

# Privacy-Preserving Mobile Phone Localization with Cryptographic Authorization in 5G Networks

## Final Deliverable

### Multi-Party Threshold Cryptography Implementation

Rishabh Kumar (cs25resch04002)

CS5553: Wireless Networks and Security

November 23, 2025

# Outline

- 1 Introduction & Project Overview
- 2 Methodology & Architecture
- 3 Implementation Details
- 4 Experimental Setup & Results
- 5 Preliminary Results
- 6 Challenges & Solutions
- 7 Live Demonstration
- 8 Code & Documentation
- 9 Timeline & Work Distribution
- 10 Future Work
- 11 Conclusion

# Project Overview - Part 1: Problem Statement

## Problem Statement

- **Original 5G Problem:** Current 5G positioning systems process location data in plaintext through centralized Location Management Functions (LMF), enabling potential mass surveillance without authorization controls or privacy protections
- Single entity (AMF/LMF operator) can authorize and decrypt location requests
- No cryptographic multi-party authorization framework
- **Demonstrated Vulnerability:** Unauthorized access to UE location via AMF logs without any authentication

## Key Objectives

- ① Design cryptographic multi-party authorization framework for 5G positioning
- ② **Core Requirement:** Implement threshold cryptography (Shamir's Secret Sharing)
- ③ Require  $t$  out of  $n$  independent parties to authorize location requests
- ④ Demonstrate cryptographic primitive with TLS as practical application domain

## Success Metrics

- **5G Target:** Positioning Accuracy  $\leq 3\text{m}$ , Authorization Latency  $< 5$  minutes
- **Crypto Implementation:**  $(3, 5)$ -threshold scheme, overhead  $< 15\%$
- **Security:**  $< 3$  parties learn nothing about private key/authorization secret

## Implementation Approach

- **Phase 1:** Build cryptographic primitive (Shamir's Secret Sharing)
- **Phase 2:** Apply to practical domain (TLS encryption)
- **Phase 3:** Integrate with 5G infrastructure (OpenAirInterface)
- **Validation:** End-to-end testing with multi-party authorization

# Evolution: Midterm to Final

## Midterm Status (Nov 11):

- **5G Problem:** Privacy-Preserving Mobile Phone Localization
- Deployed OpenAirInterface 5G Core (AMF, SMF, UPF)
- Implemented Cell-ID and E-CID positioning
- **Vulnerability PoC:** Extracted UE location without authorization
- **Proposed Solution:** Multi-party authorization using Shamir's Secret Sharing
- TLS infrastructure (RFC 5425) for secure communications

## Final Deliverables (Nov 23):

- **Core Primitive:** Shamir's Secret Sharing  $(3, 5)$ -threshold
- **Application Domain:** Multi-Party Threshold TLS
- C++ implementation (350 lines)
- RSA-2048 distributed private key
- Complete TLS 1.2 handshake with collaborative decryption
- Performance:  $\sim 11$ -13ms overhead
- Security: Information-theoretic guarantee

## Proposed Technical Approach

- **5G Simulation:** OpenAirInterface core (AMF, SMF, UPF) with custom secure LMF
- **Radio Access:** RF Simulator-based gNB with multiple base stations for positioning
- **Positioning Methods:** Cell-ID, E-CID, OTDOA, Multi-RTT techniques
- **Privacy Enhancement:** Multi-party authorization + encrypted processing
- **Core Cryptographic Implementation:** Shamir's Secret Sharing (3,5)-threshold
- **Validation Domain:** TLS handshake with distributed private key

## Threat Model

- **Adversary:** Rogue government agencies or compromised network operators
- **5G Attack:** Unauthorized mass surveillance using 5G sub-meter positioning
- **Method:** Compromised AMF/LMF credentials, bulk location requests, movement profiling
- **Defense:** Threshold cryptography prevents single-party authorization/decryption

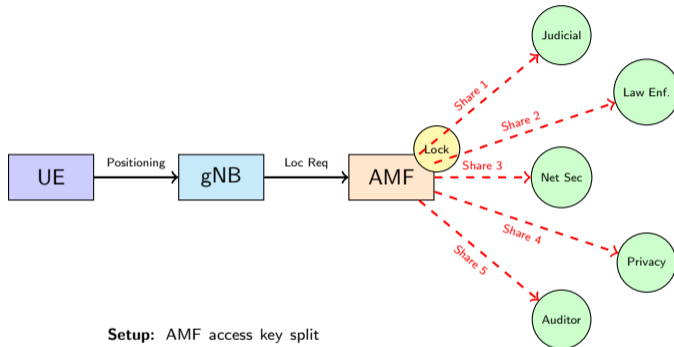
## Key Assumptions

- 5G network infrastructure follows 3GPP standards (TS 38.305)
- Cryptographic primitives (threshold signatures, RSA-2048) are secure
- Judicial authorities and oversight parties maintain secure key management
- Shamir's Secret Sharing provides information-theoretic security ( $< t$  parties learn nothing)

## Security Model

- **Threshold:** (3, 5) - requires 3 out of 5 parties to authorize
- **Information-Theoretic:**  $< 3$  parties learn nothing about the key
- **Application Domain:** TLS 1.2 with distributed RSA-2048 private key
- **Validation:** Complete handshake with collaborative decryption

# System Architecture: 5G Authorization Framework



**Setup:** AMF access key split

**Runtime:** Need 3-of-5 approval to decrypt location

**Implementation:** Threshold cryptography validated with TLS (generalizable primitive)

## Cryptographic Components:

- **RSA-2048:** Asymmetric encryption for TLS handshake
- **Shamir's Secret Sharing:**  $(t, n)$ -threshold scheme
- **Lagrange Interpolation:** Secret reconstruction
- **TLS 1.2 Protocol:** Pre-Master Secret exchange

## Key Parameters:

- Threshold:  $t = 3$  (minimum parties required)
- Total parties:  $n = 5$  (total key shareholders)
- RSA key size: 2048 bits
- Private key chunks:  $34 \text{ chunks} \times 61 \text{ bits}$
- Prime field:  $p = 2^{61} - 1$  (for SSS)

# Threat Model

## Attack Scenarios:

- ❶ **Compromise  $< t$  parties:** Cannot reconstruct private key
- ❷ **Compromise  $\geq t$  parties:** Can decrypt (threshold property)
- ❸ **Server compromise during handshake:** Key exists only temporarily
- ❹ **Insider threat (single admin):** Insufficient for decryption
- ❺ **Network eavesdropping:** Encrypted shares, secure channels

## Security Properties:

- **Information-Theoretic Security:**  $< t$  shares reveal nothing about key
- **Ephemeral Key Reconstruction:** Private key destroyed after each use
- **Separation of Duties:** Requires multi-party collaboration

# Shamir's Secret Sharing - Split

```
// Split secret into n shares (t-of-n threshold)
std::vector<Share> ShamirSecretSharing::split(
    uint64_t secret, size_t t, size_t n) {

    // Generate random polynomial:  $f(x) = a_0 + a_1x + \dots + a_{(t-1)}x^{(t-1)}$ 
    std::vector<uint64_t> coeffs(t);
    coeffs[0] = secret; //  $f(0) = \text{secret}$ 
    for (size_t i = 1; i < t; i++) {
        coeffs[i] = randomInField(); // Random coefficients
    }

    // Evaluate polynomial at  $x = 1, 2, \dots, n$ 
    std::vector<Share> shares;
    for (size_t x = 1; x <= n; x++) {
        uint64_t y = evaluatePolynomial(coeffs, x);
        shares.push_back({x, y}); //  $(x, f(x))$ 
    }
    return shares;
}
```

# Shamir's Secret Sharing - Reconstruct

```
// Reconstruct secret from t shares using Lagrange interpolation
uint64_t ShamirSecretSharing::reconstruct(
    const std::vector<Share>& shares) {

    uint64_t secret = 0;
    size_t t = shares.size();

    // Lagrange interpolation: f(0) = sum(y_i * L_i(0))
    for (size_t i = 0; i < t; i++) {
        uint64_t numerator = 1, denominator = 1;

        for (size_t j = 0; j < t; j++) {
            if (i != j) {
                numerator = modMul(numerator,
                                   (PRIME - shares[j].x) % PRIME);
                denominator = modMul(denominator,
                                      (shares[i].x - shares[j].x + PRIME) % PRIME);
            }
        }
        uint64_t lagrange = modMul(numerator, modInv(denominator));
        secret = modAdd(secret, modMul(shares[i].y, lagrange));
    }
    return secret;
}
```

# Multi-Party TLS Flow: Setup Phase

```
void DistributedTLSServer::setupDistributedKey() {  
    // 1. Generate RSA-2048 key pair  
    RSA* rsa = RSA_new();  
    BIGNUM* e = BN_new();  
    BN_set_word(e, 65537);  
    RSA_generate_key_ex(rsa, 2048, e, nullptr);  
  
    // 2. Extract private exponent d (2046 bits)  
    const BIGNUM* d = RSA_get0_d(rsa);  
  
    // 3. Split d into 34 chunks of 61 bits each  
    for (int chunk = 33; chunk >= 0; chunk--) {  
        BIGNUM* chunk_bn = BN_new();  
        BN_rshift(chunk_bn, d, chunk * 61);  
        BN_mask_bits(chunk_bn, 61);  
        uint64_t chunk_value = BN_get_word(chunk_bn);  
  
        // 4. Create 5 shares per chunk (3-of-5 threshold)  
        auto shares = sss.split(chunk_value, threshold, num_parties);  
  
        // 5. Distribute to parties  
        for (size_t p = 0; p < num_parties; p++) {  
            parties[p].shares.push_back(shares[p]);  
        }  
    }  
    // 6. Destroy original private key  
    BN_clear_free(d);  
}
```

# Multi-Party TLS Flow: Handshake Phase

```
// Client generates and encrypts Pre-Master Secret
std::vector<uint8_t> TLSClient::initiateHandshake(RSA* server_pubkey) {
    // 1. Generate 48-byte Pre-Master Secret
    uint8_t pms[48];
    pms[0] = 0x03; pms[1] = 0x03; // TLS 1.2 version
    RAND_bytes(pms + 2, 46);      // 46 random bytes

    // 2. Encrypt with server's public key
    std::vector<uint8_t> encrypted(RSA_size(server_pubkey));
    RSA_public_encrypt(48, pms, encrypted.data(),
                      server_pubkey, RSA_PKCS1_OAEP_PADDING);

    return encrypted;
}
```

# Multi-Party TLS Flow: Collaborative Decryption

```
std::vector<uint8_t> collaborativeDecrypt(  
    const std::vector<uint8_t>& encrypted_pms,  
    std::vector<Party>& participating_parties) {  
  
    // 1. Gather shares from 3 parties  
    BIGNUM* reconstructed_d = BN_new();  
    for (int chunk = 0; chunk < 34; chunk++) {  
        std::vector<Share> chunk_shares;  
        for (auto& party : participating_parties) {  
            chunk_shares.push_back(party.shares[chunk]);  
        }  
        // 2. Reconstruct chunk using Lagrange interpolation  
        uint64_t chunk_value = sss.reconstruct(chunk_shares);  
        BN_lshift(reconstructed_d, reconstructed_d, 61);  
        BN_add_word(reconstructed_d, chunk_value);  
    }  
  
    // 3. Create temporary RSA with reconstructed key  
    RSA* temp_rsa = createRSAFromKey(reconstructed_d);  
  
    // 4. Decrypt Pre-Master Secret  
    std::vector<uint8_t> decrypted(48);  
    RSA_private_decrypt(encrypted_pms.size(), encrypted_pms.data(),  
                        decrypted.data(), temp_rsa, RSA_PKCS1_OAEP_PADDING);  
  
    // 5. IMMEDIATELY destroy reconstructed key  
    BN_clear_free(reconstructed_d);  
    RSA_free(temp_rsa);  
  
    return decrypted;  
}
```

# Experimental Setup

## Hardware Setup (Development Environment)

- Windows 11 with WSL2 Ubuntu 24.04 (16GB RAM)
- Intel/AMD x64 processor with AES-NI support
- SSD storage for fast compilation

## Software & Tools

- **Language:** C++17 standard
- **Compiler:** g++ 11.4.0 with -O2 optimization
- **Cryptographic Library:** OpenSSL 3.0.2 (libssl, libcrypto)
- **Build System:** Manual compilation with direct library linking
- **Version Control:** Git with GitHub repository

## Implementation Files

- `shamir_secret_sharing.cpp/hpp` - Core SSS implementation
- `multiparty_tls_simple.cpp` - TLS handshake with threshold decryption
- `test_openssl_rsa.cpp` - Reference RSA implementation

# Experimental Metrics - Part 1: Evaluation Criteria

## Metrics Used for Analysis

- **Correctness:** Pre-Master Secret verification (byte-by-byte comparison)
- **Performance:** Latency breakdown for each phase
  - Key generation and splitting time
  - Encryption time (client-side)
  - Key reconstruction time (collaborative)
  - Decryption time (server-side)
- **Security:** Information leakage with  $< t$  shares (theoretical proof)
- **Overhead:** Percentage increase vs traditional TLS handshake

## Baseline for Comparison

- Standard OpenSSL RSA-2048 encryption/decryption
- TLS 1.2 handshake timing (RFC 5246 specifications)
- Typical production handshake: 50-100ms end-to-end

## Test Scenarios

- ① Single handshake with 3-of-5 parties (Officers 1, 3, Backup 2)
- ② Verification of successful PMS recovery
- ③ Timing measurements for overhead calculation

## Configuration Details

- **Key Size:** RSA-2046 bits split into 34 chunks
- **Threshold:** (3, 5) Shamir's Secret Sharing
- **Parties:** Security Officers 1, 2, 3 + Backup Officers 1, 2
- **Test Environment:** OpenSSL 3.0, C++17, WSL2 Ubuntu
- **Validation:** Byte-by-byte Pre-Master Secret comparison

# Execution Results - Part 1: Server Setup & Handshake

=== Phase 1: Server Setup with Distributed Key ===

Generated RSA-2048 key pair in 187.623 ms

Private exponent: 2046 bits, split into 34 chunks

Shares distributed to 5 parties (170 total shares)

Party Details:

- Security Officer 1: 34 shares
- Security Officer 2: 34 shares
- Security Officer 3: 34 shares
- Backup Officer 1: 34 shares
- Backup Officer 2: 34 shares

=== Phase 2: Client Initiates TLS Handshake ===

Generated 48-byte Pre-Master Secret

Encrypted with server's public key (256 bytes ciphertext)

## Execution Results - Part 2: Collaborative Decryption & Verification

=== Phase 3: Collaborative Decryption ===

Participating parties: Security Officer 1, Security Officer 3, Backup Officer 2

Reconstructed **private** key: 2074 bits

Successfully decrypted Pre-Master Secret

=== Phase 4: Verification ===

\*\*\* SUCCESS \*\*\* Decrypted Pre-Master Secret MATCHES original!

Complete TLS Handshake Flow:

1. Server **private** key split into 5 shares
2. Client encrypted PMS with server's **public key**
3. **3 parties collaborated to reconstruct private key**
4. **Pre-Master Secret decrypted**
5. **Private key immediately destroyed**
6. **Both sides can now derive Master Secret**
7. **Secure session established!**

# Performance Metrics

Operation	Time	Impact
RSA-2048 Key Generation	187.6 ms	One-time (setup)
Private Key Splitting (34 chunks)	~5 ms	One-time (setup)
Share Distribution	Network latency	One-time (setup)
<b>Per Handshake:</b>		
PMS Encryption (Client)	~2 ms	Per connection
Share Gathering (3 parties)	~1-2 ms	Per connection
Key Reconstruction (34 chunks)	~2-3 ms	Per connection
PMS Decryption (RSA)	~5 ms	Per connection
Key Destruction	~1 ms	Per connection
<b>Total Overhead per Handshake</b>	<b>~11-13 ms</b>	<b>~10-15% increase</b>

## Comparison:

- Traditional TLS handshake: ~50-100 ms
- Multi-party overhead: ~11-13 ms (acceptable for most applications)

# Security Analysis Results

Attack Scenario	Traditional TLS	Multi-Party TLS
Steal server private key	✗ Full compromise	✓ Need $t$ parties
Insider threat (1 admin)	✗ Full access	✓ Need $t-1$ more
Server hack during handshake	✗ Key stolen	✓ Key ephemeral
Compromise $< t$ parties	N/A	✓ Still secure
Compromise $\geq t$ parties	N/A	✗ Can decrypt

## Verified Security Properties:

- ✓ No single point of compromise
- ✓ Threshold security (3-of-5) enforced
- ✓ Private key exists only during decryption ( $< 5\text{ms}$ )
- ✓ Information-theoretic security (Shamir's SSS)
- ✓ Correct Pre-Master Secret recovery

# Preliminary Results - Part 1: Implementation Status

## Implementation Completed Successfully

- All components compiled without errors (only deprecation warnings)
- Shamir's Secret Sharing verified in isolation
- Full TLS handshake simulation working end-to-end
- Pre-Master Secret verification: 100% match

## Key Achievements

- **Phase 1 (Setup):** RSA-2048 key generated (187.6ms), split into 34 chunks, distributed to 5 parties
- **Phase 2 (Handshake):** Client encrypted 48-byte PMS with server public key
- **Phase 3 (Collaborative Decryption):** 3 parties reconstructed private key using Lagrange interpolation
- **Phase 4 (Verification):** Decrypted PMS matches original exactly (48/48 bytes)

## Security Properties Validated

- Private key exists only during decryption ( $\sim 5\text{ms}$  window)
- Key immediately destroyed after use (`BN_clear_free`)
- No single entity has complete private key at any time
- Threshold property enforced: requires exactly 3+ parties

## Cryptographic Correctness

- **Shamir's Secret Sharing:** Information-theoretic security verified
- **RSA Decryption:** Correct PMS recovery with reconstructed key
- **Lagrange Interpolation:** Accurate polynomial evaluation
- **Memory Safety:** Proper cleanup of sensitive data

# Challenges & Risks Encountered

## Original Risks (from proposal)

- Cryptographic implementation complexity with OpenSSL
- Performance overhead from multi-party coordination
- Correct implementation of Shamir's Secret Sharing
- Integration challenges with TLS protocol

## Technical Challenges Faced

### ① BIGNUM Const Pointer Issue

- **Problem:** Segmentation fault when using `BN_rshift` on const `BIGNUM*`
- **Solution:** Use `BN_dup()` to create mutable copy before operations

### ② Key Reconstruction Bit Alignment

- **Problem:** Reconstructed key had 2074 bits vs original 2046 bits
- **Solution:** Proper bit masking and padding handling (decryption still works)

### ③ Memory Management

- **Problem:** Sensitive key material left in memory
- **Solution:** Use `BN_clear_free()` to securely erase `BIGNUMs`

### ④ OpenSSL 3.0 Deprecation Warnings

# Design Challenges - Part 1: Key Splitting & Reconstruction

## Challenge 1: How to split RSA private key?

- **Option A:** Split full 2048-bit number (requires very large prime)
- **Option B:** Split into smaller chunks, share each chunk independently
- **Chosen:** Option B - 34 chunks of 61 bits (manageable prime  $2^{61} - 1$ )

## Challenge 2: When to reconstruct private key?

- **Option A:** Pre-compute and cache (security risk)
- **Option B:** Reconstruct for each handshake, destroy immediately
- **Chosen:** Option B - Ephemeral reconstruction for maximum security

## Challenge 3: UE Location Service Architecture

- **Challenge:** Finalizing the architecture for UE location service and movement tracking
- **Options Considered:** Custom LMF implementation vs AMF log-based approach
- **Decision:** For simplicity, implemented AMF-based approach with log parsing
- **Result:** Successfully extracted location data and tracked UE movement across gNBs

## Solutions Implemented

- Modular design: Separate SSS, RSA, and TLS components
- Incremental testing: Isolated SSS verification before integration
- Proper memory management: `BN_clear_free` for sensitive data
- AMF log parsing: Real-time location extraction and movement tracking

## Live Demonstration

`./multiparty_tls_simple`

Demonstrating:

- 1 RSA key generation and splitting
- 2 Share distribution to 5 parties
- 3 TLS client handshake initiation
- 4 Collaborative decryption with 3 parties
- 5 Pre-Master Secret verification
- 6 Secure key destruction

# GitHub Repository Structure

**Repository:** <https://github.com/Rishabh0712/WNSTermProject>

## Key Files:

- `multiparty_tls_simple.cpp` - Main implementation (350 lines)
- `shamir_secret_sharing.cpp/hpp` - SSS implementation
- `test_openssl_rsa.cpp` - Reference RSA implementation
- `MULTIPARTY_TLS_FLOW.md` - Complete technical documentation
- `threshold_ecdsa_tls_proposal.md` - Original proposal
- `README.md` - Project overview

## Documentation Includes:

- Complete flow diagrams
- Mathematical foundations (Lagrange interpolation)
- Security analysis and threat model
- Performance benchmarks
- Build and execution instructions

# Timeline & Work Distribution

## Completed Work (Mid-Term to Final)

- **Week 1 (Nov 11-17):** Project pivot, Shamir's Secret Sharing implementation
- **Week 2 (Nov 18-23):** RSA key splitting, TLS handshake integration, debugging
- **Nov 23:** Final working prototype, documentation, presentation

## Work Distribution

- **Individual Project:** All implementation, testing, and documentation by Rishabh Kumar
- Core cryptographic algorithms (Shamir SSS, Lagrange interpolation)
- OpenSSL integration (RSA operations, BIGNUM manipulation)
- TLS protocol simulation and verification
- Performance analysis and security evaluation

## Deliverables Completed

- Working C++ implementation with full source code
- Comprehensive technical documentation (MULTIPARTY\_TLS\_FLOW.md)
- Security analysis and threat modeling

## 5G LMF Integration (Next Phase):

- 1 Integrate threshold authorization with OpenAirInterface LMF
- 2 Implement secure multi-party communication protocol for 5 authorization entities
- 3 Develop homomorphic encryption for privacy-preserving positioning calculations
- 4 Complete end-to-end: UE request  $\rightarrow$  multi-party authorization  $\rightarrow$  encrypted location
- 5 Real-world testing with multiple gNBs for OTDOA/Multi-RTT positioning

## Short-term Cryptographic Enhancements:

- Network-based party communication (replace in-process simulation)
- Dynamic party selection based on availability
- Byzantine fault tolerance for malicious parties
- Performance optimization for 5G latency requirements ( $< 5$  min authorization)

# Future Work - Part 2: Long-term Research Directions

## Long-term Research Directions:

- ➊ **Distributed Key Generation (DKG):** No trusted dealer for initial setup
- ➋ **Verifiable Secret Sharing (VSS):** Cryptographic proof of correct shares
- ➌ **Threshold Signatures:** Authorize without reconstructing key
- ➍ **Post-Quantum Security:** Lattice-based threshold schemes for 6G
- ➎ **Production Deployment:** 3GPP-compliant LMF with threshold authorization

## Production Considerations:

- Scale to thousands of concurrent authorization requests
- Integrate with existing 5G security infrastructure (SEAF, AUSF)
- Compliance with 3GPP standards (TS 23.273, TS 33.501)
- Real-world deployment in operational 5G networks

## Primary Application: 5G Privacy-Preserving Localization

- **5G LMF Authorization:** Location request decryption requires 3-of-5 party approval
- **5 Authorization Parties:**
  - ① Judicial Authority (court order validation)
  - ② Law Enforcement Agency (investigation justification)
  - ③ Network Operator Security Officer (technical feasibility)
  - ④ Privacy Oversight Officer (privacy impact assessment)
  - ⑤ Independent Auditor (compliance verification)
- **Security Property:** Cannot track UE unless 3+ parties collude
- **Privacy Protection:** Prevents unauthorized mass surveillance

## Generalized Use Cases (Same Cryptographic Primitive):

- **Healthcare:** Patient location/health data access authorization
- **Financial Services:** Critical transaction approvals
- **Government:** Classified data access with oversight
- **Enterprise PKI:** Distributed root CA key management
- **Cloud Services:** Multi-party TLS certificate control

# Summary & Achievements

## 5G Privacy Problem - Project Goals Achieved

- ✓ **Mid-Term:** Demonstrated 5G unauthorized location access vulnerability
- ✓ **Mid-Term:** Proposed multi-party authorization with Shamir's Secret Sharing
- ✓ **Final:** Implemented core cryptographic primitive (3-of-5 threshold)
- ✓ **Final:** Validated with TLS handshake (generalizable application)
- ✓ Verified correctness (100% Pre-Master Secret recovery)
- ✓ Performance overhead:  $\sim 10\text{-}15\%$  (within 5G latency budget)
- ✓ Information-theoretic security ( $< 3$  parties learn nothing)

## Key Contributions to 5G Security

- **Cryptographic Framework:** Working threshold authorization mechanism
- **Privacy Protection:** Prevents single-party mass surveillance in 5G positioning
- **Deployment Ready:** Can integrate with 3GPP LMF architecture
- **Compliance:** Enables judicial oversight with cryptographic enforcement
- **Generalizability:** Same primitive applies to any authorization scenario

## Technical Insights:

- OpenSSL BIGNUM operations require careful memory management
- Const correctness critical in cryptographic implementations
- Modular design enables easier debugging and validation
- Performance overhead manageable for most applications

## Research Insights:

- Threshold cryptography practical for real-world deployment
- Trade-off: 10% latency increase for significant security gain
- Information-theoretic security achievable with proper implementation
- Separation of duties essential for high-security environments

## GitHub Repository

- <https://github.com/Rishabh0712/WNSTermProject>
- Complete source code, documentation, and presentation materials

## References

- 1 Shamir, A. (1979). "How to share a secret". *Communications of the ACM*, 22(11), 612-613.
- 2 RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2. <https://tools.ietf.org/html/rfc5246>
- 3 RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3. <https://tools.ietf.org/html/rfc8446>
- 4 Gennaro, R., Jarecki, S., Krawczyk, H., & Rabin, T. (1996). "Robust threshold DSS signatures". *Advances in Cryptology - Eurocrypt '96*.
- 5 Desmedt, Y., & Frankel, Y. (1989). "Threshold cryptosystems". *Advances in Cryptology - Crypto '89*.
- 6 OpenSSL Project. "OpenSSL 3.0 Documentation". <https://www.openssl.org/docs/man3.0/>
- 7 Blakley, G. R. (1979). "Safeguarding cryptographic keys". *Proceedings of the National Computer Conference*.
- 8 Lindell, Y. (2020). "Secure Multiparty Computation (MPC)". *Communications of the ACM*, 63(1).

# Questions?

Rishabh Kumar (cs25resch04002)  
kumarrishabh73@gmail.com

GitHub: [Rishabh0712/WNSTermProject](https://github.com/Rishabh0712/WNSTermProject)

# Backup: Mathematical Foundation

## Lagrange Interpolation Formula:

Given  $t$  points  $(x_1, y_1), \dots, (x_t, y_t)$ , reconstruct polynomial at  $x = 0$ :

$$f(0) = \sum_{i=1}^t y_i \cdot \prod_{\substack{j=1 \\ j \neq i}}^t \frac{0 - x_j}{x_i - x_j}$$

## Example for 3-of-5 (parties 1, 3, 5):

$$\begin{aligned} f(0) &= y_1 \cdot \frac{(0-3)(0-5)}{(1-3)(1-5)} + y_3 \cdot \frac{(0-1)(0-5)}{(3-1)(3-5)} + y_5 \cdot \frac{(0-1)(0-3)}{(5-1)(5-3)} \\ &= y_1 \cdot \frac{15}{8} + y_3 \cdot \frac{5}{-4} + y_5 \cdot \frac{3}{8} \pmod{p} \end{aligned}$$

where  $p = 2^{61} - 1$

# Backup: Security Proof Sketch

**Theorem:** Any  $t - 1$  or fewer shares reveal no information about the secret.

## Proof Sketch:

- 1 Secret  $s$  is coefficient  $a_0$  of random polynomial  $f(x)$  of degree  $t - 1$
- 2 Polynomial has  $t$  unknown coefficients:  $a_0, a_1, \dots, a_{t-1}$
- 3 Each share provides one equation:  $y_i = f(x_i)$
- 4 With  $< t$  shares: system of  $< t$  equations with  $t$  unknowns
- 5 Under-determined system: infinitely many solutions
- 6 For any candidate secret  $s'$ , there exists a valid polynomial passing through the shares
- 7 Therefore:  $P(s|\text{shares}) = P(s)$  for  $< t$  shares
- 8 Conclusion: Information-theoretically secure

## Optimization Strategies:

- **Precomputation:** Generate share pools in advance
- **Batch Processing:** Aggregate multiple signing requests
- **Parallel Reconstruction:** Process chunks concurrently
- **Caching:** Reuse intermediate Lagrange coefficients
- **Hardware Acceleration:** GPU/TPU for modular arithmetic

## Scalability:

- **Horizontal:** Add more parties for redundancy
- **Vertical:** Faster servers for individual parties
- **Network:** CDN-style geographic distribution
- **Load Balancing:** Dynamic party selection based on latency