Migrating Some Legacy e-Governance Applications to Post-Quantum Cryptography

Paper accompanying the presentation at 5th NIST PQC Standardization Conference

Petr Muzikant, Jan Willemson, Peeter Laud
Cybernetica AS
e-mail: firstname.lastname@cyber.ee

January 2024

Abstract

This paper speaks about our experience in adding support for post-quantum cryptography to a number of software components underlying and supporting the e-Governance processes of Estonia. In the paper, we briefly introduce our approach with state of the art of post-quantum engineering and our methodology. Then, we describe several Estonia's e-Government-related projects and provide an experience report about our efforts to make them quantum safe. We present some engineering problems we had to tackle with, different project-specific approaches, etc. Overall, we believe that this paper would be beneficial for the general audience interested in practical adoption of post-quantum cryptography.

1 Introduction

Estonian e-Governance depends on a number software and hardware components, some of which are almost a quarter of a century old. Recently, we have taken some of these (perhaps a bit more recent) components and augmented them with the support for PQ-secure cryptographic primitives. The tasks of the augmented components are related to user authentication, non-repudiation, and secure exchange of documents.

1.1 Challenge

Companies and organizations in general are slowly starting to give more and more focus to post-quantum cryptography, and are starting to instruct their engineers (sometimes without any background in information security) to prepare their product line for the arrival of quantum computers. These engineers may find themselves struggling due to the novelty of the technology, and lack of documentation and support. Their challenge is to implement post-quantum support in all system and architecture layers while keeping the functionality, compatibility, interoperability, and efficiency intact.

Our PQ-related work aims to improve this situation by exploring, researching, and solving engineering problems in various systems. We base our research on the Estonian e-Government and e-Services ecosystem which is one of the most advanced in Europe and which we are closely familiar with.

1.2 Overview of the Presentation

Our presentation covers our overall successful, but sometimes cumbersome attempts to implement post-quantum security in Estonia's "standard" systems for user authentication, encrypted file exchange, digital signature containers, and e-voting. From these experiences, we generalize outcomes suitable for any systems and any engineer. Still, the main focus of the presentation is on our experience, not the generalizations.

Our work is definitely not the first report of adding PQC support to an existing application. Schardong, Giron, Müller, and Custódio [1] created a PQ version of OpenID Connect. López-González, Arjona, Román, and Baturone [2] developed a PQ-safe biometric authentication framework. Yao, Matusiewicz, and Zimmer [3] introduced PQ into Security Protocol and Data Model compliant device authentication. Paul,

Scheible, and Wiemer [4] discussed PQ usage in banking protocols. The use of PQC in TLS has been described by several authors [5, 6].

However, while the listed reports describe the created PQ-secure version of their application and discuss its security and performance, the emphasis of our talk will be different. Our talk focuses on the *engineer's experience* of modifying the existing software and adding PQC support to it. We describe the engineering process in creating PQ-secure components from the existing components. Hence the focus of the talk is also different from PQC migration handbook(s) [7] that offer support to managers in directing the migration, but not to engineers that actually implement it.

The rest of this paper contains interesting details from our experience; these are covered in the presentation to varying extent. Parts of the details and the talk are based on our publication [8].

2 Approach

Our technical choices have been affected by the state of the art in PQC implementation, by the suggestions given by competent organizations (including NIST through its PQC standardization process), and by the architecture and current implementation of the components we are upgrading to PQC. In this chapter, we describe how we gathered the information that helped us to make these choices. Later, in chapter 3 we elaborate where we chose a certain technology and why.

2.1 PQ libraries

Implementations of cryptographic algorithms in various systems and applications are usually provided by standalone cryptography libraries. Thus we started by collecting information about open-source libraries dedicated to or partially supporting post-quantum algorithms. The following paragraphs contain our findings together with brief descriptions of the libraries, and our notes targeted to ease the selection for developers and engineers. In general, we acknowledge justified warnings in the descriptions of several listed libraries to not use them in production. One should always recognize the novelty and engineering immaturity of post-quantum cryptography and be prepared to accept the associated risks.

PQClean (C)[1] This library collects clean implementations of NIST-submitted algorithms under unified API (very similar to the official API required by NIST with mild modifications). It is not a standalone library, rather a source of up-to-date, revised, and tested source code – the authors encourage developers to copy the code from *PQClean* into their project and build it themselves. We have learned that this approach is particularly beneficial when developing embedded or limited systems (see section 3.1 for our experience with *PQClean* on an embedded device), where cryptographic agility is less important at runtime, but switching to different algorithms is still straight-forward thanks to the unified API.

libOQS (C)[2] OpenQuantumSafe project provides a well-maintained standalone C library *libOQS*. It also provides a collection of post-quantum Key Encapsulation Mechanisms (KEMs) and Digital Signature (DS) algorithms under a common API. On the lower level, it utilizes the previously mentioned *PQClean* as well as other sources including official submissions or GitHub repositories. Its current usability and availability is very high since OpenQuantumSafe also provides wrappers of this library for other programming languages as well as some popular cryptographic applications (e.g. *OpenSSL*, *OpenSSH*, and *OpenVPN* forks with *libOQS* as a provider).

Language or system specific libraries In case a targeted system consists of a single technology stack, one could opt for a library specific for this stack. For example, there exist *pqm4*[3] (a library optimized for ARM Cortex-M4 family of microcontrollers) and *rustpq/pqcrypto*[4] (a binding library from Rust to *PQClean*).

[1] `https://github.com/PQClean/PQClean`
[2] `https://github.com/open-quantum-safe/liboqs`
[3] `https://github.com/mupq/pqm4`
[4] `https://github.com/rustpq/pqcrypto`

BouncyCastle (Java)[5] We make some comments specific to the BouncyCastle (BC) library. Since BCv1.71, developers can use a new post-quantum cryptography BC provider, but unfortunately, this fact and the whole usage of these new features is quite poorly communicated. The only resource we could find is a very brief webpage from KeyFactor[6] (an official sponsor of the "Legion of the Bouncy Castle", a non-profit organization looking after BouncyCastle) with some examples and a link to a recording of one workshop presenting these features. General developers and engineers aiming to utilize PQ in BouncyCastle might appreciate some official documentation or whitepapers, because their only available resource is the source code itself. There is also a significant difference in how BouncyCastle interprets post-quantum cryptography objects – more on that in section 2.2.1.

Custom wrappers Another quite flexible option is to create and maintain custom-made wrappers over abovementioned libraries (most probably *libOQS* as it is the most all-rounded). This allows engineers to introduce another custom-made layer between their software and post-quantum cryptography libraries to increase interoperability without changing their code base too much. A reliable tool for creating such wrappers is *SWIG*[7], which requires to create a single file in C, where developers choose which functions from the C library to expose in the target language, which argument types it should accept, and even do some custom pre-processing.

Enhancing the currently used library Lastly, one could contribute to their currently used library by implementing PQ support and sharing it publicly.

2.2 Data structures

Another issue we had to face was the interpretation of new post-quantum cryptography objects in an already established technology stack. More precisely, how to identify them and tell the system components how to recognize them.

2.2.1 ASN.1 Notation

ASN.1 notation is the most commonly used format to store (not-only) cryptographic objects. It can be specified on every level, from X509 certificates down to exact algorithm-specific parameters or values. We acknowledge NIST's approach in PQC submissions to operate with post-quantum cryptography objects as raw bytes while the internal structure of objects is up to the authors of algorithms themselves. Apart from that, there are (some already expired) RFC drafts aiming to specify exact ASN.1 structures for different PQ objects. However, almost all of them are lacking official Object Identifiers (OIDs) for these algorithms, which are essentially required for inclusion into existing systems. Systems could work with PQ objects as raw bytes during transfer between components, but there still needs to be information about what those bytes are representing.

This is probably the main reason why OpenQuantumSafe organization came up with temporary OIDs specified in a simple and public table[8]. The following storyline is confusing and shows that proper standardization in this area is absolutely necessary. At first, they specified OIDs only for digital signature algorithms as they did not implement KEMs at the time. Then, BouncyCastle used those OIDs and came up with new ones for KEMs (without disclosing it anywhere, only deeply hidden in the source code). OpenQuantumSafe then included those in their current table and filled new ones for KEMs missed by BouncyCastle. For now, developers might want to use OIDs from the mentioned table from OpenQuantumSafe[9].

There is one significant difference between *libOQS* and *BouncyCastle* library. *libOQS* follows NIST's rule about raw bytes and implements its applications following this approach. However, *BouncyCastle* is already

[5]`https://www.bouncycastle.org/`

[6]`https://doc.primekey.com/bouncycastle/post-quantum-hybrid-cryptography-in-bouncy-castle`

[7]`https://swig.org/`

[8]`https://github.com/open-quantum-safe/oqs-provider/blob/main/ALGORITHMS.md\#oids`

[9]Update: new OID related discussion has been conducted at `https://github.com/open-quantum-safe/oqs-provider/issues/351`

only incorporating post-quantum objects based on the proposed ASN.1 drafts. Therefore the two libraries are not compatible with each other, and the developers must act accordingly[10].

2.2.2 JSON Web Algorithms

Certain applications transfer cryptographic objects utilizing JSON Web Tokens. For algorithm identification, they usually use JSON Web Algorithms [9]. E.g. in the digital signature context, *ES256* means *ECDSA using P-256 and SHA-256* (*hash-then-sign* construct). For the identification of post-quantum algorithms, only a few RFC drafts are available[11]. However, the problem is that they are defining the algorithm identifiers only for the digital signature part (e.g. *CRYDI5* means *CRYSTALS-Dilithium parameter set 5)*, but not for the hash part. The developers trying to replace the original identifiers must think about what data will be signed and verified in their components and whether it is compatible with external parts of the system. One solution could be to introduce a similar suffix (e.g. *CRYDI5-256*). Another solution might be to always use a specific hash function (e.g. SHA-512) everywhere, or not hash the signed data at all if the data to be signed is very small. Currently, it is up to the developer to determine which approach delivers the best compatibility and ease of transition.

2.3 Hybrid mode

We acknowledge the hybrid post-quantum + classical cryptography modes. However, our projects were more focused on post-quantum cryptography implementation experience, hence our additions to the components make use of only PQ algorithms. Once we figure out the details, we will start adding back the classical cryptography in a hybrid mode.

2.4 Methodology

In this section we describe the methodology we used to add support for PQ algorithms to a number of software components.

2.4.1 Preparation

Our first step for implementing post-quantum algorithms in an existing application is to identify all locations in the codebase, network and business processes, where the public key infrastructure (PKI) objects are used. We trace the data flow from start to end of the object's lifetime, as there are differences in how individual processes interact with them. Identifying these data flows helps us understand the extent of required changes in the code.

When transferring cryptographic data objects between different parts of the architecture, it is important to consider the Maximum Transmission Unit (MTU) to ensure that longer keys and signatures fit into the containers. Additionally, caution is needed when implementing the Falcon PQ digital signature algorithm as its output length is not fixed and may vary.

Data formats and possible conversions between them also have to be considered. For example, cryptographic libraries might yield raw bytes, but the rest of the system may handle data in ASN.1, Base64, or PEM encoded formats. Identifying the formats of the existing data flows is the key to ensuring the compatibility across the whole system architecture, and detect any areas that require modification.

BPMN (Business Process Model and Notation) diagrams are a format suitable for visualizing all the above-mentioned information. Figure 1 shows such preparation process for the Web-eID component described in section 3.1. PQC related objects and operations are denoted in red.

2.4.2 Technological Constraints

Before transitioning to post-quantum algorithms, it is essential to assess the technological and computational boundaries of the current system. Increased memory usage is expected when generating PQ keypairs,

[10] We have found only one not-well-maintained repository exploring the compatibility between libOQS and PQ-BouncyCastle: `https://github.com/Open-QKD-Network/oqs-bouncycastle`.

[11] Most up-to-date: `https://www.ietf.org/archive/id/draft-ietf-cose-post-quantum-signatures-01.txt`

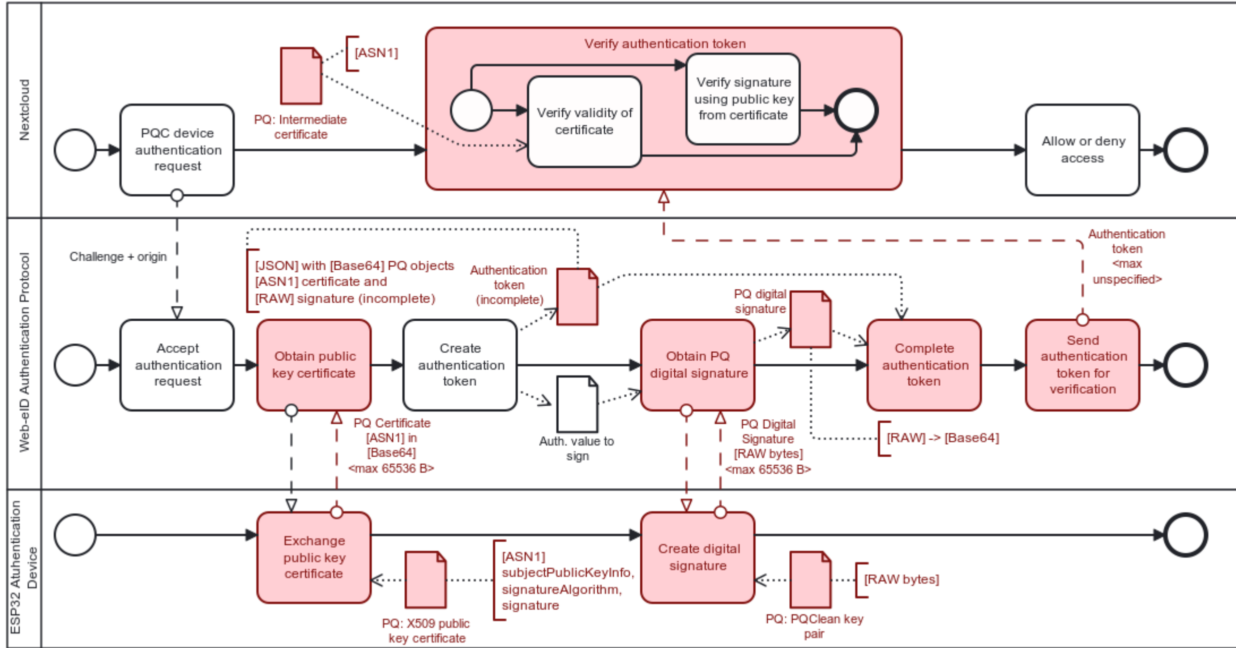

Figure 1: PQC authentication infrastructure components overview

creating signatures, and verifying them. In real-time applications, performance may be impacted. Relevant measurements can be found in [10, 11, 12, 13].

In regular applications meant to be run on desktop, laptop, or server machines, these constraints are not as significant as they are on slow networks or limited devices. In other cases, actual post-quantum algorithms need to be adjusted. For example, Gonzalez et al. [14] propose streaming public keys and signatures into the limited memory of an attached HSM component. Another work by Gonzalez and Wiggers [11] suggests using key encapsulation instead of digital signatures to reduce computational overhead.

2.4.3 Implementation

After identifying all locations and constraints, one can begin changing the codebase. We usually start at the beginning of the data lifecycle and implement post-quantum support one step at a time. Post-quantum algorithms are generally not available natively in used cryptographic libraries, therefore library extensions may be required. Data format conversions may occur during these steps, adding to the potential fragility of the implementation.

General overview of the steps is depicted in Figure 2.

3 Applications

In this chapter, we describe our experience from working with several (mostly e-Government) components and applications, together with the encountered engineering problems and how we dealt with them.

3.1 Web-eID

Web-eID[12] is a suite of applications, extensions, and tools developed by Estonian Information Systems Authority[13]. It enables authentication and digital signing with public-key cryptography on the web using smart cards. The authentication workflow is similar to the Transport Layer Security: Client Certificate Authentication (TLS-CCA) [15, Sec. 4.4.3] and it has been introduced as a successor of previously used OpenEID framework (which had certain technical challenges on the web browser platform due to its design).

[12] `https://web-eid.eu`
[13] `https://ria.ee`

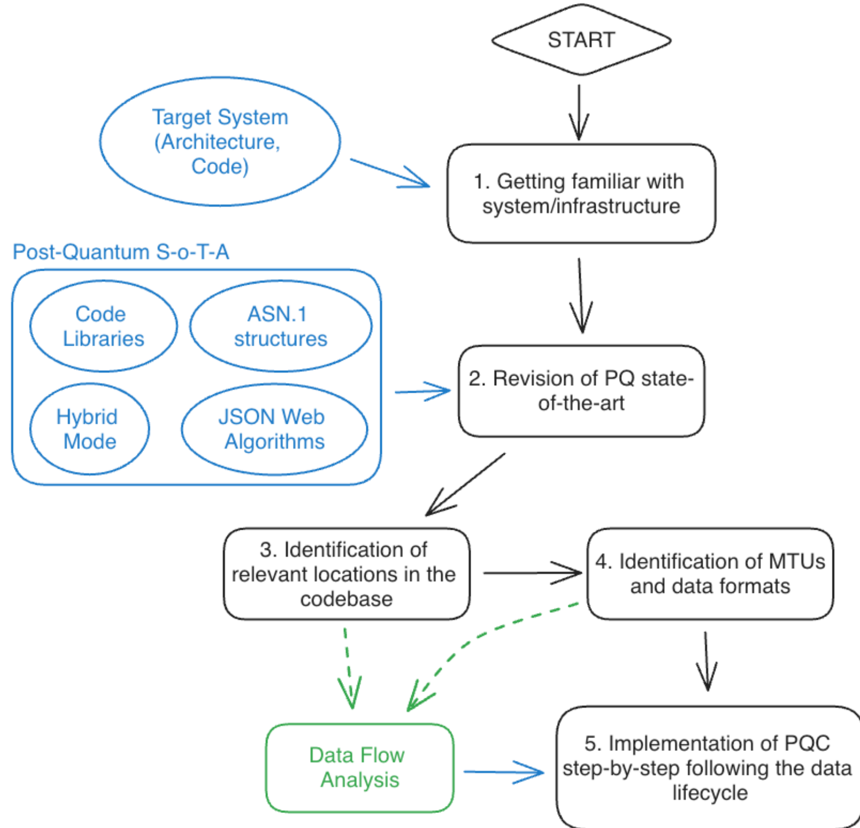

Figure 2: General PQ implementation methodology steps

Estonian citizens can use it to login into various state (e.g. taxing portal), but also private (e.g. internet banking) e-services using their national ID cards and a web browser.

The system is built upon Estonian Public Key Infrastructure, so it works with RSA and Elliptic Curve Cryptography (ECC) digital signatures. Our goal was to make it support also CRYSTALS-Dilithium and Falcon algorithms.

Our result is available at `https://github.com/Muzosh/Post-Quantum-Authentication-On-The-Web`.

Environment Setting We have decided to build this proof of concept on top of our previous work [16], where we utilized Web-eID to login users into open-source cloud service Nextcloud[14]. It consists of an instance of Nextloud running on a Docker container and a modified Web-eID version capable of working with custom-personalized smart cards. Therefore, our aim was to

1. modify an authentication device (not a smart card, more on that later) to be able to produce post-quantum digital signatures,

2. for the Web-eID framework to be able to transport and identify these signatures and incorporate them into its workflow, and

3. for the Nextcloud web application to use modified Web-eID validation library to verify these signatures, ultimately producing an authentication result.

Certification Infrastructure Authentication workflow in Web-eID is based on digitally signing certain challenges, and verifying a certificate of a public key tied to the private key which produced the signature. Therefore, we have created a post-quantum X.509 certificates using Python `asn1crypto` package, where PublicKeyInfo's algorithm field is `1.3.6.1.4.1.2.267.7.8.7` (Dilithium5 according to OpenQuantumSafe's

[14]`https://github.com/Muzosh/Smart-Card-Authentication-On-The-Web`

OIDs). Certificate data is hashed with SHA512, and the hash is then signed using our own libOQS wrapper for Python. On the other side, Web-eID validation library is modified to be able to load these certificates and verify digital signatures with them (see the next subsection for details).

Missing PQ in PHP Nextcloud is written in PHP, so we have used a Web-eID validation library written in PHP. However, there was no *libOQS* wrapper for this language, so we had to create our own using SWIG (later on, we also created similar wrappers for Python and Go, but the process is more or less the same). There is also an option to use PHP's internal extension for OpenSSL in combination with PQ-capable *OQS-OpenSSL*[15]. However, there are some limitations for algorithm identifiers, which are hard-coded.

In order to not having to implement all the logic of certificate loading and processing before verification, we chose to integrate new *libOQS* wrapper in a popular cryptography library *PHPSecLib*[16].

Embedded Device Is Not a Smart Card Multiple works in the industry have shown that post-quantum cryptography is too demanding for smart cards [17, 18, 19]. Therefore, we decided to use a different authentication device: a rather small, ESP32-S3 programmable microcontroller communicating with PCs over USB-C serial port. The development was based on the *ESP-IDF* and *PlatformIO* technology stacks, undergoing multiple stages with different experiments (mostly related to storage). In the end, it was programmed to behave very similarly to a smart card – waiting for APDU commands, processing and returning responses.

Web-eID native application knows how to communicate only with smart cards using PC/SC protocol stack (available in common operating systems). We have introduced an application-wide abstraction of an authentication device in this framework, keeping the compatibility with smart cards, but also added another module of serial devices capable of communicating via serial ports. Web-eID authors have been notified of this contribution and it can serve as a starting point for possible future implementations of other authentication devices.

Memory Management on an Embedded Device Another issue was a rather problematic memory management of post-quantum algorithms on the embedded device. ESP operating system framework allows for "tasks" (its version of processes) to have only maximum 8KB of stack memory, no matter how much memory is soldered to the device. This was nowhere near enough for *PQClean* source code to run in the main "loop" task (designated for waiting and responding to commands from PC). For security reasons, *PQClean* tries to minimize dynamic allocations, therefore it allocates all its necessary memory into the stack. To solve this issue, we had to modify the *PQClean* source-code for the device, allowing functions to dynamically allocate objects in the heap memory while utilizing safe memory management techniques (e.g. using unique pointers in C++).

Some Remaining Issues Since this project was our first experience with a post-quantum implementation, we do not view this as a final product ready to enhance the security of Estonian authentication infrastructure. First, there is no possibility to properly certify the embedded device. At this moment, there is no Secure Boot and Flash Encryption implemented. Private keys are stored in the flash memory symmetrically encrypted by a key derived from user's PIN. Also, post-quantum algorithms (Dilithium and Falcon) are hardcoded in more places than we would like. There was no room for cryptographic agility in every component due to the limitations of the platform.

3.2 CDOC 2.0

CDOC 2.0 [20] is a specification defining the process of securing, storing, and exchanging encrypted documents. It can be perceived as a file format for storing encrypted data together with intended recipients, featuring additional security measures with an optional server back-end. The current Java reference implementation[17] supports five scenarios differentiated by a specific algorithm used for encrypting all data per recipient (ECDH, RSA, and pre-shared symetric key), and whether a back-end server for key exchange is used or not.

[15] `https://github.com/Muzosh/OQS-openssl-in-PHP`
[16] `https://github.com/phpseclib/phpseclib/pull/1920`
[17] `https://github.com/open-eid/cdoc2-java-ref-impl`

We have expanded this number of scenarios by two: using CRYSTALS-Kyber-1024 with and without a key exchange server, and we will describe our experience in the following subsections.

Existing Product, New Cryptography Unlike in the previous case, where we were building an entire post-quantum authentication system with a large number of components, this application consists of only one component. Therefore, our approach has changed as well. Thanks to well defined scenarios, we were able to easily (after creating an analysis of PKI data flows) duplicate all code related to previous scenarios and make the used cryptographic library to work with post-quantum objects.

The original implementation used BouncyCastle, so a version update was required to enable post-quantum capabilities. We also learned that new BouncyCastle version can be used as a provider for Java Keytool, a popular commandline application for cryptographic object management.

Serverless Scenario We started by analysing two existing scenarios (the RSA encryption, and the ECDH key establishment), deciding that, among them, RSA is more similar to the new KEM scheme. Figures 3 and 4 depict these similarities. Note that CDOC 2.0 foresees the scenario where the same document may be encrypted for several recipients; this is depicted by using different colours in the figures.

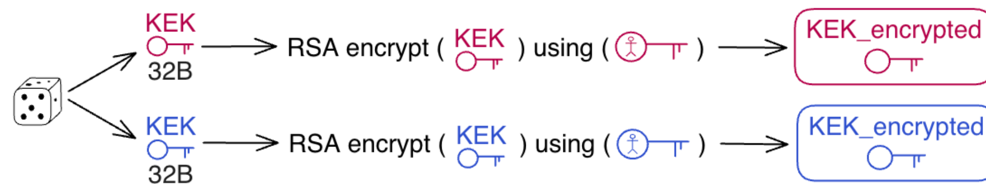

Figure 3: Encryption of Key Encryption Key in RSA serverless scenario

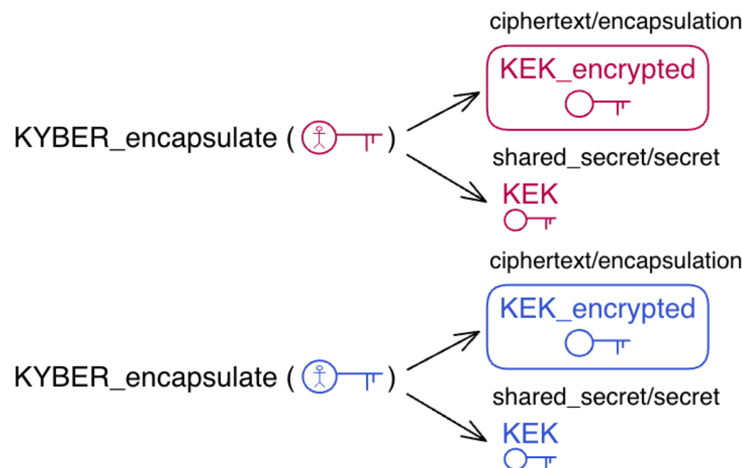

Figure 4: Encapsulation of Key Encryption Key in CRYSTALS-Kyber serverless scenario

Both approaches create a random Key Encryption Key (KEK), which is used to encrypt (using one-time-pad) the File Master Key (FMK), a key that ultimately encrypts the payload. KEK is created for each recipient individually. See Figure 5 for the overview of Recipient Capsule, an essential metadata container targeted for each recipient.

On the receiving end, the new Kyber post-quantum scenario is similar to the RSA decryption as well. See Figures 6 and 7.

Key Exchange Server Scenario Key exchange scenario disables storing the key capsules (necessary for revealing KEK, therefore FMK, therefore payload decryption) in the CDOC file header, but demands storing them on a server. The process here is similar to the previously described serverless scenario, except for one significant aspect. In order to convince the server that a recipient is eligible to obtain their key capsule, they

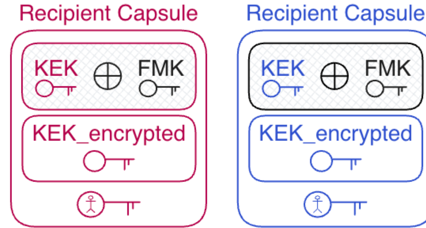

Figure 5: Recipient Capsule in transit created for each recipient

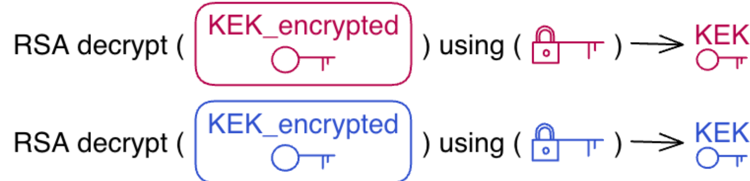

Figure 6: Decryption of Key Encryption Key in RSA serverless scenario

must first authenticate to the server using TLS client authentication with the same public key as the public key stored in the key capsule.

For Kyber scenario, this inherently means that the recipient must authenticate with a TLS certificate having Kyber public key in the `subjectPublicKeyInfo` field. Creation of these certificates was not problematic since it was similar to Web-eID. However, there are issues with such certificates at the TLS level of PQ integration.

For the purpose of releasing the key capsule, it would be enough to have a Kyber public key in the `subjectPublicKeyInfo` field and just checking at the key exchange server that this certificate is valid and trusted (i.e. signed by the certificate authority using either a classical or even a post-quantum signature scheme). However, Java SSL implementation has problems reading such certificates, thus not allowing to establish the connection. Second, the protocol currently proposes to use the TLS connection as is, so the `subjectPublicKeyInfo` field in the client certificate is actually used to decrypt the data sent by the client.

Such functionality is currently experimental – P. Schwabe, D. Stebila, and T. Wiggers proposed a PQ-KEM version of TLS in [21], and CloudFlare further elaborated with its implementation. We plan to explore this technology for this and following projects in the future.

For now, we overcame this issue by utilizing a classical certificate (i.e. ECC or RSA for both for `subjectPublicKeyInfo` as well as for signing the certificate itself), but embedding a Kyber public key into another field: `id-ce-subjectAltPublicKeyInfo (OID: 2.5.29.72)`. This allows the usage of post-quantum key exchange server scenario without significant or not-well-thought-out changes to the underlying SSL/TLS framework.

3.3 ASiC-E

Associated Signature Container Extended (ASiC-E) [22] is a standard for storing digital signatures. This standard is widely used in Estonia for sharing digitally signed containers based on the XAdES/eIDAS protocol stack. It specifies the format of the container as well as the metadata file for algorithm identification, checksums, signature values, etc.

ASiC-E formatted signatures are also used in Estonian Internet voting system [23] where the signed votes have this format. We are currently working on a PQ safe version of the voting system, including creating CRYSTALS-Dilithium signatures in ASiC-E format. We have created a fork of a Python package for ASiC-E `pyasice`[18], added a command-line interface to it and integrated our previously created Python wrapper of *libOQS*. It is able to create and verify post-quantum containers with Dilithium parameter set 3 digital signatures. There were only two issues within this project.

[18] `https://github.com/thorgate/pyasice`

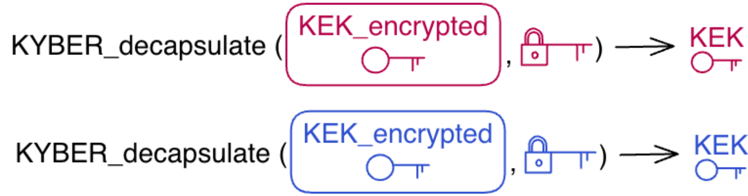

Figure 7: Decapsulation of Key Encryption Key in CRYSTALS-Kyber serverless scenario

Missing w3.org SignatureMethod Identifiers In the original protocol and package, signature identifiers were specified by URIs with `w3.org` root. For example, RSA with SHA-256 was identified with the URI `"http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"`. For our implementation, we are temporarily using `dilithium3`. This means, that official tools like *DigiDoc4 Client*[19] can open the container, but can not verify it (see Figure 8).

Signer

Notice

This is an invalid signature or malformed digitally signed file. The signature is not valid.

TECHNICAL INFORMATION

SignatureXAdES_LTA.cpp:203 Signature validation
SignatureXAdES_B.cpp:695 Failed to validate signatureValue.
Digest.cpp:144 Digest method URI 'dilithium3' is not supported.
SignatureXAdES_B.cpp:675 Unable to verify signing certificate
SignatureXAdES_B.cpp:669 Signing certificate does not contain NonRepudiation key usage flag
OCSP.cpp:211 Failed to verify OCSP response.
digital envelope routines:0 error:03000072:digital envelope routines::decode error
OCSP routines:0 error:13800082:OCSP routines::no signer key

Signature

Attribute	Value
Signer's Certificate	intermediate-cert.pq-ivxv.cyber.ee
Signer's Certificate issuer	root-cert.pq-ivxv.cyber.ee
Signature method	dilithium3
Container format	application/vnd.etsi.asic-e+zip
Signature format	EPES/time-mark
Signature policy	2.1.0
Signed file count	5
SPUri	https://www.sk.ee/repository/bdoc-spec2
Hash value of signature	30 51 30 0D 06 09 60 86 48 01 65 03 04 02
OCSP Certificate	intermediate-cert.pq-ivxv.cyber.ee
OCSP Certificate issuer	root-cert.pq-ivxv.cyber.ee
OCSP time	21.09.2023 14:54:27 +02:00
OCSP time (UTC)	21.09.2023 12:54:27 +00:00
Signing time (UTC)	21.09.2023 12:54:27 +00:00
Claimed signing time (UTC)	21.09.2023 12:54:27 +00:00

Figure 8: Verification status of PQ ASiC-e container

Automatic modification of installed modules Package `pyasice` depends on `asn1crypto` module to be able to load PQ certificates and verify them. Vanilla version of this module is pretty flexible for post-quantum usage, but there are classical algorithm OIDs hard-coded. We have used a Python Hatch framework which can include scripts to run when the package is being installed via the `pip` tool. This allowed us to automatically modify `asn1crypto` files (adding OID for Dilithium parameter set 3) during an installation of our post-quantum version of `pyasice`.

3.4 OCSP and TSA server

As mentioned above, we are currently working on a quantum-safe system for Internet voting used in Estonia. It consists of multiple research topics for post-quantum primitives and protocols (e.g. mix-nets and zero-knowledge proofs) that are very different from the primitives NIST is currently standardizing. However, there are two required services that only depend on standard public-key primitives: online certificate status protocol (OCSP) and timestamping servers. These should be migrated to PQ algorithms currently being standardized.

OCSP server Creating a PQ OCSP server was quite straightforward. We used an open-source implementation[20] which defines specific shell scripts and a Docker container with OpenSSL installed. It also has a functionality of Certificate Authority, thus allowing us to issue intermediate and client certificates.

[19] `https://github.com/open-eid/DigiDoc4-Client`
[20] `https://github.com/redislabs-training/ocsp-responder`

To make a post-quantum version of it, we added instructions to the Dockerfile to install OQS-OpenSSL and our PQ version of `pyasice`. There were no significant engineering issues during this transformation and it is working flawlessly so far.

TSA server The same can not be said about our effort to create a standards-based [24] post-quantum timestamping server, which is also required for Estonian e-voting. So far, we have failed to create a post-quantum version of it. We have put this task on hold, waiting for OpenSSL v3.3.

There are only a few open-source implementations for such a TSA. The implementation that we have focused on[21] (the only one that seemed to do exactly what we need) is written in C and it is also utilizing OpenSSL installed on the system. However, it is using OpenSSL C API, not the `openssl` command-line program. The PQ capabilities of OQS-OpenSSL are not functional in OpenSSL C API. This has been discussed with the maintainers of both OQS-OpenSSL as well as OpenSSL, and the conclusion is that the said functionality will be targeted for OpenSSL version 3.3. See `https://github.com/open-quantum-safe/oqs-provider/issues/251` for more details on this issue. While waiting for the release of this version of OpenSSL, we have prepared a Docker container with OQS-OpenSSL, where this TSA server will run. After the release, we do not expect significant difficulties in getting the TSA server to run.

Acknowledgments

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. Also funded by Estonian Research Council, grants no. PRG1780 and PRG2177.

References

[1] Frederico Schardong et al. "Post-Quantum Electronic Identity: Adapting OpenID Connect and OAuth 2.0 to the Post-Quantum Era". In: *Cryptology and Network Security*. Springer International Publishing, 2022, pp. 371–390. DOI: 10.1007/978-3-031-20974-1_20.

[2] Paula López-González et al. "A facial authentication system using post-quantum-secure data generated on mobile devices". In: *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*. ACM, Oct. 2022. DOI: 10.1145/3495243.3558761.

[3] Jiewen Yao, Krystian Matusiewicz, and Vincent Zimmer. *Post Quantum Design in SPDM for Device Authentication and Key Establishment*. Cryptology ePrint Archive, Paper 2022/1049. 2022. DOI: 10.3390/cryptography6040048. URL: https:/eprint.iacr.org/2022/1049.pdf.

[4] Sebastian Paul, Patrik Scheible, and Friedrich Wiemer. *Towards Post-Quantum Security for Cyber-Physical Systems: Integrating PQC into Industrial M2M Communication*. Cryptology ePrint Archive, Paper 2021/1563. 2021. DOI: 10.3233/JCS-210037. URL: https:/eprint.iacr.org/2021/1563.pdf.

[5] Joppe W. Bos et al. "Post-Quantum Key Exchange for the TLS Protocol from the Ring Learning with Errors Problem". In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 553–570. DOI: 10.1109/SP.2015.40.

[6] Peter Schwabe, Douglas Stebila, and Thom Wiggers. "Post-Quantum TLS Without Handshake Signatures". In: *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. Ed. by Jay Ligatti et al. ACM, 2020, pp. 1461–1480. DOI: 10.1145/3372297.3423350.

[7] *The PQC Migration Handbook*. Applied Cryptography and Quantum Algorithms Cryptology Group Netherlands National Communications Security Agency. 2023. URL: https://english.aivd.nl/publications/publications/2023/04/04/the-pqc-migration-handbook.

[21] https://github.com/kakwa/uts-server

[8] Petr Muzikant and Jan Willemson. "Deploying Post-Quantum Algorithms in Existing Applications and Embedded Devices". In: UbiSec 2023, https://research.cyber.ee/~janwil/publ/pqauth.pdf.

[9] M. Jones. *JSON Web Algorithms (JWA)*. May 2015. DOI: 10.17487/rfc7518.

[10] Sajimon P C, Kurunandan Jain, and Prabhakar Krishnan. "Analysis of Post-Quantum Cryptography for Internet of Things". In: *2022 6th International Conference on Intelligent Computing and Control Systems (ICICCS)*. IEEE, May 2022. DOI: 10.1109/iciccs53718.2022.9787987.

[11] Ruben Gonzalez and Thom Wiggers. "KEMTLS vs. Post-quantum TLS: Performance on Embedded Systems". In: *Security, Privacy, and Applied Cryptography Engineering*. Springer Nature Switzerland, 2022, pp. 99–117. DOI: 10.1007/978-3-031-22829-2_6.

[12] George Tasopoulos et al. "Performance Evaluation of Post-Quantum TLS 1.3 on Resource-Constrained Embedded Systems". In: *Information Security Practice and Experience*. Springer International Publishing, 2022, pp. 432–451. DOI: 10.1007/978-3-031-21280-2_24.

[13] Manohar Raavi et al. "QUIC Protocol with Post-quantum Authentication". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2022, pp. 84–91. DOI: 10.1007/978-3-031-22390-7_6.

[14] Ruben Gonzalez et al. "Verifying Post-Quantum Signatures in 8 kB of RAM". In: *Post-Quantum Cryptography*. Springer International Publishing, 2021, pp. 215–233. DOI: 10.1007/978-3-030-81293-5_12.

[15] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/info/rfc8446.

[16] Petr Muzikant. "Cloud Service Access Control using Smart Cards". MA thesis. Brno University of Technology, 2022.

[17] Aurélien Greuet. *Smartcard and Post-Quantum Crypto*. 2021. URL: https://csrc.nist.gov/Presentations/2021/smartcard-and-post-quantum-crypto.

[18] Luk Bettale, Marco De Oliveira, and Emmanuelle Dottax. "Post-Quantum Protocols for Banking Applications". In: *Smart Card Research and Advanced Applications*. Springer International Publishing, 2023, pp. 271–289. DOI: 10.1007/978-3-031-25319-5_14.

[19] Lukas Malina et al. "Towards Practical Deployment of Post-quantum Cryptography on Constrained Platforms and Hardware-Accelerated Platforms". In: *Innovative Security Solutions for Information Technology and Communications*. Springer International Publishing, 2020, pp. 109–124. DOI: 10.1007/978-3-030-41025-4_8.

[20] *CDOC 2.0 spetsifikatsioon, v0.9*. Tech. rep. D-19-12. Cybernetica AS, 2023.

[21] Peter Schwabe, Douglas Stebila, and Thom Wiggers. *Post-quantum TLS without handshake signatures*. Cryptology ePrint Archive, Paper 2020/534. https://eprint.iacr.org/2020/534. 2020. DOI: 10.1145/3372297.3423350. URL: https:/eprint.iacr.org/2020/534.pdf.

[22] *ETSI EN 319 162-1. Electronic Signatures and Infrastructures (ESI); Associated Signature Containers (ASiC); Part 1: Building blocks and ASiC baseline containers*. Version 1.1.1. Apr. 2016. URL: https://www.etsi.org/deliver/etsi_en/319100_319199/31914201/01.01.01_60/en_31914201v010101p.pdf.

[23] Sven Heiberg et al. "Improving the Verifiability of the Estonian Internet Voting Scheme". In: *Electronic Voting - First International Joint Conference, E-Vote-ID 2016, Bregenz, Austria, October 18-21, 2016, Proceedings*. Ed. by Robert Krimmer et al. Vol. 10141. Lecture Notes in Computer Science. Springer, 2016, pp. 92–107. DOI: 10.1007/978-3-319-52240-1_6.

[24] Robert Zuccherato et al. *Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)*. RFC 3161. Aug. 2001. DOI: 10.17487/RFC3161. URL: https://www.rfc-editor.org/info/rfc3161.