Math Companion to Soundcalc

February 3, 2026

Disclaimer: This document is a work in progress. It may contain mistakes and inaccuracies. For now, code is king.

Contents

1 Preliminaries

1.1 Fields

Fields of size q are denoted as $\mathbb{F}_q$ or simply $\mathbb{F}$. Currently supported prime fields:

- KoalaBear: $q = 2^{31} - 2^{24} + 1$
- Goldilocks: $q = 2^{64} - 2^{32} + 1$
- Mersenne31: $q = 2^{31} - 1$
- BN254: $q = 21888242871839275222246405745257275088548364400416034343698204186575808495617$

1.2 Regime-specific bounds

1.2.1 Proximity parameter

`unique_decoding.py/get_proximity_parameter()`

In UDR

$$\delta_U(\rho) = (1 - \rho)/2 \tag{1}$$

In JBR we compute the δ differently.

`johnson_bound.py/get_proximity_parameter()`

$$\delta_J(\rho, N, q) = \begin{cases} 1 - \sqrt{\rho} - \frac{\sqrt{\rho}}{100} & \text{if } q > 2^{150} \\ 1 - \sqrt{\rho} - \max(\frac{\rho}{20}, \frac{\sqrt{\rho}}{100}) & \text{otherwise} \end{cases} \tag{2}$$

1.2.2 List sizes

`unique_decoding.py/get_max_list_size()`

$$\ell = 1$$

`johnson_bound.py/get_max_list_size()`

We compute

$$\ell(\rho, N) = \frac{1}{2(1 - \sqrt{\rho} - \delta)\sqrt{\rho}}.$$

1.2.3 Batching errors

We define batching error functions

$$\epsilon_{\text{batch,pow},J}(\rho, N, q, B), \quad \epsilon_{\text{batch,pow},U}(\rho, N, q, B), \quad \epsilon_{\text{batch,lin},U}(\rho, N, q), \quad \epsilon_{\text{batch,pow},U}(\rho, N, q, B)$$

For JBR :

- Compute proximity parameter δ as in (2):
- Compute m as in `get_m()`:

$$m = 0.5 + \max\left(\left\lceil \frac{\sqrt{\rho}}{1 - \sqrt{\rho} - \delta} \right\rceil, 3\right)$$

- Compute linear error[1]:

$$\epsilon_{\text{batch,lin},J}(\rho, N, q) = \frac{N\left(2m^5 + 3m\delta\rho\right) + 3m\rho}{3\rho^{1.5}q}$$

- Compute powers error:

$$\epsilon_{\text{batch,pow},J}(\rho, N, q, B) = \epsilon_{\text{batch,lin},J}(\rho, N, q) \cdot (B-1)$$

For UDR :

- Compute linear error:

$$\epsilon_{\text{batch,lin},U}(\rho, N, q) = \frac{N}{q\rho}$$

- Compute powers error:

$$\epsilon_{\text{batch,pow},U}(\rho, N, q, B) = \epsilon_{\text{batch,lin},J}(\rho, N, q) \cdot B$$

1.2.4 DEEP-ALI errors

`circuit.py/get_DEEP_ALI_errors()`

The DEEP-ALI soundness error (with exception of the list size) and notation are taken from [Hab22b, Section 5.2]:

- Number of AIR constraints [Hab22b, p.15]: $C =$ `num_constraints`
- Overall AIR degree: $d =$ `AIR_max_degree`
- Trace increase constant [Hab22b, Remark 7]: m_c `max_combo` (set to 2 in [Hab22b])
- Trace length: N (denoted by $|H|$ in [Hab22b])
- List size: ℓ as in section 1.2.2, replaces the L^+ in [Hab22b]

The DEEP-ALI soundness error is computed as follows:

$$\epsilon_{\text{DA}} = \max\left(\frac{\ell \cdot C}{2^{b_{\text{field}}}}, \frac{\ell(d(N + m_c - 1) + (N-1))}{2^{b_{\text{field}}} - N - N/\rho}\right) \tag{3}$$

Sanity check (multi-point condition, §4.1.3 in [Hab22b]). Since our DEEP-ALI bound uses multi-point quotients (a.k.a. combo batching), we enforce the domain-sizing requirement

$$N + m_c < (1-\theta) \cdot \frac{N}{\rho}, \tag{4}$$

where ρ is the FRI rate and θ is the proximity parameter determined by the chosen regime. This condition is checked in code via an assertion in `circuit.py/_get_DEEP_ALI_errors()`.

[1] Code refers to Theorem 4.2 from BCHKS25

2 WHIR-based VM security level calculation

2.1 Notation and parameters

- Number of iterations: $M = \texttt{num\_iterations}$
- Iteration index: $i \in \{0, \ldots, M-1\}$
- Folding round index: $s \in \{0, \ldots, k-1\}$
- Extension field size: $|\mathbb{F}| = b_{\text{field}}$
- Constraint degree: $d = \texttt{constraint\_degree}$
- Folding factor: $k = \texttt{folding\_factor}$
- Log-degree in iteration i: $m_i = \texttt{log\_degrees}[i]$, and we set $m_i = m_0 - k \cdot i$
- Log-inverted rate in iteration i: $\mu_i = \texttt{log\_inv\_rates}[i]$, and we set $\mu_i = \log_2 \frac{1}{\rho} + (k-1) \cdot i$
- Iteration-round-specific RS codes:

$$\mathcal{C}_{i,s} = (\mathcal{C}_{i,s}[\rho], \mathcal{C}_{i,s}[N])_{i,s} = (2^{-\mu_i}, 2^{m_i - s})$$

- Number of OOD samples in iteration i: $w_i = \texttt{num\_ood\_samples}[i]$
- Number of queries in iteration i: $t_i = \texttt{num\_queries}[i]$
- Grinding bits:

$$g_{\text{batch}}, \quad \{g_{i,s}^{\text{fold}}\}_{i \in [M], s \in [k]}, \quad \{g_i^{\text{ood}}\}_{i \in M}, \quad \{g_i^{\text{qry}}\}_{i \in M}.$$

2.2 Bits of Security in UDR and JBR

In order to compute the bits of security in UDR and JBR regimes, we compute the following error terms using the parameters from section 2.1.

Computed in `get_security_levels_for_regime()`.

For any error term:

$$\lambda(\epsilon) = \lfloor -\log_2 \epsilon \rfloor.$$

Overall security:

$$\lambda_{\text{total}} = \min\{\lambda_{\text{batch}}, \{\lambda_{i,s}^{\text{fold}}\}_{i \in [M], s \in [k]}, \{\lambda_i^{\text{out}}\}_{i \in \{1, \ldots, M-1\}}, \{\lambda_i^{\text{shift}}\}_{i \in \{1, \ldots, M-1\}}, \lambda^{\text{fin}}\}.$$

2.3 Batching Error ϵ_{batch}

This computes the batching error with parameters from section 2.1. `get_batching_error()`

In the Johnson Bound Regime (JBR) we proceed as follows:

- Set code parameters

$$(\rho, N) = \left(2^{-\mu_0}, 2^{m_0}\right)$$

- Compute base error (cf. section 1.2.3):

$$\varepsilon_{\text{batch},JBR}^{\text{base}} = \begin{cases} \epsilon_{\text{batch,pow},J}(\rho, N, b_{\text{field}}, B) & \text{(power batching)}, \\ \epsilon_{\text{batch,lin},J}(\rho, N, b_{\text{field}}) & \text{(linear batching)}. \end{cases}$$

Whereas in the UDR we do as follows:

- Compute base error (cf. section 1.2.3):

$$\varepsilon_{\text{batch},UDR}^{\text{base}} = \begin{cases} \epsilon_{\text{batch,pow},U} & \text{(power batching)}, \\ \epsilon_{\text{batch,lin},U} & \text{(linear batching)}. \end{cases}$$

In both regimes we do as follows after grinding:

$$\epsilon_{\text{batch}} = \varepsilon_{\text{batch}}^{\text{base}} \cdot 2^{-g_{\text{batch}}}.$$

2.4 Folding Error

This section computes the error of a folding round as in `whir_based_vm.py/epsilon_fold()`, referring to Theorem 5.2 of the WHIR paper.

For iteration $i \in [M]$ and folding round $s \in [k]$ in JBR we do[2]:

- Get list size as in section 1.2.2

$$\ell_{i,s} = \ell(\mathcal{C}_{i,s}[\rho])$$

- Base error (two terms):

$$\varepsilon_{i,s}^{\text{fold,base,J}} = d \cdot \frac{\ell_{i,s}}{b_{\text{field}}} + \epsilon_{\text{batch,pow},J}(\mathcal{C}_{i,s+1}[\rho], \mathcal{C}_{i,s+1}[N], b_{\text{field}}, B = 2).$$

And in UDR the base error is

$$\varepsilon_{i,s}^{\text{fold,base,U}} = \frac{d}{b_{\text{field}}} + \epsilon_{\text{batch,pow},U}(\mathcal{C}_{i,s+1}[\rho], \mathcal{C}_{i,s+1}[N], b_{\text{field}}, B = 2).$$

After grinding:

$$\epsilon_{i,s}^{\text{fold}} = \varepsilon_{i,s}^{\text{fold,base}} \cdot 2^{-g_{i,s}^{\text{fold}}}.$$

2.5 OOD error

This computes the Out-of-Domain error as in `whir_based_vm.py/epsilon_out()`.

For iteration $i \in \{1, 2, \ldots, M-1\}$

- Base error in JBR (cf. section 1.2.2):

$$\varepsilon_i^{\text{out,base,J}} = \ell_{i,0}^2 \left(\frac{2^{m_i}}{2q}\right)^{w_{i-1}}.$$

- Base error in UDR:

$$\varepsilon_i^{\text{out,base,U}} = \left(\frac{2^{m_i}}{2q}\right)^{w_{i-1}}.$$

- After grinding:

$$\epsilon_i^{\text{out}} = \varepsilon_i^{\text{out,base}} \cdot 2^{-g_{i-1}^{\text{ood}}}.$$

2.6 Shift Error

`epsilon_shift()`

For iteration $i \in \{1, 2, \ldots, M-1\}$ in JBR

- Get δ for the iteration using (2):

$$\delta_i = \min_{s \in [k+1]} \delta_J(\mathcal{C}_{i-1,s}[\rho], \mathcal{C}_{i-1,s}[N], q)$$

- Base error (two terms):

$$\varepsilon_i^{\text{shift,base}} = (1 - \delta_i)^{t_{i-1}} + \ell_{i,0} \cdot \frac{t_{i-1} + 1}{q}.$$

Same procedure in UDR:

- Get δ for the iteration using (1):

$$\delta_i = \min_{s \in [k+1]} \delta_U(\mathcal{C}_{i-1,s}[\rho])$$

- Base error :

$$\varepsilon_i^{\text{shift,base}} = (1 - \delta_i)^{t_{i-1}} + \frac{t_{i-1} + 1}{q}.$$

After grinding:

$$\epsilon_i^{\text{shift}} = \varepsilon_i^{\text{shift,base}} \cdot 2^{-g_{i-1}^{\text{qry}}}.$$

[2] The function `epsilon_fold()` counts folding rounds from 1 to k, whereas here we count from 0 to $k-1$.

2.7 Final Round Error

`epsilon_final()`

JBR:

- Get δ for the iteration using (2):
$$\delta_{M-1} = \min_{s\in[k+1]} \delta_J(\mathcal{C}_{M-1,s}[\rho])$$

UDR:

- Get δ for the iteration using (1):
$$\delta_{M-1} = \min_{s\in[k+1]} \delta_U(\mathcal{C}_{M-1,s}[\rho])$$

Then the base error is :
$$\varepsilon^{\text{fin,base}} = (1-\delta_{M-1})^{t_{M-1}}.$$

After grinding:
$$\epsilon^{\text{fin}} = (1-\delta_{M-1})^{t_{M-1}} \cdot 2^{-g^{\text{qry}}_{M-1}}.$$

3 WHIR Proof Size Calculations

This section summarizes the proof-size formula computed in `get_proof_size_bits()`.

3.1 Initial Function Size

$$S_{\text{if}} = b_{\text{hash}}.$$

3.2 Initial Sumcheck Size

The folding proof size per round is
$$S_{\text{sumcheck}} = k(d-1)b_{\text{field}}.$$

3.3 OOD Proof Size for Iteration i

$$S_{\text{ood},i} = b_{\text{hash}} + w_i b_{\text{field}} + k(d-1)b_{\text{field}}$$

$$S_{\text{ood},M} = 2^{m_M} b_{\text{field}}$$

3.4 Query Proof Size for Iteration i

We compute
$$S_{\text{qry},i} = \begin{cases} eMP(2^{m_i+\mu_i-k}, t_i, 2^k, b_{\text{field}}, b_{\text{hash}}). & i > 0 \\ eMP(2^{m_0+\mu_0-k}, t_0, B\cdot 2^k, b_{\text{field}}, b_{\text{hash}}). & i = 0 \end{cases}$$

3.5 Total Proof Size

Collecting all terms and summing over all M iterations:

$$\boxed{S_{\text{total}} = S_{\text{if}} + S_{\text{sumcheck}} + \sum_{i\in[M]} (S_{\text{ood},i} + S_{\text{qry},i})}$$

4 FRI-based VM security level calculation

This section calculates the security level for a FRI-based VM in section 4.2.

4.1 FRI parameters

Global parameters used in the FRI analysis:

- r_{FRI} — number of FRI rounds.
- Folding factors $\widehat{\text{folds}} = [k_0, k_1, \ldots, k_{r_{FRI}-1}]$;
- t — number of queries.
- δ — proximity parameter.
- ρ — rate of the Reed-Solomon code.
- N — trace length.
- ℓ — list size.
- $b_{\text{grind},Q}$ — grinding parameter for the query phase.
- n — witness size.
- b_{hash} — number of bits in the hash function output.
- b_{proof} — proof size in bits.
- B — *batch size*. Number of functions appearing in the batched-FRI.
- b_{field} — number of bits in the extension field element.

4.2 Security level for a FRI-based VM

The security level is calculated in

`zkvms/fri_based_vm.py/get_security_levels()`

.

It is done separately for two different regimes: UDR and JBR — using the same procedure `fri_based_vm.py/get_security_l`

1. Calculate the FRI round-by-round soundness errors $\mathbf{e}_{\text{FRI}}$:

$$\mathbf{e}_{\text{FRI}} = \max(\text{e}_{\text{batch}}, \{\epsilon_i^{\text{fold}}\}_{i\in[r_{FRI}]}, \text{e}_{\text{query}})$$

2. Obtain optimal δ proximity parameter as by section 1.2.1.
3. Obtain the list size ℓ for the respective δ as by section 1.2.2.
4. Obtain the DEEP-ALI soundness error ϵ_{DA} as by section 1.2.4.
5. Compute the total soundness error as

$$\epsilon = \max(\mathbf{e}_{\text{FRI}}, \epsilon_{\text{DA}}).$$

Then the full security level in bits is the maximum of the two regimes:

$$\text{Security level} = \max(-\log_2 \epsilon_U, -\log_2 \epsilon_J).$$

4.3 Batching Error

This computes the batching error with parameters from section 2.1. `get_batching_error()`

In the Johnson Bound Regime (JBR) we compute base error as:

$$\mathrm{e}_{\text{batch}}^{\text{base,JBR}} = \begin{cases} \epsilon_{\text{batch,pow},J}(\rho, N, q, B) & \text{(power batching)}, \\ \epsilon_{\text{batch,lin},J}(\rho, N, q) & \text{(linear batching)}. \end{cases}$$

Whereas in the UDR we compute base error:

$$\mathrm{e}_{\text{batch}}^{\text{base,UDR}} = \begin{cases} \epsilon_{\text{batch,pow},U}(\rho, N, q, B) & \text{(power batching)}, \\ \epsilon_{\text{batch,lin},U}(\rho, N, q) & \text{(linear batching)}. \end{cases}$$

4.4 Commit phase errors

This is calculated in `fri_based_vm.py/get_commit_phase_error()`.

For round $i \in [r_{FRI}]$ we compute the dimension N_i as

$$N_i = \frac{N}{\prod_{j \in [i]} k_j}$$

Then the folding error in round i is computed as

$$\epsilon_i^{\text{fold}} = \begin{cases} \epsilon_{\text{batch,pow},J}(\rho, N_i, q, k_i), & \text{JBR}, \\ \epsilon_{\text{batch,pow},U}(\rho, N_i, q, k_i), & \text{UDR} \end{cases} \tag{5}$$

4.5 Query error

This is calculated in `fri_based_vm.py/get_query_phase_error()`.

Query phase error:

$$\epsilon_{\text{query}} = (1-\theta)^t \cdot 2^{-b_{\text{grind},Q}} \tag{6}$$

5 FRI proof size

This calculation is performed in

`fri.py/get_FRI_proof_size_bits()`

. The FRI proof contains two parts: Merkle roots, and one "openings" per query, where an "opening" is a Merkle path for each folding layer. For each layer we count the size that this layer contributes, which includes the root and all Merkle paths.

Initial round: one root and one path per query. We assume that for the initial functions, there is only one Merkle root, and each leaf i for that root contains symbols i for all initial functions.

Folding rounds: we assume that "siblings" for the following layers are grouped together in one leaf. This is natural as they always need to be opened together.

The proof size is calculated as follows:

$$b_{\text{proof}} = \underbrace{b_{\text{hash}} + eMP(n, t, B, b_{\text{field}}, b_{\text{hash}})}_{\text{Initial round}} + \underbrace{\sum_{0 \le i < r_{FRI}} \left(b_{\text{hash}} + eMP\left(\frac{n}{\prod_{0 \le j \le i} \widehat{\text{folds}[j]}}, t, B, b_{\text{field}}, b_{\text{hash}}\right) \right)}_{\text{Folding rounds}} \tag{7}$$

where $eMP(n, t, b, b_{\text{field}}, b_{\text{hash}})$ is the expected Merkle path size for t queries to the n leaf tree, where each leaf has b elements of b_{field} bits each, calculated as `get_size_of_merkle_multi_proof_bits_expected()`

$$eMP(n, t, b, b_{\text{field}}, b_{\text{hash}}) = \underbrace{t \cdot b \cdot b_{\text{field}}}_{\text{size of all}} + b_{\text{hash}} \sum_{1 \le d \le \lceil \log_2 n \rceil} \left\lceil 2^d \left((1 - 2^{-d})^t - (1 - 2^{1-d})^t \right) \right\rceil \tag{8}$$

6 Lookup soundness calculation

We consider logUP, the lookup argument introduced in [Hab22a].

We have a table $\mathbf{T} \in \mathbb{F}^{T\times S}$ and tables $\mathbf{L}^1, \ldots, \mathbf{L}^K$ in $\mathbb{F}^{L\times S}$ that are lookups of it, that is, $\forall\ i \in [L]\ \exists\ j \in [T]\ s.t.\ \mathbf{L}_i = \mathbf{T}_j$.

For the sake of the integration and in order to be consistent with parameters, we consider the cases when the lookup argument is proven using a univariate or multivariate IOP separately. We also consider a differentiate case when $S = 1$.

The logUP argument proves lookup relations by proving that given a vector $\vec{m} \in \mathbb{F}^{|\mathbf{T}|}$ whose elements are either 1 or 0, the lookup relation holds if and only if the following euqation is stisfied.

$$\sum_{\vec{l}\in\mathbf{L}} \frac{1}{X-\vec{l}} = \sum_{\vec{t}\in\mathbf{T}} \frac{m_{\vec{t}}}{X-\vec{t}}$$

We split the logUP protocol in two steps:

1. Compute some polynomial expression of the equation above
2. Prove consistency with $\mathbf{T}$ and $\mathbf{L}$

6.1 Univariate case

In this case, we always assume that proving the well formation of the sums is deferred to AIR constraints, and that the additional constraints this causes are already taken into account in the size of the AIR trace for the soundness of the proof of correct execution of it.

We consider the following parameters:

- F = size of the extension field $\mathbb{F}$
- T = rows of $\mathbf{T}$
- L = rows of $\mathbf{L}$
- S = number of columns of T and L
- M = number of lookups performed on $\mathbf{T}$

The expression

$$\sum_{i=1}^{L} \frac{1}{X-\sum_{s=1}^{S}\mathbf{L}_{is}Y^{s-1}} = \sum_{j=1}^{T} \frac{m_j}{X-\sum_{s=1}^{S}\mathbf{T}_{js}Y^{s-1}}$$

is replaced, upon receiving α, β uniformly sampled from $\mathbb{F}$, by the polynomial equality

$$\sum_{i=1}^{L} \frac{1}{\alpha-\sum_{s=1}^{S}\mathbf{L}_{is}\beta^{s-1}}\lambda_i(X) = \sum_{j=1}^{T} \frac{m_j}{\alpha-\sum_{s=1}^{S}\mathbf{T}_{js}\beta^{s-1}}\mu_j(X)$$

$\lambda_i(X), \mu_j(X)$ being the Lagrange interpolation polynomials of some domains of size L and T, respectively.

The probability that a prover gets a randomness that verifies without $\mathbf{L}$ being a lookup on $\mathbf{T}$ is bounded by ϵ_{sum} as follows:

6.1.1 Multi-column:

A cheating prover can be lucky and obtain α, β such that the expression above is satisfied for a table $\mathbf{L}$ that is not a lookup on $\mathbf{T}$ with probability $\epsilon_{\mathsf{sum}} \le \frac{(L+T)S}{F}$.

6.1.2 Single column:

For the single column case, we simply set $S = 1$ above and have that the soundness error is $\epsilon_{\mathsf{sum}} \le \frac{(L+T)}{F}$.

6.1.3 Aggregation:

When performing M lookups on $\mathbf{T}$, we have that the soundness error ϵ_{sum} is bounded by $M\frac{(L+T)S}{F}$.

6.1.4 Dummy Variables

If $\mathbf{T}, \mathbf{L}$ are smaller than the domain size and the tables are filled with dummy variables, those variables should be taken into account in the size of L and T.

6.2 Multivariate

We consider the following parameters:

- F = size of the extension field $\mathbb{F}$
- T = rows of $\mathbf{T}$
- L = rows of $\mathbf{L}$
- H = size of alphabet Σ
- S = number of columns of T and L
- M = number of lookups performed on $\mathbf{T}$

We assume $\mathbf{L}$ and $\mathbf{T}$ are padded to have $T = L = H$. The expression

$$\sum_{i=1}^{L} \frac{1}{X - \sum_{s=1}^{S} \mathbf{L}_{is} Y^{s-1}} = \sum_{j=1}^{T} \frac{m_j}{X - \sum_{s=1}^{S} \mathbf{T}_{js} Y^{s-1}}$$

is replaced, upon receiving α uniformly sampled from $\mathbb{F}$, by the polynomial equality

$$\sum_{\vec{x} \in \Sigma} \frac{1}{\alpha - L(\vec{x})} = \sum_{\vec{x} \in \Sigma} \frac{m(\vec{x})}{\alpha - T(\vec{x})}$$

where $\Sigma \in \mathbb{F}^{\ell}$ and $|\Sigma| = \max\{TS, LS\}$.
To convert the equation above into a polynomial one, both sides are multiplied by

$$\prod_{\vec{x} \in \Sigma} (\alpha - L(\vec{x})) \prod_{\vec{x} \in \Sigma} (\alpha - T(\vec{x}))$$

To prove step number 2. of logUP, that is, that these expressions are consistent with $\mathbf{L}$ and $\mathbf{T}$, we consider two options: (i) The logUP argument is proven independently of the AIR trace, which adds an error $\epsilon_{\mathsf{logup-sound}}$ (ii) The polynomial identities that prove consistency are reduced to univariate identities, and included in the AIR constraints as in the univariate case, which adds an error $\epsilon_{\mathsf{reduction-univ}}$.

6.2.1 Single column:

By the same reasoning as above, we have that a cheating prover can get lucky with α with probability $\epsilon_{\mathsf{sum}} \leq \frac{2H}{F}$.

6.2.2 Multi column:

As the tables $\mathbf{L}, \mathbf{T}$ are already represented as tensors, considering more than one column consists on simply expanding $\mathbf{L}, \mathbf{T}$ and modifying L, T, Σ accordingly.

6.2.3 Aggregation:

When performing K lookups on one table, they can all be verified together, as looking up a matrix of size $L \times K$ on $\mathbf{T}$. The soundness error is then bounded as $\epsilon_{\mathsf{sum}} \leq \frac{K2H}{F}$.

References

[Hab22a] Ulrich Haböck. Multivariate lookups based on logarithmic derivatives. *Cryptology ePrint Archive*, 2022.

[Hab22b] Ulrich Haböck. A summary on the fri low degree test. Cryptology ePrint Archive, Paper 2022/1216, 2022. URL: `https://eprint.iacr.org/archive/2022/1216/20241217:162441`, `arXiv:2022/1216`.