Signature Verification Process at Receiver Side

# Overview

When a client receives a payload from the server, it must verify the authenticity of the received nonce and the integrity of the data.  
This ensures that the nonce came from the correct sender and has not been tampered with.

# Verification Steps

📦 Step 1️⃣ — Receiving the Payload:

Payload structure:  
{  
 "id": "A",  
 "payloads": {  
 "B": "base64(encrypted(rA\_for\_B))",  
 "C": "base64(encrypted(rA\_for\_C))"  
 },  
 "sig": "base64(signature over (rA + A+B+C))"  
}  
Important fields:  
- payloads: Contains encrypted nonces for peers.  
- sig: Digital signature from sender.

📥 Step 2️⃣ — Decrypt Encrypted Nonce:

Decrypt the nonce intended for the receiver using their private key:  
- enc\_nonce = base64-decoded encrypted nonce.  
- peer\_nonce = decrypted nonce (actual rA).  
Python Code:  
enc\_nonce = base64.b64decode(peer\_data["payloads"][CLIENT\_ID])  
peer\_nonce = decrypt\_nonce\_from\_peer(enc\_nonce, priv\_key)

🔎 Step 3️⃣ — Rebuild the Hash Input:

Rebuild what the sender signed using:  
- peer\_nonce + sorted list of participant IDs.  
Python Code:  
hash\_input = peer\_nonce + b"".join([id.encode() for id in sorted(["A", "B", "C"])])

🔐 Step 4️⃣ — Verify the Signature:

Use sender’s public key to verify the signature:  
Python Code:  
peer\_sig = base64.b64decode(peer\_data["sig"])  
peer\_pub = load\_peer\_public\_key\_from\_cert("A")  
  
peer\_pub.verify(  
 peer\_sig,  
 hash\_input,  
 padding.PSS(mgf=padding.MGF1(hashes.SHA256()), salt\_length=padding.PSS.MAX\_LENGTH),  
 hashes.SHA256()  
)  
- peer\_pub: Sender's public key.  
- peer\_sig: Digital signature.  
- hash\_input: Rebuilt signed message.

# 🔒 Trusting the Secret

You do NOT trust the decrypted nonce immediately.   
You trust it only AFTER two verifications:  
1. Decryption succeeds using your private key.  
2. Signature verification succeeds using the sender's public key.  
  
Only if both succeed, the decrypted nonce (secret) is considered genuine and safe to use.

# 🎯 Verification Outcome

- If no exception is raised → ✅ Signature valid → Trust the received nonce.  
- If verification fails → ❌ Signature invalid → Discard the message.

# 🔐 Saving and Using the Nonce

Save the verified nonce for later use in session key derivation.  
Python Code:  
peer\_nonces["A"] = peer\_nonce

# 🛡️ Final Step: Deriving Shared Session Key

After collecting nonces from all peers:  
Python Code:  
kabc = derive\_kabc([peer\_nonces[i] for i in sorted(peer\_nonces)])  
aesgcm = AESGCM(kabc)

# 💡 Real World Analogy

Imagine:  
- A sends you a secret inside a locked box addressed to you.  
- A also seals the box with a wax seal (signature).  
- Server just carries the box — cannot open or modify it.  
You:  
1. Unlock the box with your private key (decrypt).  
2. Check the wax seal with A’s public key (verify signature).  
3. Trust the secret ONLY if the wax seal (signature) is valid.  
4. Combine secrets from all participants to build the shared session key (Kabc).