Kingdom of Saudi Arabia Ministry of Education University of Jeddah College of Computer Science and Engineering Department of Cybersecurity		المملكة العربية السعودية وزارة التعليم جامعة جدّة كلية علوم وهندسة الحاسب قسم الأمن السيبراني

Project

CCCY 312 Cryptography

First Semester 2025-2026

This page must be used as cover page of your report.

Due Date: 6 December 2025

Submission: Each Student MUST Upload his/her COPY to Backboard

Group Size: Maximum of Three Students & Minimum of Two Students (Preferred)

Student Name: Raghad Alotaibi **Student ID:** 2311917

Student Name: Dina Alharbi **Student ID:** 2310592

Student Name: Seham Alqarni **Student ID:** 2312040

Instructor Name	Section
Rawan	CY1

Contents

Accountable Threshold EdDSA: A Secure and Traceable Threshold Signature Protocol

Abstract

Threshold signatures enable a group of multiple participants to collaboratively produce a valid single digital signature without reconstructing the full private key. This capability establishes distributed trust in multi party systems such as blockchain, identity management, and secure authentication frameworks. Conventional threshold implementations of EdDSA lack integrated mechanisms for accountability and signer privacy and do not support proactive security for long term key protection. To bridge these gaps we propose a new protocol named **Accountable Threshold EdDSA**.

This scheme combines ElGamal encryption with zero knowledge proofs and incorporates a method for proactive key refresh. Our proposed protocol keeps both the threshold value and the specific set of signers confidential from external observers yet provides the ability for a designated tracer to identify signers if misconduct occurs. Furthermore the private key shares held by participants can be periodically and securely updated without modifying the group's public key thereby reducing risks from key exposure over time.

1. Introduction

Digital signatures ensure authenticity, integrity, and non-repudiation across modern communication systems. Traditional schemes such as RSA generate relatively large signatures and rely on a single private key holder, introducing a significant single point of failure.

This project presents an **Accountable Threshold EdDSA** protocol, a multi-party short digital signature scheme built on **elliptic curve cryptography (ECC)**. This design distributes key ownership among multiple authorized participants while still allowing them to jointly generate a single compact signature.

This report outlines the essential cryptographic foundations, presents the complete structure of the protocol, evaluates its security, examines relevant attacks and mitigations, compares its security properties to conventional elliptic-curve signature schemes, and assesses its compliance with national cybersecurity requirements.

2. Background and Preliminaries

2.1 Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) represents one of the most advanced and widely adopted areas of modern cryptography. It uses the set of points on a curve defined over a finite field, commonly written as:

$$y^2 = x^3 + ax + b$$

These points form a group where performing scalar multiplication is easy, but reversing it is computationally infeasible. This hardness, known as the **Elliptic-Curve Discrete Logarithm Problem (ECDLP)**, gives ECC its security [1].

ECC is widely used because it avoids the index-calculus attacks that weaken classical finite-filed systems. Modern cryptographic standards select curves that avoid known weaknesses, allowing ECC to maintain high confidence and efficiency in real-world applications such as digital signatures and secure communication [1].

2.2 Short Digital Signatures

2.2.1 Concept

A short digital signature is simply a type of digital signature that's designed to be as small as possible while still staying secure. The main idea is to shrink the number of bits used in the signature so it takes up less space and is faster to send or store, without weakening the protection it provides. This makes the signature lighter and more efficient, but still fully trustworthy when verifying it [2].

2.2.2 What Makes a Signature "Short"

A digital signature is considered short when it uses fewer bits but still keeps the same level of security. This happens because the scheme relies on elliptic curve cryptography, which gives strong protection without needing large numbers. With this type of design, the signature becomes smaller, verification is faster, and the whole process works efficiently even on devices that have limited resources. The use of small keys, compact point representation, and simple operations all help make the signature short [2].

2.2.3 Real-World Relevance

Short digital signatures are useful in environments where devices have limited storage, power, and processing capability. They are commonly used in IoT devices, medical implants, and wireless sensor networks because their small size reduces communication overhead and helps save energy while still providing secure authentication. These signatures are also suitable for government digital platforms that follow NCA requirements, since these systems rely on efficient elliptic curve signatures to support secure and scalable services [3].

2.3 Existing ECC-Based Short Signatures

2.3.2 BLS (Bohen-Lynn-Shacham) Signatures

Core Idea:
BLS signatures use a bilinear pairing that is a special mathematical function that connects two elliptic curve points and maps them into another algebraic group in a consistent way [3].

How signing works :

Hash the message $H(m)$ maps the message *m* to a point on the cu

Multiply that by your secret key x: $\sigma = H(m)^{x}$

The verifier checks the key equation:

$$e(\sigma, P) = e(H(m), Ppub)$$

where e is a bilinear pairing that satisfies the property:

$$e(aP,\ bQ) = e(P,\ Q)^{ab}$$

This property allows multiple signers signatures to be combined into **one aggregated Signature [3]**:

$$\sigma agg = \prod i\ \ \sigma i$$

Problem They Solve:
BLS compresses multiple independent signatures into a single short 48\96 byte signature. It reduces blockchain or distributed-ledger storage overhead and improves verification efficiency by combining them mathematically using the bilinear property of pairings [3].

Signature Size & Efficiency:

- Single signature: 48 bytes (in group G1) or 96 bytes (in G2) [3].
- Verification requires two pairing operations computationally heavy but done in constant time [3].

Strengths:

- Extremely compact that signature is only one elliptic curve point regardless of how many signers [3].
- aggregation friendly where dozens or hundreds of signatures compress into one [3].
- Proven secure under the **Computational Diffie Hellman (CDH)** assumption [3].

Weaknesses (Mathematical Structure):

- The pairing function e :G1×G2→GT is an algebraic map that requires finite field exponentiation and Miller's algorithm and that introduces computational overhead; each pairing costs ≈ 10× more than a scalar multiplication so the verification is heavier even though its signatures are short [3].

- BLS needs special elliptic curves that support efficient pairings (ex: BLS12-381) If you make even a tiny mistake when generating curve parameters it can break security [3].
- Requires correct subgroup checks to prevent if an input lies outside that subgroup an attacker could craft small order elements that make $e(\sigma, P) = e(H(m), Ppub)$ evaluate as true by coincidence even though σ is not a valid signature [3].

Known Attacks:

- **Rogue-key aggregation:**
  Malicious users can publish manipulated public keys (inverses or linear combinations of others keys) to forge aggregated signatures that appear valid [3].
- **Side-channel leaks:**
  The pairing computations in $e(\sigma, P) = e(H(m), Ppub)$ involve repeated finite field multiplications. If implemented naively the time taken for these multiplications can depend on secret data [timing or power variations [3].
- **Curve parameter tampering:**
  If non standard or incorrectly generated curve parameters are used (like unsafe embedding degree, nonprime group order) the hardness of the discrete logarithm problem collapses [3].
- **Small subgroup attack (also called invalid curve attack):**
  An attacker submits inputs $(\sigma,\ P,\ or\ P\square_{u}b)$ that lie outside the correct subgroup of prime order q expected by the protocol. Because of how bilinear pairings work a small subgroup wrap around after only a few multiplications so an attacker can brute force the right fake σ that still satisfies the verification equation [3].

2.3.2 Schnorr Signatures on Elliptic Curves (EC-Schnorr)

Core Idea:

Schnorr signatures are simple yet powerful the signer generates a random nonce k and computes

$$s = k + e \cdot x \,(mod n) \quad \text{with} \quad e = H(R \| P \| m)$$

Here x is the private key, $P = xG$ is the public key and H is a hash function and the verification Equation [4]: $sG = R + eP$

Problem They Solve:

Schnorr signatures eliminate the modular inversion step required by ECDSA, which was a common source of errors and side channel vulnerabilities. The Schnorr design provides mathematical simplicity and provable security based on the Elliptic Curve Discrete Logarithm Problem (ECDLP) [4].

Signature Size & Efficiency:

Always 64 bytes. Requires only two scalar multiplications for verification making it extremely fast for embedded devices or IoT systems [4].

Strengths [4]:

- Simple algebra, fast, and easy to extend to threshold signing.
- Proven secure under standard cryptographic assumptions based on well established ECDLP hardness.
- Naturally flexible supports threshold or multi signature extensions.

Weaknesses [4]:

- If two signatures reuse the same nonce k their difference reveals the private key k: $s1 - s2 = (e1 - e2) \cdot x \ (mod\ n)$ making x the private key to be solved easily
- Signature malleability If R can be represented ambiguously on the curve (compressed vs uncompressed format) attackers can slightly modify the signature without invalidating it.
- Requires truly random or deterministic nonces (RFC 6979) weak random generators can leak key material.

Known Attacks [4]:

- Nonce reuse during signing can lead to full private key recovery through simple algebraic analysis.
- Replay or malleability attacks can occur if messages are not domain separated or if signature encoding is ambiguous
- Side channel leakage via timing or power can reveal partial information about the private scalar.

2.3.3 EdDSA / ED25519 (RFC 8032)

Core Idea:

EdDSA improves Schnorr by deriving the nonce into a deterministic and side channel resistant version [5].

The signature is computed as:

$$S = r + h \cdot a \ (mod\ \ell) \quad with \quad h = H(R \parallel A \parallel m)$$

The verification equation:

$$S \cdot G = R + h \cdot A$$

The scheme uses the **Twisted Edwards** curve defined by [5]:

$$-x^2 + y^2 = 1 + dx^2y^2 \quad where \quad d =- 121666 \div 121665 \ over\ F2^{55519}$$

Problem They Solve:

Deterministic nonce generation eliminates ECDSA's catastrophic RNG failures by removing random number dependence entirely and provides constant-time execution to prevent timing leaks [5].

Signature Size & Efficiency

Fixed 64 bytes. Verification ≈ 70 µs on modern CPUs

Extremely efficient and secure for both software and hardware implementations [5].

Strengths [5]:

- No RNG dependency by deriving the nonce deterministically from the message and private key using a hash.
- Fast and constant time implementation making it “side channel hardened”.
- Compact and fast with 64 byte signatures and 32 byte public keys perfect for constrained environments or embedded hardware.

Weaknesses [5]:

- Deterministic nonces mean once the private key seed is exposed an attacker can recompute all nonces used in past signatures.
- Vulnerable to fault injection: flipping a bit in r or a during computation can create exploitable correlations.

Known Attacks [5]:

- Cache-timing and power-analysis attacks on hardware.
- Deterministic Nonce Key Recovery Attack: if an attacker ever learns the private seed they can recompute every nonce r used in past signatures or forge new ones that look valid.

3. *Proposed Protocol: Accountable Threshold EdDSA*

Threshold Accountable Privacy-Preserving Signature with Proactive Refresh (TAPS-PR) is a short and efficient elliptic-curve–based signature scheme that extends EdDSA into a distributed environment. Instead of relying on a single signer, TAPS-PR lets a group of parties collaboratively produce one compact signature. What makes it stand out is its built-in accountability: if any participant behaves maliciously or submits an invalid partial signature, the protocol provides a way to identify exactly who caused the issue [5].

3.1 Motivation for Choosing the Protocol

Accountable Threshold EdDSA introduces several enhancements that address the shortcomings of traditional schemes while maintaining compatibility with EdDSA’s signature format.

- **Accountability:** Malicious or invalid signers can be identified through cryptographic tracing, preventing undetected misuse of signing authority.
- **Quorum Privacy:** The threshold value t and the identities of participating signers remain hidden unless tracing is invoked, protecting sensitive organizational structure.
- **Traceability:** Although the quorum is private, it can be revealed in cases involving fraud or disputes, enabling controlled and auditable investigation.
- **Proactive Key Refresh:** Private key shares are periodically updated without altering the public key, preventing gradual compromise.
- **Improved Security:** TAPS-PR addresses major limitations of traditional threshold schemes.

These properties are essential for digital identity systems, distributed certification authorities, multi-party authentication, and financial authorization frameworks [5].

3.2 Security Requirements and System Model

The Accountable Threshold EdDSA protocol operates in a distributed setting with multiple signing parties, a threshold value *t*, and additional entities responsible for aggregation, verification, and accountability. Any subset of at least *t* honest participants can jointly produce a valid signature, while the protocol ensures privacy of the signing group and enables tracing under misuse [5].

Participants

- $P_1,\ldots, P_n$ **:** signing parties holding private key shares and generating partial signatures.
- Threshold value *t***:** minimum number signers required.
- **Combiner:** collects and verifies partial signatures, encrypts the final signature scalar, and outputs the final signature.
- **Verifier:** checks the validity of the resulting signature and ZK proof.
- **Tracer:** decrypts the encrypted signature scalar and identifies the actual signing quorum when accountability is invoked.

Secure Channels

The protocol requires authenticated, integrity-protected channels for exchanging [5]:

- Nonce commitments
- Partial signatures
- Zero-knowledge proofs
- Key-refresh values

Trust Assumptions

- Fewer than *t* corrupted parties cannot forge a valid threshold signature.
- The tracer performs accountability correctly when invoked.
- The combiner does not alter valid contributions.
- Hash functions and elliptic-curve group operations follow standard cryptographic assumptions.

Security Goals

- **Unforgeability:** No signature can be produced without *t* valid signers.
- **Accountability:** Signers who produce an incorrect or malicious share can be identified during tracing.
- **Robustness:** Signing succeeds as long as *t* honest parties participate.
- **Non-Repudiation:** Faulty signers cannot deny misbehavior.
- **Coalition-Resistance:** Fewer than *t* corrupted parties gain no advantage.
- **Quorum Privacy:** The threshold value and the signing quorum remain hidden from external observers unless accountability is explicitly triggered.
- **Proactive Key Protection:** Private key shares are periodically refreshed without changing public keys.

3.3 Mathematical Setup

The protocol is defined over standard elliptic-curve and hash-based primitives consistent with EdDSA. The mathematical setup establishes the algebraic structures, parameters, and notation used across key generation, signing, verification, tracing, and proactive key refresh [5].

Elliptic-Curve Group [5]

- A cyclic group formed from points on an elliptic curve, where the total number of points in the group G is a prime number q.
- A base point $g \in G$
- An additional independent generator $h \in G$ used for commitments, ElGamal encryption, and privacy-preserving operations.
- A hash function H

All group operations involving G are performed using elliptic-curve arithmetic.

Public Parameters [5]

The system initializes the following public parameters:

$$EdParams = (G, q, g, \text{curve parameters}, H)$$

Where:

- G: elliptic-curve group
- q: large prime group order
- g: base point
- H: cryptographic hash function used for nonce derivation and challenge computation
- h: auxiliary generator used for encryption and commitments

These values are publicly known and consistent with EdDSA's structure.

Although the mathematical setup is expressed over a generic elliptic-curve group G, the protocol remains fully compatible with Ed25519 which is the standard real-world implementation of EdDSA. Ed25519 operates on a twisted Edwards curve and outputs compact 64-byte signatures (R, s). The threshold logic and accountability layer introduced in this design preserve this signature structure, allowing the protocol to integrate directly into systems that already rely on Ed25519 without requiring any changes to existing formats or verification workflows [5].

Signer Key Pairs [5]

Each signer P_i:

- Generates private seed $seed_i$
- A derives a scalar $x_i \in Z_q$
- Sets: $sk_i = x_i$
- Computes public key: $pk_i = g^{xi}$

All signers public keys collectively form: $pk' = (pk_1, pk_2, \ldots, pk_n)$

Tracer Key Pair [5]

The tracer generates its private key: $sk_e \xleftarrow{\$} \mathbb{Z}_q$

And computes its public key: $pk_i \leftarrow g^{sk_e}$

Threshold and Lagrange Coefficients [5]

A threshold value *t* is selected for signing.

Given a signing quorum $J \subseteq \{1, \ldots, n\}$ with $|J| = t$

$$\lambda_i^J = \prod_{j \in J, j \neq i} \frac{j}{j-i} \; mod\ q$$

These coefficients enable correct interpolation of partial signatures.

3.4 Key Generation

The Accountable Threshold EdDSA protocol does not rely on a trusted dealer. Instead, each signer independently generates its own private share, and the public key is formed through collective contributions. The resulting signing key is never reconstructed and remains distributed among the parties throughout the system [5].

Signer Key Generation [5]

Each participant *Pi*:

1. samples a private $seed_i$ and computes $H(seed_i)$
2. derives a scalar $x_i \in Z_q$
3. sets its initial secret key as $sk_i = x_i$
4. and computes the corresponding public key $pk_i = g^{xi}$

These computations are done locally, so no private values are shared with other parties.

Public Key Assembly:

All participants broadcast their public keys pk_i. The global public key is formed as the collection:

$$pk' = (pk_1, pk_2, ..., pk_n)$$

This set is used during signing, verification, and tracing. Since each signer independently generates its own share, the final signing key is effectively a distributed secret without ever being reconstructed.

Combiner Key Generation [5]

The combiner:

1. Encrypts the threshold value *t* using ElGamal to hide it:

$$(T_0, T_1) = g^{\psi}, g^t h^{\psi} \text{ , where } \psi \xleftarrow{\$} \mathbb{Z}_q$$

2. Generates an authentication keypair (sk_{cs}, pk_{cs}).

Tracer Key Generation [5]

The tracer generates its private key: $sk_e \xleftarrow{\$} \mathbb{Z}_q$

And computes its public key: $pk_i \leftarrow g^{sk_e}$

No Secret Key Reconstruction [5]

Unlike classical DKG protocols, this scheme does not produce a single combined secret key stored by anyone. The private key exists only in the form of distributed shares:

$$sk_1, sk_2, \ldots , sk_n$$

and remains distributed throughout the protocol's lifetime.

Result

At the end of this phase:

- Each signer holds a private key share sk_i
- All signer public keys form the shared vector pk'.
- The threshold value t is hidden via ElGamal.
- The tracer holds its own key pair for accountability.

3.5 Signing Phase

The signing phase allows any quorum of t signers to collaboratively produce a single EdDSA-compatible signature. The protocol proceeds in two rounds and incorporates zero-knowledge proofs to ensure correctness, key freshness, and signer accountability without revealing the identities of the participating parties [5].

Round 1: Nonce Generation [5]

Each signer P_i:

1. Derives a nonce from its private seed and the message m

$$r_i = H(seed_i, m)$$

2. Computes the nonce commitment

$$R_i = g^{ri}$$

3. Sends R_i to all other signers in the quorum.

Round 2: Partial Signature Computation [5]

When all commitments are received, the combined nonce is computed as

$$R = \prod_{i \in J} R_i$$

All signers compute the challenge value

$$c = H(R, pk, m)$$

Each signer computes its weighted partial signature using its private key share and Lagrange coefficient:

$$s_i = r_i + \lambda_i^J \cdot c \cdot sk_i \mod q$$

They also generate zero-knowledge proof π_i (Proof S_0) to prove that the partial signature is correct and uses the latest refreshed key, preventing malicious or outdated usage, as required by TAPS-PR.

Finally, send (R_i, s_i, π_i) to the combiner.

Combiner Aggregation [5]

1. Verify each proof π_i and if any proof fails, the combiner aborts.

2. Compute aggregated signature scalar:

$$s = \sum_{i \in J} s_i \mod q$$

3. Encrypt the aggregated signature value using the tracer's public key pk_t via ElGamal:

$$ct = (c_0, c_1) = (g^{\rho}, g^{s} \cdot pk_t^{\rho}) \text{ where } \rho \xleftarrow{\$} \mathbb{Z}_q$$

4. Produce zero-knowledge proof $\pi(Proof\ S_1)$ showing that:

   - *ct* correctly encrypts a valid signature scalar consistent with *R*
   - The signature comes from exactly *t* signers
   - Identities if the signers remain hidden

5. Sign the proof with the combiner's key to produce a digital authentication tag *tg*.

Final Signature Output

$$\sigma=(R, ct, \pi, tg)$$

where:

- R is the aggregated nonce
- ct is the encrypted signature scalar
- π is proof S_1
- tg is the combiner's authentication tag

The resulting signature is the same size as a standard EdDSA signature while preserving quorum privacy and enabling accountability when required.

3.6 Zero Knowledge Proofs

Proof S_0 (Signer ZKP)

Ensures each partial signature is correct and uses the updated secret key share [5].

Proof S_1 (Combiner ZKP)

Ensures that [5]:

- ct encrypts the correct aggregated value s
- exactly t signers contributed valid shares
- the signing quorum remains hidden.

Both proofs are non-interactive, created using the Fiat–Shamir transformation. Proof S_0 is generated locally by each signer when producing a partial signature, while Proof S_1 is constructed by the combiner as part of assembling the final encrypted signature [5] [13].

3.7 Verification Phase

The verifier receives the message m and the threshold signature [5]

$$\sigma=(R, ct, \pi, tg)$$

Verification proceeds as follows [5]:

1. Check the combiner's authenticity:
   The verifier uses the combiner's public verification key to ensure that the pair (m, R, ct, π) has not been tampered with.

$$SIG.Verify(pk_{cs}, (m, R, ct, \pi), tg) = 1$$

2. Validate the zero-knowledge proof:
   The proof π demonstrates three statements simultaneously:
   - Correct EdDSA signature structure
   - Consistency of ct
   - Exactly t signers
3. Accept or reject accordingly:
   If both the combiner's tag and the zero-knowledge proof are valid, the verifier accepts σ. Otherwise, the output is rejected.

This phase ensures that the verifier learns only that the signature is valid and produced by a threshold quorum without learning which parties signed or what the threshold t is.

3.8 Accountability Mechanism

Accountability is achieved by allowing only a designated tracer to reveal which t parties generated a threshold signature. During normal operation, the identities of the signers remain fully hidden [5].

If tracer is invoked, the tracer:

1. Recovers

$$g^s = \frac{C_1}{C_0^{ske}}$$

   then identifies the signing quorum using public keys.

2. Determines which subset J of size t satisfies the signature using public keys pk_i
3. Outputs the real signing quorum if it exists, otherwise output fail

This mechanism ensures that signer identities remain private unless tracing is explicitly needed, while still enabling full attribution in case of misuse or disputed signatures.

3.9 Proactive Key Refresh

To prevent long-term key exposure, each signer updates its private key share without

changing the public key.

Key-Refresh Share Generation [5]

Each signer P_i:

1. Selects a random polynomial f_i of degree $t - 1$ such that $fi(0) = 0$
2. Compute

$$\delta_{i,j} = \sum_{\ell=1}^{t-1} a_{i,\ell} \cdot j^{\ell}$$

3. Commitment

$$A_{i,j} = g^{\delta_{i,j}}$$

4. Send $\delta_{i,j}$ and $A_{i,j}$ to party P_j

Private Key Update [5]

Party P_j verifies that $A_{i,j} = g^{\delta_{i,j}}$. If valid, keep $\delta_{i,j}$, otherwise, reject. After verifying all refresh shares and commitments, party P_j updates the private key share as:

$$sk_j' = sk_j + \sum_{i=1}^{n} \delta_{i,j}$$

4. Novelty and Contributions

Our proposed Threshold EdDSA scheme introduces several key contributions:

- It is the first EdDSA-based threshold design that combines accountability, privacy, and proactive key refresh within a single protocol [6].
- It preserves the original EdDSA signature size, ensuring compact and efficient signatures [6].
- It strengthens overall security by addressing limitations found in previous threshold signature schemes [6].

5. Security Analysis and Cryptanalysis

5.1 Underlying Hard Problem

The protocol's security relies on two main assumptions. The **Elliptic Curve Discrete Logarithm Problem (ECDLP)** ensures that private key shares cannot be recovered from their corresponding public keys. The **Decisional Diffie Hellman (DDH)** assumption underpins the **ElGamal style encryption** used to hide the signature value and the signing quorum ensuring that these encrypted components reveal no information about the underlying secrets [5].

Under the **DDH** assumption the attacker cannot distinguish between a valid mathematical relationship (g^a, g^b, g^{ab}) and a random triple (g^a, g^b, g^C) means that even if ciphertexts or commitments are observed no information about the secret exponents can be inferred. Thus the ElGamal ciphertexts and zero knowledge proofs and proactive key refresh steps remain computationally indistinguishable from random data [5].

With these both assumptions it guarantees that recovering private shares or forging valid signatures or revealing the hidden quorum is impossible therefore ensuring key secrecy, signature integrity, and quorum privacy [5].

5.2 Brute-Force Attacks

Key space resistance

Each signer's private key xi and nonce ri are uniformly random 252-bit integers within the Ed25519 group order. An exhaustive search over 2^{252} possibilities exceed all feasible computational limits [8].

Encrypted signature component

The ciphertext $ct = (c_0, c_1) = (g^r, s\cdot pk_t^r)$ protecting the aggregated signature value follows the **ElGamal encryption scheme** which is semantically secure under the **Decisional Diffie Hellman** assumption.Recovering r or s from Ct would require solving the discrete logarithm problem for g^r which remains computationally infeasible under the **Elliptic Curve Discrete Logarithm Problem** [9].

Hash derived challenges

Each challenge $e = H(R, pk, m)$ is computed using SHA-512 offering 256-bit pre image and collision resistance.
No possible algorithm can invert or find collisions in these hashes to influence signing outcomes or force predictable challenges [10].

5.3 Classical Cryptanalysis

5.3.1 Key-Recovery Attacks

Each participant holds a secret share $si = f(i)$ from a random polynomial $f(x)$ of degree t. Recovering the master secret $a0$ requires at least $t + 1$ valid shares any smaller subset produces an underdetermined system modulo ℓ. The hardness of ECDLP ensures that even correlated public information (commitments or partial signatures) cannot reveal the underlying private values [11].

During **proactive key refresh** the protocol adds a zero-sum random polynomial to $f(x)$ rotating every share without altering the global public key. This prevents long term exposure attacks, even if some historical shares are compromised.
Potential vectors such as malformed zero knowledge proofs , compromised tracer keys or correlation analysis across refresh epochs remain neutralized by the verification layer and polynomial randomness [11].

5.3.2 Forgery Attacks

Producing a valid forged signature (R, ct, π, tg) requires solving the verification equation

$$S \cdot G = R + eA$$

without possessing t or more legitimate private key shares [12].

Since $e = H(R, pk, m)$ is an unpredictable hash challenge bound to each message and $A = g\sum xi\lambda i$ depends on securely distributed shares. An attacker attempting to fabricate valid S and R must effectively compute the unknown discrete logarithm relating A and G. This directly reduces the forgery problem [12].

The **accountability layer** further strengthens unforgeability:

- Each partial signature is tied to a signer identity through a zero knowledge proof πi proving correctness of computation without revealing the private share [12].

- The combiner validates every πi before aggregation :any invalid proof aborts the signing process and precisely identifies the malicious participant [12].

Therefore **existential unforgeability under chosen-message attacks (EUF-CMA)** holds as long as fewer than t shares are compromised [12].

5.3.3 Attacks on Zero-Knowledge Proofs

Zero-knowledge proofs (ZKPs) ensure that each partial signature was computed correctly using their legitimate secret share of the private key without revealing the secret itself.

Potential attacks:

- **Rewinding Attacks:** in interactive ZKPs the attacker could "rewind" the verifier asking them to check multiple challenges to extract and exploit hidden information.

Accountable Threshold EdDSA avoids that danger by requiring every participant to commit their nonce Ri first $ci = H(Ri \,||\, ctx)$ then when they "reveal" Ri the combiner checks that it matches the earlier commitment ci.
This structure makes ZKPs non interactive / no rewinding possible [13].

- **Malicious Prover Behavior :** if a participant actively trying to cheat the system by submitting a random or incorrect proof πi to disrupt the signing process or conceal invalid math.
Accountable Threshold EdDSA avoids that danger by using publicly verifiable proofs that means that all parties can check the math on each proof.
If one participant proof does not match the expected relationship between [nonce-public key-ciphertext] verification fails only for that signer [3].

5.4 Quantum Threats

Quantum computing introduces theoretical risks to elliptic curve cryptosystems
Two famous quantum algorithms **Shor's** and **Grover's** are particularly relevant Since **Accountable Threshold EdDSA** is built on:

- elliptic-curve math ECDLP
- ElGamal encryption which also relies on the discrete logarithm problem.

both would be broken by Shor's algorithm
That means: private keys - partial key shares - ElGamal ciphertexts could all be recovered once scalable quantum computers exist.

Even with that system's accountability - ZK proofs - threshold structure don't introduce new vulnerabilities [15].

Theoretical Mitigations Against Future Quantum Attacks:

1. **Hybrid Classical and Post-Quantum Signatures**
The system can output both an elliptic-curve signature and a lightweight post-quantum signature for the same message. This hybrid approach maintains compatibility with existing infrastructure while ensuring that signatures remain secure even if ECC becomes weak under quantum attacks [14].

2. **Crypto-Agility**
The protocol should be designed in a modular way that allows signature schemes, encryption mechanisms, or hash functions to be replaced without redesigning the entire system. Crypto-agility ensures a smooth transition to post-quantum primitives when needed [14].

3. **Short Key Lifetimes and Frequent Rotation**
Reducing key lifetime and performing regular rotations limits the exposure of long-term private keys. Even if future quantum techniques weaken ECC, the window of vulnerability is minimized [14].

6. *Side-Channel Attacks Analysis*

The Accountable operations expose several implementation-level leakage points even if the mathematical protocol is secure. The most relevant side-channel threats for this protocol are:

1. Timing Attacks

Processes like nonce generation, scalar multiplication, and zero-knowledge proof construction can unintentionally reveal information if they take different amounts of time depending on the underlying secret values. When these operations are not implemented in true constant time, an attacker who measures execution delays can begin inferring patterns related to private key shares or nonce material. Even small timing variations, when collected over many runs, can gradually expose sensitive information [6].

Mitigation Strategy

Run all signature-related arithmetic in constant time, avoid secret-dependent branching, and ensure uniform execution paths for nonce generation, partial signatures, and ZK proof creation.

2. Power and Electromagnetic Analysis

Power and electromagnetic-based side-channel attacks are still some of the most effective ways to target elliptic-curve systems. When a signer runs local operations like producing a partial signature or updating a key share, the device leaks tiny physical signals that depend on the secret data being processed. If these steps aren't properly protected, attackers can analyze these signals using techniques such as Simple Power Analysis (SPA), Differential Power Analysis (DPA), or even newer deep-learning–based profiling methods. With enough measurements, these methods can reveal meaningful information about nonce values or portions of a participant's private key share [6].

Mitigation Strategy

Use masking, randomized internal computations, and noise injection to decorrelate physical leakage from sensitive values. Hardware performing signing may also need EM shielding to prevent remote probing.

3. Fault Injection Attacks

The protocol depends on each signer generating correct nonce commitments, producing valid partial signatures, and contributing honest key-update values. If an attacker manages to inject faults through voltage glitches, clock tampering, electromagnetic pulses, or other hardware interference, they can push a device into producing incorrect outputs. These faulty results can unintentionally expose patterns linked to the signer's internal state. In practice, a single corrupted partial signature or a malformed ciphertext may leak enough information to reveal part of a private key share or weaken the protocol's accountability guarantees [6].

Mitigation Strategy

Verify every received nonce commitment, partial signature, and key-refresh value with consistency checks. Reject malformed inputs and incorporate protections against unusual voltage/clock behavior. Ensuring robust validation prevents accepting adversarially faulted data.

7. *Implementation Overview*

This implementation provides a simplified demonstration of the core ideas behind Accountable Threshold EdDSA. The code simulates distributed key sharing, nonce generation, partial signing, and signature aggregation using elliptic-curve operations. It also includes a tampering test to show how the protocol detects invalid partial signatures. The following snippets highlight the main components of the implementation.

1. Initializing all cryptographic dependencies, selecting all elliptic curves, and defining the global parameters G and q used throughout the threshold signature scheme.

```
import secrets
import hashlib
import random
from dataclasses import dataclass
from typing import List, Dict, Tuple
from tinyec import registry
```

```
curve = registry.get_curve("secp256r1")
q = curve.field.n       # group order
G = curve.g             # base point
```

2. These helper functions implement hashing, point encoding, modular inversion, and Lagrange coefficient computation required for combining partial signatures.

```
def H_int(*parts: bytes) -> int:
    """Hash arbitrary bytes -> integer mod q."""
    h = hashlib.sha256()
    for p in parts:
        h.update(p)
    return int.from_bytes(h.digest(), "big") % q


def point_to_bytes(P) -> bytes:
    """Serialize EC point as x||y (32 bytes each)."""
    x = P.x.to_bytes(32, "big")
    y = P.y.to_bytes(32, "big")
    return x + y
```

```
def modinv(a: int, m: int) -> int:
    """Modular inverse a^{-1} mod m."""
    return pow(a, -1, m)


def lagrange_coefficient(i: int, J: List[int], modulus: int) -> int:
    """
    Lagrange coefficient λ_i^J for index i in quorum J:
        λ_i = ∏_{j∈J, j≠i} (j / (j - i)) mod q
    """
    num, den = 1, 1
    for j in J:
        if j == i:
            continue
        num = (num * j) % modulus
        den = (den * (j - i)) % modulus
    return (num * modinv(den % modulus, modulus)) % modulus
```

3. This section generates Shamir secret shares so that no single participant ever holds the full private key, enabling threshold security.

```python
def shamir_generate_shares(secret: int, t: int, n: int, modulus: int) -> List[int]:
    """
    Shamir (t, n) secret sharing:
        f(x) = secret + a1 x + ... + a_{t-1} x^{t-1}
        share_i = f(i)
    """
    coeffs = [secret] + [secrets.randbelow(modulus) for _ in range(t - 1)]

    def f(x: int) -> int:
        val, power = 0, 1
        for a in coeffs:
            val = (val + a * power) % modulus
            power = (power * x) % modulus
        return val

    return [f(i) for i in range(1, n + 1)]
```

4. Each signer is represented as an object capable of generating nonces and producing partial signatures using the private key share.

```python
@dataclass
class Party:
    index: int
    sk_share: int        # Shamir share
    pk_global: object    # PK = xG
    pk_share: object     # sk_share * G
```

5. Each party generates a random nonce and computes its weighted partial signature as required

```python
    def generate_nonce(self) -> Tuple[int, object]:
        """Return (k_i, R_i = k_i G)."""
        k_i = secrets.randbelow(q)
        R_i = k_i * G
        return k_i, R_i

    def partial_sign(self, k_i: int, c: int, lam_i: int) -> int:
        """Compute s_i = k_i + λ_i c sk_i mod q."""
        return (k_i + lam_i * c * self.sk_share) % q
```

6. This function verifies the final aggregated signature (R, s) by checking the Schnorr

```
def verify_threshold_signature(R, s: int, PK, message: bytes) -> bool:
    """Check sG == R + cPK for c = H(R,PK,m)."""
    c = H_int(point_to_bytes(R), point_to_bytes(PK), message)
    return s * G == R + c * PK
```

verification equation.

7. The main routine initializes the system parameters, generates the master key, distributes shares, and sets up the threshold signing environment.

```
# Parameters
n, t = 5, 3
message = b"Threshold EdDSA-style demo"

# Master secret and global public key
x = secrets.randbelow(q)
PK = x * G
print("Master secret x =", x)
print("Public key PK =", (PK.x, PK.y))

# Shamir shares (simple DKG)
shares = shamir_generate_shares(x, t, n, q)
parties = [Party(i, shares[i - 1], PK, shares[i - 1] * G)
           for i in range(1, n + 1)]

# Proactive refresh
print("\n[+] Running proactive key refresh...")
proactive_refresh(parties, t)

# Signing quorum
J = [1, 2, 3]
print("\nSigning quorum J =", J)
```

8. All selected parties broadcast their nonce commitments, which are combined to form the global nonce R.

```
nonces: Dict[int, Tuple[int, object]] = {
    i: parties[i - 1].generate_nonce() for i in J
}

R = None
for i in J:
    _, R_i = nonces[i]
    R = R_i if R is None else R + R_i
```

9. Each signer computes the partial signature using its nonce, Lagrange weight, and private key share.

```python
partials: Dict[int, int] = {}
for i in J:
    k_i, R_i = nonces[i]
    lam_i = lagrange_coefficient(i, J, q)
    s_i = parties[i - 1].partial_sign(k_i, c, lam_i)
    partials[i] = s_i
```

10. All partial signatures are combined to produce a valid threshold signature identical in structure to EdDSA.

```python
s = sum(partials.values()) % q
print("\nFinal threshold signature:")
```

11. This block demonstrates the accountability mechanism by encrypting the final signature value for later tracing.

```python
def elgamal_setup() -> Tuple[int, int, int, int]:
    """Toy ElGamal over fixed 192-bit prime."""
    p = 2**192 - 2**64 - 1
    g = 5
    x = secrets.randbelow(p - 2) + 1
    y = pow(g, x, p)
    return p, g, x, y


def elgamal_encrypt(p: int, g: int, y: int, m: int) -> Tuple[int, int]:
    """
    Additive ElGamal:
        c0 = g^k mod p
        c1 = m + y^k mod p
    """
    k = secrets.randbelow(p - 2) + 1
    c0 = pow(g, k, p)
    c1 = (m + pow(y, k, p)) % p
    return c0, c1


def elgamal_decrypt(p: int, x: int, c0: int, c1: int) -> int:
    """Decrypt additive ElGamal: m = c1 - c0^x mod p."""
    shared = pow(c0, x, p)
    return (c1 - shared) % p
```

Test Cases

1. Nonce reuse:

```
def nonce_reuse_attack_demo():
    """
    Same key sk, same nonce k, two messages m1 != m2.
    Then:
        s1 - s2 = (c1 - c2) sk mod q -> sk recovered.
    """
    print("\n========== [Attack] Nonce Reuse Demo ==========")

    m1 = input("Enter first message (m1): ").encode()
    m2 = input("Enter second message (m2): ").encode()

    sk = secrets.randbelow(q)
    print(f"[+] Hidden secret key sk = {sk}")

    reused_k = secrets.randbelow(q)
    R1, s1, c1, _ = single_schnorr_sign(sk, m1, k=reused_k)
    R2, s2, c2, _ = single_schnorr_sign(sk, m2, k=reused_k)

    print(f"c1 = {c1}")
    print(f"c2 = {c2}")
    print(f"s1 = {s1}")
    print(f"s2 = {s2}")

    num = (s1 - s2) % q
    den = (c1 - c2) % q
    recovered_sk = (num * modinv(den, q)) % q

    print(f"[ATTACK] Recovered sk = {recovered_sk}")
    print("[RESULT] Attack successful?", recovered_sk == sk)
    print("===============================================\n")
```

2. Fault Injection:

```python
n, t = 5, 3
message = b"Threshold signing under fault attack"

x = secrets.randbelow(q)
PK = x * G

shares = shamir_generate_shares(x, t, n, q)
parties = [Party(i, shares[i - 1], PK, shares[i - 1] * G)
           for i in range(1, n + 1)]

J = [1, 2, 3]
print(f"[+] Signing quorum J = {J}")

nonces: Dict[int, Tuple[int, object]] = {
    i: parties[i - 1].generate_nonce() for i in J
}

R = None
for i in J:
    _, R_i = nonces[i]
    R = R_i if R is None else R + R_i

c = H_int(point_to_bytes(R), point_to_bytes(PK), message)

partials: Dict[int, int] = {}
for i in J:
    k_i, R_i = nonces[i]
    lam_i = lagrange_coefficient(i, J, q)
    s_i = parties[i - 1].partial_sign(k_i, c, lam_i)
    partials[i] = s_i
```

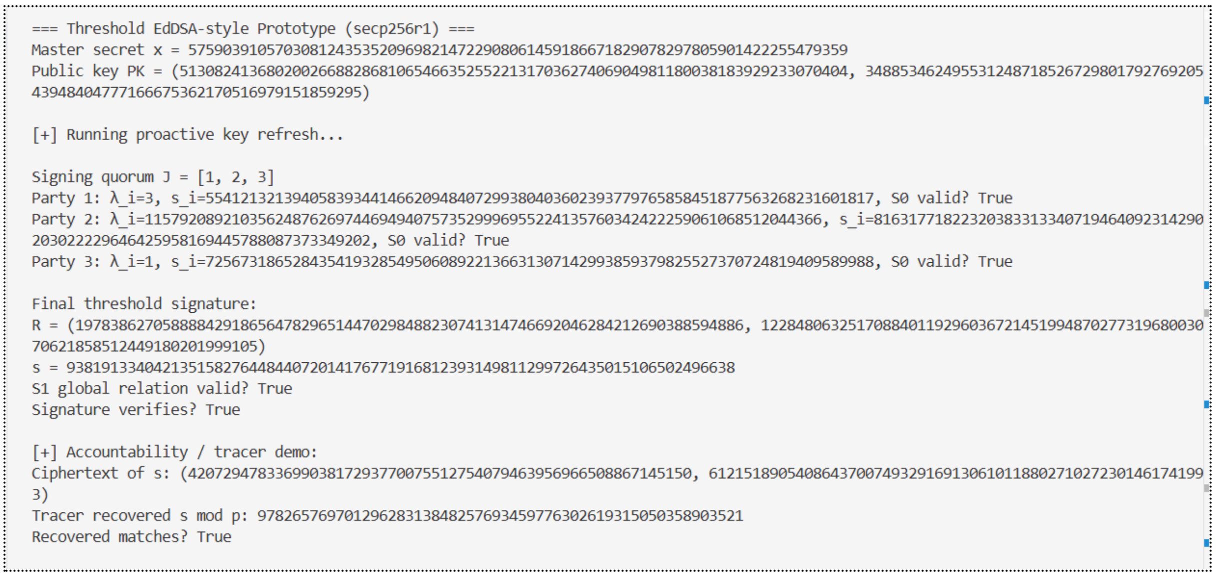
```python
s_valid = sum(partials.values()) % q
valid_ok = verify_threshold_signature(R, s_valid, PK, message)
print(f"[+] Baseline threshold verification OK? {valid_ok}")

victim = random.choice(J)
print(f"[ATTACK] Injecting fault into s_{victim}")
partials_faulty = dict(partials)
partials_faulty[victim] ^= 1  # flip LSB

print("[*] S0 checks after fault:")
for i in J:
    k_i, R_i = nonces[i]
    lam_i = lagrange_coefficient(i, J, q)
    s_i_faulty = partials_faulty[i]
    ok = proof_S0_validity(parties[i - 1], k_i, R_i, c, lam_i, s_i_faulty)
    print(f"  Party {i}: S0 valid? {ok}")

s_faulty = sum(partials_faulty.values()) % q
faulty_ok = verify_threshold_signature(R, s_faulty, PK, message)
print(f"[RESULT] Final signature verifies after fault? {faulty_ok}")
print("-> Faults break correctness and can be detected.")
print("=========================================================\n")
```

3. Partial Signature Tampering

```python
n, t = 5, 3
message = b"Threshold signing with malicious tampering"

x = secrets.randbelow(q)
PK = x * G

shares = shamir_generate_shares(x, t, n, q)
parties = [Party(i, shares[i - 1], PK, shares[i - 1] * G)
           for i in range(1, n + 1)]

J = [1, 2, 3]
print(f"[+] Signing quorum J = {J}")

nonces: Dict[int, Tuple[int, object]] = {
    i: parties[i - 1].generate_nonce() for i in J
}

R = None
for i in J:
    _, R_i = nonces[i]
    R = R_i if R is None else R + R_i

c = H_int(point_to_bytes(R), point_to_bytes(PK), message)

partials: Dict[int, int] = {}
for i in J:
    k_i, R_i = nonces[i]
    lam_i = lagrange_coefficient(i, J, q)
    s_i = parties[i - 1].partial_sign(k_i, c, lam_i)
    partials[i] = s_i
```

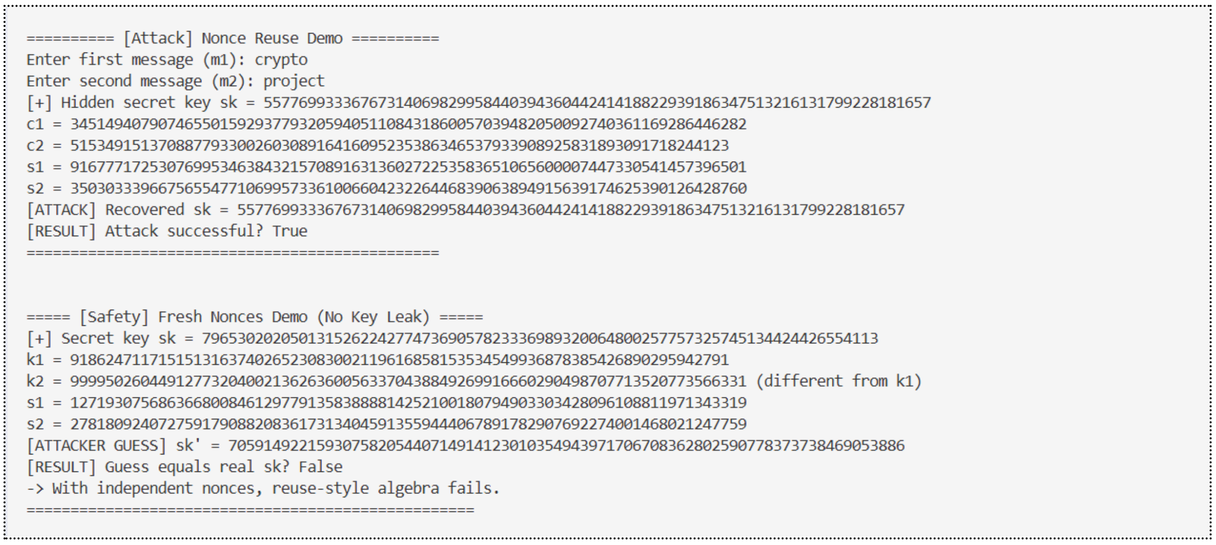
```python
s_valid = sum(partials.values()) % q
print(f"[+] Baseline signature valid? {verify_threshold_signature(R, s_valid, PK, message)}")

# Adversary tampers with one partial signature
attacker_target = random.choice(J)
print(f"[ATTACK] Maliciously tampering with s_{attacker_target}")
partials_tampered = dict(partials)
# instead of 1-bit flip, add a random non-zero offset
offset = secrets.randbelow(q - 1) + 1
partials_tampered[attacker_target] = (partials_tampered[attacker_target] + offset) % q

print("[*] S0 checks after tampering:")
for i in J:
    k_i, R_i = nonces[i]
    lam_i = lagrange_coefficient(i, J, q)
    s_i_tamp = partials_tampered[i]
    ok = proof_S0_validity(parties[i - 1], k_i, R_i, c, lam_i, s_i_tamp)
    print(f"  Party {i}: S0 valid? {ok}")

s_tampered = sum(partials_tampered.values()) % q
tampered_ok = verify_threshold_signature(R, s_tampered, PK, message)
print(f"[RESULT] Final signature verifies after tampering? {tampered_ok}")
print("-> Malicious partial modification is caught by S0 / S1 consistency.")
print("========================================================\n")
```

Final Output



```
=== Threshold EdDSA-style Prototype (secp256r1) ===
Master secret x = 5759039105703081243535209698214722908061459186671829078297805901422255479359
Public key PK = (51308241368020026688286810654663525522131703627406904981180038183929233070404, 348853462495531248718526729801792769205
43948404777166675362170516979151859295)

[+] Running proactive key refresh...

Signing quorum J = [1, 2, 3]
Party 1: λ_i=3, s_i=55412132139405839344146620948407299380403602393779765858451877563268231601817, S0 valid? True
Party 2: λ_i=115792089210356248762697446949407573529996955224135760342422259061068512044366, s_i=81631771822320383313340719464092314290
203022229646425958169445788087373349202, S0 valid? True
Party 3: λ_i=1, s_i=72567318652843541932854950608922136631307142993859379825527370724819409589988, S0 valid? True

Final threshold signature:
R = (19783862705888842918656478296514470298488230741314746692046284212690388594886, 122848063251708840119296036721451994870277319680030
70621858512449180201999105)
s = 93819133404213515827644844072014176771916812393149811299726435015106502496638
S1 global relation valid? True
Signature verifies? True

[+] Accountability / tracer demo:
Ciphertext of s: (4207294783369903817293770075512754079463956966508867145150, 612151890540864370074932916913061011880271027230146174199
3)
Tracer recovered s mod p: 97826576970129628313848257693459776302619315050358903521
Recovered matches? True
```

Nonce reuse attack:



```
========== [Attack] Nonce Reuse Demo ==========
Enter first message (m1): crypto
Enter second message (m2): project
[+] Hidden secret key sk = 5577699333676731406982995844039436044241418822939186347513216131799228181657
c1 = 34514940790746550159293779320594051108431860057039482050092740361169286446282
c2 = 515349151370887793300260308916416095235386346537933908925831893091718244123
s1 = 91677717253076995346384321570891631360272253583651065600007447330541457396501
s2 = 35030333966756554771069957336100660423226446839063894915639174625390126428760
[ATTACK] Recovered sk = 5577699333676731406982995844039436044241418822939186347513216131799228181657
[RESULT] Attack successful? True
===============================================


===== [Safety] Fresh Nonces Demo (No Key Leak) =====
[+] Secret key sk = 79653020205013152622427747369057823336989320064800257757325745134424426554113
k1 = 918624711715151316374026523083002119616858153534549936878385426890295942791
k2 = 99995026044912773204002136263600563370438849269916660290498707713520773566331 (different from k1)
s1 = 12719307568636680084612977913583888814252100180794903303428096108811971343319
s2 = 27818092407275917908820836173134045913559444067891782907692274001468021247759
[ATTACKER GUESS] sk' = 70591492215930758205440714914123010354943971706708362802590778373738469053886
[RESULT] Guess equals real sk? False
-> With independent nonces, reuse-style algebra fails.
====================================================
```

Fault injection attack:

```
========== [Attack] Fault Injection on Threshold ==========
[+] Signing quorum J = [1, 2, 3]
[+] Baseline threshold verification OK? True
[ATTACK] Injecting fault into s_3
[*] S0 checks after fault:
  Party 1: S0 valid? True
  Party 2: S0 valid? True
  Party 3: S0 valid? False
[RESULT] Final signature verifies after fault? False
-> Faults break correctness and can be detected.
==========================================================
```

Partial signature tampering:

```
====== [Attack] Partial Signature Tampering Demo ======
[+] Signing quorum J = [1, 2, 3]
[+] Baseline signature valid? True
[ATTACK] Maliciously tampering with s_2
[*] S0 checks after tampering:
  Party 1: S0 valid? True
  Party 2: S0 valid? False
  Party 3: S0 valid? True
[RESULT] Final signature verifies after tampering? False
-> Malicious partial modification is caught by S0 / S1 consistency.
========================================================
```

8. *Comparison With Existing Schemes*

Feature	BLS Signatures	Schnorr Signatures	EdDSA (Ed25519)	Proposed: Accountable Threshold EdDSA
Mathematical Foundation	Bilinear pairings: $e : G_1 \times G_2 \rightarrow G\_T$	Standard ECDLP: $s = k + e \cdot x$	Twisted Edwards curve: $-x^2 + y^2 = 1 + d \cdot x^2 y^2$	EdDSA on Curve25519, ElGamal encryption, Zero-Knowledge proofs
Signature Size	48 bytes or 96 bytes	64 bytes	64 bytes	64 bytes and encryption overhead
Key Size	48–96 bytes	32–33 bytes	32 bytes	32 bytes per participant
Verification Speed	Slow (≈10× scalar-mult cost due to pairing)	Fast (two scalar multiplications)	Very fast (deterministic arithmetic)	Slower (adds ZK-proof verification)
Accountability	None	None	None	Yes racer can identify malicious or invalid signers
Key Management	Basic	Basic	Basic	Proactive key refresh with ZK proofs
Privacy Features	None	None	None	Hides threshold value and signer identities
RNG Dependency	Low	High nonce reuse is fatal	None (deterministic nonces)	Minimal (derived deterministically from key)
Side-Channel Attack Resistance	Moderate, pairing operations may leak	Poor, secret-dependent timing	Excellent, fully constant-time	Strong, Constant time ElGamal and ZK proof implementation required
Fault Injection Resilience	None	None	None	Fault-tolerant, redundant verification and identifiable aborts detect faulty signers

9. Alignment With NCS Cybersecurity Controls

The **Accountable Threshold EdDSA** protocol aligns with multiple National Cybersecurity Authority (NCA) requirements [7].

1. **Approved Cryptographic Algorithms (Section 2.2 – Asymmetric Algorithms)**
    - Uses NCA-approved elliptic-curve cryptography, including Curve25519.
    - EdDSA is part of the ECC family permitted by NCA.

2. **ECC Key Size and Security Levels (Section 2.2)**
    - The protocol uses 256-bit ECC keys, matching NCA's recommended security levels.
    - The hardness assumptions (ECDLP, CDH, DDH) align with NCA-approved algorithms.

3. **Public Key Signature Requirements (Section 3.8)**
    - NCA recognizes ECC-based signature schemes such as ECDSA.
    - EdDSA and Threshold EdDSA belong to the same signature family and follow the same verification model.

4. **PKI Architecture Requirements (Section 5)**
    - The protocol produces standard EdDSA-format signatures fully compatible with PKI.
    - Key distribution and verification align with certificate constraints defined in NCA PKI guidelines.

5. **Key Management Controls (Section 6)**
    - Threshold key sharing improves private-key protection.
    - Proactive key refresh aligns with NCA requirements for continuous key-lifecycle security.
    - Accountability supports NCA's governance and auditing controls.

6. **National-Scale Security and Performance Requirements (Section 1)**
    - The protocol delivers short signatures, fast verification, and scalable multi-party authentication.
    - Suitable for digital identity, government services, and high-assurance systems emphasized by NCA.

10. Conclusion

This project analyzed the Accountable Threshold EdDSA protocol as a modern extension of EdDSA for distributed and privacy-preserving digital signatures. By detailing its mathematical underpinnings, protocol phases, accountability features, and security guarantees, we showed how the scheme enables multi-party signing while preserving signature compactness and Ed25519 compatibility.

The security evaluation demonstrates that the protocol maintains strong protection against classical attacks, supports proactive key refresh, and provides accountability without exposing the signing quorum. The prototype implementation, while simplified, confirms the core mechanics of threshold signing and encrypted tracing.

Overall, Accountable Threshold EdDSA offers a practical and secure foundation for digital identity systems, distributed authorization, and applications that require both privacy and auditable signer behavior.

References

[1] D. J. Bernstein and T. Lange, "Safe curves for elliptic-curve cryptography," *Information Security in a Connected World: Celebrating the Life and Work of Ed Dawson*, pp. 124–191, 2025.

[2] D. Boneh, B. Lynn and H. Shacham, "Short Signatures with Aggregation," *IEEE Transactions on Information Theory*, 2024.

[3] M. Vidaković and K. Miličević, "Performance and Applicability of Post-Quantum Digital Signature Algorithms in Resource-Constrained Environments," Algorithms, vol. 16, no. 518, 2023.

[4] Y. Peng, "A Survey on Threshold Digital Signature Schemes," *Frontiers of Computer Science*, 2026.

[5] Y. Xie et al., "Accountable and Secure Threshold EdDSA Signature and Its Applications," *IEEE Trans. Inf. Forensics & Security*, vol. 19, pp. 7033–7046, 2024.

[6] S. Belaïd and M. Rivain, "High Order Side-Channel Security for Elliptic-Curve Implementations," *IACR Trans. on Cryptographic Hardware and Embedded Systems*, vol. 2023, no. 1, pp. 238–276, 2022.

[7] National Cybersecurity Authority (NCA), "National Cryptography Standard (NCS-1:2020)," 2020.

[8] National Institute of Standards and Technology, Recommendation for Key Management, Part 1: General (Rev. 5), NIST Special Publication 800-57 Part 1 Rev. 5, May 2020.

[9] V. Shoup, “A Computational Introduction to Number Theory and Algebra,” 3rd Ed., 2021

[10] J. Josefsson and S. Josefsson, “Edwards-Curve Digital Signature Algorithm (EdDSA),” RFC 8032, IETF, Apr. 2023.

[11] C. Komlo and I. Goldberg, “FROST: Flexible Round-Optimized Schnorr Threshold Signatures,” IETF CFRG Internet-Draft draft-irtf-cfrg-frost-16, Feb. 2025.

[12] H. Zhang, Y. Liu, and Z. Chen, “Accountable and Secure Threshold EdDSA Signature and Its Applications,” Mathematics, vol. 12, no. 4, Art. 460, MDPI, Feb. 2024.

[13] A. Kiltz, J. Loss, and V. Lyubashevsky, “Fiat–Shamir for Proofs of Knowledge—Security Analysis,” Journal of Cryptology, vol. 31, no. 3, pp. 776–818, 2018

[14] H. Ward, J. Watson, E. Cooper, and H. Castro, “Hybrid Cryptographic Protocols Integrating Classical and Post-Quantum Techniques for Secure Cloud Communication,” 2025.

[15] M.-J. O. Saarinen, *“The Quantum Threat to RSA and Elliptic Curve Cryptography,”* Technical Report, Tampere University, Finland, Dec. 2024.