20 GKR Protocol

20.1 Motivation

Previous discussion, while giving certain applications of the Sum-Check protocol, still provides quite "artificial" constructions not directly applicable in the real-world use-cases. For instance, the #SAT problem is not very practical as the prover $\mathcal{P}$ cannot even compute the value H in polynomial time. Ideally, we would like to have a protocol that allows the prover to compute the value H is the polynomial time, while the verifier $\mathcal{V}$ should be able to verify the correctness of the claimed value H in the logarithmic time.

Goldwasser, Kalai, and Rothblum (GKR) described a protocol which solves exactly this issue over the arithmetical circuits, which we solved using QAP→NILP reduction in Groth16. Here we take the Sum-Check approach.

Suppose we are given the *layered* arithmetical circuit $\mathtt{C} : \mathbb{F}^n \to \mathbb{F}^m$ of size S (number of gates). The *layered* here means that the circuit $\mathtt{C}$ can be decomposed into d layers (note that GKR can be generalized to the unstructured arithmetical circuits as well). The GKR protocol allows to achieve the following performance:

- The communication consists of $\mathcal{O}(d \cdot \mathsf{polylog}(S))$ field elements.
- The verifier runs in $\mathcal{O}(n + d \cdot \mathsf{polylog}(S))$ time.
- The prover runs in $\mathcal{O}(\mathsf{poly}(S))$ time.
- The soundness error is just $\mathcal{O}(d \log(S)/|\mathbb{F}|)$.

20.2 Protocol Description

20.2.1 Circuit Representation

Again, assume we are given the circuit $\mathtt{C} : \mathbb{F}^n \to \mathbb{F}^m$ of size S, depth d, and fan-in two (i.e., each gate has at most two inputs, but might have multiple outputs). Let us number the layers of the circuit from 0 to d where 0 denotes the output layer while d is the input layer (notice that the layers are numbered in the reverse order compared to the usual circuit notation). Assume the number of gates in the layer i is S_i and without the loss of generality, we assume that S_i is the power of two: $S_i = 2^{v_i}$. Now, we are going to introduce certain functions to further build up the Sum-Check.

Gates Encodings. Suppose $W_i : \{0,1\}^{v_i} \to \mathbb{F}$ is structured so that it outputs the value of the i-th layer gate given the gate label. As usual, we denote the multilinear extension of W_i as $\widetilde{W}_i : \mathbb{F}^{v_i} \to \mathbb{F}$.

Wiring Predicates. We introduce the wiring predicates $\mathsf{in}_{1,i}, \mathsf{in}_{2,i} : \{0,1\}^{v_i} \to \{0,1\}^{v_{i+1}}$ which indicate which pairs of wairing are connected to the i-th layer gate from the layer $i+1$. It takes the label of the gate in the layer i (say, a) and the function $\mathsf{in}_{1,i}(a)$ outputs the label of the first input connected to the gate a in the layer $i+1$, while $\mathsf{in}_{2,i}(a)$ outputs the label of the second input connected to the gate a in the layer $i+1$.

Operations Encodings. Define two functions: $\mathsf{add}, \mathsf{mul} : \{0,1\}^{v_i+2v_{i+1}} \to \{0,1\}$ which take three gate labels (say, (a,b,c)) and return 1 if and only if $(b,c) = (\mathsf{in}_{1,i}(a), \mathsf{in}_{2,i}(a))$ (that is, the gate a in the layer i is connected to the gates b and c in the layer $i+1$) and a is an addition gate. Similarly, $\mathsf{mul}(a,b,c)$ returns 1 if and only if $(b,c) = (\mathsf{in}_{1,i}(a), \mathsf{in}_{2,i}(a))$ and a is a multiplication gate. We denote the MLEs of these functions as $\widetilde{\mathsf{add}}$ and $\widetilde{\mathsf{mul}} : \mathbb{F}^{v_i+2v_{i+1}} \to \mathbb{F}$, respectively.

Example. Consider the following circuit with $d = 3$ layers:

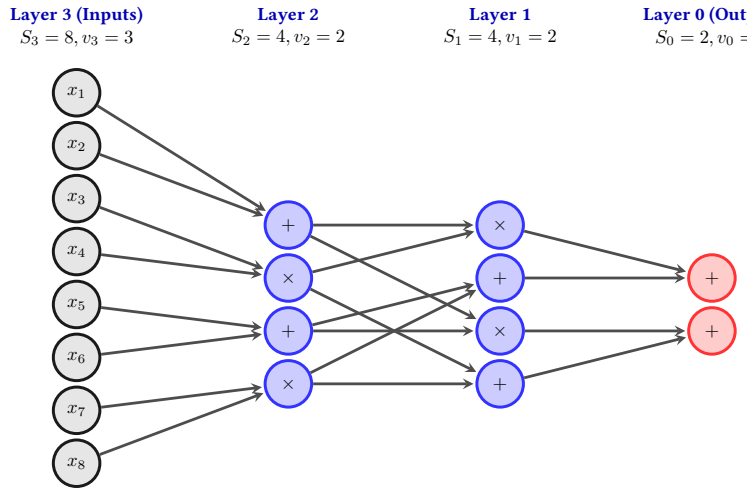

Figure 20.1: Example of a layered arithmetical circuit $\mathtt{C} : \mathbb{F}^8 \to \mathbb{F}^2$ with $d = 3$ layers.

Let us assume that we have inputted the values $\mathbf{x}^{\langle 3\rangle} = (1,2,0,1,0,2,0,1)$. Then, the value of the layer 2 is $\mathbf{x}^{\langle 2\rangle} = (3,0,2,0)$ and the layer 1 is $\mathbf{x}^{\langle 1\rangle} = (0,2,6,0)$. Finally, the output layer is $\mathbf{x}^{\langle 0\rangle} = (2,6)$. Let us now see how to encode gates, wiring predicates, and operations encodings on this circuit. Consider $W_1 : \{0,1\}^2 \to \mathbb{F}$. We have

$$W_1(0,0) = 0, \quad W_1(0,1) = 2, \quad W_1(1,0) = 6, \quad W_1(1,1) = 0.$$

Therefore, the multilinear extension of W_1 is $\widetilde{W}_1(X_1, X_2) = 2(1-X_1)X_2 + 6X_1(1-X_2)$.

Now, let us define the wiring predicates $\mathsf{in}_{1,2}, \mathsf{in}_{2,2} : \{0,1\}^2 \to \{0,1\}^3$. Consider the 2nd gate ($2 = (1,0)$) in layer 2. It is connected to the 5th and 6th gates in layer 3 (the input layer). Since $5 = (1,0,1)$ and $6 = (1,1,0)$ in the binary form, we have

$$\mathsf{in}_{1,2}(1,0) = (1,0,1), \quad \mathsf{in}_{2,2}(1,0) = (1,1,0).$$

Now, consider the operation encodings add_2 and $\mathsf{mul}_2 : \{0,1\}^8 \to \{0,1\}$. The addition gate is non-zero only on the following inputs:

$$((0,0),(0,0,0),(0,0,1)), \quad ((1,0),(1,0,0),(1,0,1)).$$

Therefore, the multilinear extension of add_2 is

$$\begin{aligned}\widetilde{\mathsf{add}}_2(X_1,X_2,Y_1,Y_2,Y_3,Z_1,Z_2,Z_3) = &(1-X_1)(1-X_2)(1-Y_1)(1-Y_2)\\ &\cdot(1-Y_3)(1-Z_1)(1-Z_2)Z_3\\ &+ X_1(1-X_2)Y_1(1-Y_2)(1-Y_3)Z_1(1-Z_2)Z_3\end{aligned}$$

Similarly, the multiplication encoding mul_2 is non-zero only on the following inputs:

$$((0,1),(0,1,0),(0,1,1)), \quad ((1,1),(1,1,0),(1,1,1)),$$

The multilinear extension of mul_2 is

$$\begin{aligned}\widetilde{\mathsf{mul}}_2(X_1,X_2,Y_1,Y_2,Y_3,Z_1,Z_2,Z_3) = &(1-X_1)X_2(1-Y_1)Y_2(1-Y_3)(1-Z_1)Z_2Z_3\\ &+ X_1X_2Y_1Y_2(1-Y_3)Z_1Z_2Z_3.\end{aligned}$$

Remark. Note that the operations encodings add_i and mul_i (and thus MLEs $\widetilde{\mathsf{add}}_i$ and $\widetilde{\mathsf{mul}}_i$) do not depend on the solution witness $\{\mathbf{x}^{\langle i\rangle}\}_{i\in[d+1]}$, while the gates encodings W_i do depend.

20.2.2 Protocol Specification

The GKR protocol consists of d rounds, one for each layer of the circuit. In the i-th round, the prover $\mathcal{P}$ claims a value for $\widetilde{W}_i(\mathbf{r}_i)$ at the randomly selected point $\mathbf{r}_i \leftarrow\!\!\$\ \mathbb{F}^{v_i}$.

At the start of the 1st round, this claim is made about the circuit outputs. Namely, if we have $S_0 = 2^{v_0}$ gates in the output layer, let $D : \{0,1\}^{v_0} \to \mathbb{F}$ denote the function which maps the gate label to the claimed output value, sent by the prover $\mathcal{P}$. The verifier picks a random point $\mathbf{r}_0 \leftarrow\!\!\$\ \mathbb{F}^{v_0}$ and evaluates $\widetilde{D}(r_0)$ in time $\mathcal{O}(S_0)$. By the Schwartz-Zippel lemma, if $\widetilde{D}(\mathbf{r}_0) = \widetilde{W}_0(\mathbf{r}_0)$, then with the soundness error of $v_0/|\mathbb{F}|^{v_0}$ the verifier is assured that these two polynomials are indeed equal. However, for obvious reasons, the verifier cannot compute $\widetilde{W}_0(\mathbf{r}_0)$ without the prover's help. So the core idea of GKR is to reduce the claim about the value of $\widetilde{W}_i(\mathbf{r}_i)$ to a claim about the value of $\widetilde{W}_{i+1}(\mathbf{r}_{i+1})$ for some randomly chosen $\mathbf{r}_{i+1} \leftarrow\!\!\$\ \mathbb{F}^{v_{i+1}}$. How? Consider the following lemma.

Lemma 20.1. The following statement holds:

$$\widetilde{W}_i(\mathbf{z}) = \sum_{\boldsymbol{b},\boldsymbol{c}\in\{0,1\}^{v_{i+1}}} \Big(\widetilde{\mathsf{add}}_i(\mathbf{z},\boldsymbol{a},\boldsymbol{b})(\widetilde{W}_{i+1}(\boldsymbol{b}) + \widetilde{W}_{i+1}(\boldsymbol{c})) + \widetilde{\mathsf{mul}}_i(\mathbf{z},\boldsymbol{b},\boldsymbol{c})\widetilde{W}_{i+1}(\boldsymbol{b})\widetilde{W}_{i+1}(\boldsymbol{c})\Big).$$

Proof. Since both sides are multilinear polynomials, it suffices to check that the equality holds for all boolean assignments $\mathbf{z} \in \{0,1\}^{v_i}$. Fix some concrete $\mathbf{z}_0 \in \{0,1\}^{v_i}$ and without the loss of generality, assume that $\mathbf{z}_0$ is the addition gate in the layer i (the case of multiplication gate is similar). In such case, all the terms $\widetilde{\mathsf{mul}}_i(\mathbf{z},\boldsymbol{b},\boldsymbol{c})$ will be zero, so we can reduce the sum down to:

$$\widetilde{W}_i(\mathbf{z}_0) = \sum_{\boldsymbol{b},\boldsymbol{c}\in\{0,1\}^{v_{i+1}}} \widetilde{\mathsf{add}}_i(\mathbf{z}_0,\boldsymbol{a},\boldsymbol{b})(\widetilde{W}_{i+1}(\boldsymbol{b}) + \widetilde{W}_{i+1}(\boldsymbol{c}))$$

Now, according to the definition of the $\widetilde{\mathsf{add}}_i$ predicate, the $\widetilde{\mathsf{add}}_i(\mathbf{z}_0,\boldsymbol{b},\boldsymbol{c})$ is non-zero (equals to 1) only if $(\boldsymbol{b},\boldsymbol{c}) = (\mathsf{in}_{1,i}(\mathbf{z}_0), \mathsf{in}_{2,i}(\mathbf{z}_0))$. Since we know that $\mathbf{z}_0$ is an addition gate, we can conclude that two such gates $\boldsymbol{b}$ and $\boldsymbol{c}$ exist. Therefore, we can rewrite the sum as:

$$\widetilde{W}_{i+1}(\mathsf{in}_{1,i}(\mathbf{z}_0)) + \widetilde{W}_{i+1}(\mathsf{in}_{2,i}(\mathbf{z}_0)) = \widetilde{W}_i(\mathbf{z}_0).$$

That said, to check the $\mathcal{P}$'s claim about $\widetilde{W}_i(\mathbf{r}_i)$, the verifier can apply the Sum-Check protocol on the function

$$f_i(\boldsymbol{b},\boldsymbol{c};\mathbf{r}_i) = \widetilde{\mathsf{add}}_i(\mathbf{r}_i,\boldsymbol{b},\boldsymbol{c})(\widetilde{W}_{i+1}(\boldsymbol{b})+\widetilde{W}_{i+1}(\boldsymbol{c}))+\widetilde{\mathsf{mul}}_i(\mathbf{r}_i,\boldsymbol{b},\boldsymbol{c})\widetilde{W}_{i+1}(\boldsymbol{b})\widetilde{W}_{i+1}(\boldsymbol{c}).$$

However, note that *the verifier $\mathcal{V}$ does not know the polynomial $\widetilde{W}_{i+1}$*. However, note that he does not need to know it until the last round, during which $\mathcal{V}$ has to make the oracle request $\mathcal{O}^{f_i}$ to compute the value of f_i at the randomly selected point $(\boldsymbol{b}^*,\boldsymbol{c}^*) \leftarrow\!\!\$\ \mathbb{F}^{2v_{i+1}}$. Evaluating $f_i(\boldsymbol{b}^*,\boldsymbol{c}^*;\mathbf{r}_i)$ requires evaluating $\widetilde{\mathsf{add}}_i(\mathbf{r}_i,\boldsymbol{b}^*,\boldsymbol{c}^*)$, $\widetilde{\mathsf{mul}}_i(\mathbf{r}_i,\boldsymbol{b}^*,\boldsymbol{c}^*)$, and $\widetilde{W}_{i+1}(\boldsymbol{b}^*)$ with $\widetilde{W}_{i+1}(\boldsymbol{c}^*)$. Note that evaluating the first two terms can be done by the verifier $\mathcal{V}$ in $\mathcal{O}(\mathsf{poly}(v_i, v_{i+1}))$. However, evaluating the last two terms requires the prover $\mathcal{P}$ to assist the verifier. So we are going to do the following: the prover $\mathcal{P}$ sends the values $z_b = \widetilde{W}_{i+1}(\boldsymbol{b}^*)$ and $z_c = \widetilde{W}_{i+1}(\boldsymbol{c}^*)$ to the verifier $\mathcal{V}$. Now the verifier has to check both these values are correct. However, here is the issue: during the sum-check protocol, the verifier can only use a single random point $\mathbf{r}_{i+1} \in \mathbb{F}^{v_{i+1}}$. So how do we check both conditions using one random point? Here is the trick.

Proposition 20.2. Let $\ell : \mathbb{F} \to \mathbb{F}^{v_{i+1}}$ be the line such that $\ell(0) = \boldsymbol{b}^*$ and $\ell(1) = \boldsymbol{c}^*$. Then, the prover $\mathcal{P}$ sends the univariate polynomial $q(X)$ claimed to be equal to $\widetilde{W}_{i+1} \circ \ell$ — the restriction of $\widetilde{W}_{i+1}$ to the line ℓ. $\mathcal{V}$ checks

whether indeed $\ell(0) = z_b$ and $\ell(1) = z_c$, then chooses a random point $\mathbf{r}^* \leftarrow\$ \mathbb{F}^{v_{i+1}}$ and checks whether $\widetilde{W}_{i+1}(\ell(\mathbf{r}^*)) = q(\mathbf{r}^*)$.

This way, the interaction between the prover and the verifier at this round ends with the new claim about the value of $\widetilde{W}_{i+1}(\mathbf{r}_{i+1})$ with $\mathbf{r}_{i+1} := \ell(\mathbf{r}^*)$.

Finally, we need to define the last round of the protocol. Here, the $\mathcal{V}$ simply evaluates $\widetilde{W}_d(\mathbf{r}_d)$ on his own, costing $\mathcal{O}(n)$ time.

Proposition 20.3. We finally summarize the GKR protocol:

- At the start of the first round, $\mathcal{P}$ sends a function $D : \{0,1\}^{v_0} \to \mathbb{F}$ claimed to equal W_0 (that is, he essentially sends the values of the output layer gates).
- $\mathcal{V}$ picks a random point $\mathbf{r}_0 \leftarrow\$ \mathbb{F}^{v_0}$ and finds $m_0 \leftarrow \widetilde{D}(\mathbf{r}_0)$.
- For each $i \in [d]$ we do the following:
  - Define the $2v_i$-variate polynomial:

$$f_i(\boldsymbol{b}, \boldsymbol{c}; \mathbf{r}_i) = \widetilde{\mathsf{add}}_i(\mathbf{r}_i, \boldsymbol{b}, \boldsymbol{c})(\widetilde{W}_{i+1}(\boldsymbol{b}) + \widetilde{W}_{i+1}(\boldsymbol{c})) + \widetilde{\mathsf{mul}}_i(\mathbf{r}_i, \boldsymbol{b}, \boldsymbol{c})\widetilde{W}_{i+1}(\boldsymbol{b})\widetilde{W}_{i+1}(\boldsymbol{c}).$$

  - $\mathcal{P}$ sends the value m_i claimed to equal $\sum_{\boldsymbol{b},\boldsymbol{c} \in \{0,1\}^{v_{i+1}}} f_i(\boldsymbol{b}, \boldsymbol{c}; \mathbf{r}_i)$.
  - $\mathcal{V}$ applies the Sum-Check protocol on the function $f_i(\cdot; \mathbf{r}_i)$ up until the last round, where the $\mathcal{V}$ has to compute the value $f_i(\boldsymbol{b}^*, \boldsymbol{c}^*; \mathbf{r}_i)$ at random $(\boldsymbol{b}^*, \boldsymbol{c}^*) \in \mathbb{F}^{2v_{i+1}}$.
  - Both prover and the verifier computes the line $\ell : \mathbb{F} \to \mathbb{F}^{v_{i+1}}$ such that $\ell(0) = \boldsymbol{b}^*$ and $\ell(1) = \boldsymbol{c}^*$. $\mathcal{P}$ sends the univariate polynomial $q(X)$ claimed to equal $\widetilde{W}_{i+1} \circ \ell$.
  - $\mathcal{V}$ computes the required value $f_i(\boldsymbol{b}^*, \boldsymbol{c}^*; \mathbf{r}_i)$ using $q(0)$ and $q(1)$ instead of $\widetilde{W}_{i+1}(\boldsymbol{b}^*)$ and $\widetilde{W}_{i+1}(\boldsymbol{c}^*)$.
  - $\mathcal{V}$ chooses a random point $\mathbf{r}^* \leftarrow\$ \mathbb{F}^{v_{i+1}}$ and sets $\mathbf{r}_{i+1} \leftarrow \ell(\mathbf{r}^*)$ and $m_{i+1} \leftarrow q(\mathbf{r}_{i+1})$.
  - The check reduces to verifying that $\widetilde{W}_{i+1}(\mathbf{r}_{i+1}) = m_{i+1}$, so proceed to the next step.
- At the last round, $\mathcal{V}$ directly checks whether $m_d = \widetilde{W}_d(\mathbf{r}_d)$.

21 Offline Memory Checking

21.1 Sum-Check-based Grand Product Protocol

This subsection describes a transparent SNARK, which may be used for proving grand product relations of the following form:

$$\mathcal{R}_{GP} = \left\{(p \in \mathbb{F}, \mathbf{v} \in \mathbb{F}^m) \colon p = \prod_{i=0}^{m} v_i\right\} \tag{33}$$

Without loss of generality, assume that m is a power of 2. We can think of $\mathbf{v}$ as a vector of evaluations of a $\log m$-variate multilinear polynomial $v(x)$ over $\{0,1\}^{\log m}$ in a natural fashion. We assume the prover first opens p and then commits to $v(x)$; the following protocol then verifies both the polynomial's validity and the corresponding grand-product equality derived from that commitment.

Lemma 21.1. A scalar p and a vector $\mathbf{v}$ satisfies the relation $\mathcal{R}_{GP}$ if and only if there exists a multilinear polynomial f in $\log m + 1$ variables such that $f(1,\ldots,1,0) = p$ and $\forall x \in \{0,1\}^{\log m}$ the following hold:

$$f(0,x) = v(x)$$
$$f(1,x) = f(x,0) \cdot f(x,1)$$

Such polynomial f has the following construction:

- $f(1,\ldots,1) = 0$
- For all $\ell \in [\log m]$ and $x \in \{0,1\}^{\log m - \ell}$:
$$f(1^\ell, 0, x) = \prod_{y \in \{0,1\}^\ell} v(x,y)$$

One can simply visualize the structure of f as a binary tree.

Example 21.1. Let $m = 4$ ($\log m = 2$), $\mathbf{v} = \{1,2,3,4\}$, consequently $p = 1 \times 2 \times 3 \times 4 = 24$, then:

$$v(x_1,x_2) = 1 + 2x_1 + x_2$$
$$v(x_1,x_2): \quad v(0,0) = 1,\ v(0,1) = 2,\ v(1,0) = 3,\ v(1,1) = 4.$$

Now, we define f as follows:

$$f(0,0,0) = 1,\ f(0,0,1) = 2,\ f(0,1,0) = 3,\ f(0,1,1) = 4,$$

and:

$$
\begin{aligned}
&f(1,0,0) = f(0,0,0) \times f(0,0,1) = 1 \times 2 = 2,\\
&f(1,0,1) = f(0,1,0) \times f(0,1,1) = 3 \times 4 = 12,\\
&f(1,1,0) = f(1,0,0) \times f(1,0,1) = 2 \times 12 = 24 = p,\\
&f(1,1,1) = 0.
\end{aligned}
$$

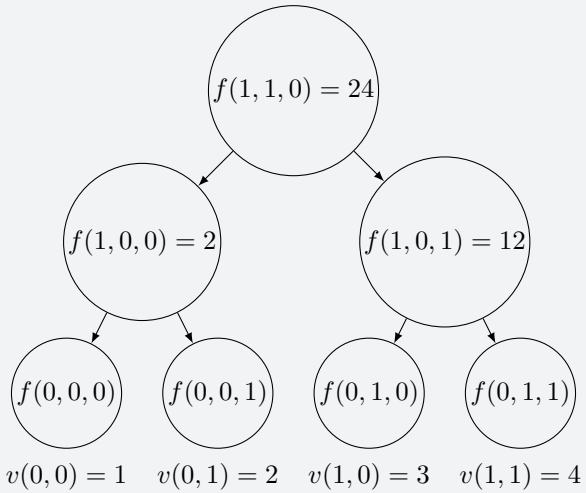

Illustration: *Binary tree of the Grand Product constraints for* $v(x_1, x_2) = 1 + 2x_1 + x_2$.

Then, to check that $\forall x \in \{0,1\}^{\log m}$ the equation $f(1,x) = f(x,0) \cdot f(x,1)$ holds, we can use a sum-check protocol to prove the evaluation of g that is referred to a MLE of $f(1,x) - f(x,0) \cdot f(x,1)$:

$$g(t) = \sum_{x \in \{0,1\}^{\log m}} \widetilde{\mathrm{eq}}(t,x) \cdot (f(1,x) - f(x,0) \cdot f(x,1))$$

By the Schwartz–Zippel lemma, except for a soundness error of $\frac{\log m}{|\mathbb{F}|}$ (which should be negligible), $g(\tau) = 0$ for τ uniformly random in $\mathbb{F}^{\log m}$ if and only if $g = 0$, which implies that $f(1,x) - f(x,0) \cdot f(x,1) = 0$ for all $x \in \{0,1\}^{\log m}$.

Similarly, to prove that $v(x) = f(0,x)$ for all $x \in \{0,1\}^{\log m}$ it suffices to prove that $v(\gamma) = f(0,\gamma)$ for a public coin $\gamma \in \mathbb{F}^{\log m}$.

Thus, to prove the existence of f and hence the grand product relationship, it

suffices to prove, for some verifier selected random $\tau, \gamma \in \mathbb{F}^{\ell}$, that:

$$0 = \sum_{x \in \{0,1\}^{\log m}} \widetilde{\mathrm{eq}}(x, \tau) \cdot (f(1, x) - f(x, 0) \cdot f(x, 1)) \tag{34}$$

$$f(0, \gamma) = v(\gamma) \tag{35}$$

$$f(1, \ldots, 1, 0) = p \tag{36}$$

This can be achieved by running the sum-check protocol between $\mathcal{P}$ and $\mathcal{V}$, where $\mathcal{V}$ has oracle access to v and f. Additionally, $\mathcal{V}$ evaluates the f, v at the random point γ to verify that $f(0, x) = v(x)$. So, the sum-check-based protocol for the grand products looks as follows:

Algorithm 5: Sum-Check-based Protocol for Grand Products

$\mathcal{P}$: Compute polynomials $v \in \mathbb{F}^{\log m}[x]$, $f \in \mathbb{F}^{\log m + 1}[x]$ such that

$$p = \prod_{x \in \{0,1\}^{\log m}} v(x) \quad \text{and} \quad f, v \text{ satisfy (34), (35), (36).}$$

$\mathcal{P}$: $C_f \leftarrow \mathrm{Commit}(f)$; $C_v \leftarrow \mathrm{Commit}(v)$; send C_f, C_v to $\mathcal{V}$.

$\mathcal{V}$: Choose random $\tau, \gamma \in \mathbb{F}^{\log m}$ and send them to $\mathcal{P}$.

$\mathcal{P}$: Compute

$$g(x) \;=\; \widetilde{\mathrm{eq}}(x, \tau) \left(f(1, x) - f(x, 0)\, f(x, 1)\right).$$

$\mathcal{P}$ **&** $\mathcal{V}$: Run SumCheckProtocol$(0,\ g,\ C_f)$

```
/* we assume this sum-check runs over the commitment to f, not
   directly on g.                                              */
```

$\mathcal{V}$: $r \leftarrow \mathrm{Query}(C_f, (1, \ldots, 1, 0))$.

if $r \neq p$ **then**

  $\mathcal{V}$ **rejects**.

end

$\mathcal{V}$: $a \leftarrow \mathrm{Query}(C_f, (0, \gamma))$, $v(\gamma) \leftarrow \mathrm{Query}(C_v, \gamma)$.

if $a \neq v(\gamma)$ **then**

  $\mathcal{V}$ **rejects**.

end

Remark. Note, that during the call to the sum-check protocol, we pass the polynomial g in a couple with the commitment to f. Here we assume that each invocation of the Query method to g inside the sum-check instance will be more complex: **instead of evaluating a single function and returning result, we evaluate f at three different points, check the proofs and return results according to the construction of g.**

21.2 Randomized Permutation Check

The goal of the randomized permutation check is to verify, with high probability, whether two sequences of tuples are permutations of each other, without

performing a full sort or pairwise comparison.

Definition 21.2 (Reed-Solomon Fingerprinting)**.** Let $\mathbf{a} \in \mathbb{F}^n$, then for a random $\gamma \in \mathbb{F}$, the Reed-Solomon fingerprinting of $\mathbf{a}$ is defined as:

$$h_\gamma(\mathbf{a}) = \sum_{i \in n} a_i \cdot \gamma^i.$$

$h_\gamma(\mathbf{a})$ uniquely identifies the sequence $\mathbf{a}$ with high probability, i.e., let $\mathbf{b} \in F^n$ and $\mathbf{a} \neq \mathbf{b}$, then, according to the Schwartz-Zippel lemma:

$$\Pr[h_\gamma(\mathbf{a}) = h_\gamma(\mathbf{b})] \leq \frac{n}{|\mathbb{F}|}.$$

Definition 21.3 (Randomized Permutation Check)**.** Let A and B be two multisets of tuples in $\mathbb{F}^n$. Define

$$\mathcal{H}_{\tau,\gamma}(X) \;=\; \prod_{x \in X} \big(h_\gamma(x) - \tau\big).$$

Then comparing $\mathcal{H}_{\tau,\gamma}(A)$ and $\mathcal{H}_{\tau,\gamma}(B)$ yields a randomized test for whether A and B are permutations of one another. Concretely:

- (Completeness) If $A = B$ (as multisets), then

  $$\mathcal{H}_{\tau,\gamma}(A) = \mathcal{H}_{\tau,\gamma}(B)$$

  with probability 1 over uniform $\tau, \gamma \in \mathbb{F}$.
- (Soundness) If $A \neq B$, then

  $$\Pr\left[\,\mathcal{H}_{\tau,\gamma}(A) = \mathcal{H}_{\tau,\gamma}(B)\right] \;\leq\; \frac{\max(|A|, |B|)}{|\mathbb{F}|}.$$

In other words, by first "hashing" each tuple via the map function h_γ and then taking the τ-shifted product, one obtains a fingerprint that is invariant under permutation but unlikely to collide on distinct sets.

Example 21.2. Consider all operations in $\mathbb{F}_7$, and set $n = 3, \tau = 5$, and $\gamma = 3$. Let

$$A = \{(1,2,3),\ (4,0,6)\}, \quad B = \{(4,0,6),\ (1,2,3)\},$$

First compute the Reed–Solomon fingerprints modulo 7:

$$h_\gamma(1,2,3) = 1 \cdot 3^0 + 2 \cdot 3^1 + 3 \cdot 3^2 = 1 + 6 + 6 = 13 \equiv 6,$$
$$h_\gamma(4,0,6) = 4 \cdot 1 + 0 \cdot 3 + 6 \cdot 2 = 4 + 0 + 12 = 16 \equiv 2.$$

Now form the shifted products:

$$\mathcal{H}_{\tau,\gamma}(A) = (6-5)\,(2-5) = 1 \cdot (-3) \equiv 4,$$
$$\mathcal{H}_{\tau,\gamma}(B) = (2-5)\,(6-5) = (-3) \cdot 1 \equiv 4,$$

so the test *accepts* A vs. B (they are indeed permutations).

Now consider a non-permutation B':

$$B' = \{(1,2,3),\ (2,1,3)\}.$$

$$h_\gamma(2,1,3) = 2 \cdot 1 + 1 \cdot 3 + 3 \cdot 2 = 2 + 3 + 6 = 11 \equiv 4.$$
$$\mathcal{H}_{\tau,\gamma}(B') = (6-5)\,(4-5) = 1 \cdot (-1) \equiv 6 \neq 4,$$

so the test *rejects* A vs. B'. In this way a single choice of (γ, τ) distinguishes permutations from non-permutations.

21.3 Offline Memory Checking

This protocol lets you, after doing a bunch of read/writes from an untrusted memory, verify in one quick algebraic step that every answer really came from the same original contents – without re-running any reads.

For the first time it's not so obvious, why we may need need such a protocol, so let's start with a simple example.

Example 21.3. Consider Alice, who stores two values on Bob's dedicated server at addresses 0 and 1. Initially, Bob's memory contains

$$M \;=\; \{(0,100),\ (1,200)\}.$$

Alice then performs the following operations in sequence:

1. Alice writes a new value 150 at address 0. Bob updates his memory to $\{(0,150),(1,200)\}$.
2. Alice reads from address 0 and obtains the reply 150.
3. Alice reads from address 1 and (honestly) obtains 200.

At this point, both replies individually look correct. However, Bob can cheat on the very last step by returning, say, 300 instead of 200:

$$(1,200) \;\longrightarrow\; (1,300).$$

Alice only sees the single read result "300 at address 1," which could simply be the true value after some write she missed—and she has no immediate way to detect the inconsistency.

In other words, without keeping an auditable record of all reads, Alice cannot later prove that all replies came from a single, immutable memory

> state. This gap is exactly what an *offline memory check* fills: after all reads are done, Alice runs one fast verification that guarantees "every read you saw really did come from one fixed initial memory + your writes," and thus catches any cheating by Bob.

Each memory cell can be described as a tuple $(\mathsf{addr}, \mathsf{val}, \mathsf{counter})$, where addr is the address of the memory cell, val is the value stored at that address, and $\mathsf{counter}$ is the number of times the memory cell has been accessed. You can think of a counter as a timestamp. The memory is a vector of memory cels.

The protocol utilizes four sets of tuples:

- **init** – contains the initial memory state, where all counters are set to 0;
- **write** – contains memory cels that represent write operations, where val is the value stored at the memory address addr after the specified counter;
- **read** – contains memory cels that represent read operations, where val is the value read from the memory address addr at the specified counter;
- **final** – contains the final memory state, where all counters are set to the last value after the last read/write operation.

At the beginning, **init** is populated with the initial memory state, where all counters are set to 0, indicating that the values were first accessed during the initialization phase, while the **read**, **write** are empty. Then, for each read operation, the untrusted memory is queried at addr, returning a pair $(\mathsf{val}, \mathsf{counter})$, and writing a tuple $(\mathsf{addr}, \mathsf{val}, \mathsf{counter})$ to the **read** set. For each write operation a corresponding tuple $(\mathsf{addr}, \mathsf{newval}, \mathsf{counter} + 1)$, where newval is a value being written, is added, with counter incremented by one, to the **write** set.

We assume that before each write operation, the read operation is performed. And the read operation is always followed by a write operation to the same address. Thus, to make a plain read operation, the prover first reads the value from the memory, then writes it back to the memory.

After all reads and writes are done, the **final** set is populated with the final memory state. Now, the verifier can check the consistency of all the operations by verifying the following equation:

$$\mathbf{read} \cup \mathbf{final} = \mathbf{write} \cup \mathbf{init}.$$

Thus, using randomized permutation check, one can rewrite the equation as

follows:

$$\mathcal{H}_{\tau,\gamma}(\mathbf{read}) \cdot \mathcal{H}_{\tau,\gamma}(\mathbf{final}) = \mathcal{H}_{\tau,\gamma}(\mathbf{write}) \cdot \mathcal{H}_{\tau,\gamma}(\mathbf{init}),$$

$$\prod_{(a,v,t)\in\mathbf{read}} \big(h_\gamma(a,v,t) - \tau\big) \prod_{(a,v,t)\in\mathbf{final}} \big(h_\gamma(a,v,t) - \tau\big) =$$

$$= \prod_{(a,v,t)\in\mathbf{write}} \big(h_\gamma(a,v,t) - \tau\big) \prod_{(a,v,t)\in\mathbf{init}} \big(h_\gamma(a,v,t) - \tau\big),$$

$$\prod_{(a,v,t)\in\mathbf{read}} \big((a\gamma^2 + v\gamma + t) - \tau\big) \prod_{(a,v,t)\in\mathbf{final}} \big((a\gamma^2 + v\gamma + t) - \tau\big) =$$

$$= \prod_{(a,v,t)\in\mathbf{write}} \big((a\gamma^2 + v\gamma + t) - \tau\big) \prod_{(a,v,t)\in\mathbf{init}} \big((a\gamma^2 + v\gamma + t) - \tau\big).$$

Example 21.4. Suppose the we track a single *row-memory*, the algorithm performs three reads and two writes, the initial memory is given by the following table:

Address	0	1	2	3
Value	2	5	7	9

Initial memory (counter 0):

$$\mathbf{init} = \{(0,2,0),\ (1,5,0),\ (2,7,0),\ (3,9,0)\},$$

while

$$\mathbf{read} = \varnothing \quad \text{and} \quad \mathbf{write} = \varnothing.$$

Trace 1 - consistent execution

step	**operation**	$\Delta\mathbf{read}_{\text{step}}$	$\Delta\mathbf{write}_{\text{step}}$
1	read(1) $\to$ $(5,0)$	$(1,5,0)$	-
2	write((1,6))	-	$(1,6,1)$
3	read(2) $\to$ $(7,0)$	$(2,7,0)$	-
4	write((2,7))	-	$(2,7,1)$

After the four steps

$$\mathbf{read} = \{(1,5,0),\ (2,7,0)\},$$
$$\mathbf{write} = \{(1,6,1),\ (2,7,1)\}.$$

And we can fill **final** as:

$$\mathbf{final} = \{(0,2,0),\ (1,6,1),\ (2,7,1),\ (3,9,0)\}$$

One can clearly see that $\mathbf{read} \cup \mathbf{final} = \mathbf{write} \cup \mathbf{init}$.

$$\{(1,5,0),(2,7,0)\} \cup \{(0,2,0),(1,6,1),(2,7,1),(3,9,0)\} =$$
$$= \{(1,6,1),(2,7,1)\} \cup \{(0,2,0),(1,5,0),(2,7,0),(3,9,0)\}$$

Trace 2 - inconsistent execution
Let the second reading return a wrong value – a, instead of 5. So, the new trace would look like:

step	**operation**	$\Delta\mathbf{read}_{\text{step}}$	$\Delta\mathbf{write}_{\text{step}}$
1	read(1) $\rightarrow$ $(a, 0)$	$(1, a, 0)$	-
2	write((1,6))	-	$(1, 6, 1)$

After the two steps

$$\mathbf{read} = \{(1, a, 0)\},$$
$$\mathbf{write} = \{(1, 6, 1)\},$$

The verifier checks $\mathbf{read} \cup \mathbf{final} = \mathbf{write} \cup \mathbf{init}$:

$$\{(1, a, 0)\} \cup \{(0, 2, 0), (1, 6, 1), (2, 7, 0), (3, 9, 0)\} \neq$$
$$\neq \{(1, 6, 1)\} \cup \{(0, 2, 0), (1, 5, 0), (2, 7, 0), (3, 9, 0)\}$$

and *rejects*.

Actually, no multiset final that depends on the valid values and addresses can reconcile them.