Deploying Post-Quantum Algorithms in Existing Applications and Embedded Devices

Petr Muzikant and Jan Willemson[0000−0002−6290−2099]

Cybernetica AS, Mäealuse 2/1, Tallinn 12618, Estonia,
firstname.lastname@cyber.ee

Abstract. This document studies the current state of post-quantum cryptography implementation feasibility, providing general approaches that developers and security engineers can utilize to start integrating today. First, we analyze the current state of the art in the field of available cryptographic libraries and standards for algorithm interpretations and encodings. Then, we provide few implementation challenges that rose from our experiments and how to handle them. Lastly, we have built a proof-of-concept implementation by creating a post-quantum version of a modern web authentication framework. Our work introduces post-quantum support in multiple open-source libraries that together enable web-service administrators to authenticate their users with Dilithium-5 or Falcon-1024 secured electronic identities. Among other components, our proof-of-concept also includes a client side solution for key management using programmable embedded device.

Keywords: Post-Quantum Cryptography Integration; Authentication; Embedded Development; Post-Quantum Implementation; Dilithium; Falcon

1 Introduction

The basic ideas behind quantum computers were laid out already in 1980s [1]. Peter Shor's seminal paper from 1994 showed how the principles of quantum computing can be applied to solve classically hard computational problems, which in turn can lead to breaking of the currently standardized asymmetric cryptographic algorithms [2].

Even though quantum computers sufficiently powerful for breaking, say, 2048-bit RSA or 256-bit elliptic curve algorithms are not yet available, the cryptographic community has been working on their post-quantum (PQ) alternatives for more than a decade. This process was formalized by the National Institute of Standards and Technology (NIST) from United States that made a public call in 2016 to obtain candidates for post-quantum key establishment mechanisms (KEM) and digital signatures. In 2022, NIST selected the first four algorithms (one KEM and three signatures) to be standardized [3].

However, standardization is only the first step of the long process of actual deployment in real-life information systems. A major challenge in this process

is rolling out the support for post-quantum algorithms in all the layers of the communication protocols, starting from the client devices and ending with the back-ends of the services.

This is the problem setting where our current paper draws its inspiration from. We decided to focus on engineering aspects of post-quantum protocol implementations and build a complete proof-of-concept infrastructure supporting post-quantum algorithms in all the components. As real off-the-shelf cryptographic hardware providing post-quantum primitives is not yet available, we also built an end user device allowing to generate and apply post-quantum keys for authentication protocols. The paper presents the architecture of our solution together with the challenges we faced and solutions we propose to them.

2 Background

Several works have implemented post-quantum (PQ) algorithms into authentication frameworks. Schardong, Giron, Müller, and Custódio [4] created a PQ version of *OpenID Connect*. López-González, Arjona, Román, and Baturone [5] developed a PQ-safe biometric authentication framework. Yao, Matusiewicz, and Zimmer [6] introduced PQ into Security Protocol and Data Model compliant device authentication. Lastly, Paul, Scheible, and Wiemer [7] discussed PQ usage in banking protocols. Our work takes a more developer-friendly approach, providing a general method for implementing PQ algorithms into existing applications. We demonstrate this by creating a PQ Web Authentication Infrastructure with an embedded device for client-side key management, but at the same time by sharing generalized remarks applicable elsewhere.

2.1 Post-Quantum Algorithm Libraries

Several post-quantum cryptographic libraries are available for use in applications.

PQClean[1] is a C library that aggregates NIST-submitted algorithms with a unified Application Programming Interface (API). It enables easy integration of a single PQ algorithm into existing applications, with minimal effort required to add more later. Developers are encouraged to copy the code, adjust common libraries, and compile it.

For developers who prefer a compiled, higher-level library, *libOQS*[2] (written in C) is a great option. It features code from *PQClean* as well as other sources, and offers language wrappers for C++, Python, Java, Go, .NET, and Rust. There are also high-level libraries built on *libOQS*, such as *OQS-OpenSSL*, *OQS-OpenSSH*, and *OQS-OpenVPN*.[3]

[1] See `https://github.com/PQClean/PQClean`.
[2] See `https://github.com/open-quantum-safe/liboqs`.
[3] A full listing is available at `https://openquantumsafe.org/applications/`.

Other available post-quantum cryptographic libraries include *libpqcrypto*[4], *rustpq/pqcrypto*[5], and *pqm4*[6].

2.2 Post-Quantum ASN.1 Structures

Abstract Syntax Notation One (ASN.1) structures for post-quantum cryptographic objects are essential for successful integration into existing applications, particularly in X.509 certificate usage. Yet, no standards currently exist. NIST's submission rules require authors of post-quantum algorithms to encode their algorithm-specific structures of keys, signatures, and ciphertexts into a single byte string [8].

Multiple Request for Comments (RFC) drafts propose ASN.1 structures for most objects from currently selected-to-be-standardized PQ algorithms.[7] The current version of *OQS-OpenSSL* works with arbitrary Object Identifiers (OIDs) from the OpenQuantumSafe organization for different algorithms.[8]

It is also possible to use `subjectPublicKeyInfo` with `algorithmIdentifier` specified from the list of OIDs and `subjectPublicKey` as PQ digital signature in byte string format. Snetkov and Vakarjuk [9] suggest using `subjectAltPublicKeyInfo`, `altSignatureAlgorithm`, and `altSignatureValue` attributes to maintain classical cryptography functional.

2.3 Post-Quantum JSON Web Algorithms

JSON Web Algorithm (JWA) is an RFC [10] that registers cryptographic algorithms and identifiers for JSON Web Signature and JSON Web Encryption, which are widely used for securely transferring data over the network. For example, `ES384` stands for ECDSA digital signature algorithm using P-384 and SHA-384 hash algorithm. At the time of writing this article, there are no RFC drafts for post-quantum JWAs.

However, there are drafts[9] for JSON Web Key, which specifies only the digital signature algorithm part and does not provide a hash function. For example, `CRYDI5` means CRYSTALS-Dilithium algorithm parameter on the 5th security level.

To use post-quantum algorithms in JWAs, a constant specified hash function can be used (e.g., SHA-512, which is recommended by NIST as quantum-safe). When implementing post-quantum digital signatures in existing applications that use JWAs, `CRYDI5` could be handled as CRYSTALS-Dilithium-5 + SHA-512.

[4] See `https://libpqcrypto.org/index.html`.
[5] See `https://github.com/rustpq/pqcrypto`.
[6] See `https://github.com/mupq/pqm4`.
[7] See `https://github.com/open-quantum-safe/oqs-provider/issues/89`.
[8] See `https://github.com/open-quantum-safe/oqs-provider/blob/main/ALGORITHMS.md#oids`.
[9] See `https://www.ietf.org/archive/id/draft-prorock-cose-post-quantum-signatures-01.html`.

2.4 Hybrid Mode

Hybrid mode combines classical cryptography with post-quantum cryptography. While this approach incurs more performance, memory, and storage overhead, it also eliminates potential threats from both classical and quantum world. Post-quantum algorithms ensure the longevity of data confidentiality, while classical cryptography guards against potential emerging threats on unexplored PQ cryptography.

Several RFC drafts propose ASN.1 structures for hybrid modes.[10] However, the question remains as to how these two types of algorithms should be coupled together.

A novel method is proposed by Ghinea *et al.* [11] to improve unforgeability. It involves prepending labels to signed messages, as opposed to concatenating classical and post-quantum signatures, or sequentially using the digital signature given with one algorithm as an input for another signature algorithm.

3 Post-Quantum Implementation Challenges

Implementing post-quantum algorithms in existing applications can be difficult due to the continual progress of and uncertainty in post-quantum technology. This may lead to numerous engineering issues for developers and cybersecurity engineers when trying to make their systems quantum-resistant.

In this section we will provide general ideas to aid implementation of post-quantum algorithms into existing applications.

3.1 Identifying relevant locations

Identifying locations in the codebase, network and business processes where public key infrastructure (PKI) objects are used is the first step in implementing post-quantum algorithms in an existing application. Tracing the data flow from start to end of the object's lifetime is necessary to identify different approaches towards non-PQ and PQ objects, as there are differences in how other processes interact with them. Identifying these data flows helps in understanding the extent of changes necessary for implementing post-quantum algorithms in an application.

When transferring cryptographic data objects between different parts of the architecture, it is important to consider the Maximum Transmission Unit (MTU) to ensure longer keys and signatures fit into containers. Additionally, caution is required when implementing the Falcon PQ digital signature algorithm as its output length may vary.

[10] See `https://datatracker.ietf.org/doc/draft-ounsworth-pq-composite-encryption/01/`, `https://datatracker.ietf.org/doc/draft-ounsworth-pq-composite-keys/`, and `https://datatracker.ietf.org/doc/draft-truskovsky-lamps-pq-hybrid-x509/01/`.

Data formats and possible conversions between them during the object's lifetime also require detailed consideration. For example, cryptographic libraries might yield raw bytes, but the rest of the system handles data transfer in ASN.1, Base64, or PEM encoded formats. Identifying the format of existing data flows is the key to ensuring compatibility across the system architecture and detecting any areas that require modification.

A format suitable for this purpose is a Business Process Model and Notation (BPMN) diagram. An example of a BPMN diagram from our implementation is shown in Figure 2.

3.2 Technological and computational 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 key-pairs, creating signatures, and verifying them. In real-time applications, performance may be impacted. Further measurements can be found in [12], [13], [14], [15], section 5, and our source code.

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 other limited devices. In other cases, actual post-quantum algorithms need to be adjusted. For example, Gonzalez *et al.* [16] proposes streaming public keys and signatures into the limited memory of an attached HSM component. Another work by Gonzales and Wiggers [13] suggests using key encapsulation instead of digital signatures to reduce computational overhead.

In section 4, we will also provide an example of how to overcome constraint problems by switching from smart cards to embedded programmable microcontrollers, and how we adjusted the algorithm to allocate its objects on the heap, instead of stack memory.

3.3 Implementing PQ algorithms in the codebase

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

We refer to 2.1 for existing extensions (e.g. *OpenSSL* or popular programming languages). If none are suitable, *SWIG*[11] can be used to generate wrappers from C implementations of PQ algorithms (see section 4.4). New wrapper can be used directly, or as an extension to the used cryptographic library.

[11] See `https://swig.org/`.

4 Practical results

We present a proof-of-concept implementation for a complete authentication infrastructure with post-quantum cryptography.[12] We set up a usable, modern, cross-platform and open-source combination of components to authenticate users to a web application using post-quantum digital signature algorithms (Dilithium-5 and Falcon-1024). We describe the implementation process and present the engineering problems we met together with the solutions we propose.

4.1 Architecture

To create a proof-of-concept PQ-enabled authentication system, we needed to select three key components of the system – *back-end server with a web service*, *authentication framework*, and *authentication device*.

1. We chose open-source cloud storage Nextcloud[13] as our *back-end server with a web service*. It serves as an electronic identity verifier, authenticating users and granting them access to the web application.
2. We chose the open-source *authentication framework* Web-eID, developed by the Estonian Information Authority, as a successor of Open-EID[14]. Both are cross-platform solutions used for authenticating citizens on the web using national electronic ID cards. For more information, see section 4.2.
3. We chose an ESP32 microcontroller as our *authentication device*, which holds the information needed to establish electronic identity (e.g., a private key for signing challenge nonces). This was due to the post-quantum constraints discussed in section 3.2, which prevented us from using smart cards. Further details can be found in section 4.3.

These three top-level components are supported by multiple lower-level libraries/repositories that needed to be adjusted and contributed to as well. See section 4.4 for more information. Figure 1 provides an overview of all components and their relations.

4.2 Authentication Data Flow

Web-eID is a suite of applications, extensions, and tools that enable authentication and digital-signing with public-key cryptography on the web, similar to Transport Layer Security: Client Certificate Authentication (TLS-CCA). For more information, see chapter 2.4 in [17] or the official website[15]. We present our analysis of the PKI object data lifetime in Figure 2. It displays all the relevant components that we needed to focus on when implementing support for post-quantum algorithms.

[12] Repository index with source code and additional information can be found at `https://github.com/Muzosh/Post-Quantum-Authentication-On-The-Web`.
[13] See `https://nextcloud.com/`.
[14] See `https://github.com/open-eid`.
[15] See `https://web-eid.eu/`.

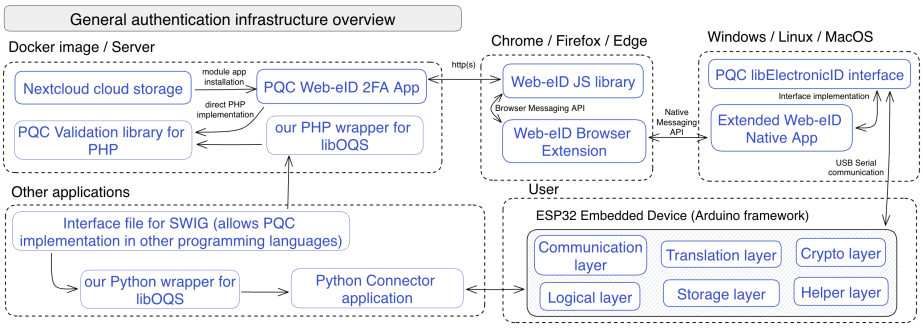

Fig. 1: PQC authentication infrastructure components overview

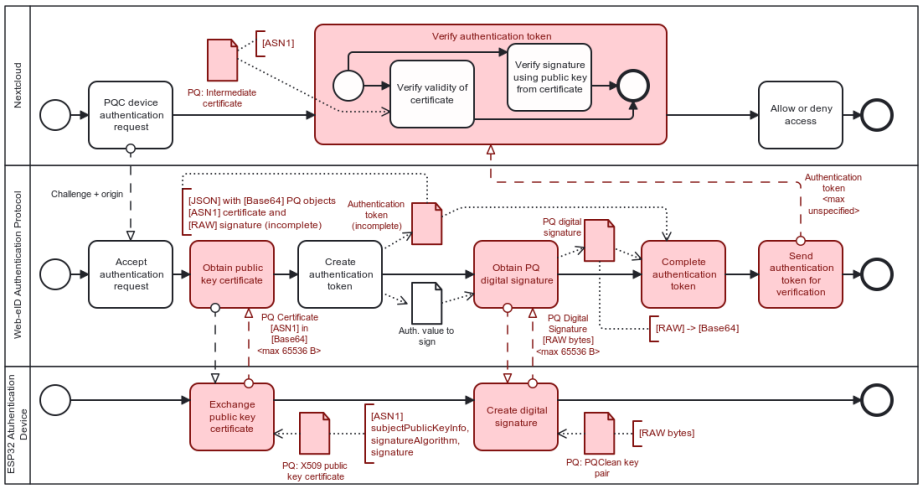

Fig. 2: Post-quantum data flow analysis

4.3 Authentication Device

Web-eID supports only smart cards using the Personal Computer/Smart Card (PC/SC) protocol stack. However, current cards lack the RAM and CPU capabilities to perform post-quantum digital signatures [18]. In [19], authors suggest using KEM algorithms to implement quantum-resistant banking protocols, but using KEM in our work would have required significant changes to the Web-eID authentication protocol. Lastly, [20] discusses implementations of PQ cryptography on constrained platforms, including smart cards. The authors conclude that there are no algorithms that can be run on off-the-shelf programmable smart cards, and one must use more powerful platforms.

Therefore, we decided to use an ESP32 system-on-chip microcontroller device as our authentication device, as they are sufficiently powerful and widely available low-cost products. We used two devices: *DFRobot FireBeetle 2* and the more

powerful *LilyGO T-Display-S3* (for more technical details and benchmarks, see section 5.2).

To make such a device Web-eID compatible, we developed a full-fledged firmware for the embedded device to imitate the functionality of the Estonian ID card. It receives custom APDU commands and sends the appropriate APDU responses after proper user PIN authorization. We used PlatformIO[16] with Arduino-ESP32 framework to enable Arduino features on the ESP32 platform.

Our authentication device does not use the PC/SC protocol stack to send data to the PC, but USB serial communication instead. To make this possible, we contributed with an application-wide abstraction layer in the Web-eID application to include USB serial devices. This layer allows developers to create custom serial devices and integrate them with modern authentication protocols.

The following list contains a subset of notable considerations and problems when developing an application for an ESP32 embedded device.

- *Device driver* – in this work, we consider using an authentication device connected to the PC via a USB port. Therefore, a USB-to-serial driver is required for the PC to recognize the device when connected. For *FireBeetle*, we used an installable *CH34X driver*. For *T3-Display-S3*, a device capability *USB CDC* (USB Communications Device Class), which is a common and recognizable interface, was used. The latter option has the benefit of being recognized in all current operating systems without needing to install any software in advance. However, there is a slight delay of 1-2 seconds after inserting the device.
- *Storage options* – to manage PIN states and generated key pairs, persistent file storage is essential. Initially, we opted for *LittleFS*, an open-source filesystem designed for embedded devices. Further investigation revealed that the ESP32 framework offers multiple storage API options for flash memory [21]. We decided to use *NVS*, a key-value flash storage with a maximum of 508KB data limitation, due to its faster write and read speeds, thus shortening authentication time.
- *Debugging* – to debug an ESP32 microcontroller, an external debugging probe must be connected to the device and PC. An alternative option is to print debug data to the serial buffer, but this interferes with the authentication protocol as it shares the serial buffer (PC expects pre-determined number of bytes in responses). To address this, we used a separate NVS namespace for logging and saving debug messages to the flash memory. After an operation, logs can be read from the serial buffer (and erased) on demand.
- *Serial communication on PC* – we initially used the *Boost::Asio*[17] C++ library to communicate with the PC device. However, to simplify the process (mainly using timeout and listing available ports features) and to reduce

[16] See `https://platformio.org/`.
[17] See `https://www.boost.org/doc/libs/1_82_0/doc/html/boost_asio.html`.

dependencies, we transitioned to the *QtSerialPort*[18] library, which is part of the Qt framework. For the Python application, we used *pyserial*[19].
- *Buffer size* – in the Arduino framework, `Serial.setRxBufferSize()` must be called before `Serial.begin()`, not after.
- *USB MODE* – we learned that in order for the device to work properly after reconnection during authentication,[20] `ARDUINO_USB_CDC_ON_BOOT` build flag must be on, and `ARDUINO_USB_MODE` must be off at the same time.

4.4 Post-Quantum Implementations

In this section, we will provide details of our experience implementing post-quantum cryptography throughout the entire architecture, including low-level components.

General Remarks For implementing key pair generation, digital signature, or verification with post-quantum algorithms, we used *PQClean* for the embedded device and *libOQS* for the rest of the codebase (see section 2.1 for their descriptions). If an application already uses a cryptographic library such as OpenSSL, one approach is to search for a post-quantum version of it (or contribute one as open-source) – e.g. *OQS-OpenSSL* can be installed using the *libOQS* library. Alternatively, the application logic can be split into two branches: if classical algorithms are required, use the existing library; if post-quantum algorithms are needed, use raw implementation libraries like *libOQS*. We used both approaches in our authentication infrastructure.

PQ on Embedded Device The biggest challenge when implementing post-quantum algorithms on embedded devices is memory management. As mentioned before, *PQClean* is the most suitable library in this situation, but it still needs to be adjusted.

On ESP32 platforms, all operations are managed by FreeRTOS [22], an operating kernel system for embedded devices. Therefore, even the `loop()` function runs as a task, with a predetermined 8KB stack in RAM. As stated in [16] and tested by us, 8KB is not sufficient for post-quantum algorithms.

To address this issue, we created a new FreeRTOS task with a larger stack allocation. Its purpose is to call a specified PQ function from the *PQClean* API and return the result. However, this approach is not consistent across multiple devices, as the upper memory limit for the stack allocation is not equal to the total free memory available (due to ESP32 having memory allocations for different purposes [23]). We also occasionally encountered errors stating that the task could not be created, even after checking for free memory.

[18] See `https://doc.qt.io/qt-6/qtserialport-index.html`.
[19] See `https://github.com/pyserial/pyserial`.
[20] More information about the issue can be found at `https://esp32.com/viewtopic.php?f=19&t=33762`

Therefore, we allocated only 32MB of stack for these tasks and minimized memory allocation to the stack in the *PQClean* implementation, moving it to the heap memory. We changed large declarations in functions to use `malloc` and `free` functions. For even safer memory management, we chose to rewrite this code from C to C++ and use `std::unique_ptr` objects, which handle other objects' memory allocation for their lifetime.

Additionally, we converted all arrays to pointers in function signatures to prevent them from being copied to the stack when the function is called. This guarantees that every post-quantum operation has 32MB of memory for stack allocations and all algorithm-related objects are allocated dynamically on the heap, which prevents runtime errors.

PQ in Transit Before implementing post-quantum algorithms on either side of the architecture, we checked that the increased (and variable in the case of Falcon) size of post-quantum data (such as public keys, digital signatures or certificates) would not break communication by sending mock data.

Web-eID transmits a custom authentication token in JSON format, consisting of a DER-encoded unverified public key certificate and a raw signature, both Base64 encoded. It uses multiple APIs, like the Browser Messaging, Native Messaging and HTTP API, to transfer the token between the components. These APIs have no significant size restrictions for Dilithium-5 or Falcon-1024 objects, so there was no need to make any codebase changes.

Algorithm identifiers are encoded in the ASN.1 structure of the unverified certificate and as a JSON Web Algorithm in the `algorithm` field of the Web-eID authentication token. During device personalization (generating key pairs, obtaining public keys, and creating client certificates), unofficial, *libOQS*-compatible OIDs (e.g. `1.3.6.1.4.1.2.267.7.8.7` for Dilithium-5) and drafted JSON Web Key identifiers (e.g. `CRYDI5` for Dilithium-5 + constant SHA512 for the hash function) are used.

PQ in Existing Architecture In this section, we discuss the implementation of post-quantum cryptographic algorithms in the rest of our authentication architecture. We made changes to the following components.

- *Nextcloud Web-eID 2FA* – an installable Nextcloud application that allows using Web-eID authentication result as the second factor.[21] A switch to the PQC Web-eID AuthToken Validation Library for PHP was required (see below).
- *PQC Web-eID AuthToken Validation Library for PHP* – pre-quantum version of this library used both *OpenSSL* and *PHPSecLib*, so we adjusted both:
  - *OQS-OpenSSL* – the OpenQuantumSafe organization provides an *OpenSSL@1.1* fork and *OpenSSL@3* extension capable of post-quantum algorithms.[22] However, the current version of internal OpenSSL PHP

[21] See `https://github.com/Muzosh/nextcloud_twofactor_webeid`.
[22] See `https://openquantumsafe.org/applications/tls.html`.

extension only supports functions that do not require algorithm identification (e.g. `openssl_verify`, `openssl_sign`, not `openssl_pkey_new`). The reason for this is that DSA, DH, RSA and EC algorithms are hard-coded, and there is no way of specifying post-quantum algorithm identifiers from PHP source code.[23]
    - *PQC-PHPSecLib* – we provide an open-source contribution in the form of preparation for post-quantum algorithms with *Dilithium-5* reference implementation. This uses either *OQS-OpenSSL* or our new PHP extension of C++ *libOQS* library created with *SWIG*. Before any function is run, `ASN1::loadOIDs` must be called with all new post-quantum algorithm object identifiers.
- *liboqs-php* and *liboqs-python* – OpenQuantumSafe offers several language wrappers for its *libOQS* library, but does not include a PHP wrapper. We created one using *SWIG*, an application that requires C++ interface for generating wrappers. We used this wrapper in our *PQC-PHPSecLib*. We also created a Python wrapper for our device personalization application (for creating post-quantum certificates and testing the device). *SWIG*-specific remapping was needed to transform PHP `string` and Python `bytearray` into C++ `uint8_t*` and vice versa.
- *PQC-Web-eID Application* – apart from the already mentioned new application-wide abstraction layer, an *ElectronicID* interface (required for each supported card/device) was implemented for our new authentication device. No post-quantum functions were required in this part, and data transit was described in section 4.4.
- *Device Connector* – this console application allows administrators to initialize and issue our authentication devices, test Web-eID compatibility and post-quantum capabilities. It uses the *liboqs-python* library as it is written in Python.

5 Benchmarks

In this section, we present the results from our measurements and explain some of our choices based on these results.

5.1 Chosen Algorithms

Web-eID authentication protocol utilizes digital signature schemes, so we chose three post-quantum signature algorithms that NIST selected as finalists in 2022: *Dilithium*, *Falcon* and *Sphincs+* [24]. Out of these, we have chosen parameters that provide NIST security level 5.[24] In case of *Sphincs+* (which has multiple variants of level 5) we chose the most performant one.

[23] More about this issue at `https://github.com/open-quantum-safe/openssl/issues/433`.

[24] See `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization/Evaluation-Criteria/Security-(Evaluation-Criteria)`.

We measured the durations of key pair generation, digital signature creation, and digital signature verification of the post-quantum algorithms in the *libOQS* library on a dockerized version of Debian on an Apple M2 Pro chip. Tab. 1 shows the relevant object sizes for the three selected algorithms, and Tab. 2 displays the measurement results (averages from 30 retries).

Table 1: Object sizes of three selected algorithms

Algorithm	Public key size [B]	Private key size [B]	Signature size [B]
Dilithium-5	2952	4864	4595
Falcon-1024	1793	2305	1280
SPHINCS+-SHAKE256-256f-s	64	128	49856

Table 2: Performance measurement of three selected algorithms with PHPv8.1 interpreter in dockerized Debian on M2 Pro chip

Algorithm	Keypair gen. time [ms]	Sign. gen. time [ms]	Verif. time [ms]
Dilithium-5	1.314×10^{-7}	2.366×10^{-7}	1.134×10^{-7}
Falcon-1024	3.392×10^{-5}	7.658×10^{-6}	6.440×10^{-8}
SPHINCS+-SHAKE256-256f-s	4.926×10^{-6}	1.059×10^{-4}	2.541×10^{-6}

Sphincs+ is not suitable for embedded devices due to its large signatures. *Falcon* has smaller object sizes than *Dilithium*, but its signature operation takes 32 times longer. Thus, *Dilithium-5* was chosen as most suitable algorithm for our authentication infrastructure.

5.2 Digital Signature Creation Duration

Fig. 3 illustrates the evolution of on-device signature creation during the development phase (the values shown are averages from 100 retries). Initially, we used the *DFRobot FireBeetle 2* device with an ESP32-E chip, 520KB of SRAM, and 32MB of storage without encrypting the content in the flash memory (blue column). To address the lack of Hardware Security module/Trusted Platform Module, we introduced data encryption (using quantum-safe AES256-OCB), with the symmetric key derived from the user PIN (green column, resulting in a longer duration). After evaluating storage options (discussed in section 4.3), we moved some (grey column, leading to a significant decrease) or all (yellow column, slight decrease) stored data to NVS storage.

Switching to the *LilyGO T-Display-S3* device with an ESP32-S3 chip, 8MB of PSRAM, and 16MB of storage (red column), significantly decreased the digital signature creation time to 0.196 seconds. The reason is mostly higher computational power of the new ESP32 chip generation. For comparison, a *JavaCard*

SLJ52GCA150 (jTOP SLE78 Estonian ID card platform) achieved an ECDSA over P-382 elliptic curve signature creation in 0.262 seconds [17]. We also implemented the *Falcon-1024* algorithm (purple column) for demonstration purposes, which resulted in a significant increase in duration.

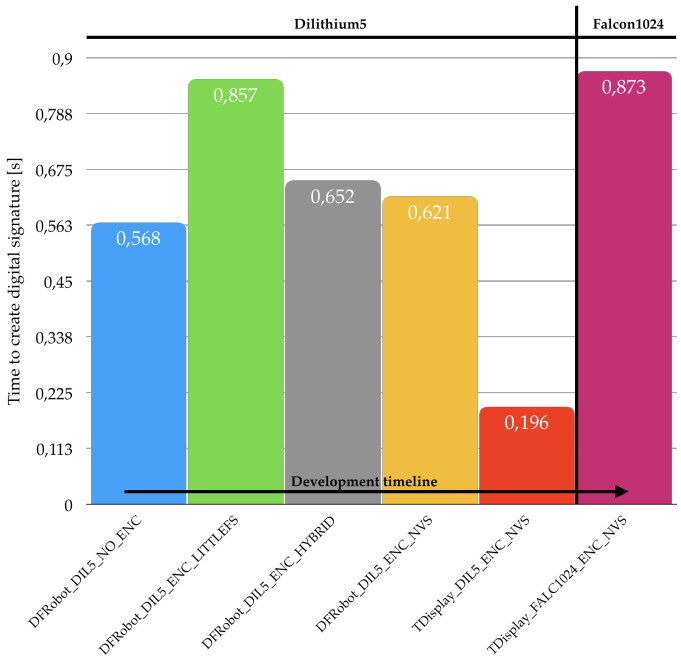

Fig. 3: Digital signature benchmark on ESP32 devices

5.3 User Experience Considerations

Our authentication infrastructure is in the proof-of-concept phase, so we can not provide exact evaluation of the whole authentication process. In our testing scenario, with a Nextcloud server running in a dockerized Debian on an Apple M2 Pro chip and a post-quantum enabled Web-eID, the user has to go through two more clicks and one PIN input. This is a lot more noticeable than the cryptographic operations running in the background. For a detailed view of the process from the administrator and user point of view, refer to chapters 4.1 and 4.2 in [17], which describe a similar authentication process with a smart card.

The USB CDC interface introduces a 1-2 second delay after device insertion. This delay is only noticeable during authentication if the device is inserted during the process. If inserted beforehand, the device is initialized in the background and ready when authentication starts.

6 Conclusions and Further Work

Post-quantum cryptography implementations impose several engineering problems, such as longer key/signature sizes and higher computational requirements. In this article, we discussed these problems, their possible solutions, and provided an example reference implementation in the form of post-quantum-enabled, modern, and open-source authentication infrastructure . We also developed an embedded client authentication device (with custom firmware and management console) and example server-side Nextcloud cloud storage using two-factor authentication.

Our work demonstrates the feasibility of integrating post-quantum cryptography into existing authentication infrastructures and provides a starting point for those interested in exploring the practical implications of post-quantum cryptography today.

For future work, we plan to publish post-quantum extensions of the used public repositories, such as Web-eID parts, *PHPSecLib*, and *liboqs-php* libraries. This step will probably require some development in the ASN.1 structures standardization. We would also like to create one Docker implementation of the whole infrastructure for easier installation and testing, and optimize the authentication device codebase for even faster results.

We also consider changing the hardware to a microcontroller with an integrated trusted execution environment and hardware secure module, so we do not have to encrypt stored data ourselves. Lastly, we plan to perform a full security analysis of this infrastructure and compare it with existing analyses.

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.

References

[1] John Preskill. "Quantum computing 40 years later". In: *arXiv preprint arXiv:2106.10522* (2021). URL: https://arxiv.org/abs/2106.10522.

[2] Peter W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. IEEE Computer Society, 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700. URL: https://doi.org/10.1109/SFCS.1994.365700.

[3] *Post-Quantum Cryptography*. National Institute of Standards and Technology. 2023. URL: https://csrc.nist.gov/projects/post-quantum-cryptography.

[4] 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. URL: https://doi.org/10.1007%2F978-3-031-20974-1_20.

[5] 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. URL: https://doi.org/10.1145%2F3495243.3558761.

[6] 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.

[7] 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.

[8] Douglas Stebila. *Key Format*. 2019. URL: https://github.com/open-quantum-safe/liboqs/issues/507.

[9] Nikita Snetkov and Jelizaveta Vakarjuk. *Integrating post-quantum cryptography to UXP*. Tech. rep. D-2-499. Cybernetica AS, 2022. URL: https://cyber.ee/uploads/PQC_UXP_report_8c97d91552.pdf.

[10] M. Jones. *JSON Web Algorithms (JWA)*. Tech. rep. May 2015. DOI: 10.17487/rfc7518. URL: https://doi.org/10.17487%2Frfc7518.

[11] Diana Ghinea et al. *Hybrid Post-Quantum Signatures in Hardware Security Keys*. Cryptology ePrint Archive, Paper 2022/1225. 2022. URL: https:/eprint.iacr.org/2022/1225.pdf.

[12] 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. URL: https://doi.org/10.1109%2Ficiccs53718.2022.9787987.

[13] 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. URL: https://doi.org/10.1007%2F978-3-031-22829-2_6.

[14] 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. URL: https://doi.org/10.1007%2F978-3-031-21280-2_24.

[15] 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. URL: https://doi.org/10.1007%2F978-3-031-22390-7_6.

[16] 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. URL: https://doi.org/10.1007%2F978-3-030-81293-5_12.

[17] Petr Muzikant. "Cloud Service Access Control using Smart Cards". MA thesis. Brno University of Technology, 2022. URL: http://hdl.handle.net/11012/208378.

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

[19] 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. URL: https://doi.org/10.1007%2F978-3-031-25319-5_14.

[20] 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. URL: https://doi.org/10.1007%2F978-3-030-41025-4_8.

[21] *Storage API*. Espressif Systems (Shanghai) Co. Ltd. 2023. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/storage/index.html.

[22] *FreeRTOS*. 2023. URL: https://www.freertos.org/.

[23] *ESP32 Memory Types*. Espressif Systems (Shanghai) Co., Ltd. 2023. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/memory-types.html.

[24] *NIST Post-Quantum: Selected Algorithms 2022*. National Institute of Standards and Technology. 2022. URL: https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.