Integrating Fast Multiparty Threshold ECDSA with Fast Trustless Setup into TLS 1.3

Internal Security Engineering Draft

November 2025

Contents

1 Executive Summary

This document combines a comprehensive proposal and detailed handshake workflow for integrating a modern Fast Multiparty Threshold ECDSA scheme together with a trustless Distributed Key Generation (DKG) protocol into existing TLS 1.3 infrastructure without altering wire semantics. The integration decentralizes private key control, limits blast radius of node compromise, preserves client transparency, and maintains low-latency server authentication.

1.1 What is Threshold Cryptography?

Threshold cryptography is a cryptographic primitive that distributes trust among multiple parties. Instead of a single entity holding a complete private key, the key is split into multiple *shares* distributed across N participants. Any subset of T participants (the *threshold*) can cooperate to perform cryptographic operations (e.g., signing), but fewer than T participants learn nothing about the private key.

Key Properties:

- **Secret Sharing:** Private key x never exists in one location; instead, shares $x_1, x_2, \ldots, x_N$ are distributed such that x can be reconstructed (implicitly, not explicitly) only with $\geq T$ shares.
- **Threshold Signing:** To sign message m, at least T parties contribute partial signatures σ_i; these are combined into a standard signature σ verifiable with the public key.
- **Robustness:** Compromise of up to $T - 1$ parties reveals no information about the key; the system remains secure.
- **Availability:** As long as $\geq T$ honest participants are online, operations proceed normally.

Why Threshold ECDSA for TLS? Traditional TLS deployments store private keys in single locations (servers, HSMs). If compromised, attackers can impersonate servers indefinitely. Threshold ECDSA eliminates this single point of failure: an attacker must compromise $\geq T$ geographically and administratively separated nodes simultaneously—a significantly harder attack. Moreover, the scheme is *transparent to clients*: standard TLS handshakes proceed unchanged; clients verify signatures using the same public key as before.

Trustless Setup (DKG): Traditional secret sharing requires a trusted dealer to split the key. A *Distributed Key Generation (DKG)* protocol allows participants to jointly generate a keypair without any party ever knowing the full private key, eliminating the dealer as a point of trust or failure.

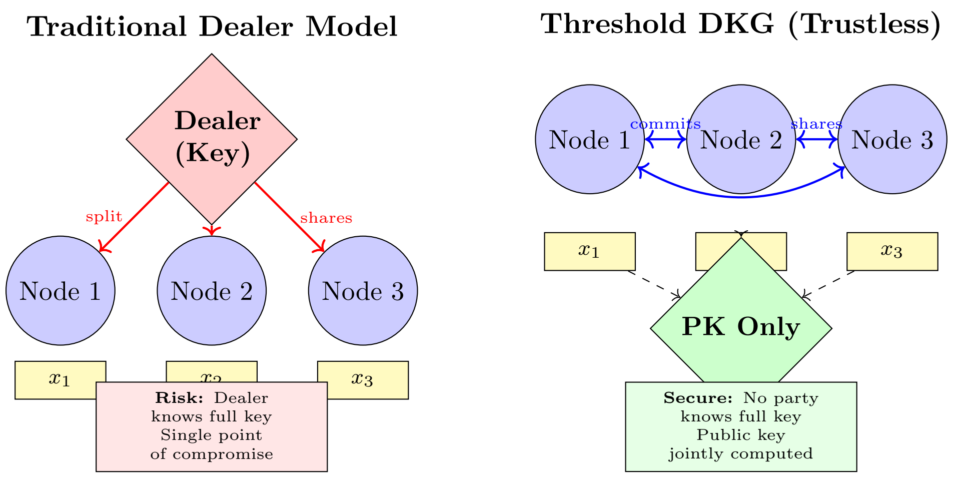
Traditional Dealer Model
Dealer (Key)
split
shares
Node 1
Node 2
Node 3
x_1
x_2
x_3
Risk: Dealer knows full key Single point of compromise
Threshold DKG (Trustless)
Node 1
commits
Node 2
shares
Node 3
x_1
x_3
PK Only
Secure: No party knows full key Public key jointly computed

Figure 1: Comparison of traditional dealer-based secret sharing vs. trustless Distributed Key Generation (DKG). In DKG, participants exchange commitments and shares via multi-party computation; the private key is never materialized.

2 Objectives

- Threshold-protect server certificate keys: public key retained, private key never reconstructed.
- Trustless (dealerless) key setup with robustness against up to f malicious/Byzantine participants.
- Zero client modifications; RFC 8446 compliance.
- Added signing latency median < 10 ms; strong auditability and rotation workflows.
- Scalable precomputation for high throughput (hundreds of handshakes per second).

3 Scope and Non-Goals

In Scope: Server cert thresholdization, optional internal threshold CA, TLS CertificateVerify signing, OCSP/SCT support, operational tooling.
Out of Scope (initial): Client certificate threshold, PQC threshold primitives, cross-organizational federation.

4 Stakeholders

Security Engineering, Platform/Infra, Compliance/Governance, SRE/DevOps, Application Teams.

5 Background and Motivation

5.1 The Problem with Centralized Private Keys

Traditional TLS deployments rely on a single private key stored on the server or in a Hardware Security Module (HSM). While HSMs provide tamper resistance, they remain *logical single points of failure*:

- **Key Exfiltration:** If compromised (firmware exploit, supply chain attack, insider threat), the attacker obtains the complete key.
- **Impersonation Risk:** A stolen key allows indefinite server impersonation until certificate revocation propagates.
- **Limited Blast Radius Control:** HSM compromise affects all certificates sharing that key.
- **Operational Rigidity:** Key rotation requires certificate reissuance and careful propagation windows.

5.2 Threshold Cryptography as a Solution

Threshold ECDSA disperses key material across N geographically and administratively separated nodes. A quorum of T nodes collaborates to produce signatures, but fewer than T reveals *nothing* about the private key. Benefits include:

- **Defense in Depth:** Attacker must compromise $\geq T$ nodes simultaneously across different security domains.
- **Graceful Degradation:** Up to $N - T$ nodes can fail or be taken offline without service disruption.

- **Auditability:** Every signature involves multiple parties; logs provide multi-party attestation.
- **Proactive Security:** Share refresh protocols allow re-randomizing shares periodically, mitigating slow key leakage.

5.3 Fast Variants and Trustless Setup

Modern schemes (Lindell–Nof, Gennaro–Goldfeder) achieve:

- **Two-round signing** with aggressive precomputation of ephemeral nonces, enabling latencies competitive with local HSM operations (< 10 ms added).
- **Trustless DKG** eliminating the dealer: participants jointly generate the keypair via multiparty computation without any party learning the full key.

These advances make threshold ECDSA practical for high-throughput, latency-sensitive deployments like TLS servers at scale.

6 Requirements

6.1 Functional

- Distributed key generation for curve P-256 (primary) or secp256k1 (optional) with configurable (N, T).
- Produce valid ECDSA (r, s) signatures for TLS CertificateVerify.
- Proactive share refresh without altering public key.
- High availability: tolerate $< T$ offline nodes.

6.2 Security

- Unforgeability under threshold ECDSA assumptions.
- Confidential share storage; hardened memory; encrypted rest state.
- Audit trail: transcript hash, quorum identity, aggregated R.
- Nonce safety; bias resistance; misbehavior detection and eviction.

6.3 Performance

Median added latency ≤ 10 ms; $p99 \leq 25$ ms. Horizontal scaling via precompute workers.

6.4 Operational

Automated health, rotation workflow, disaster recovery with threshold backups.

7 Threat Model

Adversaries: external attackers, insiders, network interceptors, malicious signer nodes. Goals: prevent key reconstruction with $< T$ shares, prevent nonce reuse, detect rogue-key attempts, maintain attribution.

8 Architecture Overview

8.1 Components

Signer Nodes (share custody) Coordinator (orchestration) Auditor Service Precomputation Workers PKI Integration Layer Monitoring Stack.

8.2 Data Flow Summary

1. **DKG:** Broadcast commitments and shares; derive public key Q.
2. **Precompute:** Generate nonce tuples (k_i, R_i) and auxiliary inversion data.
3. **Signing:** Aggregate $R = \sum R_i$, derive r, compute partial s_i, aggregate s.
4. **TLS Frontend:** Consumes (r, s) as if locally generated.

9 Cryptographic Protocol Details

9.1 Distributed Key Generation

Each party selects polynomial f_i of degree $T-1$; share for party p is $s_p = \sum_i f_i(p)$. Commitments (Pedersen/Feldman) ensure verifiability; abort on inconsistency. Public key $Q = (\sum_i f_i(0))G$.

9.2 Precomputation

Signers compute pools of nonces k_i and points $R_i = k_i G$ with auxiliary values for fast inversion. Coordinator tracks availability.

9.3 Signing

Given transcript hash m, each signer returns partial $s_i = k_i^{-1}(m + r x_i)$ (scheme variant dependent). Final $s = \sum s_i \mod n$, low-s normalization applied.

9.4 Share Refresh

Proactive refresh via secondary polynomials randomizes shares; Q unchanged.

9.5 Misbehavior Detection

Invalid partials trigger blame protocols requiring commitment proofs; malicious nodes quarantined.

10 TLS 1.3 Integration Points

The threshold signing mechanism integrates seamlessly into TLS 1.3 by substituting only the *CertificateVerify* ECDSA signing operation. All other protocol steps remain unchanged:

- **Certificate Message:** Contains the public key Q derived from the threshold DKG. Clients use this public key for standard ECDSA verification.

- **CertificateVerify Signature:** Instead of using a local private key, the TLS frontend (via OpenSSL 3 provider or PKCS#11 engine) invokes the threshold signing coordinator with the transcript hash TH_1. The coordinator orchestrates partial signature collection and aggregation, returning a standard ECDSA (r, s) pair.
- **Session Tickets:** Unaffected (symmetric key encryption).
- **OCSP Responses, SCTs, CRL Signing:** Can reuse the same threshold signing path if the issuing key is also threshold-protected.
- **Client Perspective:** Completely transparent. The client performs standard RFC 8446 handshake verification; no knowledge of or changes required for threshold backend.

Implementation Hook: The OpenSSL 3 Provider API allows custom signing backends. The threshold provider intercepts `EVP_PKEY_sign` calls, marshals the digest to the coordinator, awaits the threshold signature, and returns it as if computed locally. Fallback mechanisms handle quorum unavailability gracefully (e.g., retry, alert, or optional secondary conventional key per policy).

11 Operational Workflow

11.1 Lifecycle Phases

1. **Bootstrap (DKG):** Deploy signer nodes with ephemeral bootstrap identities; run distributed key generation protocol; each node stores its share x_i encrypted at rest; public key Q published and certificate issued by CA.
2. **Normal Signing:** TLS frontend requests signatures for incoming handshakes; coordinator selects T healthy signers; partials aggregated; (r, s) returned.
3. **Precompute Pool Maintenance:** Background workers on signer nodes generate nonce tuples (k_i, R_i) and register with coordinator; pool size monitored (target: e.g., 1000 tuples); autoscaling triggered if depletion risk detected.
4. **Key Rotation:** Run parallel DKG for new keypair Q'; issue certificate for Q'; dual-publish both certificates during transition window; after propagation (e.g., 48h), deprecate old Q; archive logs and securely erase old shares.
5. **Proactive Refresh:** Periodic (e.g., weekly) share refresh via polynomial re-sharing; Q unchanged but shares x_i randomized, mitigating long-term partial leakage accumulation.
6. **Incident Response:** If node compromise suspected, exclude from quorum; run emergency refresh or expedited rotation; forensic audit of signing logs.
7. **Decommission:** Securely wipe share storage; revoke node membership; update coordinator configuration; verify no residual key material.

12 High Availability

Example: $N = 7, T = 4$. Latency-aware quorum selection; fallback policy on insufficient quorum; optional secondary conventional key (policy-gated).

13 Performance Considerations

Intra-DC RTT (2–3 ms) + partial compute (¡2 ms) + aggregation (¡1 ms) yields target median 5–8 ms. Parallel precompute amortizes expensive inversions.

14 API Contract (Draft)

Listing 1: Signing API Request/Response

```
POST /v1/sign
{ "hash": "<hex>", "context": "TLS-1.3-CertificateVerify", "nonce_policy":
    "fresh" }

Response:
{ "r": "<hex>", "s": "<hex>", "pubkey": "<hex>",
  "quorum": ["S1","S3","S5","S6"],
  "precompute_id": "uuid", "timestamp": "2025-11-13T12:34:56Z" }
```

Errors: 409 QuorumInsufficient, 424 PrecomputeDepleted, 400 InvalidHashFormat, 500 InternalAggregationError.

15 Logging and Audit

Append-only log (Kafka or similar) with hashing chain; daily Merkle root published for transparency.

16 Security Analysis Summary

- Share Theft (¡T): No reconstruction.
- Nonce Bias: Verifiable commitments; single-use enforcement.
- Rogue-Key: Commitment + ZK proof checks.
- DoS by Malicious Signer: Adaptive quorum; eviction.
- Coordinator Compromise: No forging without shares.

17 Testing and Validation

Unit (arithmetic invariants), integration (local cluster handshake), fuzzing (commitment/partial corruption), chaos (mid-round node termination), performance benchmarks, timing side-channel probes, compliance mapping.

18 Implementation Phases

Phase	Duration	Deliverables
0 Research	2 wks	Protocol selection, curve finalization
1 Prototype	4 wks	Minimal cluster, REST API, tests
2 TLS Integration	3 wks	OpenSSL provider, handshake demo
3 Hardening	4 wks	Audit log, refresh, fault tolerance
4 Rotation & CA	3 wks	Key rotation flow, threshold CA (opt.)
5 Production Rollout	4 wks	Monitoring, DR plan, docs

19 Risk and Mitigation

Risk	Impact	Mitigation
Protocol Complexity Bugs	Invalid/forgeable signatures	Use vetted libraries; formal verification of arithmetic core
Latency Spikes	Handshake slowdown	Precompute pools; autoscale; SLO alerts
Coordinator Bottleneck	Throughput cap	Stateless multi-instance; idempotent requests
Share Leakage Over Time	Gradual compromise	Weekly proactive refresh; hardened memory handling
Malicious Node Abort Storm	Availability degradation	Blame protocol; eviction; rapid replacement provisioning
Operational Misconfig	Downtime/security gap	IaC validation; canary deployment; config policy checks

20 Future Extensions

Client-side threshold auth, PQC threshold (Dilithium/FROST-like EdDSA), multi-cloud distribution, transparency log publication, confidential computing enclaves.

21 Compliance and Governance

Documented DKG ceremony, RBAC for maintenance, audited break-glass processes, periodic third-party review.

22 Deployment Model

Kubernetes: Signer nodes as StatefulSets (AZ anti-affinity), coordinators as Deployments with HPA, Vault/Sealed Secrets for share sealing.

23 Open Questions

Threshold parameters (N, T); curve choice (performance vs acceptance); library selection vs custom implementation; inclusion timing for threshold CA; audit publication cadence.

24 Acceptance Criteria

Valid TLS handshake to unmodified client with threshold-generated signature meeting latency SLO; successful share refresh without key change; simulated $< T$ compromise fails to reconstruct key; zero-downtime rotation observed over 24h.

25 Summary

Decentralizing TLS private keys via fast threshold ECDSA materially reduces compromise risk while preserving protocol compatibility and performance. Structured phased rollout mitigates complexity and ensures auditability.

26 Detailed TLS 1.3 Handshake Workflow (Threshold Integration)

26.1 Legend

C: Client; FE: TLS Frontend; COORD: Coordinator; S_i: Signer node; PK: Public key; PrePool: Precomputed nonce pool.

26.2 Pre-Handshake Preparation

DKG completion, membership list, precompute pool fill, health checks, secure authenticated channels.

26.3 Standard Phases

ClientHello; ServerHello; EncryptedExtensions; Certificate; **CertificateVerify (threshold signing)**; Finished; Application Data.

26.4 Happy Path Step-by-Step

1. C $\rightarrow$ FE: ClientHello.
2. FE selects cipher suite / group; returns ServerHello.
3. Key schedule derivation (HKDF) by FE.
4. FE sends EncryptedExtensions.
5. FE sends Certificate containing PK.
6. Transcript hash TH_1 computed.
7. FE submits threshold SignRequest(TH_1) to COORD.
8. COORD selects quorum Q of size T with available precompute tuples.
9. COORD instructs each signer to participate using chosen tuple.
10. Each signer computes partial (R_i, s_i) and returns with proof.
11. COORD aggregates $R = \sum R_i$, derives r, verifies partials, computes $s = \sum s_i$ (normalizes low-s).

12. COORD returns final (r, s) to FE.
13. FE sends CertificateVerify with (r, s).
14. FE sends Finished.
15. Client verifies signature (standard ECDSA over transcript) and sends Finished.
16. Secure channel established; application data flows.

26.5 Failure Paths

Insufficient Quorum: Error returned; FE retries/backoff or policy fallback.

Precompute Depletion: Refill triggered; request retried post replenishment.

Invalid Partial: Blame protocol; malicious node evicted; round restarted.

Coordinator Failure: FE resubmits idempotent request to alternate coordinator.

Client Abort: Consumed nonces discarded to avoid reuse.

26.6 Security Checks During Signing

Nonce single-use enforcement; low-s normalization; transcript hash validation; audit logging (request id, quorum, aggregated R hash).

26.7 Complete TLS 1.3 Handshake with Threshold Signing (Illustrated)

26.8 Latency Budget (Illustrative)

Component	Time (ms)
FE $\rightarrow$ COORD Dispatch	0.3
Fan-out to Signers	0.5
Partial Compute (parallel)	1.5
Aggregate & Verify	0.7
Return to FE	0.3
Total Added (median)	≈ 3.3

26.9 Monitoring Metrics

- `tls_threshold_sign_duration_ms` (Histogram): End-to-end signing latency distribution.
- `tls_threshold_quorum_size` (Gauge): Number of signers participating per request.
- `tls_threshold_precompute_pool_depth` (Gauge): Available nonce tuples per signer.
- `tls_threshold_abort_count` (Counter): Failed signing attempts (timeout, invalid partial).
- `tls_threshold_malicious_flagged` (Counter): Nodes flagged by blame protocol.

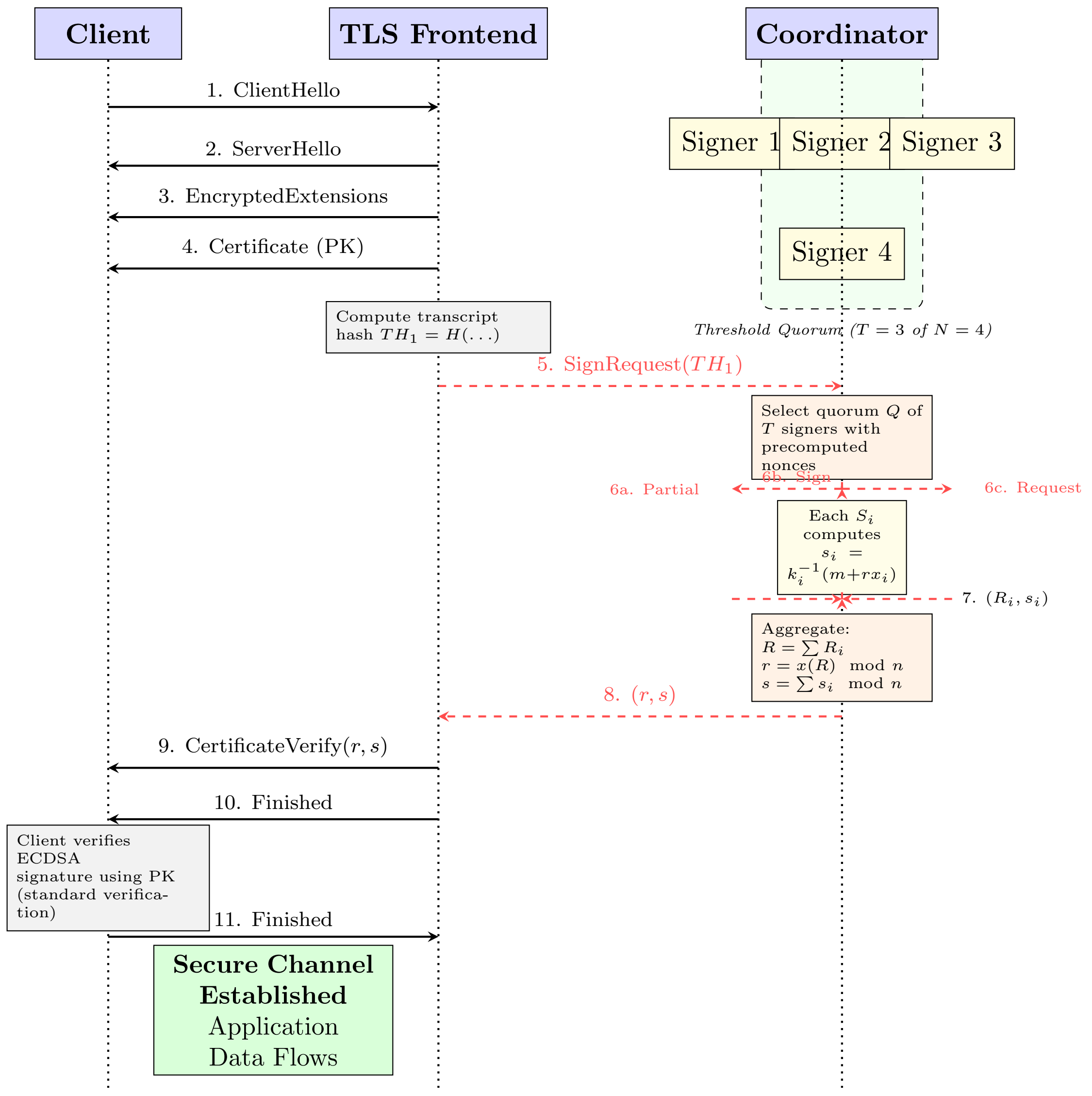
Client
TLS Frontend
Coordinator
1. ClientHello
2. ServerHello
3. EncryptedExtensions
4. Certificate (PK)
Signer 1
Signer 2
Signer 3
Signer 4
Threshold Quorum ($T = 3$ of $N = 4$)
Compute transcript hash $TH_1 = H(\ldots)$
5. SignRequest(TH_1)
Select quorum Q of T signers with precomputed nonces
6a. Partial
6b. Sign
6c. Request
Each S_i computes $s_i = k_i^{-1}(m+rx_i)$
7. (R_i, s_i)
Aggregate: $R = \sum R_i$ $r = x(R) \mod n$ $s = \sum s_i \mod n$
8. (r, s)
9. CertificateVerify(r, s)
10. Finished
Client verifies ECDSA signature using PK (standard verification)
11. Finished
Secure Channel Established
Application Data Flows

Figure 2: Complete TLS 1.3 handshake with threshold ECDSA signing integration. Steps 5–8 (red dashed) represent the threshold signing subprotocol invoked during CertificateVerify generation. Client remains unaware of the distributed signing process.

- `tls_threshold_requests_total` (Counter): Total signing requests received.
- `tls_threshold_coordinator_queue_depth` (Gauge): Pending requests at coordinator.

27 References

- RFC 8446: TLS 1.3.
- Lindell, Nof (2021): Fast Secure Two-Party ECDSA.
- Gennaro, Goldfeder (2018): Fast Multiparty Threshold ECDSA.
- SEC 1: Elliptic Curve Cryptography.
- OpenSSL Provider API Docs.
- FROST Draft: Round-Optimized Schnorr Threshold Signatures.