computation”. In: *Proceedings of the 34th IEEE Symposium on Security and Privacy*. Oakland ’13. 2013, pp. 223–237.

[XZS22] Tiancheng Xie, Yupeng Zhang, and Dawn Song. “Orion: Zero Knowledge Proof with Linear Prover Time”. In: *Proceedings of the 42nd Annual International Cryptology Conference*. CRYPTO ’22. 2022, pp. 299–328.

[ark] arkworks contributors. *`arkworks` zkSNARK ecosystem*. 2022. URL: `https://arkworks.rs`.