

Post-Quantum Cryptography in Kotlin Multiplatform

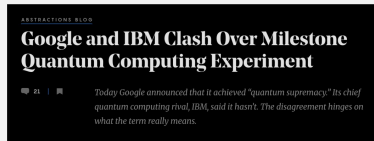
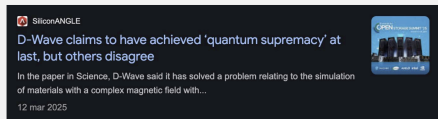
GDG Vicenza DevFest 2026

Emmanuele Villa

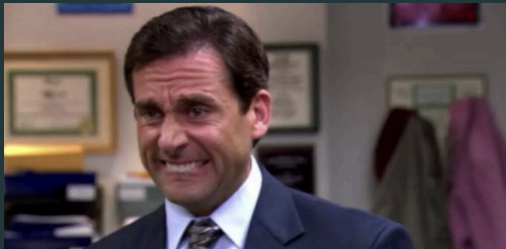
Why Quantum matters (now)?

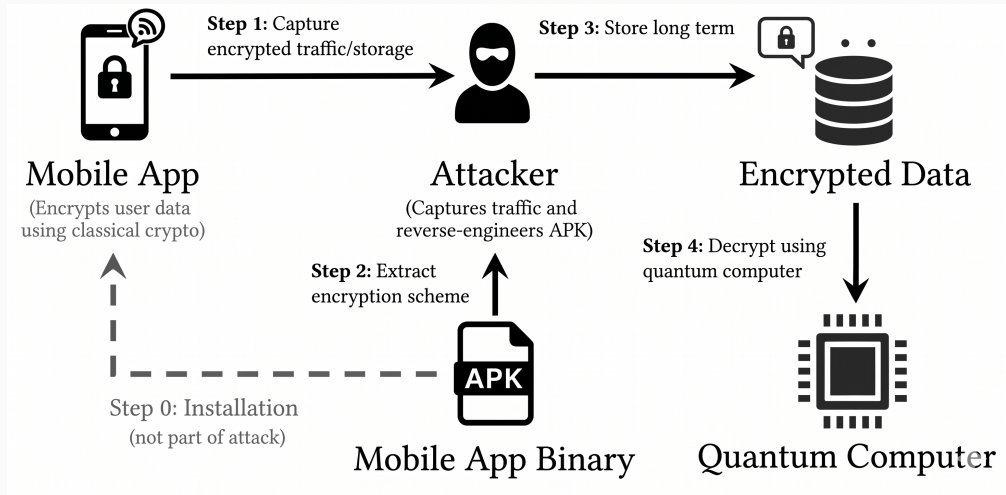


- We are still in the **NISQ** era: we can demonstrate **quantum advantage** on narrow benchmarks, but “quantum supremacy” has no universally agreed metric.
- Research is pushing toward **larger and more stable systems** (e.g., the 10 000-qubit milestone) by improving **fault-tolerance** and error correction.



Are we safe now?





Assessing and Enhancing Quantum Readiness in Mobile Apps (2025)
Joseph Strauss, Krishna Upadhyay, A. B. Siddique, Ibrahim Baggili, Umar Farooq

Data at danger

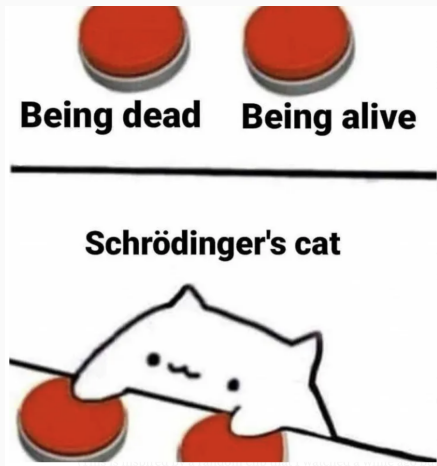
If the information must stay secret for years/decades, it is at risk:

- **Personal data** (health, legal, demographic).
- **Financial records** (transactions, accounts, contracts).
- **Corporate secrets** (IP, source code, strategic plans).
- **Critical infrastructure** (diplomatic, military, infrastructure communications).



What quantum can do





- A qubit can be in a **superposition**: a weighted combination of 0 and 1 until you measure it.
- The result also depends on **how you measure**.
- With **entanglement**, qubits share correlations so algorithms can exploit **interference** to speed up some computations.

Cryptographic primitives

Using Base64 as “encryption”:



Cryptographic primitives

Hash functions

SHA-3



Symmetric encryption

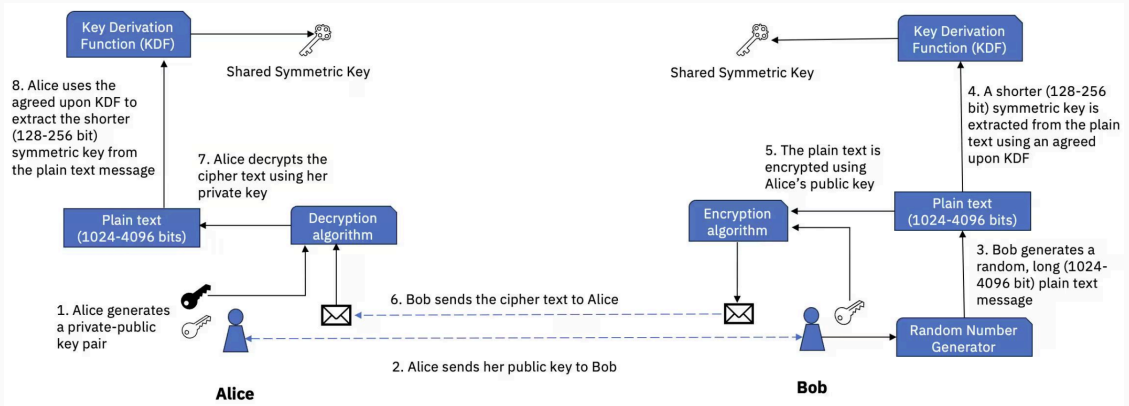
AES-256



Asymmetric encryption

RSA, DH/ECC





Non interactive key transport

How to break them

Primitive	Classical attack	Quantum attack
Asymmetric	RSA: GNFS - DH/ECC: Pollard's Rho	Shor's algorithm
Symmetric (AES)	Brute-force key search	Grover's algorithm
Hash functions (SHA)	Collisions: birthday attack	Preimage: Grover's algorithm

RSA key generation

1. Prime number extraction

Choose two large primes p and q , then compute $n = pq$.

Example

$$p = 17$$

$$q = 23$$

$$n = pq = 391$$

RSA key generation

1. Prime number extraction

Choose two large primes p and q , then compute $n = pq$.

2. Euler's totient function

Compute $\varphi(n) = (p - 1)(q - 1)$.

Example

$$p = 17$$

$$q = 23$$

$$n = pq = 391$$

$$\varphi(n) = (p - 1)(q - 1) = 352$$

RSA key generation

1. Prime number extraction

Choose two large primes p and q , then compute $n = pq$.

2. Euler's totient function

Compute $\varphi(n) = (p - 1)(q - 1)$.

3. Finding the exponent

Choose e coprime with $\varphi(n)$: $\gcd(e, \varphi(n)) = 1$.

Example

$$p = 17$$

$$q = 23$$

$$n = pq = 391$$

$$\varphi(n) = (p - 1)(q - 1) = 352$$

$$e = 65537$$

RSA key generation

1. Prime number extraction

Choose two large primes p and q , then compute $n = pq$.

2. Euler's totient function

Compute $\varphi(n) = (p - 1)(q - 1)$.

3. Finding the exponent

Choose e coprime with $\varphi(n)$: $\gcd(e, \varphi(n)) = 1$.

4. Compute the modular inverse

Find d such that $de \equiv 1 \pmod{\varphi(n)}$.

Example

$$p = 17$$

$$q = 23$$

$$n = pq = 391$$

$$\varphi(n) = (p - 1)(q - 1) = 352$$

$$e = 65537$$

$$d = 65$$

RSA key generation

1. Prime number extraction

Choose two large primes p and q , then compute $n = pq$.

2. Euler's totient function

Compute $\varphi(n) = (p - 1)(q - 1)$.

3. Finding the exponent

Choose e coprime with $\varphi(n)$: $\gcd(e, \varphi(n)) = 1$.

4. Compute the modular inverse

Find d such that $de \equiv 1 \pmod{\varphi(n)}$.

5. Key pair

Public key (e, n) and private key (d, n) .

Example

$$p = 17$$

$$q = 23$$

$$n = pq = 391$$

$$\varphi(n) = (p - 1)(q - 1) = 352$$

$$e = 65537$$

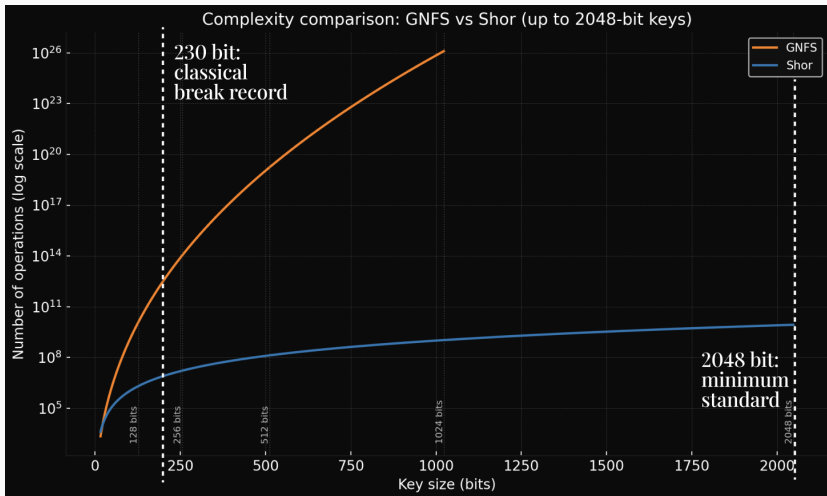
$$d = 65$$

$$\text{public key} = (e, n) = (65537, 391)$$

$$\text{private key} = (d, n) = (65, 391)$$

Factoring breaks RSA

If I can factor $n = 391$, I can recover p and q and compute the private key.



Classical GNFS vs Quantum Shor's algorithm for factoring.

Quantum-resistant

- **Symmetric (AES)**

Grover gives a quadratic speedup $\Rightarrow$ use larger keys (e.g. AES-256).

- **Hash (SHA-2/SHA-3)**

Grover gives a quadratic speedups $\Rightarrow$ use longer outputs (e.g. SHA-384/512).

Quantum-unsafe

- **Asymmetric (RSA, DH/ECC)**

Shor breaks factoring and discrete log in polynomial time.

[Submitted on 11 Jul 2025]

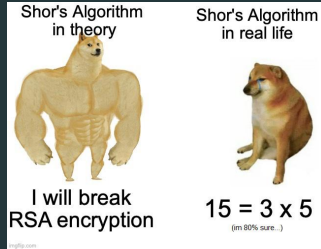
Breaking a 5-Bit Elliptic Curve Key using a 133-Qubit Quantum Computer

Steve Tappeconic

This experiment breaks a 5-bit elliptic curve cryptographic key using a Shor-style quantum attack. Executed on IBM's 133-qubit `ibm_torino` with Qiskit Runtime 2.0, a 15-qubit circuit, comprised of 10 logical qubits and 5 ancilla, interferes over an order-32 elliptic curve subgroup to extract the secret scalar k from the public key relation $Q = kP$, without ever encoding k directly into the oracle. From 16,384 shots, the quantum interference reveals a diagonal ridge in the 32×32 QFT outcome space. The quantum circuit, over 67,000 layers deep, produced valid interference patterns despite extreme circuit depth, and classical post-processing revealed $k = 7$ in the top 100 invertible (a, b) results. All code, circuits, and raw data are publicly available for replication.

Longest key factored by a quantum computer.

State of the art in cryptography



Study on 4000 apps (2025)

75%

Quantum-unsafe

- RSA, MD5 and SHA-1 are among the most used algorithms in mobile applications.

100%

Quantum-resistant

- Algorithms like SHA-256 or AES/CBC-256 are quantum-resistant, but not quantum-safe.

0%

Quantum-safe

- None of the apps implemented completely quantum-safe algorithms.

Assessing and Enhancing Quantum Readiness in Mobile Apps (Strauss et al., 2025).

DATA COLLECTED: JANUARY 9, 2026

Social media, banking, and shopping apps are not yet quantum-resistant

92% of analyzed apps are not yet quantum-resistant, and only two messaging apps and TikTok have implemented quantum-resistant cryptography.

	Not quantum-resistant	Researching quantum-resistance	Quantum-resistant
Messaging apps	     	  	 
Social media apps	  	     	
Banking apps	       	 	
Shopping apps	       	 	



This visual is licensed under a Creative Commons Attribution 4.0 International license — <https://creativecommons.org/licenses/by/4.0/>



Surfshark study on the top 40 installed apps (Jan 2026)

OpenSSH supports a number of cryptographic key agreement algorithms considered to be safe against attacks from quantum computers. We recommend that all SSH connections use these algorithms.

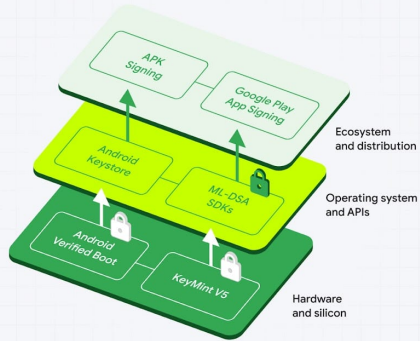
OpenSSH has offered post-quantum key agreement (*KexAlgorithms*) by default since release 9.0 (April 2022), initially via the `sntrup761x25519-sha512` algorithm. More recently, in OpenSSH 9.9, we have added a second post-quantum key agreement `mlkem768x25519-sha256` and it was made the new default scheme in OpenSSH 10.0 (April 2025).

To encourage migration to these stronger algorithms, OpenSSH 10.1 will warn the user when a non post-quantum key agreement scheme is selected, with the following message:

```
** WARNING: connection is not using a post-quantum key exchange algorithm.  
** This session may be vulnerable to "store now, decrypt later" attacks.  
** The server may need to be upgraded. See https://openssh.com/pq.html
```

openSSH uses quantum-safe algorithms for key exchange since version 9.0 (2022).

Android 17 Post-Quantum Chain of Trust



The full-stack vision illustrates how Android 17 secures the entire ecosystem, establishing a continuous chain of mathematical certainty from the physical hardware boot sequence through to the global distribution of applications via Google Play.

Android 17 introduces quantum-safe algorithms for the keystore

0.029%

Quantum-safe algorithms

- 6 004 out of 20 556 816 connections were quantum safe, mostly from openSSH and Google servers.

Post-Quantum Cryptography (PQC) Network Instrument: Measuring PQC Adoption Rates (Sowa et al., 2024).

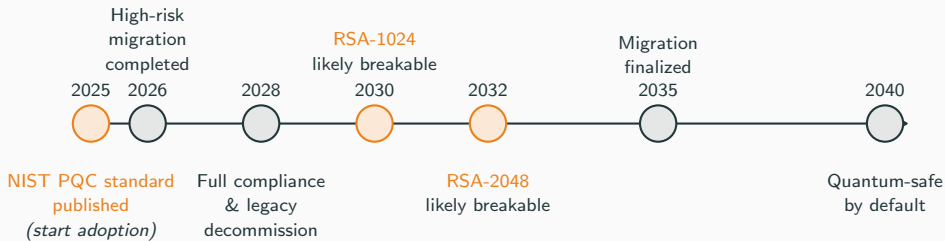
Post Quantum Cryptography

RSA Cryptosystem: *exists*

Quantum Computers:



Roadmap



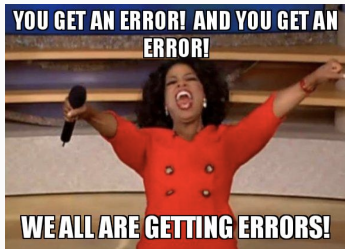
Instead of RSA, ECDH...

Algorithm	Family	Application
CRYSTALS-Kyber	Lattice-based	Key encapsulation
CRYSTALS-Dilithium	Lattice-based	Digital signatures
FALCON	Lattice-based	Lightweight digital signatures
SPHINCS+	Hash-based	Digital Signature

Module-Lattice-Based Key-Encapsulation Mechanism

Based on the **Learning with errors** (LWE) problem:

- Start from a function that is easy to compute.
- Add a small random **error** to the output.
- Inversion becomes hard: noise hides the exact value.

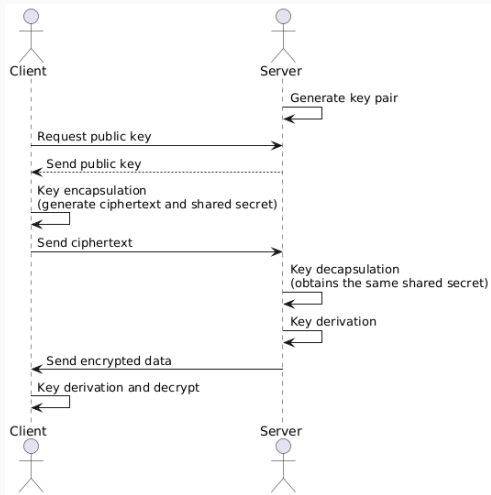


A quantum safe app





- github.com/ronhombre/KyberKotlin
- A **100% Kotlin Multiplatform** implementation of **ML-KEM** (Final).
- Includes **super fast optimizations**.



KyberKotlin flow: key generation, encapsulation, decapsulation, key derivation, encryption and decryption.

 Kotlin Multiplatform Wizard

English

New Project

Templates Gallery

Project Name

KyberKMP

Project ID

com.shadowings.kyberkmp

 **Android**

With Compose Multiplatform UI framework based on Jetpack Compose

☒

 **iOS**

☐

 **Desktop**

☐

 **Web**

☐

 **Server**

With [Ktor](#) framework

☒

☒ Include tests

 DOWNLOAD

Creation of a compose-app with backend.

Key creation



Input:

- Public matrix A of polynomials
- Secret vector s
- Small random error e

Output:

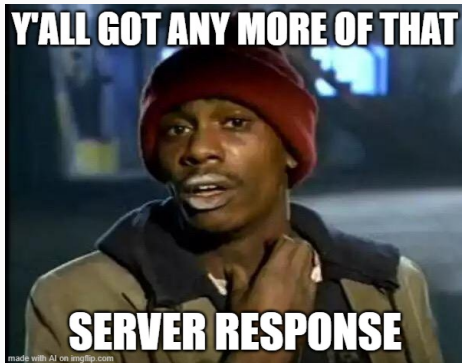
- Public key $t = A \cdot s + e$

Key creation — Server

```
class KyberService {  
    /**  
     * Kyber KEM keypair using ML-KEM-512 parameter set.  
     * Generated once during service initialization,  
     * but it should be rotated periodically in a real application.  
     */  
    private val param = KyberParameter.ML_KEM_512  
    private val keyPair: KyberKEMKeyPair =  
        KyberKeyGenerator.generate(parameter = param)  
  
    fun getPublicKeyBytes(): ByteArray =  
        keyPair.encapsulationKey.key.fullBytes  
}
```



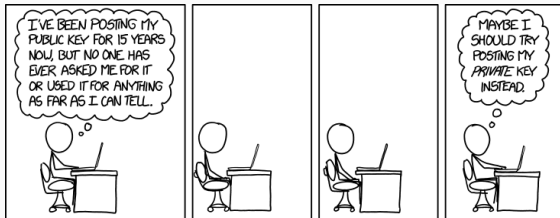
Key creation — Server



```
routing {  
    get("/publicKey") {  
        PublicKeyHandler().handle(call)  
    }  
}  
  
class PublicKeyHandler : KoinComponent {  
    private val kyberService: KyberService by inject()  
  
    suspend fun handle(call: ApplicationCall) {  
        val keyBytes = kyberService.getPublicKeyBytes()  
        call.respondBytes(  
            keyBytes,  
            ContentType.Application.OctetStream  
        )  
    }  
}
```

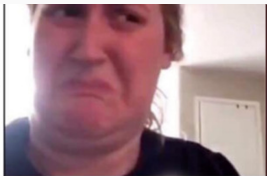
Key retrieval — Client

```
class PublicKeyRepository(  
    private val client: HttpClient  
) {  
    suspend fun getPublicKey(): ByteArray {  
        val response = client.get("${baseUrl}/publicKey")  
        val bytes = response.readRawBytes()  
        return bytes  
    }  
}
```



Key encapsulation

ALGORITHMS



**RANDOMIZED
ALGORITHMS**



Before each request, the client calculates:

$m \leftarrow \text{random byte array}$

$r \leftarrow \text{random vector}$

$e_1, e_2 \leftarrow \text{random error}$

$u = A \cdot r + e_1$

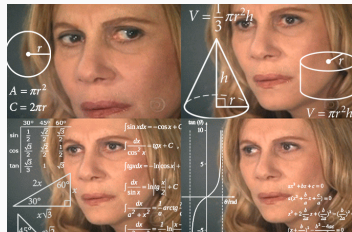
$v = t \cdot r + e_2 + \text{Encode}(m)$

ciphertext = (u, v)

Key encapsulation — Client

```
class PublicKeyRepository(
    private val client: HttpClient
) {
    suspend fun getPublicKey(): KyberEncapsulationResult {
        val response = client.get("${baseUrl}/publicKey")
        val bytes = response.readRawBytes()
        val encapsulationKey =
            KyberEncapsulationKey.Companion.fromBytes(bytes)
        return encapsulationKey.encapsulate()
    }
}

/**
 * A class for ML-KEM Encapsulation Results.
 * Contains Shared Secret Key and CipherText.
 */
@JsExport
class KyberEncapsulationResult(
    private val sharedSecretKey_: ByteArray, // This is calculated from m
    private val cipherText_: KyberCipherText // This is (u,v)
)
```

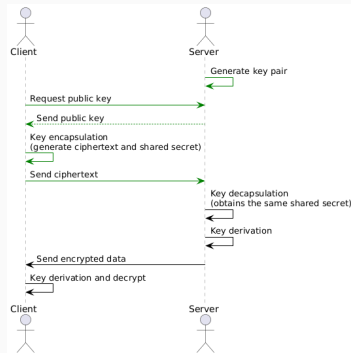




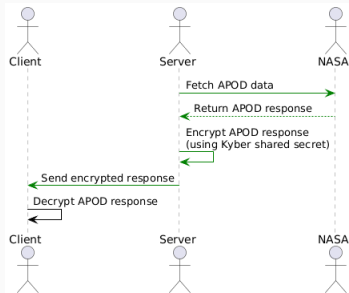
NASA Astronomy Picture Of the Day (APOD)

Data request — Client

```
class ApodRepository() {  
    suspend fun getApod(): ApodResponse {  
        // 1. Obtain Kyber ciphertext  
        val keys = publicKeyRepository.getPublicKey()  
  
        // 2. POST ciphertext to server  
        val request = ApodRequest(  
            cipherText = Base64Utils.encode(  
                keys.cipherText.fullBytes  
            ),  
            requestId = Uuid.random().toString(),  
            deviceId = Uuid.random().toString()  
        )  
  
        val responseString = client.post("${baseUrl}/apod") {  
            contentType(ContentType.Application.Json)  
            setBody(request)  
        }.body<String>()  
        // ...  
    }  
}
```



Data preparation — Server



```
routing {
    get("/apod") {
        ApodHandler().handle(call)
    }
}

class ApodHandler : KoinComponent {
    private val crypto: ICrypto by inject()

    suspend fun handle(call: ApplicationCall) {
        val apodResponse = fetchApod(httpClient)
        val encryptedResponse = crypto.encryptApodResponse(
            kyberService = kyberService,
            apodRequest = apodRequest,
            apod = apodResponse
        )
        val json = JsonUtil.strictJson
            .encodeToString(encryptedResponse)
        call.respond(json)
    }
}
```

Key decapsulation

Server State & Goal

- **Receives:** Ciphertext (u, v) from the client.
- **Already has:** Private key s and public key $t = A \cdot s + e$.

Noise Removal (Rounding)

- The combined error term $(e \cdot r + e_2 - s \cdot e_1)$ is small by design.
- Modular rounding filters out this noise to recover $\text{Encode}(m)$ exactly.
- From $\text{Encode}(m)$, it recovers m and then the shared key.

Mathematical Steps

$$v - s \cdot u = (t \cdot r + e_2 + \text{Encode}(m)) - s \cdot (A \cdot r + e_1)$$

$$\text{[Substitute } t = A \cdot s + e]$$

$$= (A \cdot s + e) \cdot r + e_2 + \text{Encode}(m) - s \cdot (A \cdot r + e_1)$$

$$\text{[Distribute terms / Expand products]}$$

$$= A \cdot s \cdot r + e \cdot r + e_2 + \text{Encode}(m) - s \cdot A \cdot r - s \cdot e_1$$

$$\text{[Cancel } A \cdot s \cdot r \text{ and } s \cdot A \cdot r]$$

$$= \text{Encode}(m) + \underbrace{(e \cdot r + e_2 - s \cdot e_1)}_{\text{combined error}}$$

Key decapsulation — Server

```
override fun encryptApodResponse(
    kyberService: KyberService,
    apodRequest: ApodRequest,
    apod: ApodResponse
): ApodSecureResponse {

    // 1. Receive Kyber ciphertext encapsulating shared secret
    val kyberCipherText = KyberCipherText.fromBytes(ciphertext)

    // 2. Decapsulate shared secret using Kyber private key
    val sharedSecret = kyberService.keyPair.decapsulationKey
        .decapsulate(kyberCipherText)

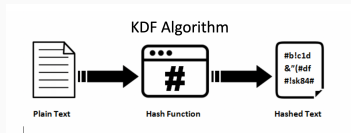
    // ...
}
```



Server

- We don't use the raw shared key directly: it is only an **intermediate secret**.
- Our secret may be shorter than the length required by symmetric algorithms.
- Different use cases may need independent keys.
- Deriving session keys prevents long-term compromise if one session key is exposed.

Key derivation — Server



```
// 3. Generate a cryptographically random 12-byte nonce (GCM standard).
val nonce = ByteArray(Constants.GCM_NONCE_SIZE_BYTES)
SecureRandom().nextBytes(nonce)

// 4. Create Additional Authenticated Data (AAD) metadata
val additionalData = AdditionalAuthenticatedData(
    requestId = apodRequest.requestId,
    deviceId = apodRequest.deviceId,
)
val aad = JsonUtil.strictJson
    .encodeToString(additionalData)
    .encodeToByteArray()

// 5. Derive AES-256 key via HKDF (HMAC-SHA256) with
val info = ContextInfo(
    requestId = apodRequest.requestId,
    deviceId = apodRequest.deviceId,
    nonce = Base64Utils.encode(nonce)
)

val derivedKey = crypto.deriveAES256Key(
    sharedSecret = sharedSecret,
    salt = nonce,
    info = JsonUtil.strictJson
        .encodeToString(info)
        .encodeToByteArray()
)
```

deriveAES256Key() — Server

```
override fun deriveAES256Key(  
    sharedSecret: ByteArray,  
    salt: ByteArray,  
    info: ByteArray  
): ByteArray {  
    // Extract step: use HMAC-SHA256 with salt as key and sharedSecret as input,  
    // to produce a fixed-length pseudorandom key (PRK).  
    val prk = hMacSHA256(salt, sharedSecret)  
  
    // Expand step: use the PRK to generate the desired key length,  
    // incorporating the context info (domain separation).  
    val result = hkdfExpand(prk, info, Constants.AES_KEY_SIZE_BYTES)  
  
    // Clear sensitive intermediate key material from memory  
    prk.fill(0)  
    return result  
}
```



“SHA-256 is not Quantum Safe”

— *Emmanuele Villa*

Why we still use SHA-256

01 — Quantum resistance

- Grover reduces security from 2^{256} to 2^{128} operations.
- Still extremely hard and practically secure for key derivation.

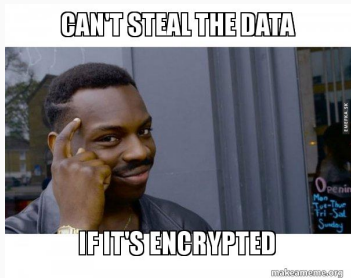
02 — Role of the KEM

- The KEM protects the **private key**.
- Breaking SHA-256 only exposes a single message.
- Breaking the private key compromises all communications.

03 — Conclusion

- Not theoretically quantum-safe, but practical risk is low.
- Protecting the private key is more critical than protecting individual messages.

Encrypt data — Server



```
// 6. Zero out shared secret immediately after use
sharedSecret.fill(0)

// 7. Encrypt APOD plaintext with AES-256-GCM (derived key + nonce + AAD)
val encrypted = crypto.encryptAESGCM(
    key = derivedKey,
    plaintext = JsonUtil.strictJson
        .encodeToString(apod)
        .encodeToByteArray(),
    nonce = nonce,
    aad = aad
)

// 8. Zero out derived AES key after encryption
derivedKey.fill(0)

return ApodSecureResponse(
    nonce = Base64.getEncoder().encodeToString(nonce),
    data = Base64.getEncoder().encodeToString(encrypted)
)
```

Data decryption — Client

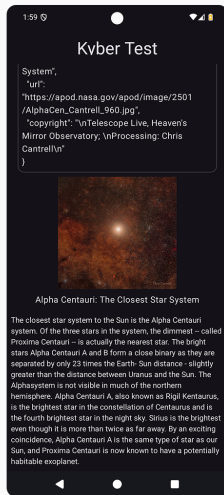
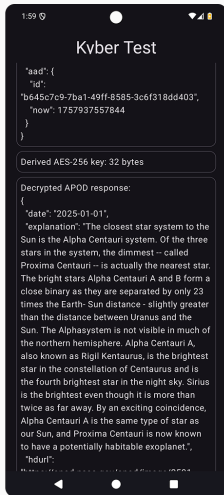
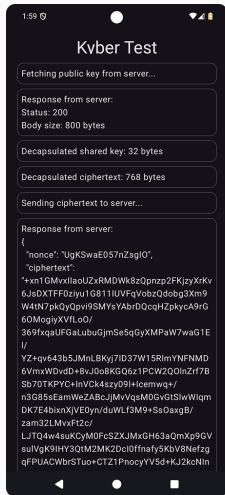


```
class ApodRepository(private val client: HttpClient) {  
    suspend fun getApod(): ApodResponse {  
        // ... (Send ciphertext & get encrypted payload)  
  
        // 3. Deserialize JSON & decode Base64 data  
        val secure = Json.decodeFromString<ApodSecureResponse>(responseString)  
        val nonceBytes = Base64.getDecoder().decode(secure.nonce)  
        val encrypted = Base64.getDecoder().decode(secure.data)  
  
        // 4. Derive the SAME AES-256 key via HKDF  
        val derivedKey = crypto.deriveAES256Key(  
            sharedSecret = keys.sharedSecretKey, // Already encapsulated!  
            salt = nonceBytes,  
            info = contextInfoBytes  
        )  
  
        // 5. Decrypt payload via AES-256-GCM  
        val decrypted = crypto.decryptAESGCM(  
            key = derivedKey, ciphertext = encrypted,  
            nonce = nonceBytes, aad = aadBytes  
        )  
  
        // 6. Wipe sensitive keys & return result  
        keys.sharedSecretKey.fill(0)  
        derivedKey.fill(0)  
        return Json.decodeFromString(decrypted.decodeToString())  
    }  
}
```

Run the stuff

```
# Run the server
./gradlew server:run

# Run the app
./gradlew app:installDebug
adb shell am start -n
    org.shadowings.kybertest/.presentation.activity.MainActivity
```



The demo app that retrieves the NASA Astronomy Picture Of the Day (APOD) from a server, using Kyber for key exchange and AES-GCM for encryption.

Resources

- 01 — Standards and specifications
- 02 — Academic references
- 03 — Software and implementations
- 04 — QC resources

Thank you!

emmanuelevilla.com/qc

