Funded by the European Union under Grant Agreement No. 101087529. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or European Research Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

Co-funded by the European Union

Integrating Post-Quantum Cryptography into Existing Systems Today

CHESS

Cyber-security Excellence Hub in Estonia and South Moravia

ESSCaSS 16.8.2023

Petr Muzikant - petr.muzikant@cyber.ee

[All references are in the form of clickable links]

CYBERNETICA

Presentation outline

- **Introduction**
- **Current state of PQC**
  - Libraries, ASN.1, JSON Web Algorithms, Hybrid modes
- **General implementation challenges**
  - Relevant locations, technological constraints, implementation in the codebase
- **Practical results, remarks, and examples**
  - PQ authentication framework, PQ-CDOC2, PQ-IVXV
- **Conclusions**

Introduction

- **Standardization** of PQC (Post-Quantum Cryptography) is *only* the **first step in a long process of actual deployment** in real-life IT systems
- Challenge:
  - Rolling out PQ support in all system and architecture layers
  - While ensuring functionality, compatibility, interoperability (and security)
- Our work:
  - Focus on engineering aspects of PQ protocol implementations
  - Exploring current options
  - Remarks and tips

Current state of PQC

Libraries, ASN.1, JSON Web Algorithms, Hybrid modes

PQ Libraries

- PQClean (C)
  - aggregates NIST-submitted algorithms with unified API
- libOQS (C)
  - higher-level library (provides submitted algorithms until NIST round 4)
  - wrappers for C++, Python, Java, Go, .NET, and Rust
  - applications built with libOQS (OpenSSL, OpenSSH, OpenVPN forks)
- libpqcrypto (C)
  - similar to libOQS, not maintained anymore? (last update in 2018)
- BouncyCastle (Java), rustpq/pqcrypto (Rust), pqm4 (C, Cortex-M4)

PQ ASN.1 structures

- Essential for PQ-X509, but also used in other applications
- No standards exist yet - NIST requires raw bytes
- Multiple RFC drafts for specific PQ algorithms
  - private and public keys with specific attributes/parameters
  - e.g. `DilithiumPrivateKey` (contains `nonce`, `tr`, `s1`, `s2`, `t0`, etc.)
- Differences in PQ libraries
  - e.g. *libOQS* returns raw bytes, *BouncyCastle* returns proposed ASN1 objects
- PQ Object Identifiers (OIDs): OQS defined their own, BouncyCastle expanded with KEMs

PQ JSON Web Algorithms (RFC 7518)

- Usage: *JW Signature*
- Sig. format: *(DIGSIG + HASH identification)*
- Example: *"ES384" means "ECDSA using P-384 curve and SHA-384"*
- No RFC drafts for PQ JWAs, but there is RFC draft for PQ JW Encodings:
  - e.g. *CRYDI5 = CRYSTALS-Dilithium parameter set 5*
  - only DIGSIG identification, no HASH?
    - *always use SHA512?*
    - *CRYDI5-256/384/512?*
    - *wait for another RFC?*

Hybrid mode (PQ + classic crypto)

- Post-Quantum cryptography:
  - ensures the longevity of data protection
- Classical cryptography:
  - protects against emerging threats on unexplored PQC
- Most common modes:
  - concatenation, sequential
  - both can have their issues → nothing concrete yet
  - *Ghinea et al.* propose novel method to improve unforgeability of hybrid dig. sig. when using ECDSA

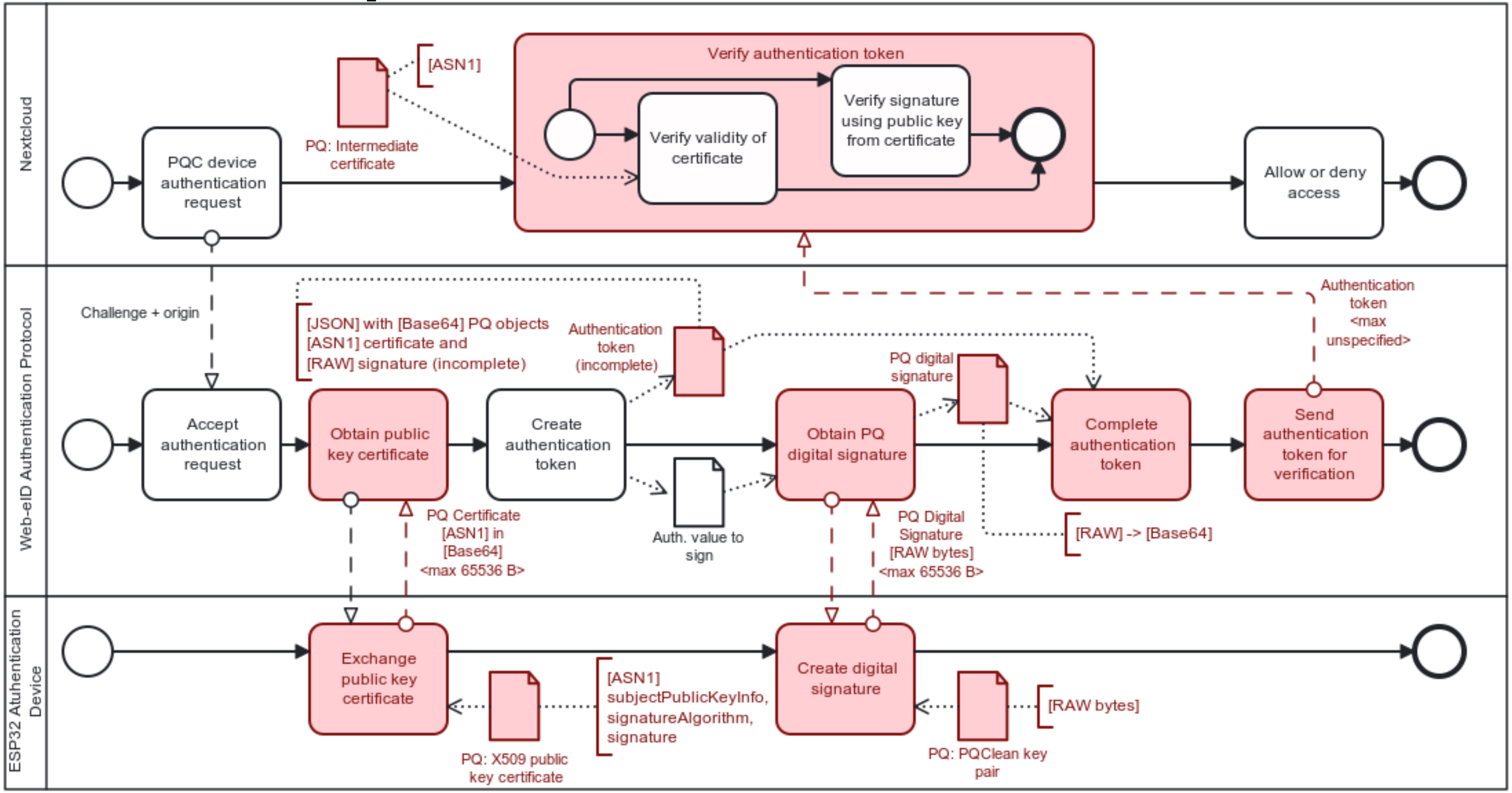
General implementation challenges

Relevant locations, technological constraints, implementation in the codebase

Identifying relevant locations

- First step in implementing PQC in an existing application
- **Identify all PKI** (public key infrastructure) **objects from the start of their lifetime** to their ends
  - helps to understand the extent of required changes
- **Beware of MTU** (Maximum Transmission Unit) when transferring PKI objects between different system components
  - bigger objects, variable size (Falcon)
- **Beware of changing data formats** (ASN1, Base64, PEM, ...) between components

BPMN example



Technological constraints

- Assess the technological and computational **boundaries of the current system**
  - Increased **performance**, **memory**, and **storage overhead**
  - Limited devices and slow networks
- Protocol adjustment examples:
  - streaming public keys and signatures into the limited memory of a HSM component
  - use key encapsulation instead of digital signatures
  - allocate all objects in heap instead of limited stack memory (our case)

Implementing PQ algorithms in the codebase

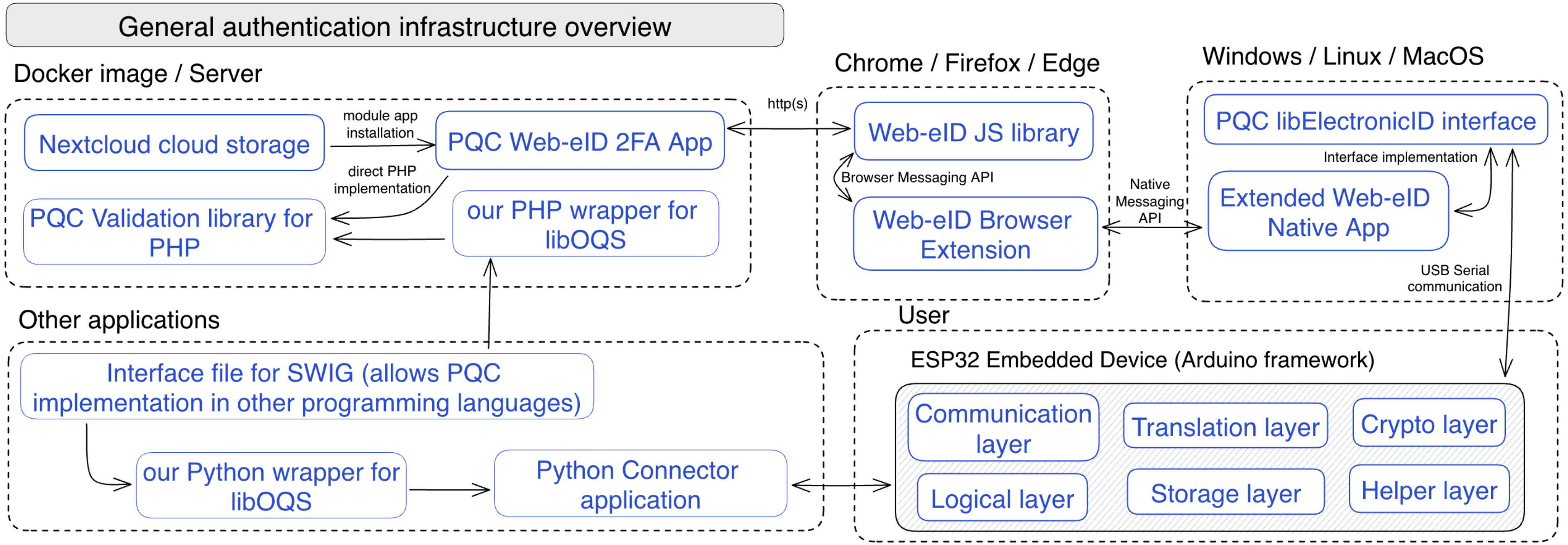
- Start at the **beginning of the data lifecycle**
- Implement post-quantum support **one step at a time**
- **Extensions or complete swaps** of cryptographic libraries might be required
  - if not available, create your own using SWIG
- Expect future changes - standardization is not over!

Practical results, remarks, and examples

PQ authentication framework, PQ-CDOC2, PQ-IVXV

Project: PQ Authentication Framework (PoC)

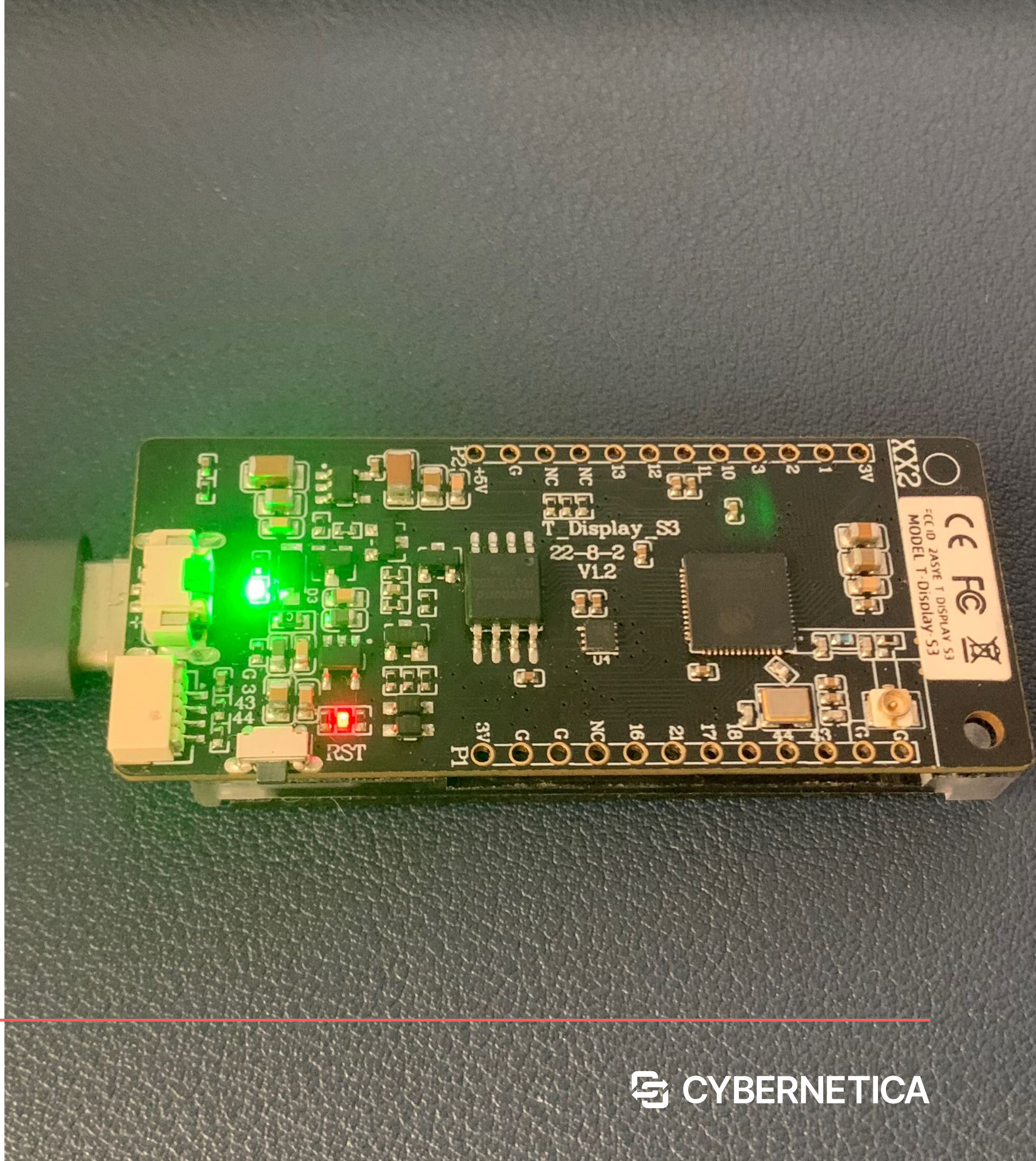
- Nextcloud cloud storage + Web-eID + embedded device
  - authentication result based on Dilithium5 or Falcon1024 signatures
  - multiple low-level PQ-capable components



CYBERNETICA

Embedded devices

- Smart cards are not suitable yet(?)
  → LilyGO T-Display-S3
- Problematic memory management:
  - Limited to 8 KB of stack RAM
  - *PQClean* allocates to a stack a lot
- Solved by adjusting *PQClean* code by using:
  - `malloc` and `free` functions
  - `std::unique_ptr` (C++ v11)



libOQS extensions

- Available wrappers for C++, Python, Java, Go, .NET, and Rust
  - ~~PHP~~? → SWIG wrapper generator!
- C/C++ interface definition required
  - → *liboqs-php*, *liboqs-python*
- Some remapping was required:
  - PHP's `string` + Python's `bytearray` into C++'s `uint8_t*`
  - and vice versa

PQ in PHP

- **OpenSSL usage → OQS-OpenSSL**
  - v1.1 fork with built-in functions:
    - complicated installation, PHP rebuild required
    - built-in functions have only DSA, DH, RSA, and EC hardcoded
  - v1.1 fork with command execution using `exec()`, `system()`, etc.:
    - works, but is not practical
  - v3 extension provider with combined usage (recommended):
    - extends regular OpenSSL@3
    - some built-in functions do not require algorithm identifier (e.g. `openssl_verify()`)
- **PHPSecLib → our PQC-PHPSecLib fork**
  - uses *OQS-OpenSSL* or *liboqs-php* (based on availability)

Project: PQ-CDOC2

- **CDOC2** = new version of *Encryption DigiDoc Format* (in development)
  - specification defining the process of **securing and exchanging encrypted messages**, similar to CMS (Cryptographic Message Syntax)
  - reference implementation in Java (uses BouncyCastle)
  - expects RSA, EC or symmetric keys
- **PQ-CDOC2** = CDOC2 expanded to accept **CRYSTALS-Kyber keys**
  - updated version of BouncyCastle to include PQ algorithms (v1.74)
  - expanded the codebase by following RSA/EC objects

PQ in BouncyCastle

- Not well documented
- `org.bouncycastle.pqc.*` packages
- Works with actual algorithm parameters from ASN1 drafts
  - vs raw bytes in libOQS
  - e.g. `KyberPublicKeyParameters` has `t` and `rho`

```
static {
    java.security.Security.addProvider(
        new org.bouncycastle.pqc.jcajce.provider.BouncyCastlePQCProvider()
    );
}
```

PQ in BouncyCastle

```java
public static AsymmetricCipherKeyPair generateKeyPair(KyberParameters params) throws
NoSuchAlgorithmException {
    KyberKeyPairGenerator kpGen = new KyberKeyPairGenerator();
    kpGen.init(new KyberKeyGenerationParameters(Crypto.getSecureRandom(), params));

    return kpGen.generateKeyPair();
}

SubjectPublicKeyInfoFactory.createSubjectPublicKeyInfo(
    (KyberPublicKeyParameters) kpGen.getPublic()
);

PrivateKeyInfoFactory.createPrivateKeyInfo(
    (KyberPrivateKeyParameters) kpGen.getPrivate()
);
```

PQ in BouncyCastle

```java
public static SecretWithEncapsulation kyberEncapsulate(KyberPublicKey kyberPublicKey)
        throws GeneralSecurityException {
    try {
        KyberPublicKeyParameters keyParams = (KyberPublicKeyParameters) PublicKeyFactory
                .createKey(kyberPublicKey.getEncoded());

        KyberKEMGenerator kem = new KyberKEMGenerator(Crypto.getSecureRandom());
        return kem.generateEncapsulated(keyParams);
    } catch (IOException | NoSuchAlgorithmException e) {
        throw new GeneralSecurityException(e);
    }
}
```

PQ in BouncyCastle

```java
public static byte[] kyberDecapsulate(byte[] encapsulation, KyberPrivateKey
kyberPrivateKey)
        throws GeneralSecurityException {
    try {
        KyberPrivateKeyParameters keyParams = (KyberPrivateKeyParameters)
PrivateKeyFactory
                .createKey(kyberPrivateKey.getEncoded());

        KyberKEMExtractor kem = new KyberKEMExtractor(keyParams);
        return kem.extractSecret(encapsulation);
    } catch (IOException e) {
        throw new GeneralSecurityException(e);
    }
}
```

PQ-CDOC2: key-server scenario

- Scenario with key exchange server ensures:
  - possibility of decrypting only once even after private key compromise
  - possibility of encrypted message's expiry date
- Scenario requires:
  - public key **in TLS client certificate** == public key of a recipient **in encrypted message**
  - i.e. client needs to provide a valid certificate, where **subjectPublicKeyInfo is a KYBER key** = problem in current Java SSL implementation
- Solution:
  - use X509 extension `id-ce-subjectAltPublicKeyInfo (2.5.29.72)`

PQ Java Keytool

- keytool = command for managing a keystore of cryptographic objects
- PQ BouncyCastle → PQ Java Keytool
- e.g. to generate .p12 with Dilithium keypair and self-signed certificate:

```
keytool \
    -providerpath bcprov-jdk18on-175.jar \
    -provider org.bouncycastle.pqc.jcajce.provider.BouncyCastlePQCProvider \
    -genkeypair \
    -keyalg Dilithium5 \
    -alias cdoc20-client-pqc-CA \
    -keystore cdoc20clientpqcCA.p12 \
    -storepass passwd \
    -sigalg Dilithium5 \
    -dname "CN=cdoc20-client-pqc-CA,OU=ISRI,O=CyberneticaAS,L=Brno,S=Czechia,C=CZ"
```

Project: PQ-IVXV (El. voting scheme)

- Quite a challenge ahead of us:
  - ensure quantum-safety of electronic voting process
- Will require PQ versions of **more advanced cryptographic primitives**
  - vote encryption, mix-nets, ZK proofs
- Current implementation is written in:
  - Java (pure implementations of ElGamal → new dependency?)
  - Go (CIRCL Library?)
- New project-specific scheme will be probably required for vote encryption (we can't use standard KEM algorithms)

Conclusions

- Implementing PQC today is **possible, but complicated**:
  - different libraries → different approaches
  - not well documented
  - computational constraints
  - standardization is not finished
- But it is definitely **worth a try**!
  - all quantum-computer **timelines are** (most probably) **estimates**
  - better to be **prepared and safe** than **sorry and late** ;)
- There is big space for **open-source PQ contributions**

References

For developers and engineers:

- links in presentation slides
- PQ Authentication Framework (all components)
- OQS-OpenSSL in PHP remarks (documentation)
- PQ-CDOC2: currently in private development repo, may appear in official one some day, contact me for more details

Petr Muzikant, petr.muzikant@cyber.ee

https://cyber.ee/
info@cyber.ee
cybernetica
CyberneticaAS
cybernetica_ee
Cybernetica