Unified Threshold Cryptography for Omnichain Asset Custody: FROST, CGGMP21, and Taproot Integration

Zach Kelling

Lux Industries

2021 (Revised August 2025)

Abstract

We present a unified threshold cryptographic framework for omnichain asset custody deployed in the Lux Teleport bridge protocol. The framework integrates two complementary threshold signature schemes—FROST (Flexible Round-Optimized Schnorr Threshold) for Ed25519 and BIP-340 Taproot chains, and CGGMP21 for threshold ECDSA on secp256k1 chains—to provide native signature verification on every major blockchain without on-chain multisig overhead. FROST covers 12 chain families (Solana, TON, Sui, Aptos, Cosmos, Polkadot, NEAR, Stellar, Cardano, Algorand, Tezos, Bitcoin Taproot) with 2-round signing producing standard 64-byte Schnorr signatures. CGGMP21 covers 5 chain families (Ethereum and all EVM chains, TRON, XRPL, StarkNet, legacy Bitcoin) with identifiable abort producing standard 65-byte ECDSA signatures. Both protocols share a common distributed key generation (DKG) phase with no trusted dealer, a proactive key refresh mechanism that rotates shares without changing the public key, and a 7-day timelocked signer rotation with on-chain visibility. We prove unforgeability under $t-1$ static corruption for both protocols in the random oracle model, give a formal composition theorem establishing that the dual-protocol system maintains security when sharing infrastructure, and present gas cost measurements for signature verification across 17 chain families. The system is implemented in Go at `github.com/luxfi/threshold` and `github.com/luxfi/mpc`, deployed in production securing cross-chain assets on Lux Teleport.

1 Introduction

Cross-chain bridges must hold custody of assets on source chains while issuing representations on destination chains. The security of billions of dollars in locked value depends on how the custodial key is managed. Three approaches dominate production deployments:

1. **Multisig**: k-of-n on-chain multisignature. Transparent but expensive (linear gas cost in k) and chain-specific (not all chains support native multisig).
2. **Trusted committee**: A centralized signer or small committee holds the private key. Cheap verification but catastrophic failure mode.
3. **Threshold signatures**: t-of-n MPC-based signing produces a single standard signature indistinguishable from a single-signer output. Constant verification cost. No on-chain multisig infrastructure required.

Threshold signatures are the only approach that achieves both constant on-chain cost and distributed trust. However, the blockchain ecosystem uses two incompatible signature families: ECDSA (secp256k1) for Ethereum, Bitcoin legacy, TRON, and XRPL; and Schnorr/EdDSA (Ed25519 or BIP-340) for Solana, Cardano, TON, Polkadot, Cosmos, and Bitcoin Taproot. No single threshold protocol covers both.

Contribution. We present a dual-protocol threshold framework:

- **FROST** [1] for Schnorr/EdDSA chains: 2-round threshold signing, 64-byte signatures, compatible with Ed25519 and BIP-340.
- **CGGMP21** [2] for ECDSA chains: threshold ECDSA with identifiable abort, 65-byte signatures, compatible with secp256k1.
- **Shared infrastructure**: common DKG, key refresh, signer rotation, and monitoring layer for both protocols.
- **Formal composition**: proof that sharing infrastructure between the two protocols does not degrade security of either.

The framework is deployed in Lux Teleport, securing cross-chain transfers across 18 native bridge programs and 270 registered chain IDs. On-chain verification uses a single MPC group address—the aggregate public key from DKG—rather than individual signer keys: $\texttt{ecrecover}(\text{digest}, \sigma) = \texttt{mpcGroupAddress}$. Individual signer identities exist for accountability and identifiable abort only.

1.1 Notation

We use the following notation throughout. Let $\mathbb{G}$ denote a cyclic group of prime order q with generator G. For a set S of signers with $|S| = t$, the Lagrange coefficient of party i in S is $\lambda_{i,S} = \prod_{j \in S, j \neq i} \frac{j}{j-i} \bmod q$. We write $H : \{0,1\}^* \to \mathbb{F}_q$ for a collision-resistant hash function modeled as a random oracle. $[n]$ denotes $\{1, \ldots, n\}$. $x \xleftarrow{\$} S$ denotes uniform sampling from set S.

2 System Model and Definitions

Definition 2.1 (Threshold Signature Scheme)**.** *A (t, n)-threshold signature scheme* TSS *consists of four algorithms:*

- $\mathsf{DKG}(1^\lambda, t, n) \to (\mathsf{pk}, \{\mathsf{sk}_i\}_{i=1}^n)$*: Distributed key generation with no trusted dealer.*
- $\mathsf{Sign}(m, S, \{\mathsf{sk}_i\}_{i \in S}) \to \sigma$*: Threshold signing with $|S| \geq t$.*
- $\mathsf{Verify}(m, \sigma, \mathsf{pk}) \to \{0,1\}$*: Standard single-key verification.*
- $\mathsf{Refresh}(\{\mathsf{sk}_i\}_{i=1}^n) \to \{\mathsf{sk}_i'\}_{i=1}^n$*: Proactive share refresh preserving* pk*.*

Definition 2.2 (Unforgeability under $t-1$ Corruption)**.** TSS *is unforgeable under static $t-1$ corruption if for all PPT adversaries $\mathcal{A}$ controlling at most $t-1$ parties:*

$$\Pr\left[\mathsf{Verify}(m^*, \sigma^*, \mathsf{pk}) = 1 \;\middle|\; \begin{array}{l} (\mathsf{pk}, \{\mathsf{sk}_i\}) \leftarrow \mathsf{DKG}(1^\lambda, t, n) \\ (m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(\cdot)}(\mathsf{pk}, \{\mathsf{sk}_i\}_{i \in C}) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

where $|C| \leq t-1$ and m^* was not queried to Sign*.*

Definition 2.3 (Identifiable Abort)**.** TSS *satisfies identifiable abort if whenever* Sign *fails to produce a valid signature, the honest parties can identify at least one malicious party from the protocol transcript with overwhelming probability.*

2.1 Adversary Model

We consider a static adversary $\mathcal{A}$ that corrupts up to $t-1$ of n parties before the protocol begins. Corrupted parties may deviate arbitrarily from the protocol (Byzantine faults). Communication is authenticated: $\mathcal{A}$ can delay but not forge messages between honest parties. We model hash functions as random oracles.

2.2 Chain Coverage Model

Let $\mathcal{C}_{\text{FROST}}$ and $\mathcal{C}_{\text{ECDSA}}$ partition the set of supported blockchains by their native signature verification:

$$\mathcal{C}_{\text{FROST}} = \{\text{Ed25519 chains}\} \cup \{\text{BIP-340 chains}\}$$
$$\mathcal{C}_{\text{ECDSA}} = \{\text{secp256k1 chains}\}$$

For any chain $c \in \mathcal{C}_{\text{FROST}} \cup \mathcal{C}_{\text{ECDSA}}$, Lux Teleport holds assets under a threshold public key pk_c such that on-chain verification uses the chain's native Verify function—no custom verifier contracts, no multisig wrappers.

3 FROST Protocol

FROST (Flexible Round-Optimized Schnorr Threshold signatures) [1] provides (t, n)-threshold Schnorr signatures in two rounds. We describe the protocol over a generic group $\mathbb{G}$ and specialize to Ed25519 and BIP-340 in Section 5.

3.1 Distributed Key Generation

FROST DKG uses Pedersen's verifiable secret sharing [6] with Feldman commitments [7].

Algorithm 1 FROST Distributed Key Generation

Require: Threshold t, participants n, generator G

Ensure: Public key Y, shares $\{s_i\}_{i=1}^n$

for each party $i \in [n]$ **do**
  $a_{i,0} \xleftarrow{\$} \mathbb{F}_q$ ▷ secret coefficient
  $a_{i,k} \xleftarrow{\$} \mathbb{F}_q$ for $k \in [t-1]$
  Define $f_i(x) = \sum_{k=0}^{t-1} a_{i,k} x^k$
  Broadcast $C_{i,k} = a_{i,k} \cdot G$ for $k = 0, \ldots, t-1$
  Send $f_i(j)$ to party j over secure channel
end for
for each party $j \in [n]$ **do**
  Verify: $f_i(j) \cdot G \stackrel{?}{=} \sum_{k=0}^{t-1} j^k \cdot C_{i,k}$
  Compute share $s_j = \sum_{i=1}^n f_i(j) \bmod q$
  Compute public key $Y = \sum_{i=1}^n C_{i,0}$
end for
return $(Y, \{s_i\}_{i=1}^n)$

Lemma 3.1 (DKG Correctness)**.** *If all parties are honest, then s_i is a valid Shamir share of $s = \sum_{i=1}^{n} a_{i,0}$ under threshold t, and $Y = s \cdot G$.*

Proof. Define $f(x) = \sum_{i=1}^{n} f_i(x)$. Then $f(0) = \sum_{i=1}^{n} a_{i,0} = s$ and $s_j = f(j)$ for each j. Since each f_i has degree $t-1$, f has degree $t-1$, so $\{(j, s_j)\}_{j=1}^{n}$ are Shamir shares with threshold t. The public key satisfies $Y = \sum_{i=1}^{n} C_{i,0} = \sum_{i=1}^{n} a_{i,0} \cdot G = s \cdot G$. □

3.2 Two-Round Signing

Algorithm 2 FROST Threshold Signing

Require: Message m, signer set S with $|S| \geq t$, shares $\{s_i\}_{i \in S}$, public key Y
Ensure: Signature $\sigma = (R, z)$

Round 1: Commitment

for each signer $i \in S$ **do**
  $d_i, e_i \xleftarrow{\$} \mathbb{F}_q$
  $(D_i, E_i) \leftarrow (d_i \cdot G,\ e_i \cdot G)$
  Broadcast (i, D_i, E_i)
end for

Round 2: Response

$B = \{(i, D_i, E_i)\}_{i \in S}$ ▷ commitment list
for each signer $i \in S$ **do**
  $\rho_i \leftarrow H_1(i, m, B)$ ▷ binding factor
end for
$R \leftarrow \sum_{i \in S} (D_i + \rho_i \cdot E_i)$
$c \leftarrow H_2(R, Y, m)$ ▷ Schnorr challenge
for each signer $i \in S$ **do**
  $z_i \leftarrow d_i + \rho_i \cdot e_i + \lambda_{i,S} \cdot s_i \cdot c$
  Broadcast z_i
end for
$z \leftarrow \sum_{i \in S} z_i$
Verify: $z \cdot G \stackrel{?}{=} R + c \cdot Y$
return $\sigma = (R, z)$

Theorem 3.2 (FROST Correctness)**.** *If all signers in S follow the protocol, the resulting signature (R, z) satisfies the Schnorr verification equation $z \cdot G = R + c \cdot Y$.*

Proof.

$$\begin{aligned} z \cdot G &= \left(\sum_{i \in S} z_i \right) \cdot G \\ &= \sum_{i \in S} (d_i + \rho_i e_i + \lambda_{i,S} s_i c) \cdot G \\ &= \sum_{i \in S} (D_i + \rho_i E_i) + c \sum_{i \in S} \lambda_{i,S} s_i \cdot G \\ &= R + c \cdot s \cdot G = R + c \cdot Y \end{aligned}$$

where the last equality uses the Lagrange interpolation property $\sum_{i \in S} \lambda_{i,S} s_i = s$. □

3.3 Chain Coverage

FROST produces signatures compatible with the following chain families:

Chain Family	Curve	Sig. Format	Size
Solana	Ed25519	(R, s)	64 B
TON	Ed25519	(R, s)	64 B
Sui (Ed25519 mode)	Ed25519	(R, s)	64 B
Aptos	Ed25519	(R, s)	64 B
Cosmos/IBC	Ed25519	(R, s)	64 B
Polkadot/Substrate	Sr25519*	(R, s)	64 B
NEAR	Ed25519	(R, s)	64 B
Stellar	Ed25519	(R, s)	64 B
Cardano	Ed25519	(R, s)	64 B
Algorand	Ed25519	(R, s)	64 B
Tezos	Ed25519	(R, s)	64 B
Bitcoin Taproot	secp256k1†	(R_x, s)	64 B

Table 1: FROST chain coverage. *Polkadot Sr25519 uses Ristretto Schnorr; FROST adapts via the Ristretto encoding. †BIP-340 uses secp256k1 with Schnorr verification; see Section 5.

4 CGGMP21 Protocol

CGGMP21 [2] provides (t, n)-threshold ECDSA with identifiable abort. The protocol separates into an offline presigning phase and a fast online signing phase.

4.1 Key Generation

CGGMP21 DKG extends Feldman VSS with Paillier encryption and zero-knowledge proofs

to enable multiplicative-to-additive (MtA) share conversion during signing.

Algorithm 3 CGGMP21 Distributed Key Generation

Require: Threshold t, participants n, generator G on secp256k1
Ensure: Public key Q, shares $\{x_i\}_{i=1}^n$, Paillier keys $\{(N_i, p_i, q_i)\}$

for each party $i \in [n]$ **do**
  $x_i^{(0)} \xleftarrow{\$} \mathbb{F}_q$, define VSS polynomial f_i of degree $t-1$
  Generate safe primes p_i, q_i; set $N_i = p_i q_i$
  Broadcast $X_i = x_i^{(0)} \cdot G$ with commitment
  Prove Π_{sch}: knowledge of $x_i^{(0)}$ w.r.t. X_i
  Prove Π_{mod}: N_i is a Blum integer
  Prove Π_{prm}: ring-Pedersen parameters are well-formed
  Send $f_i(j)$ encrypted under party j's Paillier key
end for
for each party $j \in [n]$ **do**
  Verify all proofs $\Pi_{\mathsf{sch}}, \Pi_{\mathsf{mod}}, \Pi_{\mathsf{prm}}$
  Decrypt and verify Feldman commitments
  $x_j \leftarrow \sum_{i=1}^n f_i(j) \bmod q$
end for
$Q \leftarrow \sum_{i=1}^n X_i$
return $(Q, \{x_i\}, \{N_i, p_i, q_i\})$

4.2 Presigning (Offline Phase)

The presigning protocol generates nonce shares and MtA conversion values that enable fast online signing. Each presignature is single-use.

Algorithm 4 CGGMP21 Presigning

Require: Signer set S with $|S| \geq t$, shares $\{x_i\}_{i \in S}$
Ensure: Presignature record (R, k_i, χ_i) for each $i \in S$

for each signer $i \in S$ **do**
  $k_i, \gamma_i \xleftarrow{\$} \mathbb{F}_q$
  $K_i \leftarrow k_i \cdot G$, $\Gamma_i \leftarrow \gamma_i \cdot G$
  Broadcast K_i with Π_{enc} proof
end for
MtA Conversion:
for each pair $(i, j) \in S \times S$, $i \neq j$ **do**
  Run $\mathsf{MtA}(k_i, \gamma_j) \to (\alpha_{ij}, \beta_{ij})$
    such that $\alpha_{ij} + \beta_{ji} = k_i \cdot \gamma_j$
  Run $\mathsf{MtA}(k_i, x_j) \to (\hat{\alpha}_{ij}, \hat{\beta}_{ij})$
end for
for each signer $i \in S$ **do**
  $\delta_i \leftarrow k_i \gamma_i + \sum_{j \neq i} (\alpha_{ij} + \beta_{ji})$
  $\chi_i \leftarrow k_i x_i + \sum_{j \neq i} (\hat{\alpha}_{ij} + \hat{\beta}_{ji})$
end for
$\delta \leftarrow \sum_{i \in S} \delta_i$ ▷ reconstruct $k \cdot \gamma$
$\Gamma \leftarrow \sum_{i \in S} \Gamma_i$ ▷ reconstruct $\gamma \cdot G$
$R \leftarrow \delta^{-1} \cdot \Gamma, \quad r \leftarrow R_x \bmod q$
return $(R, r, \{k_i, \chi_i\}_{i \in S})$

4.3 Online Signing

Given a presignature and message m, online signing requires a single round:

Algorithm 5 CGGMP21 Online Signing

Require: Message m, presignature $(R, r, \{k_i, \chi_i\}_{i \in S})$
Ensure: ECDSA signature (r, s)

for each signer $i \in S$ **do**
  $\sigma_i \leftarrow k_i \cdot H(m) + r \cdot \chi_i$
  Broadcast σ_i
end for
$s \leftarrow \sum_{i \in S} \sigma_i \bmod q$
Normalize: if $s > q/2$ then $s \leftarrow q - s$ ▷ low-s form
Verify: ECDSA.Verify$(m, (r, s), Q) \stackrel{?}{=} 1$
if verification fails **then**
  Identify aborter via $\Pi_{\mathsf{log}*}$ proofs ▷ identifiable abort
end if
return (r, s)

Theorem 4.1 (CGGMP21 Correctness)**.** *If all signers follow the protocol, then* $s = k^{-1}(H(m) + r \cdot x) \bmod q$ *where* $k = \sum_{i \in S} k_i \cdot \lambda_{i,S}$ *and* x *is the secret key.*

Proof. The MtA conversion ensures $\sum_{i \in S} \chi_i = k \cdot x$ and $\sum_{i \in S} \delta_i = k \cdot \gamma$ where $\gamma = \sum_{i \in S} \gamma_i \cdot \lambda_{i,S}$. Then:

$$\begin{aligned} s &= \sum_{i \in S} \sigma_i = \sum_{i \in S} (k_i \cdot H(m) + r \cdot \chi_i) \\ &= k \cdot H(m) + r \cdot k \cdot x = k(H(m) + r \cdot x) \end{aligned}$$

Since $R = (k \cdot \gamma)^{-1} \cdot (\gamma \cdot G) = k^{-1} \cdot G$, the value $r = R_x$ corresponds to $k^{-1} \cdot G$, and $s = k(H(m) + rx)$. After division by k in the verification equation, this yields a valid ECDSA signature. □

4.4 Chain Coverage

CGGMP21 produces standard ECDSA signatures for the following chain families:

Chain Family	Curve	Sig. Format	Size
Ethereum / EVM	secp256k1	(v, r, s)	65 B
TRON	secp256k1	(v, r, s)	65 B
XRPL	secp256k1	DER-encoded	70–72 B
StarkNet*	secp256k1	(r, s)	64 B
Bitcoin (legacy)	secp256k1	DER-encoded	70–72 B

Table 2: CGGMP21 chain coverage. *StarkNet account abstraction allows arbitrary signature schemes; the Lux Teleport StarkNet adapter uses secp256k1 ECDSA via a signature verifier account.

5 Bitcoin Taproot Integration

Bitcoin Taproot (BIP-341 [4]) uses Schnorr signatures per BIP-340 [3] over secp256k1 with x-only public keys. This requires specific adaptations to the FROST protocol.

5.1 BIP-340 Schnorr Signatures

BIP-340 defines Schnorr verification as follows. Given message m, signature (R_x, s) where R_x is the 32-byte x-coordinate of R, and x-only public key P_x:

1. Lift P_x to point P with even y-coordinate.
2. Lift R_x to point R with even y-coordinate.
3. Compute $e = H_{\text{BIP0340/challenge}}(R_x \| P_x \| m)$.
4. Accept iff $s \cdot G = R + e \cdot P$.

5.2 FROST KeygenTaproot

Standard FROST keygen produces a full public key $Y = (Y_x, Y_y)$. BIP-340 requires x-only keys (implicitly even y-coordinate). We define KeygenTaproot:

Algorithm 6 FROST KeygenTaproot

Require: Standard FROST DKG output $(Y, \{s_i\}_{i=1}^n)$
Ensure: x-only public key P_x, adjusted shares $\{s'_i\}$

if Y has odd y-coordinate **then**
  $s'_i \leftarrow q - s_i$ for all $i \in [n]$ ▷ negate all shares
  $P \leftarrow -Y$ ▷ now has even y-coordinate
else
  $s'_i \leftarrow s_i$ for all i
  $P \leftarrow Y$
end if
$P_x \leftarrow$ x-coordinate of P ▷ 32 bytes
return $(P_x, \{s'_i\}_{i=1}^n)$

5.3 FROST SignTaproot

Signing must also ensure R has even y-coordinate per BIP-340:

Algorithm 7 FROST SignTaproot

Require: Message m, signer set S, adjusted shares $\{s'_i\}$, x-only key P_x
Ensure: BIP-340 signature (R_x, z)

Execute FROST Rounds 1–2 (Algorithm 2) to obtain (R, z)
if R has odd y-coordinate **then**
  $R \leftarrow -R$
  $z \leftarrow q - z$ ▷ negate response
end if
$R_x \leftarrow$ x-coordinate of R
$e \leftarrow H_{\text{BIP0340/challenge}}(R_x \| P_x \| m)$
Verify: $z \cdot G \stackrel{?}{=} R + e \cdot P$
return (R_x, z) ▷ standard 64-byte BIP-340 signature

Proposition 5.1 (Taproot Compatibility)**.** *Signatures produced by Algorithm 7 pass BIP-340 verification by any standard Bitcoin Taproot verifier.*

Proof. The negation steps ensure both R and P have even y-coordinates, matching the BIP-340 convention. The challenge hash uses R_x and P_x per the BIP-340 tagged hash specification. The verification equation $z \cdot G = R + e \cdot P$ is the standard Schnorr equation. Since the output is a pair (R_x, z) of two 32-byte values, it conforms to the 64-byte BIP-340 signature format. □

5.4 Tapscript Integration

For Taproot outputs using Tapscript (BIP-342 [5]), the threshold public key P_x can be placed in the key-path spend, enabling direct Schnorr spending without revealing the threshold structure. This provides:

- **Privacy**: On-chain, the UTXO is indistinguishable from a single-signer Taproot output. The threshold structure is never revealed.
- **Efficiency**: Key-path spends are the cheapest Bitcoin transaction type (1 input witness: 64-byte signature).
- **Fallback**: Script-path spends can encode alternative spending conditions (e.g., time-locked recovery) in the Tapscript tree.

6 Key Management

Both FROST and CGGMP21 share a common key management layer providing distributed key generation, proactive refresh, and signer rotation.

6.1 Distributed Key Generation

Both protocols use Feldman VSS [7] for DKG with no trusted dealer. The shared structure is:

1. Each party i samples a random polynomial f_i of degree $t-1$ and broadcasts Feldman commitments $C_{i,k} = a_{i,k} \cdot G$.
2. Parties exchange encrypted shares and verify against commitments.
3. The public key is $\mathsf{pk} = \sum_{i=1}^{n} C_{i,0}$.

The difference is that CGGMP21 additionally requires Paillier key generation and zero-knowledge proofs ($\Pi_{\mathsf{mod}}, \Pi_{\mathsf{prm}}, \Pi_{\mathsf{sch}}$) for the MtA sub-protocol.

Remark 6.1. *DKG is the most computationally expensive phase. For FROST, DKG completes in 12–18ms for a 3-of-5 configuration. For CGGMP21, Paillier safe prime generation dominates at 800–1200ms for 2048-bit moduli. DKG runs once per key; its cost is amortized over all subsequent signing operations.*

6.2 Proactive Key Refresh

Key refresh replaces all shares without changing the public key, limiting the window during which a compromised share is useful.

Algorithm 8 Proactive Key Refresh

Require: Current shares $\{s_i\}_{i=1}^{n}$, threshold t
Ensure: New shares $\{s'_i\}_{i=1}^{n}$ with same public key

1: **for** each party $i \in [n]$ **do**
2: Sample $g_i(x)$ of degree $t-1$ with $g_i(0) = 0$
3: Broadcast commitments $G_{i,k} = g_{i,k} \cdot G$ for $k = 0, \ldots, t-1$
4: Verify $G_{i,0} = \mathcal{O}$ (identity point) ▷ ensures zero constant
5: Send $g_i(j)$ to party j
6: **end for**
7: **for** each party $j \in [n]$ **do**
8: Verify commitments: $g_i(j) \cdot G = \sum_k j^k \cdot G_{i,k}$
9: $s'_j \leftarrow s_j + \sum_{i=1}^{n} g_i(j) \bmod q$
10: **end for**

Lemma 6.1 (Refresh Preserves Public Key)**.** *After refresh, the public key is unchanged:* $\mathsf{pk}' = \mathsf{pk}$.

Proof. The new secret is $s' = s + \sum_{i=1}^{n} g_i(0) = s + 0 = s$. Therefore $\mathsf{pk}' = s' \cdot G = s \cdot G = \mathsf{pk}$. □

Lemma 6.2 (Forward Security)**.** *An adversary who compromises $t-1$ shares from epoch e and $t-1$ shares from epoch $e+1$ cannot reconstruct the secret key, provided at least one honest party participated in the refresh.*

Proof. Shares from different epochs lie on different polynomials. The adversary holds $t-1$ points on each of two degree-$(t-1)$ polynomials sharing the same constant term. Reconstructing s from $t-1$ points on a single polynomial is infeasible by Shamir's information-theoretic security. Combining shares across epochs does not help because the refresh polynomials g_i are independently random—the adversary cannot correlate shares across epochs without knowing at least one honest party's refresh polynomial. □

6.3 Signer Rotation

Signer rotation changes the set of authorized signers (e.g., replacing a decommissioned node). Unlike key refresh, rotation changes the set $[n]$ itself.

Definition 6.1 (Signer Rotation)**.** *A signer rotation from set $\mathcal{P} = \{P_1, \ldots, P_n\}$ to set $\mathcal{P}' = \{P'_1, \ldots, P'_{n'}\}$ with overlap $\mathcal{P} \cap \mathcal{P}' \geq t$ proceeds as:*

1. ***Announce** (on-chain): New MPC group address is posted with a configurable rotation delay (default 24h, governor-settable $\in [0, 7d]$).*
2. ***Reshare**: After delay expiry, t members of $\mathcal{P}$ run an LSS dynamic resharing protocol to generate shares for $\mathcal{P}'$ under the same public key (0.14ms for $3 \to 5$, 3.5ms for $50 \to 100$).*
3. ***Activate**: $\mathcal{P}'$ proves liveness by producing a threshold signature, and the on-chain registry transitions to the new group address.*
4. ***Revoke**: Old shares from $\mathcal{P} \setminus \mathcal{P}'$ are zeroed.*

The rotation delay is an operational parameter, not a security invariant—the MPC threshold *is* the trust model. Setting the delay to 0 enables instant rotation when operationally required (e.g., emergency signer replacement). The delay provides on-chain visibility: any observer can verify the upcoming rotation and challenge it (e.g., via governance) before activation.

Theorem 6.3 (Rotation Security)**.** *If the adversary controls fewer than t parties in both $\mathcal{P}$ and $\mathcal{P}'$, then rotation preserves unforgeability.*

Proof. The resharing protocol is a DKG where the secret s is determined by the old shares rather than fresh randomness. From the perspective of $\mathcal{P}'$, the shares are identically distributed to fresh DKG shares (a random polynomial of degree $t'-1$ with $f(0) = s$). Security follows from the DKG security theorem applied to $\mathcal{P}'$ with the fixed secret s. □

7 Security Analysis

7.1 FROST Unforgeability

Theorem 7.1 (FROST Unforgeability [1])**.** *FROST is unforgeable under $t-1$ static corruption in the random oracle model, assuming the hardness of the Discrete Logarithm Problem (DLP) in $\mathbb{G}$.*

Proof sketch. The proof reduces to the unforgeability of standard Schnorr signatures via a simulation argument. Given a forger $\mathcal{F}$ that breaks FROST with $t-1$ corrupt parties, we construct a reduction $\mathcal{R}$ that breaks the Schnorr signature scheme:

1. $\mathcal{R}$ receives Schnorr public key Y^* from the Schnorr challenger.
2. $\mathcal{R}$ simulates DKG by embedding Y^* as the aggregate key, using the simulator's ability to program random oracle H_2.
3. $\mathcal{R}$ answers $\mathcal{F}$'s signing queries by querying its own Schnorr signing oracle: the $t-1$ corrupt shares plus the simulated honest party's response produce a valid threshold signature.
4. When $\mathcal{F}$ outputs a forgery (m^*, σ^*) on an unqueried message, $\mathcal{R}$ extracts a Schnorr forgery using the forking lemma [14] applied to the random oracle.

The reduction is tight up to a factor of q_H (number of random oracle queries) due to the forking lemma. □

7.2 CGGMP21 Security

Theorem 7.2 (CGGMP21 Unforgeability with Identifiable Abort [2])**.** *CGGMP21 is unforgeable under $t-1$ static corruption in the UC framework, assuming the hardness of ECDSA (equiva-*

lently, DLP on secp256k1), semantic security of Paillier encryption, and soundness of the zero-knowledge proofs $\Pi_{\mathsf{enc}}, \Pi_{\mathsf{log}*}, \Pi_{\mathsf{aff}-\mathsf{g}}$.

Proof sketch. The UC proof proceeds through a sequence of hybrid games:

1. **Game 0**: Real protocol execution.
2. **Game 1**: Replace honest parties' Paillier encryptions with encryptions of zero. Indistinguishable by Paillier semantic security.
3. **Game 2**: Simulate ZK proofs. Indistinguishable by zero-knowledge.
4. **Game 3**: Extract adversary's inputs from ZK proofs. If extraction fails, identify the aborting party (identifiable abort).
5. **Game 4**: Simulate honest parties' signing shares using knowledge of the adversary's inputs. Indistinguishable by Paillier security.
6. **Game 5**: Ideal functionality. A forgery in this game implies breaking ECDSA.

Identifiable abort follows from the soundness of $\Pi_{\mathsf{log}*}$: if a party's partial signature is inconsistent with its presigning commitments, the ZK proof identifies the deviation. □

7.3 Composition Security

Lux Teleport runs FROST and CGGMP21 within the same signer network, sharing communication infrastructure and operator identities. We prove this composition is safe.

Theorem 7.3 (Protocol Composition)**.** *Running FROST (over Ed25519) and CGGMP21 (over secp256k1) in parallel with the same signer set does not degrade the security of either protocol, provided:*

1. *The DKG for each protocol uses independent randomness.*
2. *The signing nonces for each protocol are independently sampled.*
3. *The hash functions H_{FROST} and H_{ECDSA} use domain-separated prefixes.*

Proof. FROST operates over Ed25519 (a twisted Edwards curve over $\mathbb{F}_{2^{255}-19}$) and CGGMP21 operates over secp256k1 (a Weierstrass curve over $\mathbb{F}_{2^{256}-2^{32}-977}$). The group orders, generators, and hash domains are distinct. Since the DKG and signing randomness are independent, the protocols' random oracle queries are domain-separated, and the underlying hard problems (DLP on Ed25519 vs. DLP on secp256k1) are independent, a compromise of one protocol's secret key reveals no information about the other. Formally, the joint simulation in the UC framework decomposes into independent simulations for each protocol, and the overall advantage is bounded by the sum of individual advantages. □

7.4 Failure Probability Analysis

For production deployment with threshold t of n signers, the probability that signing fails due to fewer than t honest signers being available is:

$$P_{\text{fail}} = \sum_{k=0}^{t-1} \binom{n}{k} p^k (1-p)^{n-k} \tag{1}$$

where p is the per-signer availability probability. Table 3 shows values for Lux Teleport's production configuration.

(t, n)	$p = 0.95$	$p = 0.99$	$p = 0.999$
$(3, 5)$	2.3×10^{-3}	9.8×10^{-6}	1.0×10^{-8}
$(5, 7)$	4.1×10^{-4}	2.0×10^{-8}	2.1×10^{-12}
$(7, 10)$	1.5×10^{-4}	3.6×10^{-10}	3.6×10^{-16}
$(67, 100)$	3.7×10^{-8}	$< 10^{-40}$	$< 10^{-100}$

Table 3: Signing failure probability by threshold configuration and signer availability.

8 Gas Cost Analysis

A critical advantage of threshold signatures over multisig is constant on-chain verification cost. Table 4 compares verification costs.

Ethereum. On EVM chains, `ecrecover` costs 3,000 gas. A CGGMP21 threshold signature requires exactly one `ecrecover` call, verifying against the MPC group address: `ecrecover`$(\text{digest}, \sigma) ==$ `mpcGroupAddress`. Individual signers are not visible on-chain; only the aggregate key matters for verification. A k-of-n Gnosis Safe multisig requires k `ecrecover` calls

Chain Family	TSS Verify	k-of-n Multisig	Savings	Verify Method	Protocol
Ethereum / EVM	21,000 + 3,000	21,000 + 3,000k	$(k-1) \times 3{,}000$	`ecrecover` precompile	CGGMP21
Bitcoin Taproot	1 vByte witness	$k \times 64$ vB witness	$(k-1) \times 64$ vB	OP_CHECKSIG (Schnorr)	FROST
Bitcoin Legacy	1 DER sig	k DER sigs	$(k-1)$ sigs	OP_CHECKMULTISIG	CGGMP21
Solana	1 Ed25519 verify	k Ed25519 verifies	$(k-1)$ verifies	Ed25519 program	FROST
TON	∼0.005 TON	∼0.005k TON	$(k-1) \times 0.005$	Ed25519 TVM opcode	FROST
Cosmos / IBC	1 sig verify	k sig verifies	$(k-1)$ verifies	`ed25519_verify`	FROST
XRPL	1 ECDSA sig	k signer entries	$(k-1)$ entries	Native ECDSA	CGGMP21
Cardano	1 Ed25519 verify	k verifies	$(k-1)$ verifies	Plutus native	FROST
Polkadot	1 Sr25519 verify	k verifies	$(k-1)$ verifies	Runtime verify	FROST
Sui	1 Ed25519 verify	k verifies	$(k-1)$ verifies	Move native	FROST
Aptos	1 Ed25519 verify	k verifies	$(k-1)$ verifies	Move native	FROST
NEAR	1 Ed25519 verify	k verifies	$(k-1)$ verifies	Runtime verify	FROST
Stellar	1 Ed25519 verify	k signer weights	$(k-1)$ weights	Core verify	FROST
Algorand	1 Ed25519 verify	k verifies	$(k-1)$ verifies	AVM opcode	FROST
Tezos	1 Ed25519 verify	k verifies	$(k-1)$ verifies	Michelson	FROST
TRON	21,000 + 3,000	21,000 + 3,000k	$(k-1) \times 3{,}000$	TVM ecrecover	CGGMP21
StarkNet	1 ECDSA verify	k ECDSA verifies	$(k-1)$ verifies	Account verifier	CGGMP21

Table 4: On-chain signature verification costs: threshold signature (single verify) vs. k-of-n multisig. Threshold signatures always cost one single-signature verification regardless of t and n. Multisig cost scales linearly in k.

plus contract overhead (∼40,000 gas base). For the Teleport production configuration of $t = 67$, $n = 100$: threshold costs 24,000 gas (base + 1 ecrecover); equivalent multisig costs 241,000 gas (base + 67 ecrecover). Savings: 90%.

Bitcoin Taproot. A Taproot key-path spend using FROST requires 64 bytes of witness data (one Schnorr signature). An equivalent k-of-n using OP_CHECKMULTISIGVERIFY requires $k \times 72$ bytes (DER-encoded ECDSA signatures) plus the redeem script. For $t = 5$: FROST uses 64 vB; multisig uses 360+ vB. Savings: 82%.

Solana. Ed25519 signature verification on Solana costs approximately 1,400 compute units. A FROST signature requires one verification. A multisig requires k verifications plus program logic overhead (∼5,000 CU base). For $t = 5$: FROST costs 6,400 CU; multisig costs 12,000 CU. Savings: 47%.

9 Comparison with Alternatives

We compare the FROST+CGGMP21 approach against three deployed alternatives for cross-chain custody.

BLS Aggregation (Wormhole-style). Wormhole [17] uses BLS12-381 aggregate signatures from a guardian set. BLS aggregation provides compact multi-signer proofs but requires a custom on-chain verifier (BLS precompile or Solidity implementation) costing 180,000+ gas on EVM. BLS12-381 is not natively supported on any major chain except Ethereum (EIP-2537). Coverage is limited to chains where the verifier can be deployed.

TSS GG20 (tBTC v2). tBTC v2 [18] uses the GG20 protocol [9] for threshold ECDSA. GG20 requires 8 rounds of communication (vs. CGGMP21's offline+1 or FROST's 2 rounds) and lacks identifiable abort—a malicious party can cause signing to fail without detection. GG20 covers only secp256k1 chains. No Ed25519 or Taproot support.

Property	**FROST + CGGMP21** (Teleport)	**BLS Aggregation** (Wormhole)	**TSS GG20** (tBTC v2)	**Multisig** (Gnosis Safe)
Sig. scheme	Schnorr + ECDSA	BLS12-381	ECDSA (GG20)	ECDSA
On-chain format	Native single sig	Custom verifier	Native single sig	k sigs + contract
Verify cost (EVM)	3,000 gas	180,000 gas	3,000 gas	3,000k gas
Chain coverage	17 families	EVM + custom	secp256k1 only	EVM only
Rounds (signing)	2 (FROST) / 1+pre (CGGMP)	1	8	0 (async)
Identifiable abort	Yes (CGGMP21)	No	No (GG20)	N/A
Key refresh	Yes	No (fixed keys)	Yes	N/A
Bitcoin Taproot	Yes (native)	No	No	No
Ed25519 chains	Yes (native)	No	No	No
Corruption threshold	$t-1$ of n	$t-1$ of n	$t-1$ of n	$k-1$ of n

Table 5: Comparison of threshold/multisig schemes for cross-chain custody.

Multisig (Gnosis Safe). On-chain multisig is the simplest approach but has fundamental limitations: (1) gas cost scales linearly in the number of signers k; (2) threshold changes require contract transactions; (3) not available on chains without smart contracts (Bitcoin, Stellar); (4) reveals the threshold structure on-chain.

9.1 Security Property Comparison

Property	**Ours**	**BLS**	**GG20**	**Multisig**
EUF-CMA	✓	✓	✓	✓
Id. abort	✓	–	–	N/A
Key refresh	✓	–	✓	–
Forward sec.	✓	–	✓	–
Privacy*	✓	–	✓	–
UC security	✓†	✓	–	N/A

Table 6: Security properties. *Privacy = threshold structure not revealed on-chain. †CGGMP21 is UC-secure; FROST is proven in the ROM.

10 Implementation

The framework is implemented in Go across two packages:

- `github.com/luxfi/threshold` — Core threshold cryptography: FROST (Ed25519, BIP-340), CGGMP21 (secp256k1), DKG, key refresh, resharing.
- `github.com/luxfi/mpc` — MPC coordination daemon: signer orchestration, nonce management, chain adapters, key storage, rotation ceremonies.

10.1 Architecture

The MPC daemon (`mpckeyd`) runs on each signer node and exposes a gRPC interface for signing requests. The coordination flow:

1. A bridge relayer detects a cross-chain transfer request.
2. The relayer broadcasts a signing request to the MPC network.
3. The coordinator (rotating leader) selects the signer set S and initiates the protocol.
4. Signers execute FROST or CGGMP21 depending on the target chain's signature scheme.
5. The resulting signature is submitted to the destination chain.

10.2 Protocol Selection

Chain adapters implement a common `Signer` interface:

```
type Signer interface {
    KeyGen(t, n int) (*KeyShare, error)
    Sign(msg []byte, signers []int)
        (*Signature, error)
    Refresh() (*KeyShare, error)
    PublicKey() []byte
    Protocol() string // "frost" or "cggmp21"
}
```

Each chain adapter maps from the chain's transaction format to the appropriate protocol:

```
func AdapterFor(chain Chain) Signer {
    switch chain.SigType() {
    case Ed25519, BIP340:
        return frost.NewSigner(chain)
    case ECDSA:
        return cggmp.NewSigner(chain)
    }
}
```

10.3 Nonce Management

Nonce reuse in threshold Schnorr signatures is catastrophic: reusing a nonce pair (d_i, e_i) in two signing sessions with different binding factors leaks the secret share s_i. The implementation enforces:

1. **Single-use nonces**: Each (d_i, e_i) pair is stored with a unique session ID and deleted after use. The nonce store is append-only with cryptographic deletion.
2. **Hedged nonces**: Following RFC 6979 [16] principles, nonces are derived from $\mathsf{HKDF}(\mathsf{sk}_i, \text{session\_id}, \text{aux\_rand})$ where aux_rand provides fresh entropy as a hedge against RNG failure.
3. **Nonce pre-generation**: For latency-sensitive paths, nonce pairs are pre-generated in batches and stored encrypted at rest. Each nonce is marked as consumed atomically with protocol execution.
4. **Broadcast serialization**: A `sync.Mutex` on `StoreBroadcastMessage` prevents a race condition where concurrent FROST signing sessions could overwrite each other's commitment messages in the broadcast store. Without this lock, two sessions arriving simultaneously could corrupt the commitment list B, leading to inconsistent binding factors and signing failure.

10.4 Performance

Benchmarks on production hardware (AMD EPYC 7763, 2.45 GHz, 64 cores, Ubuntu 22.04, Go 1.22):

Operation	**Protocol**	$(3, 5)$	$(67, 100)$
DKG	FROST	14 ms	820 ms
DKG	CGGMP21	1,100 ms	48,000 ms
Sign	FROST	8 ms	180 ms
Presign	CGGMP21	120 ms	4,200 ms
Sign (online)	CGGMP21	2 ms	12 ms
Refresh	Both	12 ms	680 ms
LSS Reshare ($3 \to 5$)	Both	0.14 ms	—
LSS Reshare ($50 \to 100$)	Both	—	3.5 ms

Table 7: Latency benchmarks (median over 1,000 runs, single-machine simulation with network latency of 0ms between parties). LSS (Lux Secret Sharing) dynamic resharing enables threshold changes (e.g., 3-of-5 → 3-of-7) without DKG restart.

In production with geographically distributed signers (median inter-signer RTT 45ms), end-to-end signing latency is dominated by network round-trips: FROST signing completes in $2 \times 45\text{ms} + 8\text{ms} \approx 98\text{ms}$; CGGMP21 online signing completes in $1 \times 45\text{ms} + 2\text{ms} \approx 47\text{ms}$ (with pre-computed presignatures).

11 Formal Security Proofs

We state the main security theorems with full proof structure. Formal machine-checked proofs in Lean 4 are available at `github.com/luxfi/proofs/tree/main/lean/Crypto`.

11.1 FROST Unforgeability (Full Proof)

Theorem 11.1 (FROST-UF)**.** *Let* FROST *be the (t, n)-threshold Schnorr signature scheme over group $\mathbb{G}$ of order q. For any PPT adversary $\mathcal{A}$ making at most q_s signing queries and q_H random oracle queries, and corrupting at most $t-1$ parties:*

$$\mathsf{Adv}^{\mathsf{uf}}_{\mathsf{FROST}}(\mathcal{A}) \leq \frac{q_H + q_s + 1}{q} + \sqrt{\frac{q_H}{q}} + \mathsf{Adv}^{\mathsf{dl}}_{\mathbb{G}}(\mathcal{B})$$

where $\mathcal{B}$ is a DLP solver with runtime polynomial in the runtime of $\mathcal{A}$.

Proof. We construct a sequence of games:

Game 0. Real FROST experiment with $t-1$ corrupted parties. $\mathcal{A}$ receives the public key Y, the corrupted shares $\{s_i\}_{i \in C}$, and access to a signing oracle.

Game 1. Replace the random oracle H_2 with a programmable random oracle. The reduction $\mathcal{R}$ programs responses to be consistent. This is a syntactic change; $|\Pr[G_1] - \Pr[G_0]| = 0$.

Game 2. $\mathcal{R}$ receives DLP challenge $Y^* = x^* \cdot G$ and embeds it as the honest party's public key share. Specifically, $\mathcal{R}$ sets $Y_h = Y^* - \sum_{i \in C} \lambda_{i,S}^{-1} \cdot s_i \cdot G$ where h is the simulated honest party. $\mathcal{R}$ does not know s_h but can simulate signing by programming H_2 to produce challenges c such that the honest party's response z_h can be computed from the random oracle output. This requires $\mathcal{R}$ to know c before committing to R, which is possible because $\mathcal{R}$ programs H_2. $|\Pr[G_2] - \Pr[G_1]| \leq (q_H + q_s)/q$ due to potential programming conflicts.

Game 3 (Extraction). When $\mathcal{A}$ outputs forgery $(m^*, (R^*, z^*))$ on an unqueried message, $\mathcal{R}$ applies the generalized forking lemma [15]: rewind $\mathcal{A}$ with a different random oracle response at the forgery point to obtain a second forgery $(m^*, (R^*, z^{*\prime}))$ with challenge $c' \neq c$. From $z^* - z^{*\prime} = (c - c') \cdot x^*$, $\mathcal{R}$ recovers $x^* = (z^* - z^{*\prime})/(c - c')$, solving DLP.

The forking probability is at least $(\epsilon - (q_H + 1)/q)^2/q_H$ where ϵ is $\mathcal{A}$'s advantage, giving the stated bound after rearrangement. □

11.2 CGGMP21 Identifiable Abort

Theorem 11.2 (CGGMP21 Identifiable Abort)**.** *If the CGGMP21 signing protocol fails to produce a valid signature, the honest parties can identify at least one malicious party with probability $\geq 1 - \mathsf{negl}(\lambda)$.*

Proof. The identification mechanism works as follows. In the presigning phase, each party i commits to its nonce share k_i via $K_i = k_i \cdot G$ with a zero-knowledge proof Π_{enc} of correct encryption. In the MtA sub-protocol, each party proves correct computation via $\Pi_{\mathsf{aff-g}}$. In online signing, each party i broadcasts σ_i with a proof $\Pi_{\mathsf{log*}}$ that σ_i is consistent with its committed values (k_i, χ_i).

If $s = \sum_i \sigma_i$ does not yield a valid ECDSA signature, at least one σ_i is inconsistent. The $\Pi_{\mathsf{log*}}$ proof for that party will fail verification, identifying the malicious party. Soundness of $\Pi_{\mathsf{log*}}$ ensures that an honest party's proof always verifies, so only a malicious party can be identified.

Formally: let E be the event that signing fails. Conditioned on E, the probability that all $\Pi_{\mathsf{log*}}$ proofs verify is bounded by the soundness error of $\Pi_{\mathsf{log*}}$, which is $\mathsf{negl}(\lambda)$. Therefore, with probability $\geq 1 - \mathsf{negl}(\lambda)$, at least one malicious party's proof fails. □

11.3 Key Refresh Forward Security

Theorem 11.3 (Forward Security under Proactive Refresh)**.** *With key refresh every Δ time units, an adversary who compromises $t-1$ shares in any single epoch cannot forge signatures for other epochs, even if the adversary later compromises $t-1$ different shares in a subsequent epoch.*

Proof. Let $\{s_i^{(e)}\}$ and $\{s_i^{(e+1)}\}$ denote shares in epochs e and $e+1$. The adversary knows $\{s_i^{(e)}\}_{i \in C_e}$ and $\{s_j^{(e+1)}\}_{j \in C_{e+1}}$ with $|C_e| = |C_{e+1}| = t - 1$.

Shares are related by $s_i^{(e+1)} = s_i^{(e)} + \sum_j g_j(i)$ where g_j are degree-$(t-1)$ polynomials with $g_j(0) = 0$. The adversary can compute $s_i^{(e+1)} - s_i^{(e)} = \sum_j g_j(i)$ for indices in $C_e \cap C_{e+1}$. However, $\sum_j g_j$ is a degree-$(t-1)$ polynomial with known zero constant term, determined by $t-1$

coefficients. Even if $|C_e \cap C_{e+1}| = t-2$ (maximum overlap given $|C_e| = |C_{e+1}| = t-1$ and $n \geq t+1$), the adversary obtains $t-2$ points on a degree-$(t-1)$ polynomial plus the constraint $g(0) = 0$, giving $t-1$ constraints on $t-1$ unknowns. This determines g uniquely, allowing the adversary to convert shares across epochs.

However, this requires C_e and C_{e+1} to overlap in $t-2$ indices. If the signer set has $n \geq 2t$, a mobile adversary corrupting $t-1$ parties per epoch with $|C_e \cap C_{e+1}| \leq t-2$ still cannot reconstruct s: the adversary has at most $t-1$ independent share values per epoch, which is information-theoretically insufficient for a degree-$(t-1)$ polynomial.

The forward security guarantee holds provided the adversary does not corrupt the same $t-1$ parties across two consecutive epochs and the honest parties' refresh polynomials remain secret. □

12 Related Work

Threshold Schnorr. FROST [1] optimized threshold Schnorr to 2 rounds from the 3-round GJKR protocol [10]. ROAST [12] adds robustness to FROST by running parallel signing sessions. MuSig2 [11] provides 2-round multi-signatures (all-of-n) for Bitcoin Taproot but does not support threshold ($t < n$) configurations.

Threshold ECDSA. GG18 [8] introduced practical threshold ECDSA. GG20 [9] improved efficiency but lacked identifiable abort. CGGMP21 [2] achieved UC security with identifiable abort. Doerner et al. [13] optimized the 2-party case.

BLS-based bridges. Wormhole [17] uses BLS12-381 guardian signatures. BLS aggregation is compact but requires custom verifiers on non-BLS chains (180,000+ gas on EVM). EIP-2537 adds BLS precompiles to Ethereum but is not yet widely deployed on L2s.

MPC wallets. Fireblocks [19] and ZenGo [20] deploy threshold signatures for institutional custody. Both use variants of GG20/CGGMP21 for ECDSA but do not address Ed25519 chains or Bitcoin Taproot.

13 Conclusion

We have presented a unified threshold cryptographic framework that covers 17 blockchain families through two complementary protocols: FROST for Schnorr/EdDSA chains and CGGMP21 for ECDSA chains. The framework provides:

1. **Universal coverage**: Every major blockchain is supported with native signature verification—no custom verifier contracts.
2. **Constant on-chain cost**: One signature verification regardless of threshold size, yielding up to 90% gas savings over multisig.
3. **Strong security**: Unforgeability under $t-1$ corruption (Theorems 11.1, 7.2), identifiable abort (Theorem 11.2), forward security under proactive refresh (Theorem 11.3), and safe protocol composition (Theorem 7.3).
4. **Bitcoin Taproot**: FROST KeygenTaproot produces x-only public keys (BIP-340) for key-path spends indistinguishable from single-signer outputs.
5. **MPC group address**: On-chain verification uses a single aggregate key (`ecrecover` == `mpcGroupAddress`), not individual signers.
6. **Operational safety**: Configurable rotation delay (default 24h, 0 for instant, max 7d) with on-chain visibility; proactive key refresh without public key change; LSS dynamic resharing for threshold changes.
7. **Production performance**: 98ms end-to-end FROST signing, 47ms CGGMP21 online signing with presignatures, 0.14ms LSS reshare ($3 \rightarrow 5$), in geographically distributed deployment.

The framework is deployed in Lux Teleport, securing cross-chain asset transfers across 270+ blockchains. Implementation is open-source at `github.com/luxfi/threshold` and `github.com/luxfi/mpc`.

References

[1] C. Komlo and I. Goldberg, "FROST: Flexible Round-Optimized Schnorr Threshold Signatures," *Selected Areas in Cryptography (SAC)*, 2020. https://eprint.iacr.org/2020/852

[2] R. Canetti, R. Gennaro, S. Goldfeder, N. Makriyannis, and U. Peled, "UC Non-Interactive, Proactive, Threshold ECDSA with Identifiable Aborts," *ACM CCS*, 2020. Revised 2021. https://eprint.iacr.org/2021/060

[3] P. Wuille, J. Nick, and T. Ruffing, "BIP-340: Schnorr Signatures for secp256k1," Bitcoin Improvement Proposal, 2020. https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki

[4] P. Wuille, J. Nick, and A. Towns, "BIP-341: Taproot: SegWit version 1 spending rules," Bitcoin Improvement Proposal, 2020. https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki

[5] P. Wuille, J. Nick, and A. Towns, "BIP-342: Validation of Taproot Scripts," Bitcoin Improvement Proposal, 2020. https://github.com/bitcoin/bips/blob/master/bip-0342.mediawiki

[6] T. P. Pedersen, "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing," *CRYPTO*, 1991.

[7] P. Feldman, "A Practical Scheme for Non-Interactive Verifiable Secret Sharing," *FOCS*, 1987.

[8] R. Gennaro and S. Goldfeder, "Fast Multiparty Threshold ECDSA with Fast Trustless Setup," *ACM CCS*, 2018. https://eprint.iacr.org/2019/114

[9] R. Gennaro and S. Goldfeder, "One Round Threshold ECDSA with Identifiable Abort," IACR ePrint 2020/540, 2020. https://eprint.iacr.org/2020/540

[10] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure Distributed Key Generation for Discrete-Log Based Cryptosystems," *Journal of Cryptology*, 2003.

[11] J. Nick, T. Ruffing, and Y. Seurin, "MuSig2: Simple Two-Round Schnorr Multi-Signatures," *CRYPTO*, 2021. https://eprint.iacr.org/2020/1261

[12] T. Ruffing, V. Ronge, E. C. Fletcher, J. Griffy, and D. Schröder, "ROAST: Robust Asynchronous Schnorr Threshold Signatures," *ACM CCS*, 2022. https://eprint.iacr.org/2022/550

[13] J. Doerner, Y. Kondi, E. Lee, and A. Shelat, "Secure Two-party Threshold ECDSA from ECDSA Assumptions," *IEEE S&P*, 2018.

[14] D. Pointcheval and J. Stern, "Security Proofs for Signature Schemes," *EUROCRYPT*, 1996.

[15] M. Bellare and G. Neven, "Multi-Signatures in the Plain Public-Key Model and a General Forking Lemma," *ACM CCS*, 2006.

[16] T. Pornin, "RFC 6979: Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)," IETF, 2013. https://www.rfc-editor.org/rfc/rfc6979

[17] Wormhole Foundation, "Wormhole: A Generic Message Passing Protocol," 2023. https://wormhole.com/whitepaper

[18] Keep Network, "tBTC v2: A Decentralized Bitcoin Bridge," 2022. https://docs.threshold.network/applications/tbtc-v2

[19] Fireblocks, "MPC-CMP: Threshold ECDSA from ECDSA Assumptions," 2023. https://www.fireblocks.com/what-is-mpc/

[20] ZenGo, "Threshold Signatures: The Future of Private Keys," 2019. https://zengo.com/mpc-wallet/