# TrX Implementation Note

## 1 Notation and Preliminaries

### 1.1 Cryptographic Groups and Pairings

Let $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ be a pairing-friendly elliptic curve setting where:

- $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are cyclic groups of prime order $p$
- $\mathbb{F} = \mathbb{F}_p$ is the scalar field
- $g_1 \in \mathbb{G}_1$ (also denoted $g$) and $h \in \mathbb{G}_2$ (also $g_2$) are generators
- $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a bilinear pairing

### 1.2 System Parameters

- $n$ = number of validators (parties), typically $n \in \{4, 7, 10, \ldots\}$
- $t$ = threshold for decryption ($1 \leq t < n$), typically $t = \lceil 2n/3 \rceil$ for Byzantine fault tolerance
- $B$ = maximum batch size (number of transactions), e.g., $B = 128$ or $B = 1024$
- $K$ = number of decryption contexts per epoch (e.g., $K = 100{,}000$ blocks)
- $\mathsf{ctx} = (\mathsf{block\_height}, \mathsf{context\_index})$ = decryption context binding

### 1.3 Cryptographic Primitives

- $\mathsf{com} \in \mathbb{G}_1$ = transaction batch's commitment (KZG polynomial commitment)
- $\mathsf{pd}_i \in \mathbb{G}_2$ = partial decryption from validator $i$
- $\tau \in \mathbb{F}$ = trusted setup trapdoor for KZG (must remain secret after setup)
- $\kappa_{\mathsf{ctx}} \in \mathbb{F}$ = randomness for context $\mathsf{ctx}$ (prevents cross-context replay attacks)

### 1.4 Public Parameters

We distinguish between the original TrX parameters and the implementation parameters:

- **Original TrX:** $\mathsf{pp}$ includes a context-indexed CRS $\{ g^{\kappa_{\mathsf{ctx}}\tau^j} \}_{\mathsf{ctx}\in[K], j\in[B]}$ and $h^\tau$, where $g \in \mathbb{G}_1$ and $h \in \mathbb{G}_2$.
- **Implementation:** $\mathsf{pp.params}$ are Tess parameters for silent keygen/aggregation, and $\mathsf{pp.srs}$ is a KZG SRS with $\{[g_1^{\tau^k}]\}_{k=0}^d$ and $\{[g_2^{\tau^k}]\}_{k=0}^d$.

### 1.5 Hash Functions

We model the following as random oracles:

**Original TrX:**

- $H_1 : \{0,1\}^* \to \mathbb{G}_1$ (hash-to-curve for $\mathbb{G}_1$)
- $H_F : \mathbb{G}_1 \times D \to \mathbb{F}$ (hash to field for tags)
- $\mathsf{KDF} : \mathbb{G}_T \to \{0,1\}^{|m|}$ (key derivation function for one-time pad encryption)

**Implementation:**

- $H_{\text{sig}} : \{0,1\}^* \to \{0,1\}^{256}$ (SHA-256 for Ed25519 signatures)
- $H_{\text{com}} : \{0,1\}^* \to \mathbb{F}$ (hash-to-field for commitment coefficients):

$$H_{\text{com}}(\mathsf{tx.ct}, \mathsf{tx.ad}, \mathsf{ctx}) = \mathsf{hash\_to\_scalar}(\text{SHA256}(\mathsf{tx.ct} \,\|\, \mathsf{tx.ad} \,\|\, \mathsf{ctx.height} \,\|\, \mathsf{ctx.index}))$$

# 2 Original TrX Scheme

## 2.1 Setup

**Algorithm 1** Setup

1: Sample $\tau \leftarrow \mathbb{F}$ and $\{\kappa_{\mathsf{ctx}} \leftarrow \mathbb{F}\}_{\mathsf{ctx}\in[K]}$
2: Publish CRS $\mathsf{crs} = \{g^{\kappa_{\mathsf{ctx}}\tau^j}\}_{\mathsf{ctx}\in[K], j\in[B]}$ and $h^\tau$

## 2.2 Key Generation

**Algorithm 2** KeyGen

1: Sample secret $\mathsf{trx}.sk \leftarrow \mathbb{F}$ and create $(t,n)$ shares $\{[\mathsf{trx}.sk]_i\}$
2: Set $pk = h^{\mathsf{trx}.sk}$ and $pk_i = h^{[\mathsf{trx}.sk]_i}$
3: Output encryption key $\mathsf{trx}.ek = (pk, pk^\tau)$ and decryption key $\mathsf{trx}.dk = \mathsf{trx}.sk$

## 2.3 Encryption

**Algorithm 3** Transaction Encryption

**Ensure:** User's account credentials $(\mathsf{Ed25519}.vk, \mathsf{Ed25519}.sk)$

1: Sample $\alpha \xleftarrow{\$} \mathbb{F}$
2: Compute $tag \leftarrow H_F(\mathsf{Ed25519}.vk, ad)$
3: Set $ct^{(1)} := pk^{\alpha(\tau - tag)} = (pk^\tau)^\alpha \cdot pk^{-tag\times\alpha}$
4: Set $ct^{(2)} := h^\alpha$
5: Set $ct^{(3)} := \mathsf{KDF}(e(H_1(pk), pk)^\alpha) \oplus m$
6: Set $\sigma = \mathsf{Ed25519.Sign}(\mathsf{Ed25519}.sk; \mathsf{Ed25519}.vk, ad, ct^{(1)}, ct^{(2)}, ct^{(3)})$
7: Output $\mathsf{ct} = (ct^{(1)}, ct^{(2)}, ct^{(3)})$ and $\mathsf{auth} = (\mathsf{Ed25519}.vk, \sigma)$

The value $tag = H_F(\mathsf{Ed25519}.vk, ad)$ acts as a per-tx tag that is embedded in $ct^{(1)}$ via the factor $(\tau - tag)$. This ensures that later KZG-based openings for the batch polynomial can cancel the $(\tau - tag)$ term. The mask $\mathsf{KDF}(e(H_1(pk), pk)^\alpha)$ is a one-time pad derived from a pairing, so semantic security follows from the hardness of extracting $\alpha$ without threshold shares. The Ed25519 signature binds $(ct^{(1)}, ct^{(2)}, ct^{(3)}, ad)$ to the sender so invalid ciphertexts are filtered before batching.

### 2.4 Share Partial Decryption

**Algorithm 4** Partial Decryption

1: Remove ciphertexts whose signatures fail verification
2: For remaining set, compute $f(X)$ with roots $f(H_F(\mathsf{Ed25519}.vk, ad)) = 0$ for each tag $tag_j$
3: Compute the KZG commitment of $f$: $\mathsf{com} = g^{\kappa_{\mathsf{ctx}} \cdot f(\tau)}$ via kappa SRS
4: Output share $\mathsf{pd}_i = (H_1(pk) \cdot \mathsf{com}^{-1})^{[\mathsf{trx}.sk]_i}$

The polynomial $f(X)$ is defined so that $f(tag_j) = 0$ for each valid ciphertext tag $tag_j$. The commitment $\mathsf{com} = g^{\kappa_{\mathsf{ctx}} f(\tau)}$ is a KZG evaluation of $f$ at the hidden trapdoor $\tau$, randomized by $\kappa_{\mathsf{ctx}}$ to bind it to the context. Each share $\mathsf{pd}_i$ is computed as a witness for $(H_1(pk) \cdot \mathsf{com}^{-1})$ under the secret share $[\mathsf{trx}.sk]_i$, so any verifier can later check the pairing relation against $pk_i$ and ensure the share is tied to the committed batch.

### 2.5 Share Verification

**Algorithm 5** Verify Partial Decryption

1: Remove ciphertexts whose signatures fail verification
2: Compute $\mathsf{com}$ as in Partial Decryption
3: Accept iff $e(H_1(pk) \cdot \mathsf{com}^{-1}, pk_i) = e(\mathsf{pd}_i, h)$

### 2.6 Combine and Decrypt

**Algorithm 6** Decryption

1: Given $t$ valid shares, interpolate $\omega$ and check $e(H_1(pk) \cdot \mathsf{com}^{-1}, pk) = e(\omega, h)$
2: Compute $\mathsf{com}$ and polynomial $f(X)$ as in Partial Decryption
3: For each $j$ with valid proof:
$q_j(X) = f(X)/(X - H_F(\mathsf{Ed25519}.vk, ad))$
$\pi_j = g^{\kappa_{\mathsf{ctx}} \cdot q_j(\tau)}$
$m_j = ct^{(3)} \oplus \mathsf{KDF}(e(\pi_j, ct^{(1)}) \cdot e(\omega, ct^{(2)}))$
4: Output $m_j$ (or $\perp$ if invalid)

*Interpolation detail.* Let $S \subseteq [n]$ be the index set of $t$ valid shares, and let $\lambda_i^S = \prod_{j \in S \setminus \{i\}} \frac{j}{j-i}$ be the Lagrange coefficients at 0. Each share has the form $\mathsf{pd}_i = (H_1(pk) \cdot \mathsf{com}^{-1})^{[\mathsf{trx}.sk]_i}$, so the interpolated witness is

$$\omega = \prod_{i \in S} \mathsf{pd}_i^{\lambda_i^S} = \left(H_1(pk) \cdot \mathsf{com}^{-1}\right)^{\sum_{i \in S} \lambda_i^S \cdot [\mathsf{trx}.sk]_i} = \left(H_1(pk) \cdot \mathsf{com}^{-1}\right)^{\mathsf{trx}.sk}.$$

Then $e(\omega, h) = e(H_1(pk) \cdot \mathsf{com}^{-1}, h^{\mathsf{trx}.sk}) = e(H_1(pk) \cdot \mathsf{com}^{-1}, pk)$, so $\omega$ satisfies the required pairing equation.

## 3 Tess-Based TrX Implementation (Code-Level View)

This section describes the implemented scheme in this repository, which uses Tess for silent setup and threshold encryption. The implementation keeps KZG batch commitments for integrity, but uses Tess ciphertext and share aggregation. The steps below map directly to code in `src/crypto/*`.

### 3.1 Global Setup and Epoch Setup

**Algorithm 7** Global and Epoch Setup

$\mathsf{pp.params} \leftarrow \mathsf{Tess.ParamGen}(n, t)$
$\mathsf{pp.srs} \leftarrow \mathsf{KZG.Setup}(\text{max\_batch\_size})$
For each context index $\mathsf{ctx} \in \{0, \ldots, C-1\}$:
  sample $\kappa_{\mathsf{ctx}} \xleftarrow{\$} \mathbb{F}$
  set $\mathsf{kappa\_srs}_{\mathsf{ctx}} = \{\kappa_{\mathsf{ctx}} \cdot [g_1^{\tau^k}]\}_{k=0}^{d}$

**Details.** The Tess parameters $\mathsf{pp.params}$ include Lagrange powers for validator indices, enabling silent key generation without interaction. The KZG SRS is generated once at global setup for the maximum batch degree. For each epoch, the implementation derives a set of randomized contexts ("kappa setups") by multiplying the SRS powers by a fresh scalar $\kappa_{\mathsf{ctx}}$. This yields a per-context randomized commitment key while reusing the same $\tau$.

### 3.2 Silent Key Generation and Aggregation

Each validator runs Tess `keygen_single_validator` to obtain $(\mathsf{sk}_i, \mathsf{pk}_i)$ non-interactively. A coordinator aggregates public keys into $\mathsf{PK}$ using Tess deterministic aggregation.

For validator $i \in [n]$, the key generation proceeds as:

1. Sample secret key: $\mathsf{sk}_i \xleftarrow{\$} \mathbb{F}$
2. Compute BLS public key using Lagrange basis evaluation:

$$\mathsf{pk}_i = g_1^{\mathsf{sk}_i \cdot L_i(0)}$$

   where $L_i(X)$ is the $i$-th Lagrange basis polynomial over points $\{1, 2, \ldots, n\}$:

$$L_i(X) = \prod_{j \in [n] \setminus \{i\}} \frac{X - j}{i - j}$$

   Evaluated at $X = 0$:

$$L_i(0) = \prod_{j \in [n] \setminus \{i\}} \frac{0 - j}{i - j} = \prod_{j \in [n] \setminus \{i\}} \frac{-j}{i - j}.$$

3. Non-interactive aggregation:

$$\mathsf{PK} = \prod_{i=1}^{n} \mathsf{pk}_i = \prod_{i=1}^{n} g_1^{\mathsf{sk}_i \cdot L_i(0)} = g_1^{\sum_{i=1}^{n} \mathsf{sk}_i \cdot L_i(0)}$$

Note that here we did not perform any randomization during aggregation. We may require a proof of possession of BLS public keys for security.

**Threshold Property.** This construction ensures that any subset $S \subseteq [n]$ with $|S| = t$ can reconstruct the aggregate secret key via Lagrange interpolation:

$$\mathsf{sk}_{\text{agg}} = \sum_{i=1}^{n} \mathsf{sk}_i \cdot L_i(0) = \sum_{i \in S} \lambda_i^S \cdot \mathsf{sk}_i$$

where $\lambda_i^S = \prod_{j \in S \setminus \{i\}} \frac{j}{j-i}$ are the Lagrange interpolation coefficients.

**Algorithm 8** Phase 1: Silent KeyGen

1: Each validator $i$ samples $\mathsf{sk}_i \xleftarrow{\$} \mathbb{F}$
2: Compute $\mathsf{pk}_i = g_1^{\mathsf{sk}_i \cdot L_i(0)}$ using Tess Lagrange basis in $\mathsf{pp.params}$
3: Coordinator aggregates $\mathsf{PK} = \prod_{i=1}^{n} \mathsf{pk}_i = g_1^{\sum_i \mathsf{sk}_i \cdot L_i(0)}$
4: Store public keys $\{\mathsf{pk}_i\}_{i=1}^{n}$ for share verification

## 3.3 Encryption

The client encrypts using Tess threshold encryption scheme:

$$\mathsf{ct} \leftarrow \mathsf{Tess.Enc}(\mathsf{PK}, \mathsf{pp.params}, t, m)$$

**Tess Ciphertext Structure.** A Tess ciphertext has the form $\mathsf{ct} = (\gamma, \mathsf{payload})$ where:

$$\begin{aligned} r &\xleftarrow{\$} \mathbb{F} \\ \gamma &= h^r \in \mathbb{G}_2 \\ K &= e(g_1, \mathsf{PK})^r = e(g_1, g_1^{\mathsf{sk}_{\text{agg}}})^r = e(g_1, h)^{r \cdot \mathsf{sk}_{\text{agg}}} \in \mathbb{G}_T \\ \mathsf{payload} &= m \oplus \mathsf{KDF}(K) \end{aligned}$$

The ciphertext can only be decrypted with $t$ threshold shares that collectively reconstruct $K$.

**Client Signature Binding.** To prevent malleability and ensure authenticity, the client signs the entire ciphertext:

$$\sigma_{\mathsf{tx}} = \mathsf{Ed25519.Sign}_{\mathsf{sk}_{\text{client}}}(H_{\text{sig}}(\mathsf{ct}.\gamma \,\|\, \mathsf{ct.payload} \,\|\, \mathsf{ad}))$$

This binds the Ed25519 signature to all ciphertext components and associated data, preventing tampering.

**Algorithm 9** Phase 2: Client Encryption

1: Sample $r \xleftarrow{\$} \mathbb{F}$
2: Compute $\gamma = h^r \in \mathbb{G}_2$
3: Compute $K = e(g_1, \mathsf{PK})^r = e(g_1, h)^{r \cdot \mathsf{sk}_{\text{agg}}} \in \mathbb{G}_T$
4: Compute $\mathsf{payload} = m \oplus \mathsf{KDF}(K)$
5: Set $\mathsf{ct} = (\gamma, \mathsf{payload})$
6: Compute $\sigma_{\mathsf{tx}} = \mathsf{Ed25519.Sign}(H_{\text{sig}}(\mathsf{ct} \,\|\, \mathsf{ad}))$
7: **Output:** $(\mathsf{ct}, \mathsf{ad}, \sigma_{\mathsf{tx}}, \mathsf{vk}_{\text{client}})$

## 3.4 Transaction batch's commitment and evaluation proofs

**Polynomial Construction.** Given a batch of $B$ transactions $\{\mathsf{tx}_j\}_{j=0}^{B-1}$, construct polynomial coefficients by hashing:

$$c_j = H_{\text{com}}(\mathsf{tx}_j.\mathsf{ct} \,\|\, \mathsf{tx}_j.\mathsf{ad} \,\|\, \mathsf{ctx}) \in \mathbb{F}$$

Build polynomial with these hash coefficients:

$$p(x) = \sum_{j=0}^{B-1} c_j x^j = c_0 + c_1 x + c_2 x^2 + \cdots + c_{B-1} x^{B-1}$$

**KZG Commitment.** Using the trusted setup SRS $\{g_1^{\tau^j}\}_{j=0}^{d}$ where $\tau \in \mathbb{F}$ is the trapdoor:

$$\mathsf{com} = [p(\tau)]_1 = g_1^{p(\tau)} = g_1^{\sum_{j=0}^{B-1} c_j \tau^j} = \prod_{j=0}^{B-1} (g_1^{\tau^j})^{c_j}$$

**Evaluation Proofs.** For each transaction at index $j$, generate a KZG opening proof that $p(j+1) = c_j$:

1. Compute quotient polynomial via polynomial division:

$$q_j(x) = \frac{p(x) - p(j+1)}{x - (j+1)} = \frac{p(x) - c_j}{x - (j+1)}$$

2. Commit to quotient:

$$\pi_j = [q_j(\tau)]_1 = g_1^{q_j(\tau)}$$

**Verification Equation.** A verifier checks using pairings:

$$e(\mathsf{com} - g_1^{c_j}, h) \stackrel{?}{=} e(\pi_j, h^\tau \cdot h^{-(j+1)}) = e(\pi_j, h^{\tau-(j+1)})$$

**Context Binding with $\kappa$ Randomization.** This is used to prevent replay attacks across different batches:

$$\mathsf{com}_{\mathsf{ctx}} = g_1^{\kappa_{\mathsf{ctx}} \cdot p(\tau)} = \prod_{j=0}^{B-1} (\kappa_{\mathsf{ctx}} \cdot g_1^{\tau^j})^{c_j}$$

where $\kappa_{\mathsf{ctx}} \in \mathbb{F}$ is context-specific randomness from the kappa setup.

**Algorithm 10** Phase 3: TransactionBatch Commit and Proofs

**for** $j = 0$ to $B-1$ **do**
  $c_j \leftarrow H_{\text{com}}(\mathsf{tx}_j.\mathsf{ct} \,\|\, \mathsf{tx}_j.\mathsf{ad} \,\|\, \mathsf{ctx})$
**end for**
Build polynomial $p(x) = \sum_{j=0}^{B-1} c_j x^j$ from hash coefficients
Retrieve $\kappa$ SRS for context $\mathsf{ctx}$: $\{\kappa_{\mathsf{ctx}} \cdot g_1^{\tau^j}\}_{j=0}^{d}$
Compute commitment:
  $\mathsf{com} = \prod_{j=0}^{B-1} (\kappa_{\mathsf{ctx}} \cdot g_1^{\tau^j})^{c_j} = g_1^{\kappa_{\mathsf{ctx}} \cdot p(\tau)}$
**for** $j = 0$ to $B-1$ **do**
  Compute quotient $q_j(x) = \frac{p(x)-c_j}{x-(j+1)}$ via polynomial division
  Compute proof $\pi_j \leftarrow (g_1^{\kappa_{\mathsf{ctx}} \cdot \tau^0})^{q_j(\tau, \text{coeffs})} = g_1^{\kappa_{\mathsf{ctx}} \cdot q_j(\tau)}$
  **Verification:** $e(\mathsf{com} \cdot g_1^{-\kappa_{\mathsf{ctx}} \cdot c_j}, h) \stackrel{?}{=} e(\pi_j, h^{\tau-(j+1)})$
**end for**
**Output:** Commitment $\mathsf{com}$ and proofs $\{\pi_j\}_{j=0}^{B-1}$

## 3.5 Partial Decryption

Each validator produces a Tess partial decryption by exponentiating the ciphertext component:

$$\mathsf{pd}_i = \mathsf{ct}.\gamma^{\mathsf{sk}_i} = (h^r)^{\mathsf{sk}_i} = h^{r \cdot \mathsf{sk}_i} \in \mathbb{G}_2$$

This is the core Tess threshold decryption operation. The partial decryption is bound to the specific transaction via the ciphertext $\gamma$ component.

**Threshold Decryption Correctness.** Recall the Tess ciphertext encrypts message $m$ as:

$$\mathsf{payload} = m \oplus \mathsf{KDF}(e(g_1, h)^{r \cdot \mathsf{sk}_{\text{agg}}})$$

Given $t$ partial decryptions $\{\mathsf{pd}_i = h^{r \cdot \mathsf{sk}_i}\}_{i \in S}$ where $|S| = t$, reconstruct:

$$h^{r \cdot \mathsf{sk}_{\text{agg}}} = \prod_{i \in S} \mathsf{pd}_i^{\lambda_i^S} = \prod_{i \in S} (h^{r \cdot \mathsf{sk}_i})^{\lambda_i^S} = h^{r \cdot \sum_{i \in S} \lambda_i^S \cdot \mathsf{sk}_i}$$

where $\lambda_i^S = \prod_{j \in S \setminus \{i\}} \frac{j}{j-i}$ are Lagrange coefficients satisfying:

$$\sum_{i \in S} \lambda_i^S \cdot \mathsf{sk}_i \cdot L_i(0) = \mathsf{sk}_{\text{agg}}$$

Then recover the encryption key:

$$K = e(g_1, h^{r \cdot \mathsf{sk}_{\text{agg}}}) = e(g_1, h)^{r \cdot \mathsf{sk}_{\text{agg}}}$$

and decrypt:

$$m = \mathsf{payload} \oplus \mathsf{KDF}(K)$$

**Optional BLS Signatures.** Validators may optionally sign their shares for additional authenticity. This is implemented in the SDK's `generate_signed_partial_decryption` but is not mandatory in the core protocol.

$$\sigma_i^{\text{BLS}} = \mathsf{BLS.Sign}_{\mathsf{sk}_i^{\text{BLS}}}(\mathsf{ctx} \,\|\, \mathsf{tx\_index} \,\|\, \mathsf{pd}_i \,\|\, \mathsf{com} \,\|\, \mathsf{ad})$$

**Algorithm 11** Phase 4: Share Generation and partial decryption

1: **Input:** Ciphertext $\mathsf{ct} = (\gamma, \mathsf{payload})$, secret share $\mathsf{sk}_i$
2: Compute $\mathsf{pd}_i \leftarrow \gamma^{\mathsf{sk}_i} = (h^r)^{\mathsf{sk}_i} = h^{r \cdot \mathsf{sk}_i}$ ▷ Tess partial decryption
3: Attach metadata: $(\mathsf{validator\_id}_i, \mathsf{ctx}, \mathsf{tx\_index})$ to $\mathsf{pd}_i$
4: **Optional:** Sign $\sigma_i^{\text{BLS}} \leftarrow \mathsf{BLS.Sign}(\mathsf{ctx} \,\|\, \mathsf{tx\_index} \,\|\, \mathsf{pd}_i \,\|\, \mathsf{com} \,\|\, \mathsf{ad})$
5: If signed, attach signature $\sigma_i^{\text{BLS}}$ and verification key $\mathsf{vk}_i^{\text{BLS}}$ to share
6: **Output:** $(\mathsf{pd}_i, \mathsf{validator\_id}_i, \mathsf{ctx}, \mathsf{tx\_index}, \sigma_i^{\text{BLS}}$ (if signed))

## 3.6 Share Verification

Before combining shares, validators or combiners can verify that a partial decryption is correctly formed using a pairing check. This ensures malicious validators cannot provide invalid shares.

**Verification Equation.** For a valid partial decryption, we require:

$$\mathsf{pd}_i = \gamma^{\mathsf{sk}_i} = (h^r)^{\mathsf{sk}_i}$$

Given the validator's public key $\mathsf{pk}_i = g_1^{\mathsf{sk}_i \cdot L_i(0)}$, we verify using bilinearity:

$$e(\mathsf{pk}_i, \gamma) \stackrel{?}{=} e(g_1, \mathsf{pd}_i)$$

**Correctness** If $\mathsf{pd}_i$ is correctly formed:

$$\begin{aligned} e(\mathsf{pk}_i, \gamma) &= e(g_1^{\mathsf{sk}_i \cdot L_i(0)}, h^r) \\ &= e(g_1, h)^{r \cdot \mathsf{sk}_i \cdot L_i(0)} \\ e(g_1, \mathsf{pd}_i) &= e(g_1, h^{r \cdot \mathsf{sk}_i \cdot L_i(0)}) \\ &= e(g_1, h)^{r \cdot \mathsf{sk}_i \cdot L_i(0)} \end{aligned}$$

## 3.7 Combine and Decrypt

Given $t$ partial decryptions for each transaction, the combiner uses Tess Lagrange interpolation to reconstruct the decryption key.

**Lagrange Interpolation in the Exponent.** Given a threshold set $S \subseteq [n]$ with $|S| = t$ and partial decryptions $\{\mathsf{pd}_i = h^{r \cdot \mathsf{sk}_i}\}_{i \in S}$:

1. Compute Lagrange coefficients for interpolation at point 0:

$$\lambda_i^S = \prod_{j \in S \setminus \{i\}} \frac{j}{j-i} \in \mathbb{F}$$

2. Aggregate partial decryptions via exponentiation:

$$D = \prod_{i \in S} \mathsf{pd}_i^{\lambda_i^S} = \prod_{i \in S} (h^{r \cdot \mathsf{sk}_i})^{\lambda_i^S} = h^{r \cdot \sum_{i \in S} \lambda_i^S \cdot \mathsf{sk}_i} = h^{r \cdot \mathsf{sk}_{\text{agg}}}$$

3. Compute decryption key via pairing:

$$K = e(g_1, D) = e(g_1, h^{r \cdot \mathsf{sk}_{\text{agg}}}) = e(g_1, h)^{r \cdot \mathsf{sk}_{\text{agg}}}$$

4. Decrypt the payload via XOR:

$$m = \mathsf{ct.payload} \oplus \mathsf{KDF}(K)$$

**Algorithm 12** Phase 5: Aggregate Decrypt

**Input:** Batch context with $\{\mathsf{tx}_j\}_{j=0}^{B-1}$, commitment $\mathsf{com}$, proofs $\{\pi_j\}$, shares $\{\mathsf{pd}_{i,j}\}$
Verify KZG proofs: $\forall j, e(\mathsf{com} \cdot g_1^{-\kappa_{\text{ctx}} \cdot c_j}, h) \stackrel{?}{=} e(\pi_j, h^{\tau-(j+1)})$
**if** signed shares **then**
    Verify BLS signatures: $\forall i \in S, \mathsf{BLS.Verify}(\mathsf{vk}_i^{\text{BLS}}, \sigma_i^{\text{BLS}}, \cdots)$
**end if**
**for** each transaction $j \in [B]$ **do**
    Collect threshold set $S_j \subseteq [n]$ with $|S_j| \geq t$ for transaction $j$
    Compute Lagrange coefficients: $\lambda_i^{S_j} = \prod_{k \in S_j \setminus \{i\}} \frac{k}{k-i}$
    Aggregate: $D_j \leftarrow \prod_{i \in S_j} (\mathsf{pd}_{i,j})^{\lambda_i^{S_j}} = h^{r_j \cdot \mathsf{sk}_{\text{agg}}}$
    Compute key: $K_j \leftarrow e(g_1, D_j) = e(g_1, h)^{r_j \cdot \mathsf{sk}_{\text{agg}}}$
    Decrypt: $m_j \leftarrow \mathsf{tx}_j.\mathsf{payload} \oplus \mathsf{KDF}(K_j)$
**end for**
**Output:** Plaintexts $\{m_0, m_1, \ldots, m_{B-1}\}$

# 4 Comparison

The implementation uses a hybrid approach that combines Tess threshold encryption with TrX's KZG batching layer. The key differences are:

- **Original TrX Paper:** Uses custom witness encryption based on Pairing Product Equations (PPEs). The CRS contains $\{g^{\kappa_{\text{ctx}} \tau^j}\}$ and the batch polynomial $f$ is defined by roots $H_F(\mathsf{Ed25519}.vk, ad)$. Shares are bound to the commitment: $\mathsf{pd}_i = (H_1(pk) \cdot \mathsf{com}^{-1})^{[\mathsf{trx}.sk]_i}$ with $\mathsf{com} = g^{\kappa_{\text{ctx}} f(\tau)}$.
- **This Implementation:** Uses Tess threshold encryption for the core encryption/decryption operations. Shares are computed as $\mathsf{pd}_i = \mathsf{ct}.\gamma^{\mathsf{sk}_i}$ (Tess partial decryption). Batch integrity is enforced separately via KZG commitments over hash coefficients $H_{\text{com}}(\mathsf{ct}, \mathsf{ad}, \mathsf{ctx})$.