Research project. STADS code: KIREPRO1PE. (2025), 1–17

IT UNIVERSITY OF COPENHAGEN

ARTICLE TYPE

Developer-centred security challenges in post-quantum cryptographic APIs

Jannik Egelund Verdoner

IT University of Copenhagen
Author for correspondence: , Email: jave@itu.dk.

Abstract

Cryptographic APIs bridge the gap between theoretical cryptography and practical software implementation. While classical standards are well-established, developers often struggle to use cryptographic APIs correctly, resulting in security vulnerabilities—an issue likely to be exacerbated by the adoption of post-quantum cryptographic schemes, many of which remain under active development. This project evaluates the usability of APIs implementing the schemes published by NIST, with a focus on ML-KEM and ML-DSA. We assess implementation difficulty, documentation quality, and the security assumptions placed on developers. Our findings identify common pitfalls and inform a selection of APIs suitable for future qualitative studies involving software developers. These studies aim to provide deeper insights into improving API design and documentation to reduce misuse and support secure post-quantum adoption.

Keywords: API Usability · Cryptography · Post-Quantum Cryptography · Usable Security · Human Factors · Information Security

I. INTRODUCTION

Developer-centric security recognises that most software developers are not experts in cryptography or cybersecurity, yet they are frequently tasked with implementing and integrating cryptographic functionality into their applications. This challenge is well documented by Tahaei et al. [1], who identify numerous usability issues that developers encounter when working with cryptographic APIs—ranging from insufficient documentation to problematic API design choices that increase the likelihood of misuse.

This reality places significant importance on the usability, design, and accessibility of cryptographic APIs. Fischer et al. [2] have shown that although the security community has produced a wealth of powerful cryptographic schemes and protocols, the gap between theoretical advances and their secure, widespread deployment in real-world systems remains substantial. A key factor contributing to this gap is the difficulty developers face when working with cryptographic APIs, which are often complex, poorly documented, or easy to misuse—frequently without the developer's awareness.

Although a broader ecosystem of tools exists to support secure software development—including static analysis tools, secure coding environments, and testing utilities—cryptographic APIs remain particularly critical [3], as even minor implementation errors can completely compromise system security. Recognising this, recent research has increasingly focused on improving the usability of cryptographic APIs by enhancing their documentation, reducing complexity, and aligning their design with the expectations and workflows of typical developers. This challenge becomes even more pressing in the context of post-quantum cryptography (PQC), where the transition to quantum-resistant cryptographic schemes is expected to introduce new implementation complexities and unfamiliar abstractions for software developers. On 13 August 2024, the US National Institute of Standards and Technology (NIST) published standards for three PQC schemes: one key encapsulation mechanism, ML-KEM [4], and two digital signature schemes, ML-DSA [5] and SLH-DSA [6], all believed to be quantum-resistant.

As standardisation efforts—such as those led by NIST—move theoretical PQC constructions into the hands of applied cryptography engineers responsible for designing production-ready cryptographic APIs, it is essential that these APIs are developed with a strong emphasis on usability. Poor API design can hinder secure adoption, regardless of the mathematical hardness of the underlying cryptographic schemes.

A. *Our Contributions*

In this paper, we review six post-quantum cryptographic APIs that implement one or more of the ML-KEM and ML-DSA schemes. Our evaluation combines manual and automated methods to assess API usability, focusing on implementation difficulty, documentation quality, and the implicit or explicit security assumptions made by API designers.

In addition to analysing the APIs themselves, we implemented sample code for each, using the official documentation available on GitHub and, where applicable, supplementary manuals (see Appendix 1). These implementations are evaluated using software engineering metrics commonly employed in industry—namely, cyclomatic complexity, lines of code, Halstead volume, and the maintainability index.

The study concludes by identifying APIs that appear well suited for future research, such as a master's thesis project based on planned interviews with students and professional software developers who possess some cybersecurity expertise but represent typical end users of such cryptographic APIs.

II. BACKGROUND

A. Post-quantum cryptography

Post-quantum cryptography (PQC) refers to cryptographic schemes believed to remain secure against adversaries equipped with fault-tolerant quantum computers. Classical public-key schemes such as RSA, ElGamal, and DSA rely on number-theoretic problems that can be efficiently solved by quantum algorithms for integer factorisation and discrete logarithms, most notably Shor's algorithm [7], introduced in 1994, and a faster variant proposed by Oded Regev in 2025 [8]. If physicists succeed in building fault-tolerant quantum computers, classical public-key schemes will no longer offer guarantees of confidentiality or authenticity. In 2024 Chevignard et al. [9] demonstrated that Shor's algorithm executed on a fault-tolerant quantum computer with 648 logical qubits is capable of solving a 224-bit discrete logarithm instance in a 2048-bit prime group. Scaling this to 1.730 logical qubits would suffice to break RSA-2048 encryption.

IBM and Microsoft have set the period 2029–2035 as a target for achieving fault-tolerant quantum computing with 1.000 or more logical qubits [10, 11]. In anticipation of this, the US National Institute of Standards and Technology (NIST) and the United Kingdom's National Cyber Security Centre have set 2035 as the deadline by which IT systems in large organisations and critical national infrastructure must complete their transition to PQC [12, 13]. Similarly, a joint declaration from the national security agencies of 18 EU member states has set 2030 as the migration deadline [14].

The threat is further intensified by the harvest now, decrypt later strategy, in which encrypted data is collected today with the intention of decrypting it in the future, once quantum capabilities are realised. As a result, even communications considered secure at present may be compromised retroactively. This provides strong motivation to adopt post-quantum cryptography well before fault-tolerant quantum computers are built—a concern often referred to as the "Y2Q" problem. In response, PQC schemes are being actively developed and standardised, most notably through the NIST PQC Standardisation Project. As of March 2025, the project is in its fourth round of standardisation, and three PQC schemes have been selected to date: two based on the hardness of lattice problems and one on the hardness of finding hash preimages.

i. Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM)

ML-KEM [4] is a key encapsulation mechanism (KEM) based on the CRYSTALS-Kyber scheme [15], designed to establish a shared secret key between two communicating parties. This shared secret can subsequently be used for symmetric-key cryptography. The correctness property of the scheme ensures that both parties derive the same shared key, i.e., that $K' = K$. ML-KEM comprises the following algorithms:

- **Key generation**, which generates an encapsulation key and a corresponding decapsulation key.
- **Encapsulation**, Uses the encapsulation key to generate a shared secret key and an associated ciphertext.
- **Decapsulation**, Uses the decapsulation key to produce a shared secret key from a ciphertext.

The security of ML-KEM is based on the hardness of the Module Learning with Errors (MLWE) problem, a generalisation of the standard Learning with Errors (LWE) problem, which is believed to be resistant to quantum attacks. The construction of ML-KEM proceeds in two stages: first, a public-key encryption scheme is built from the MLWE assumption; this is then converted into a key encapsulation mechanism using the Fujisaki–Okamoto transform. ML-KEM defines three security category sets—512, 768, and 1024—corresponding to NIST security categories 1, 3, and 5, respectively.

ii. Module-Lattice-Based Digital Signature Standard (ML-DSA)

ML-DSA [5] is a digital signature scheme based on the CRYSTALS-Dilithium algorithm [16], which can be used to detect unauthorised modifications to data and to authenticate the identity of a signatory. The correctness property ensures that any message signed with a valid signing key will be accepted as valid when verified using the corresponding verification key; that is, $Verify(pk, m, \sigma)$, where σ is the signature. ML-DSA comprises the following algorithms:

- **Key generation**, which generates a public–private key pair.
- **Sign**, which generates an ML-DSA signature on a given message.
- **Verify**, which checks the validity of a signature σ on a message m.

The security of ML-DSA is also based on the hardness of the Module Learning with Errors (MLWE) problem, as described above. The scheme is constructed by instantiating a Schnorr-style signature scheme using the Fiat–Shamir heuristic applied to an interactive proof, in which a prover demonstrates knowledge of matrices A, S_1, and S_2 to a verifier who knows A and $T = AS_1 + S_2$. ML-DSA defines three security category sets—44, 65, and 87—corresponding to NIST security categories 2, 3, and 5, respectively, where each numeral denotes the dimension of the matrix A.

iii. Stateless Hash-Based Digital Signature Standard (SLH-DSA)

SLH-DSA [6] is a digital signature scheme based on SPHINCS$^+$ [17]. The scheme is constructed from other hash-based signature primitives, specifically the few-time signature scheme FORS and the multi-time signature scheme XMSS. To sign a message, a hash of the message is used to select a specific FORS key, which then signs the hash. The resulting FORS signature, together with a hypertree of XMSS signatures that authenticates the selected FORS public key, constitutes the final SLH-DSA signature. All components derive from a single public root and a private seed.

The security of SLH-DSA is based on the assumed hardness of finding preimages for cryptographic hash functions, along with other related properties of these functions.

III. METHODOLOGY

The evaluation of the post-quantum cryptographic APIs considers four key aspects: API design, documentation, code examples, and security. Recent research in developer-centric security has primarily employed developer interviews, in which participants are asked to complete a task (e.g., implementing a cryptographic API), followed by a questionnaire. As our study is based on self-evaluation using assessment techniques that do not involve external participants, some of the selected metrics may introduce a degree of bias.

To demonstrate practical use of the APIs, we implement the ML-KEM and ML-DSA cryptographic schemes, relying solely on the official documentation provided with each API. At the time of selection, we identified only one API that implemented SLH-DSA; consequently, we chose to focus our implementation efforts on ML-KEM and ML-DSA.

A. Selection of APIs to Review

In selecting APIs for review, two key considerations emerged: (1) the distinction between cryptographic schemes and cryptographic protocols, and (2) the choice of programming language. Our initial survey of available APIs revealed that most implementations provided support for ML-KEM or ML-DSA as standalone schemes, requiring the developer to integrate them manually into larger cryptographic protocols such as TLS or OpenSSL. In other words, these APIs were not yet embedded in production-ready protocol implementations. As a result, we chose to implement the cryptographic schemes directly, independent of any surrounding protocol.

B. Evaluating API Design, Documentation, and Security

Given the wide range of available metrics for assessing API usability, we limited our evaluation to a targeted subset. These metrics are loosely informed by Nielsen's heuristic evaluation principles [18], which have been adapted to API usability by Myers and Stylosv[19], as well as the measurable usability attributes outlined in the API Concepts Framework by Scheller and Kühn [20]. An overview of the selected metrics is provided in Table 1 on the following page.

We apply a combination of automated and manual techniques to evaluate the APIs against these metrics. Manual evaluation is carried out by the authors through direct inspection of the API code and accompanying documentation, assessing the degree to which each API satisfies the defined criteria. For the automated evaluation, we employ the following tools:

i. Automated analysis tools

We use several tools to evaluate code and text quality. Lizard is an open-source static analysis tool that measures metrics such as cyclomatic complexity, lines of code and function parameters. [21] (see section *C* for a short introduction). ctag indexes source code symbols like functions and classes into a tags file to support structural and naming pattern analysis [22]. The Readability Analyzer is a web-based tool that assesses textual clarity and complexity using metrics like the Flesch Reading Ease [23].

C. Evaluating API example code

To evaluate the code examples, we employ metrics that are commonly used in industrial settings [24], as well as time-on-task, which we use to assess the number of hours required to implement ML-KEM and ML-DSA using the official API documentation and code examples, where available. See below:

i. Cyclomatic complexity

Cyclomatic complexity, introduced by Thomas J. McCabe [25], is a software metric that quantifies the structural complexity of a program. It measures the number of linearly independent paths through the source code, as derived from its control-flow graph. Higher values indicate greater complexity, which can affect testability and maintainability, and may correlate with an increased likelihood of errors.

ii. Lines of code

Lines of code is a fundamental software metric that quantifies the size of a computer program by counting the number of lines in its source text.

iii. Halstead volume

Halstead Volume, a key metric from Maurice Halstead's Software Science book [26], quantifies program size and complexity. It is calculated from the total occurrences and unique types of operators and operands in source code. This metric represents the information content of a program or the mental effort required for its development.

iv. Maintainability index

The maintainability index is a software metric introduced by Oman and Hagemeister [27], intended to quantify the relative ease of maintaining source code. It produces a single indicative value by combining cyclomatic complexity, lines of code, and the Halstead volume, yielding an index in the range 0 to 100 that reflects the relative maintainability of the programme. Higher values indicate better maintainability. The index MI is defined as:

$$MI = \max(0, (171 - 5.2 * \ln(V) - 0.23 * (M) \\ -16.2 * \ln(LoC)) * 100/171) \quad (1)$$

where V is the Halstead volume, M is the cyclomatic complexity and LoC is lines of code. We use the refined maintainability index equation proposed by the Microsoft Corporation [24], which produces a final index value as an integer between 0 and 100. An index value of 0–9 indicates low maintainability, 10–19 suggests moderate maintainability, and 20–100 represents good maintainability. The metric serves as a practical tool for assessing code quality and identifying areas that may require refactoring to improve long-term sustainability.

v. Time-on-task

Using a timer, we measure how long it takes to implement ML-KEM and ML-DSA relying solely on the official API documentation and any accompanying code examples.

Table 1. API metrics

Metrics	Method of evaluation
API design	
Lines of code Size of the API. Total lines of code and lines of code for crypto code only.	Automatic evaluation using Lizard
Number of functions Number of functions or methods exposed to the API user.	Automatic evaluation using Lizard
Parameter ordering Parameter organisation and consistency in argument ordering across functions with similar types (e.g. consistent ordering of msg, sig, and pubk in overloaded signature scheme functions)	Manual evaluation
Parameter complexity Number of parameters per function required to implement ML-KEM and ML-DSA; functions with many parameters increase cognitive load and the likelihood of developer error.	Automatic evaluation using Lizard
Function error type Consistency in how functions return results and errors.	Manual evaluation
Naming convention Class and function naming conventions (e.g. PascalCase, lowerCase, snake_case)	Automatic evaluation using ctag
Standardized names of cryptographic schemes Consistency with established cryptographic naming conventions established by the NIST standards.	Manual evaluation
security category Specification of the security category in the API. Is it defined explicitly by the developer (e.g. as an integer value) or implicitly through selection of the scheme (e.g. "ML-KEM-512")	Manual evaluation
Abstraction level Level of abstraction and exposure of implementation details (e.g. extent to which internal aspects of the ML-KEM cryptographic scheme are made public)	Manual evaluation
API documentation	
Readability score Representation and organisation of documentation, based on the Flesch reading ease score (0-100) as per Kincaid et al. [28]; a score of 40 or higher is recommended for official documents in the USA.	Semi-automatic evaluation using Readability Analyzer
Security explanation clarity Explanation of critical security aspects in the documentation (e.g. recommended parameter lengths and known vulnerabilities)	Manual evaluation
Documentation coverage Extent to which the documentation provides the necessary information for developers to understand and use the API effectively.	Manual evaluation
Security best practices Extent to which the documentation explains API usage according to cryptographic best practices.	Manual evaluation
Deprecation & update warnings Extent to which the documentation provides clear warnings about API deprecations or updates, given that many post-quantum APIs are still in the research stage.	Manual evaluation
API security	
Developer assumptions Extent to which the documentation specifies the recommended background for developers (e.g. cryptographic expertise)	Manual evaluation
Misuse resistance Extent to which the API promotes secure default configurations (e.g. default values for security parameters and entropy generation) and prevents selection of insecure combinations.	Manual evaluation

IV. RESULTS

A. APIs under review

With respect to the choice of programming language, we elected to focus on APIs implemented in C or C++. Many high-level APIs, such as Google's Tink, rely on cryptographic schemes that are implemented in these languages. The following APIs were selected during our initial screening:

- Google Tink [29]
- AWS-LC [30]
- liboqs [31]
- mlkem-native [32]
- wolfCrypt [33]
- Botan3 [34]
- ML-KEM API [35]
- ML-DSA API [36]

Our review of these APIs revealed that direct comparison between Google Tink and AWS-LC and the other libraries was not meaningful. Google Tink is a multi-language library that incorporates multiple low-level APIs to implement cryptographic schemes, providing a high-level interface for developers. Similarly, AWS-LC is derived from Google's BoringSSL and OpenSSL for backend implementation. In contrast, the other APIs provide low-level, standalone implementations of post-quantum schemes and do not rely on external cryptographic backends.

As a result, we selected six APIs for detailed evaluation—all implemented in C or C++. Of these, three are maintained by open-source collaborations, one by a private corporation, and two by a cryptographic engineer affiliated with a public research institute.

A positive outcome of this selection is that the APIs under review have been developed within a range of organisational contexts—for instance, by open-source communities on GitHub as well as by internal teams in corporate and academic environments. This diversity has enabled us to examine a broad spectrum of design approaches and gain varied perspectives on cryptographic API usability.

i. liboqs

liboqs [31] is an open-source API developed in C, with a C++ wrapper [37], and is part of the Open Quantum Safe project, which is supported by the Post-Quantum Cryptography Alliance under the Linux Foundation. liboqs provides a collection of implementations of post-quantum key encapsulation mechanism and digital signature schemes. Although the API includes multiple post-quantum schemes—such as FrodoKEM and Falcon—the only schemes that implement the NIST standards are ML-KEM and ML-DSA. The API is currently not recommended for use in production environments to protect sensitive data. The authors of liboqs state that the purpose of the API is cryptographic research and prototyping.

The API is actively maintained by developers from the Open Quantum Safe project and by cryptography researchers at various universities, such as the University of Waterloo in Canada.

ii. mlkem-native

mlkem-native [32] is an open-source API developed in C, and is part of the PQ Code Package project, which is supported by the Post-Quantum Cryptography Alliance under the Linux Foundation. mlkem-native provides a secure, formally verified memory and type safe, and portable C90 implementation of the NIST standard ML-KEM that is suitable for deployment in production environments.

The API is actively maintained by developers from the Open Quantum Safe and PQ Code Package projects, as well as by cryptographers employed by Amazon.com, Inc.

iii. wolfCrypt

wolfCrypt [33], developed in C, is part of the open source wolfSSL embedded SSL library, which is designed for embedded devices, real-time operating systems, and other resource-constrained environments. A subset of the wolfCrypt API has been FIPS 140-3 validated— a requirement for cryptographic APIs used by the federal government in the United States. wolfCrypt provides implementations of the NIST standards ML-KEM, ML-DSA, and SLH-DSA that are suitable for deployment in production environments. The API is actively maintained by wolfSSL Inc.

iv. botan3

botan3 [34] is an open-source API developed in C++ by Jack Lloyd. botan3 provides implementations of various post-quantum schemes, such as FrodoKEM and Classic McEliece, as well as the NIST standards ML-KEM, ML-DSA, and SLH-DSA, which are suitable for deployment in production environments. botan3 is actively maintained by Jack Lloyd, along with hundreds of contributors via GitHub.

v. ml-kem API

ml-kem [35] is an open-source API developed in C++ by Anjan Roy, a cryptographic engineer at cryptography research center, Technology Innovation Institute, Abu Dhabi. ml-kem provides an implementation of the NIST standard ML-KEM. The library only includes known answer tests (KATs) for verifying functional correctness. Therefore, the implementation is not suitable for deployment in production environments. The API is not actively maintained, the latest update on GitHub occurred on 6th of march, 2025.

vi. ml-dsa API

ml-dsa [36] is an open-source API developed in C++ by Anjan Roy, a cryptographic engineer at the cryptography research center, Technology Innovation Institute, Abu Dhabi. ml-dsa provides an implementation of the NIST standard ML-DSA., supporting parameter sets ML-DSA-44, ML-DSA-65, and ML-DSA-87 as defined in FIPS 204. The library only includes known answer tests (KATs) for verifying functional correctness. Therefore, the implementation is not suitable for deployment in production environments. The API is not actively maintained, the latest update on GitHub occurred on 7th of march, 2025.

B. API design results

The results obtained using Lizard and ctag, as described in section 3, are presented in the section on API surface area below. The analysis performed with Lizard indicates that liboqs, wolfCrypt, and botan3 are orders of magnitude larger than the other APIs under review. Although their surface areas are substantial (see figures 1 and 2), our focus lies on a specific subset of each API—namely, the portions of the source code that implement the post-quantum NIST standards.

All evaluations of API design—except for lines of code, number of functions, and level of abstraction—are based solely on the source files that a developer is expected to interact with when implementing ML-KEM and ML-DSA. The relevant source files are shown in figure 3.

i. API surface area

Figures 1, 2, and 3 provide an overview of the size of the APIs under review, including total lines of code, the number of functions, and the number of parameters per function.

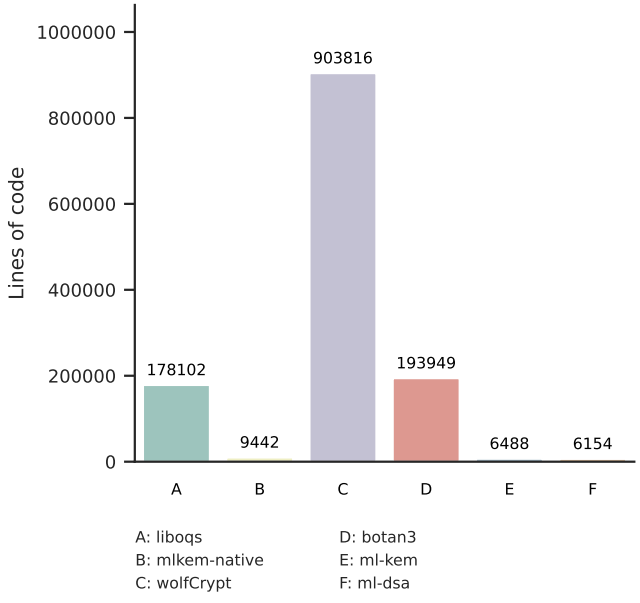

Figure 1. Lines of code

One reason for the larger size of wolfCrypt compared to the other APIs is its long development history; it has been actively maintained since 2004 by Todd Ouska and the company wolfSSL Inc. In contrast, liboqs began in 2014 as a research initiative at the University of Waterloo under the Open Quantum Safe project, while botan3 is maintained solely by Jack Lloyd with contributions from the open-source community.

mlkem-native implements only ML-KEM; however, as it is a production-ready API, it has been hardened against various types of side-channel attacks, including KyberSlash3 discovered by Bernstein et al [38] and Clangover4 by discovered by Purnal [39] in 2023 and 2024 respectively. Moreover, the code has been formally verified using the C Bounded Model Checker.

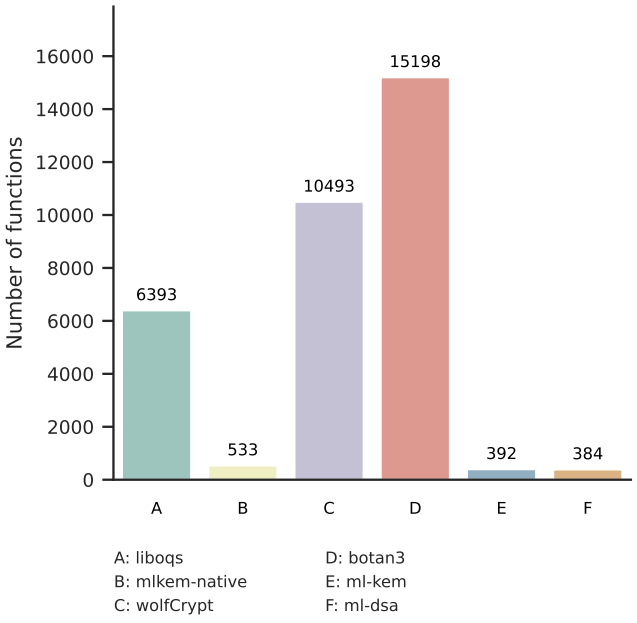

Figure 2. Number of functions

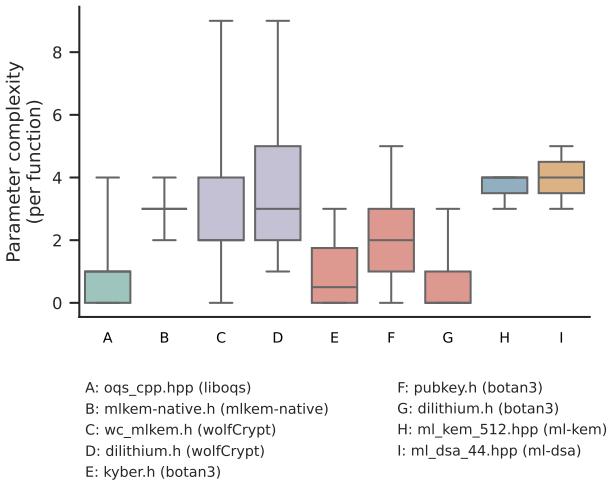

Figure 3. Parameter complexity

Figure 3 shows that the source file oqs_cpp.hpp in liboqs exhibits a low median parameter complexity with minimal variance. This is attributable to the design of its core functions: the key generation function takes no parameters, while the encapsulation and decapsulation functions each take a single argument—the public key and the ciphertext, respectively. The verify function accepts the highest number of parameters: the message, signature, context, and public key. The remaining functions primarily serve as getters or lookup utilities.

The source file mlkem-native.h in mlkem-native exhibits similarly low variance in parameter complexity, which can be attributed to the limited number of macros exposed to

developers. The macros used for key generation, encapsulation, and decapsulation accept three and two arguments, respectively and updates the input parameters as a side effect.

In contrast, the wc_mlkem.h and dilithium.h source files in wolfCrypt are designed to provide the developer with full control over the cryptographic scheme. This includes parameters for the security level and entropy generation, as well as settings related to specific embedded hardware architectures and heap memory placement. For instance, the key generation function in wc_mlkem.h has two versions: a standard key generation function and an optimised variant with additional parameters for memory placement, accepting five and seven parameters, respectively.

Like liboqs, botan3 also exposes numerous getter and lookup functions to the developer. The kyber_h and dilithium_h interfaces are designed for general-purpose use of the Kyber and Dilithium schemes with the option of using the ML-KEM and ML-DSA variants; accordingly, the key generation function accepts an entropy generator, a security level, and an scheme identifier (e.g., ML_KEM) as arguments.

In the ml-kem and ml-dsa APIs, the developer supplies variables to the functions, which are updated as a side effect. For example, the key generation function returns void; thus, the developer must provide the z and d seeds, along with variables for the public and private keys, which will be populated by the function.

That said, the key generation, encapsulation and decapsulation functions across all APIs under review adhered to a maximum of four parameters, as recommended by Scheller and Kühn [20].

ii. Parameter ordering

In the liboqs API, the functions implementing ML-KEM provide only a single function for encapsulation and decapsulation, and therefore do not employ overloading. For ML-DSA, however, there exist multiple similarly named functions for signing messages and verifying signatures. These functions are consistent in their organisation of parameters. See Listing 1 for the parameter organisation of the sign functions.

```
sign(bytes message)
sign_with_ctx_str(bytes message, bytes context)
```

Listing 1. liboqs sign parameter ordering

In mlkem-native, two macros are provided for key generation. The first macro takes as input the public and private keys, which it populates as a side effect; the second macro accepts an additional parameter for randomness (entropy).

botan3 uses constructors, rather than functions, for key generation in both ML-KEM and ML-DSA. The constructors follow a consistent pattern: the first constructor overload is used to generate a new key pair, while the remaining two constructors are used to import an existing key pair instead of generating a new one. For encapsulation and decapsulation in ML-KEM, and for signing and verification in ML-DSA, botan3 provides only a single function for the developer to use in each case.

In wolfCrypt, the standard key generation and encapsulation functions in wc_mlkem.h, along with their memory-optimised variants, do not appear to follow a consistent or logical parameter ordering. This inconsistency increases the difficulty for developers attempting to use these functions. For example, in both the standard encapsulation function and its memory-optimised counterpart, certain parameters appear to be arranged arbitrarily. In dilithium.h, on the other hand, the various variants of the sign and verify functions appear to follow a consistent parameter ordering, in which any additional parameters are listed first, followed by the remaining parameters. See listing 2 for an example of the illogical parameter ordering in wc_mlkem.h.

```
void mlkem_encapsulate(const sword16* pub,
    sword16* bp, sword16* v,
    const sword16* at, sword16* sp, const
    sword16* ep, const sword16* epp,
    const sword16* m, int kp);

int mlkem_encapsulate_seeds(const sword16* pub,
    MLKEM_PRF_T* prf, sword16* bp,
    sword16* tp, sword16* sp, int kp, const byte
    * msg, byte* seed,
    byte* coins);
```

Listing 2. wolfCrypt encapsulation parameter ordering

The ML-KEM and ML-DSA APIs do not use functional overloading or similarly named functions (e.g., encapsulate(...) and encapsulate_with_entropy(...)). Consequently, no conclusions can be drawn regarding the role of the parameter in this context.

iii. Error types

Errors in liboqs are handled exclusively by C++ standard library runtime error objects, which are thrown inside the function of the cryptographic scheme object if failure occurred.

In mlkem-native, error handling is delegated to the developer, who can check whether the value returned by the relevant macro (e.g., key encapsulation) is 0 for success or -1 for failure. mlkem-native is the only API that relies exclusively on macros for the use of ML-KEM.

wolfCrypt also delegates error handling to the developer. The API is designed with embedded systems in mind and therefore avoids the use of exception handling. However, functions in wolfCrypt return status codes of type int, indicating 0 for success and 1 for error. In combination with the API error message header, this allows the developer to implement custom error handling mechanisms, such as exceptions or static assertions.

botan3 employs a combination of C++ standard library invalid argument error objects, static assertions such as BOTAN_ASSERT_UNREACHABLE, and custom error objects (for encoding and internal mechanisms) for error handling. Every function in the Kyber and Dilithium source files has a return type that the developer may utilise for their own error handling.

The ml-kem and ml-dsa APIs do not incorporate error handling within the API functions themselves, leaving it to

the developer to implement their own error handling. The key generation functions for both ml-kem and ml-dsa have a void return type, which cannot be utilised for custom error handling. In contrast, the functions for encapsulation, decapsulation, verification, and signing return boolean values. In the example implementations provided with the APIs, error handling is performed using static assertions.

Refer to table 2 for an overview of how error handling is done in the APIs.

Table 2. Summary of error handling

std error handling[a]	custom error handling	no error handling
liboqs	botan3[b]	ml-kem native
botan3		wolfCrypt
		ml-kem
		ml-dsa

a C++ standard library
b Including static assertion

iv. Naming conventions

We used ctags to examine the naming convention policies, if any, employed in the APIs under review. If an API uses multiple naming conventions, e.g., camelCase and snake_case we assume that no consistent convention is enforced.

Based on the output of ctags and manual inspection of representative code examples, we conclude that all APIs under review consistently employ the snake_case naming convention for functions.

A possible explanation for the consistent use of snake_case across all APIs may be the growing popularity of the Python programming language, which adopts snake_case for functions and variables [40].

v. Standardized names of cryptographic schemes

All the APIs use the standardised names defined by the NIST standards 203 and/or 204. However, the manner in which these names are used varies across the APIs.

In liboqs, the scheme is specified by a string value in combination with a security level.

In mlkem-native, the macros reference MLKEM, but the final function that the developer ultimately invokes includes only KEM in its name. Nonetheless, since the API exclusively implements the ML-KEM cryptographic scheme, the developer is expected to infer that it refers to the post-quantum version of KEM.

In wolfCrypt, the functions for both ML-KEM and ML-DSA explicitly include the corresponding NIST standard names as part of the function identifiers.

In botan3, ML-KEM and ML-DSA are initialised in the kyber.h and dilithium.h source files via enumeration values (e.g., ML_KEM_1024).

In the ml-kem and ml-dsa APIs, the scheme name and security level are encoded in the class name. For example, instantiating the class ml_kem_512.hpp selects ML-KEM with security level 512.

vi. security category

The APIs under review expose the security category to the developer in different ways, but all provide options for selecting the category setting as defined in the NIST standards FIPS 203 and FIPS 204. ML-KEM uses categories 1, 3, and 5. ML-DSA uses 2, 3 and 5. Each categority sets a size for the keys. In liboqs, for both ML-KEM and ML-DSA, the security category is selected as part of the scheme name, specified as a string. An error is returned if liboqs does not find a matching string. See listing 3.

```
string kem_primitive = "ML-KEM-512";
KeyEncapsulation recipient{kem_primitive};
bytes public_key = recipient.generate_keypair();
```

Listing 3. liboqs security category

Setting the security category with mlkem-native for ML-KEM is considerably more involved, as the parameter is specified as a compilation definition required by the linker when compiling the code example. See listing 4.

```
target_compile_definitions(cryptoAPI PRIVATE
    MLK_CONFIG_PARAMETER_SET = 512
    MLK_CONFIG_API_PARAMETER_SET =
    MLK_CONFIG_PARAMETER_SET
    MLK_CONFIG_API_NAMESPACE_PREFIX =
    PQCP_MLKEM_NATIVE_MLKEM512
)
```

Listing 4. mlkem-native security category

wolfCrypt specifies the security category as a preprocessor directive that defines a macro for ML-KEM and ML-DSA. In the case of ML-KEM, the macro is a numerical value—512, 768, or 1024—consistent with mlkem-native. For ML-DSA, in contrast to liboqs, the security category is specified as a level from 1 to 5. See appendix G in the wolfSSL documentation []

The macro can then be passed as an argument to the key generation function. See listing 5.

```
// ML-KEM security category
wc_MlKemKey_Init(temp_key_ptr.get(),
    WC_ML_KEM_512, nullptr, INVALID_DEVID);

// ML-DSA security category
wc_ml_dsa_make_key(&rng, ML_DSA_L5, &key);
```

Listing 5. wolfCrypt security category

In botan3 the security category is consistent with liboqs, for both ML-KEM and ML-DSA the parameter is selected with the scheme name, specified as a enumeration value. See listing 6.

```
enum Mode {
    ...
    ML_DSA_4x4,
    ML_DSA_6x5,
    ML_DSA_8x7,
};

enum Mode {
    ...
    ML_KEM_512,
    ML_KEM_768,
    ML_KEM_1024,
```

```
};
```

Listing 6. botan security category

For the ml-kem and ml-dsa APIs, the security category is selected with the scheme name, which is determined by the class. The developer selects a class and then uses the corresponding key generation, encapsulation and decapsulation, or sign and verify methods provided by the relevant class. See Listing 7.

```
#include "ml_kem/ml_kem_512.hpp"
ml_kem_512::keygen(d_seed, z_seed, pkey, skey);
ml_kem_512::encapsulate(msg, pkey, ciphertext,
    sender_key);
ml_kem_512::decapsulate(skey, ciphertext,
    receiver_key);

#include "ml_dsa/ml_dsa_44.hpp"
ml_dsa_44::keygen(xi, pkey, skey);
ml_dsa_44::sign(rnd, skey, msg, {}, sig);
ml_dsa_44::verify(pkey, msg, {}, sig);
```

Listing 7. ml-kem/ml-dsa security category

Refer to table 3 for an overview of how each API set the security category.

Table 3. Summary of security category initialisation

String	Enumeration	Compile definition	Macro	Class
liboqs	botan3	mlkem-native	wolfCrypt	ml-kem
				ml-dsa

vii. Abstraction level

liboqs, botan3, ml-kem, and ml-dsa are situated at the higher end of the abstraction spectrum. In liboqs and botan3, the internal workings of the cryptographic scheme are encapsulated by an object that maintains internal state and reports errors for use in developer-side error handling. The ml-kem and ml-dsa APIs populate variables in place, thereby managing the internal state of the scheme, while making full use of the C++ standard library.

mlkem-native and wolfCrypt occupy the lower end of the abstraction spectrum, being optimised for portability across multiple hardware architectures (e.g., Intel, ARM, AMD) and for high throughput performance. The internal state of the schemes is managed via individually initialised variables, which are populated in place. The mlkem-native design is separated into a fixed frontend and two modular backends responsible for arithmetic and FIPS 202 operations [41]. These backends are performance-sensitive components that may be implemented in either C or assembly. Developers may either employ the backends provided by the API or implement their own using C, AArch64, or x86_64.

wolfCrypt, by contrast, adopts a more rigid architecture that does not support the use of custom backends—e.g., it does not allow developers to use wc_kem.h as a frontend in conjunction with a custom backend implementation for specific optimisations. Nevertheless, the routines in wolfCrypt rely on return codes and intermediate function initialisations, providing considerable flexibility. This enables developers to assign and specify memory locations for each memory allocation within the codebase.

C. API documentation results

i. Readability

As a starting point for reporting on API documentation readability, we use the Datayze Readability Analyzer [23] to investigate the difficulty of reading the official documentation provided by each API. All APIs provide documentation on their GitHub pages, and wolfCrypt and botan3 offer additional documentation for developers in the form of manuals available on their respective web pages [42] and [43] and a YouTube channel. Due to the size of the documentation manuals (1.766 pages for wolfCrypt and 350 pages for botan3), only a subset of pages was included in the readability analysis. Specifically, five pages from Chapter 10 ("wolfCrypt Usage Reference") were included from the wolfCrypt manual, and eight pages from Chapter 8 ("API Reference") were included from the botan3 manual. We copied the text on GitHub by inserting the raw README file into the readability analyser, using the Flesch Reading Ease (FRE) readability score, where a higher score is easier to read. See figure 4 for the API readability scores.

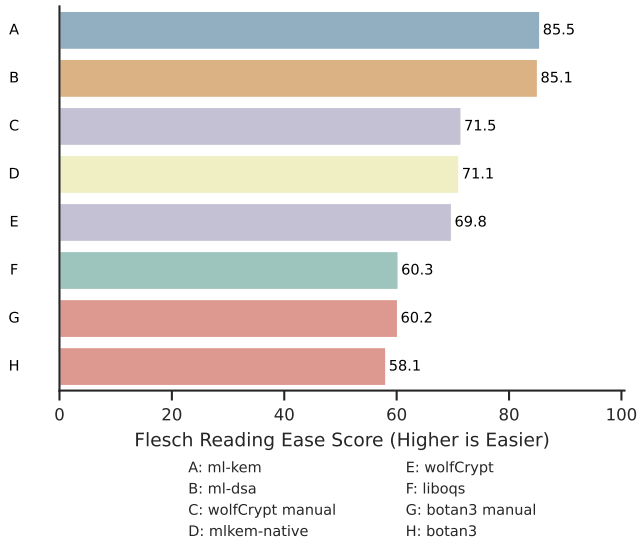

Figure 4. Readability score with FRE

The high readability scores of the ml-kem and ml-dsa APIs can be attributed to the fact that their documentation is short, concise, and direct, without an abundance of technical details. It is not surprising that the manuals exhibit slightly higher scores than their GitHub counterparts for wolfCrypt and botan3; documentation hosted on GitHub is commonly intended to provide developers with quick access to implementation details and examples during code development. Manuals, on the other hand, are designed to offer careful explanations with comprehensive technical details and generally require greater attention to read.

ii. Documentation coverage

All the APIs under review provide documentation, quickstart guides, and usage examples. In the liboqs documentation, there is an overview of the API's functionality, including a list of all schemes and their associated security levels. The API provides examples in both C and C++, which the developer can readily integrate into their own project.

In mlkem-native, the documentation provides a quickstart guide at the beginning and offers extensive descriptions of how the developer can implement a custom backend. It is clear from the documentation that the API is intended for developers who wish to tinker directly with the ML-KEM design to suit their own needs, whether for performance optimisation or support for hardware architectures not covered by the API. The documentation provides eight examples demonstrating how to build the source code, including single-level builds, multi-level builds, custom FIPS 202 builds, and single compilation unit builds.

In wolfCrypt, a substantial portion of the GitHub documentation [33] is devoted to commercial material, such as reasons to choose the API, size comparisons with OpenSSL, and a release log detailing new additions, enhancements, and optimisations. Nevertheless, the GitHub documentation also provides multiple code examples demonstrating how to use the API, including client and server, echo client and echo server, and SCTP examples. The examples are quite advanced and contain a substantial amount of code (for instance, the client .c file contains 4.941 lines of code). The developer must determine independently which parts are needed and how to incorporate them into their own project. The wolfCrypt manual provides a detailed introduction to wolfSSL, guidance on building the code, and detailed descriptions of each code example found on GitHub. If the developer requires more technical details, they can consult the wolfSSL and wolfCrypt API reference chapters in the manual, which provides comments on every cryptographic scheme and their functions. A chapter in the appendices describes how to use wolfSSL and wolfCrypt for post-quantum cryptography.

In botan3, the documentation on GitHub provides a brief introduction to the API, followed by a section on released versions of botan3 and another section describing the API's functionality, without detailing the security levels of each cryptographic cheme or protocol. botan3 includes 47 short code examples, each demonstrating how to use a single scheme, such as ML-KEM, or a protocol, and how to generate entropy using the API. Most examples contain fewer than 100 lines of code, making it relatively straightforward for the developer to incorporate the example code into their own projects. The botan3 manual provides detailed descriptions for each scheme and protocol available in the API, along with warnings about potential pitfalls.

In the ml-kem and ml-dsa APIs, the documentation briefly describes how the ML-KEM and ML-DSA schemes work. Both APIs provide a single example, which is short and easy for the developer to integrate into their own project.

None of the API documentations explicitly mentions usage according to best practices in cryptography, although it can be inferred that the provided code examples offer an indication of how the API authors intend developers to use their code.

iii. Security documentation

With regard to security documentation, liboqs contains a section specifically dedicated to security. It states that there are no known vulnerabilities in the implemented schemes, but advises developers to implement post-quantum schemes only once the NIST standardisation project has concluded, and therefore does not recommend the API for use in production environments. The overall documentation provides an overview of all available KEM and signature schemes, along with their associated security levels. However, no recommendations regarding appropriate security levels are given; it is left to the developer to determine the suitable settings.

In mlkem-native, two sections address security. One explicitly states that the API has been formally verified using a model checker, while the other explains in detail the positive effects of certain design choices on security, including the prevention of recently discovered side-channel attacks such as KyberSlash [38] and clangover [39]. The documentation mentions the available security levels but does not provide recommendations on which settings should be used.

The wolfCrypt GitHub documentation provides limited details regarding security, apart from a few update notes concerning the deprecation of cryptographic schemes and significant changes to existing schemes, as well as a brief section under "Enhancements and Optimisations" dedicated to minor code fixes, such as improving the compatibility of post-quantum cryptography with Chromium browsers. In the wolfCrypt manual, Appendix G is dedicated to post-quantum cryptography. This chapter includes a table describing the security levels for both ML-KEM and ML-DSA. For example, ML-KEM security level 512 corresponds to a public key size of 800 bytes, a private key size of 1632 bytes, a ciphertext size of 768 bytes, and a shared secret size of 32 bytes. However, the developer is assumed to have a minimal background in cryptography to understand the implications of these parameters for security.

In botan3, the GitHub documentation does not mention security at all, except for a security advisories page in the botan3 manual for cases where a developer has identified security issues in the API. However, the documentation explicitly states that if the developer requires more information, he or she must refer to the manual. In the manual, there is no mention of recommended security levels or known vulnerabilities. Nevertheless, it is explicitly stated that the developer is assumed to have some knowledge of cryptography, and several external books are recommended as preparatory reading before using the API.

Neither the ml-kem nor the ml-dsa API documentation on GitHub includes a dedicated section on security, nor do they state known vulnerabilities or describe the security levels. However, the documentation clearly states that the APIs have not yet been audited and should not be considered safe to use.

iv. Deprecation and update warnings

Refer to table 4 for an overview of which APIs under review include deprecation and/or update warnings in their documentations.

Table 4. Inclusion of deprecation and update warnings in API documentation

	Deprecation warning	Update warning
liboqs	no	yes[a]
mlkem-native	no	no
wolfCrypt	yes	yes
botan3	yes[b]	yes
ml-kem	no	no
ml-dsa	no	no

a Release log. Warnings mixed with updates.
b In manual. Some cryptographic schemes include a deprecation warning in their description.

Even though some of the APIs under review do not include deprecation or update warnings, this may simply indicate that there has not yet been a need for such warnings.

D. API security results

i. Developer assumptions

The documentation and codebases of the APIs under review do not consistently specify the level of cryptographic knowledge expected of developers. Nevertheless, implicit assumptions about developer expertise often become apparent in various ways. For instance, in low-level programming, it is common for the inner workings of an API to be exposed to the developer, allowing greater control over, for example, memory allocation. As more functionality becomes the responsibility of the developer, so too does the correct use of low-level cryptographic schemes. Consequently, a higher level of cryptographic expertise is implicitly assumed.

The liboqs API does not explicitly provide recommendations regarding the expected level of cryptographic knowledge. However, under the "Limitations and Security" section of the GitHub documentation, it is stated that the API is intended for cryptographic research and prototyping. It is therefore likely that the target users are based in academia. That said, the functions in the ml-kem class expose no more than four parameters to the developer, and both the key generation and encapsulation functions accept only a single parameter each (see figure 3).

In mlkem-native the documentation states that the API is built with a fixed frontend and offers the option to define custom backends where specific optimisations are required by the developer. This implicitly suggests that the API authors assume developers possess sufficient cryptographic expertise to implement their own schemes or edit existing schemes for use in the backend.

In the case of wolfCrypt, neither the manual nor the GitHub documentation specifies recommended prerequisites. Nonetheless, an inspection of the code examples and selected sections of the manual indicates that the API is intended for experienced cryptography engineers. As shown in figure 3, some of the functions expose up to nine parameters, most of which pertain to randomisation, memory management and device identification. Key generation and message encapsulation are not performed via a single function call; for example, initialising the key involves six intermediate function calls before the keys are fully created and can be referenced via public and private key variables (see create_keypair function in wolfCrypt_ml_kem.h on GitHub. See appendix 1). This requires developers to have a deep understanding of both the API and underlying cryptographic principles, as delegating more responsibility to the developer necessitates greater attention to the correct use of cryptographic schemes.

botan3 is the only API under review that explicitly states a prior knowledge of cryptography is assumed before using the code. On the first page of the botan3 manual [43], the documentation specifies the expected level of cryptographic understanding and provides a list of recommended books for developers to consult in advance.

From the documentation of ml-kem and ml-dsa, it can be inferred that the API author assumes users of the code possess prior knowledge of cryptography. For example, the ml-kem documentation includes a brief explanation of how ML_KEM operates at a theoretical level and notes that the scheme is indistinguishable under chosen-ciphertext attacks (IND-CCA secure). However, neither API explicitly states that prior knowledge of cryptography is recommended before using the code.

ii. Misuse resistance

When it comes to misuse resistance, all the APIs under review with the exception of ml-kem and ml-dsa provide detailed guidance on how to mitigate side-channel attacks, which represent the most common risk arising from misuse, as identified in the case study by Lazar et al. [44].

liboqs provides basic safeguards against misuse through clear API boundaries, metadata, and build-time configuration. However, it lacks strong runtime misuse resistance e.g., constant-time checks, misuse-resistant defaults and error code management.

In mlkem-native, it is explicitly stated that the cryptographic code operates in constant time, thereby providing resistance against most side-channel attacks.

wolfCrypt provides an extensive list of error codes (more than 200), which can assist developers in avoiding insecure combinations of cryptographic primitives. However, it remains the developer's responsibility to make appropriate use of these error codes.

In botan3, the cryptographic primitives and schemes have been designed with side-channel prevention in mind. The API authors maintain an active log in the manual, detailing how the code mitigates specific attacks and providing recommendations on how to avoid them. For example, the API offers a secure memory allocator for use with the C++ Standard Library's vector data structure. botan3 also have some protection against unintended use, and will catch an exception in the code if wrong parameters are used for cryptographic schemes.

In ml-kem and ml-dsa, there is no misuse protection.

E. API code results

The code examples for key encapsulation and digital signatures, based on the official documentation provided are written in C++ 23. Since the APIs under review are intended for professional software developers or researchers who are likely to implement post-quantum cryptographic schemes within existing codebases, our code examples implement a structured template based on best practices in cryptographic protocol engineering to simulate a potential real-world scenario. This approach is based on the lectures by Kleppmann and Hugenroth on cryptographic protocol engineering in Cambridge, UK in 2025 [45].
The code template enforces a strict ordering of valid operations on an object by combining a finite-state machine with the typestate analysis pattern, which was proposed by Biffle for the Rust programming language in 2019 [46]. In essence, it defines a protocol that dictates the execution order of the program, which is verified at compile time. If a developer deviates from the protocol, the code will fail to compile. We implement one enforcement mechanism for key encapsulation and another for digital signatures. See table 5 for the API code metrics and time-on-task results for each API we implemented.

i. Code metrics

We used the Lizard static analysis tool to measure the average cyclomatic complexity across all functions within a single header file (e.g., mlkem-native.h), as well as the number of lines of code, excluding comments.

Additionally, we implemented a custom Halstead volume analyser, which takes a source file as input (See appendix 1). Given the cyclomatic complexity, lines of code, and Halstead volume values, we manually computed the maintainability index for each API under review.

ii. Time-on-task results

Prior to each time-on-task session, we had already downloaded the API, included it in our development environment via the CMake configuration, and prepared the typestate pattern. The purpose of the time-on-task measurements was to assess how long it took to implement ML-KEM and ML-DSA using only the documentation provided by the API. Issues related to the development environment, such as IDE configuration, API installation, or inclusion of necessary files, were excluded from the time-on-task measurements. Time-on-tasks longer than one hour are measured in approximate time, e.g. 1 hour and 47 minutes become 2 hours.

Table 5. API example code metrics

	liboqs	ml-kem native	wolfCrypt	botan3	ml-kem	ml-dsa
CRYSTALS-Kyber						
Cyclomatic complexity	1,1	1,2	3,2	1,7	1,4	N/A
Lines of code (total)[a]	105	131	302	167	139	N/A
Lines of code (crypto only)[a]	24	32	220	40	35	N/A
Halstead volume	5370,23	7093,77	17538,88	10246,11	8853,35	N/A
Maintainability index[b]	29,64	26,69	15,75	23,21	25,43	N/A
Time-on-task	2 hours	1 hour	15 hours	2 hours	30 minutes	N/A
CRYSTALS-Dilithium						
Cyclomatic complexity	1,1	N/A	3,4	1,9	N/A	1,2
Lines of code (total)[a]	95	N/A	238	189	N/A	148
Lines of code (crypto only)[a]	25	N/A	145	59	N/A	27
Halstead volume	4674,78	N/A	13860.82	8703,61	N/A	7298.69
Maintainability index[b]	31,01	N/A	18,70	22,50	N/A	25,45
Time-on-task	1 hour	N/A	2 hours	3 hours	N/A	23 minutes

a Not including comments.
b Uses lines of code (total) for LoC variable

V. FINAL EVALUATION AND CONCLUSIONS

Here we will compare the results for API design, code examples, documentation, and security for each API under review according to the criteria defined in the project statement:

- Difficulty in understanding and implementing the post-quantum cryptographic schemes.
- Quality of API documentation and examples of intended use.
- Security assumptions expected by API designers

First the metrics related to API design, as presented in Section *B* of the Results will be compared with the code metrics in table 5. After we evaluate the documentation and security metrics and finally, we conclude by identifying which APIs we will choose for further investigation—for instance, in the context of a master's thesis or another research project involving developer interviews and implementation tasks using the chosen APIs.

A. Difficulty in understanding and implementing the post-quantum cryptographic schemes

Implementing the APIs under review was a mixed experience. First we surveyed best practices to consider when implementing cryptography in code, and stumbled upon the typestate pattern, which enforces a protocool ordering on compile-time. At first this pattern was helpful since we first explicitly stated which ordering of operations we want for implementing ML-KEM and ML-DSA. As time went on, however, typestate got annoying to work with as we had to implement the same protocol over and over for e.g. ML-KEM. With that said, we believe typestate should be used more in practical cryptographic engineering.

i. Implementing ML-KEM

Implementing ML-KEM using the ml-kem and mlkem-native libraries was relatively straightforward, requiring only 35 and 32 lines of cryptographic code, respectively (see table 5). Each implementation primarily involved one-line function calls that populated locally defined variables as side-effects. These calls included four and two arguments for key generation, four and three arguments for key encapsulation, and three arguments each for key decapsulation, respectively.

By contrast, liboqs and botan3 employ a KEM encapsulation object to store and manage the function arguments for key encapsulation (one and three arguments) and decapsulation (one and two arguments). This design facilitates improved error handling and object lifetime management. As a result, it was easier to manage the cryptographic code, since the API object tracked initialised variables internally within the class and issued warnings when unsafe combinations of cryptographic primitives were used.

wolfCrypt was by far the most complex API to work with, requiring 220 lines of cryptographic code to implement ML-KEM. Both key encapsulation and decapsulation consist of five intermediate initialisation steps, each involving explicit memory allocation and deallocation for the keys. Interpreting return codes at each step was essential for debugging and understanding failure modes. The implementation of ML-KEM using wolfCrypt took over 15 hours.

As shown in table 5, the data support our assessment that the APIs under review—with the exception of wolfCrypt—were straightforward to implement. Although the maintainability index is relatively low across all code examples, the scores still fall within the range considered to indicate good maintainability (20–100). This can be attributed to the fact that key encapsulation and decapsulation are implemented as one-line function calls.

ii. Implementing ML-DSA

Our experience developing ML-DSA was simpler than with ML-KEM for two reasons: (1) the scheme required fewer implementation steps, and (2) having implemented ML-KEM first, we had already gained experience with the APIs under review.

Implementing ML-DSA using wolfCrypt was, once again, challenging—requiring 145 lines of cryptographic code—than with the other APIs. This is primarily due to the fact that each stage of initialising the ML-DSA scheme comprises multiple intermediate initialisation steps, including memory allocation and deallocation for the keys. However, the implementation was facilitated by prior experience gained from using wolfCrypt to implement the ML-KEM scheme.

As with the ML-KEM counterparts, signing and verification in liboqs and botan3 are concise: one-line function calls for liboqs and two lines for botan3. Both APIs employ objects to handle signing and verification operations. The signing object functions accept one and three arguments, while the verification object functions accept one and two arguments, respectively.

In the ml-dsa API, signing and verification are also performed using one-line function calls, but with locally defined variables populated as side effects.

An inspection of table 5 shows that the maintainability indices for the ML-DSA examples are comparable to those of their ML-KEM counterparts. This is to be expected, as the APIs under review exhibit similar design characteristics, and the ML-KEM and ML-DSA schemes are likely implemented by the same authors. Once again, wolfCrypt remains the exception, owing to the fact that the API involves multiple intermediary steps for signing and verification and requires explicit memory management.

Although botan3 and wolfCrypt are general-purpose cryptographic libraries, each comprising more than 100,000 lines of code, it was not necessary to understand unrelated components of the APIs in order to implement the ML-KEM and ML-DSA schemes. It sufficed to comprehend the overall design philosophy of the APIs and the specific implementation details of ML-KEM and ML-DSA.

B. Quality of API documentation and examples of intended use

We begin our evaluation of the documentation with a brief readability analysis. All the APIs under review received a Flesch Reading Ease score exceeding the recommended range of 40–50 for official documents, as established by various agencies in the United States. Scores between 50 and 30 correspond to college-level reading.

With the exception of ml-kem and ml-dsa, all APIs under review feature extensive documentation—hosted on GitHub, or in the case of wolfCrypt and botan3, in formal manuals—detailing how to use the API to implement ML-KEM and ML-DSA. This documentation includes sections on getting started, usage examples, security considerations, and known vulnerabilities. wolfCrypt and botan3 are general-purpose cryptographic APIs designed for multiple use cases, and their documentation is accordingly extensive (1.766 pages for wolfCrypt and 350 pages for botan3), with post-quantum cryptography (PQC) comprising only a small portion.

With the exception of wolfCrypt, the code examples are straightforward and relatively easy to incorporate into existing codebases. While wolfCrypt contains more examples than the other APIs under review, the authors have chosen to consolidate these examples into a few large header files, making it more difficult for software developers to identify which parts of the code are relevant. For instance, the client.h file in the server-to-client example contains 4.941 lines of code and implements multiple cryptographic schemes. In contrast, the ML-KEM example for liboqs comprises only 48 lines of code.

Nevertheless, an evaluation of the documentation reveals that the APIs are targeted at different groups of software developers requiring PQC. Accordingly, preferences regarding the documentation may vary depending on the intended audience. For example, mlkem-native provides detailed instructions on how to build and integrate a custom backend into the API—an approach likely to appeal to cryptographic engineers who require low-level access to the source code. While the other APIs also permit such interaction due to their open-source nature, they are not explicitly designed for this purpose. wolfCrypt, on the other hand, provides extensive documentation in its manual on memory management, and includes a chapter on best practices for embedded systems when generating private keys and using digital signatures (Chapter 12), which is particularly relevant for embedded systems engineers.

Selecting an API solely on the basis of its documentation is not feasible in our case, as liboqs, mlkem-native, wolfCrypt, and botan3 all provide comprehensive documentation and multiple PQC code examples to support usage.

C. Security assumptions expected by API designers

Our review of the documentation reveals that botan3 is the only API under review that explicitly states a prior knowledge of cryptography is assumed before using the API. The manual notes that the API is not recommended for use without a minimum background in cryptography and provides guidance on where software developers can acquire this knowledge. The remaining APIs do not explicitly require such a background, but this expectation can be inferred from the documentation. For example, the mlkem-native documentation offers detailed instructions on how to build custom backends using the API; wolfCrypt functions expose up to nine parameters and require multiple intermediate function calls to generate private and public keys for ML-KEM.

This delegation of responsibility to the software developer in mlkem-native and wolfCrypt suggests that a thorough understanding of cryptographic principles is assumed. liboqs explicitly states that the API is intended for academic research in post-quantum cryptography, thereby also implicitly assuming familiarity with cryptography.

Implicit assumptions of cryptographic knowledge in API documentations is likely one of several factors contributing to the difficulties software developers encounter when working with cryptographic APIs, and in particular PQC APIs that will stay in the research stage until the NIST PQC standardisation project finishes in 2027.

D. Final Choice of APIs

The APIs reviewed in this study address API usability challenges in different ways and target a variety of audiences, ranging from cryptographic engineers to academic researchers and embedded systems developers. Based on our practical implementation experience and the criteria defined for the project—implementation difficulty, documentation quality, and expected security knowledge—we synthesise our findings as follows:

- `ml-kem` and `ml-dsa` are minimal APIs that lack comprehensive documentation and robust error handling. Although their simplicity makes them easy to implement, they are neither intended for production use nor suitable for broader usability research. We therefore exclude them from consideration in future usability studies.
- `liboqs` uses objects to handle cryptographic functionality such as encapsulation, decapsulation, signing, verification, and error handling, thereby making correct API usage more straightforward. However, its academic focus and limited real-world applicability render it unsuitable for studies targeting professional developers.
- `botan3` likewise uses objects to manage cryptographic functionality and error handling. It provides a developer manual covering both API usage and relevant security considerations, such as strong secure defaults that cause the code to raise an exception if "unexpected" parameters are used—e.g., in the implementation of ML-DSA verification—and protection against side-channel attacks. Notably, `botan3` is the only API that explicitly states the level of cryptographic knowledge expected of the developer.
- `wolfCrypt` relies heavily on return codes and includes a header file with detailed descriptions of each cryptographic error. A comprehensive developer manual is provided, which documents each API function clearly and includes best practices for cryptographic engineering in real-time systems.

- `mlkem-native` is the only API among those reviewed that has been formally verified using a model checker. It presumes expert knowledge of cryptography, as it is designed to enable developers to implement custom backends for the ML-KEM scheme, with a focus on optimisations tailored to specific hardware architectures. The documentation includes example backends that can be adapted for such purposes.

The metrics we selected proved effective in excluding APIs not intended for production use—for example, those with inadequate documentation, insufficient error handling, or weak misuse resistance.

We recommend that future API usability studies—such as a master's thesis or a follow-up research project involving developer interviews and implementation tasks—focus on `mlkem-native`, `wolfCrypt`, and `botan3`.

References

1. Tahaei, M. & Vaniea, K. *A Survey on Developer-Centred Security* in *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)* (2019), 129–138.
2. Fischer, K. *et al. The Challenges of Bringing Cryptography from Research Papers to Products: Results from an Interview Study with Experts* in *33rd USENIX Security Symposium (USENIX Security 24)* (USENIX Association, Philadelphia, PA, Aug. 2024), 7213–7230. ISBN: 978-1-939133-44-1. https://www.usenix.org/conference/usenixsecurity24/presentation/fischer.
3. Klemmer, J. H. *et al. "Make Them Change it Every Week!": A Qualitative Exploration of Online Developer Advice on Usable and Secure Authentication* in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Association for Computing Machinery, Copenhagen, Denmark, 2023), 2740–2754. ISBN: 9798400700507. https://doi.org/10.1145/3576915.3623072.
4. Of Standards, N. I. & Technology. *Module-Lattice-Based Key-Encapsulation Mechanism Standard* tech. rep. Federal Information Processing Standards Publications (FIPS) 203, 13 August, 2024 (U.S. Department of Commerce, Washington, D.C., 2024).
5. Of Standards, N. I. & Technology. *Module-Lattice-Based Digital Signature Standard* tech. rep. Federal Information Processing Standards Publications (FIPS) 204, 13 August, 2024 (U.S. Department of Commerce, Washington, D.C., 2024).
6. Of Standards, N. I. & Technology. *Stateless Hash-Based Digital Signature Standard* tech. rep. Federal Information Processing Standards Publications (FIPS) 205, 13 August, 2024 (U.S. Department of Commerce, Washington, D.C., 2024).
7. Shor, P. *Algorithms for quantum computation: discrete logarithms and factoring* in *Proceedings 35th Annual Symposium on Foundations of Computer Science* (1994), 124–134.
8. Regev, O. An Efficient Quantum Factoring Algorithm. *J. ACM* **72.** ISSN: 0004-5411. https://doi.org/10.1145/3708471 (Jan. 2025).
9. Chevignard, C., Fouque, P.-A. & Schrottenloher, A. *Reducing the Number of Qubits in Quantum Factoring* Cryptology ePrint Archive, Paper 2024/222. 2024. https://eprint.iacr.org/2024/222.
10. Gambetta, J. *Development & Innovation and Roadmap* Roadmap (IBM (International Business Machines Corporation), 2024). https://www.ibm.com/quantum/assets/IBM_Quantum_Developmen_&_Innovation_Roadmap_Explainer_2024-Update.pdf.
11. Aasen, D. *et al. Roadmap to fault tolerant quantum computation using topological qubit arrays* 2025. arXiv: 2502.12252 [quant-ph]. https://arxiv.org/abs/2502.12252.
12. Of Standards, N. I. & Technology. *Transition to Post-Quantum Cryptography Standards* tech. rep. NIST IR 8547 ipd (U.S. Department of Commerce, Washington, D.C., 2024).
13. Centre, N. C. S. *Timelines for migration to post-quantum cryptography* Roadmap (National Cyber Security Centre, 2025). https://www.ncsc.gov.uk/guidance/pqc-migration-timelines.
14. Für Sicherheit in der Informationstechnik, B. *Securing Tomorrow, Today: Transitioning to Post-Quantum Cryptography* Roadmap (European Union, 2024). https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Crypto/PQC-joint-statement.pdf?__blob=publicationFile&v=5.
15. Bos, J. *et al. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM* in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)* (2018), 353–367.
16. Ducas, L. *et al.* CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems* **2018,** 238–268. https://tches.iacr.org/index.php/TCHES/article/view/839 (Feb. 2018).
17. Bernstein, D. J. *et al. The SPHINCS+ Signature Framework* in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (Association for Computing Machinery, London, United Kingdom, 2019), 2129–2146. ISBN: 9781450367479. https://doi.org/10.1145/3319535.3363229.
18. Nielsen, J. *10 Usability Heuristics for User Interface Design* Documentation (Nielsen Norman Group, 1994). https://www.nngroup.com/articles/ten-usability-heuristics/.
19. Myers, B. A. & Stylos, J. Improving API usability. *Commun. ACM* **59,** 62–69. ISSN: 0001-0782. https://doi.org/10.1145/2896587 (May 2016).
20. Scheller, T. & Kühn, E. Automated measurement of API usability: The API Concepts Framework. *Information and Software Technology* **61,** 145–162. ISSN: 0950-5849. https://www.sciencedirect.com/science/article/pii/S0950584915000178 (2015).
21. Martin-Haugh, S. *Lizard code complexity analyser* Documentation (CERN, 2017). https://github.com/terryyin/lizard.
22. Jelveh, R. *ctag* Documentation (Universal-ctags organization). https://github.com/universal-ctags/ctags.
23. Tyler, S. *Readability Analyzer* Web based tool (datayze). https://datayze.com/readability-analyzer.
24. Mikejo5000 *et al. Code metrics values* Documentation (Microsoft Corporation, 2023). https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2022.
25. McCabe, T. A Complexity Measure. *IEEE Transactions on Software Engineering* **SE-2,** 308–320 (1976).
26. Halstead, M. H. *Elements of Software Science (Operating and programming systems series)* ISBN: 0444002057 (Elsevier Science Inc., USA, 1977).
27. Oman, P. & Hagemeister, J. *Metrics for assessing a software system's maintainability* in *Proceedings Conference on Software Maintenance 1992* (1992), 337–344.
28. Kincaid, J. P., Fishburne, J., P., R., Rogers, R. L. & Chissom, B. S. Derivation of New Readability Formulas (Automated Readability Index, Fog Count and Flesch Reading Ease Formula) for Navy Enlisted Personnel. ISSN: 0001-0782. https://apps.dtic.mil/sti/citations/ADA006655 (1975).
29. Google. *Tink* Cryptography (). https://developers.google.com/tink.
30. Amazon. *AWS-lc* Cryptography (). https://github.com/aws/aws-lc.
31. Project, O. Q. S. *liboqs* Cryptography (Linux Foundation). https://github.com/open-quantum-safe/liboqs.
32. Project, P. C. P. *liboqs-cpp* Cryptography (Linux Foundation). https://github.com/pq-code-package/mlkem-native.
33. Ouska, T. *wolfCrypt* Cryptography (wolfSSL Inc.). https://github.com/wolfSSL/wolfssl.
34. Lloyd, J. *botan3* Cryptography (randombit). https://github.com/randombit/botan/tree/master.

35. Roy, A. *ml-kem library* Cryptography (). https://github.com/itzmeanjan/ml-kem.

36. Roy, A. *ml-dsa library* Cryptography (). https://github.com/itzmeanjan/ml-dsa.

37. Project, O. Q. S. *liboqs-cpp* Cryptography (Linux Foundation). https://github.com/open-quantum-safe/liboqs-cpp/blob/main/README.md.

38. Bernstein, D. J. *et al.* KyberSlash: Exploiting secret-dependent division timings in Kyber implementations, 1–30. https://kyberslash.cr.yp.to/papers.html (2025).

39. Purnal, A. *ml-dsa library* Cryptanalysis (KU Leuven, 2024). https://github.com/antoonpurnal/clangover?tab=readme-ov-file.

40. Van Rossum, G. *PEP 8* Documentation (Python Software Foundation, 2025). https://peps.python.org/pep-0008/.

41. Of Standards, N. I. & Technology. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions* tech. rep. Federal Information Processing Standards Publications (FIPS) 202, 4 August 2015 (U.S. Department of Commerce, Washington, D.C., 2015).

42. Ouska, T. *wolfCrypt manual* Cryptography (wolfSSL Inc.). https://www.wolfssl.com/documentation/manuals/wolfssl/wolfSSL-Manual.pdf.

43. Lloyd, J. *botan3 manual* Cryptography (randombit). https://botan.randombit.net/handbook/botan.pdf.

44. Lazar, D., Chen, H., Wang, X. & Zeldovich, N. *Why does cryptographic software fail? a case study and open problems* in *Proceedings of 5th Asia-Pacific Workshop on Systems* (Association for Computing Machinery, Beijing, China, 2014). ISBN: 9781450330244. https://doi.org/10.1145/2637166.2637237.

45. Kleppmann, M. & Hugenroth, D. *Cryptography and Protocol Engineering* Lecture notes (University of Cambridge, 2025). https://www.cl.cam.ac.uk/teaching/2425/P79/p79-notes.pdf.

46. Biffle, C. L. *The Typestate Pattern in Rust* Documentation (Python Software Foundation, 2019). https://cliffle.com/blog/rust-typestate/.

Appendix 1. API code example implementations

Find code examples here:
https://github.itu.dk/jave/Post_quantum_cryptography_research_project

Appendix 2. Use of Generative AI

In this paper, we used ChatGPT (GPT-4-turbo) to improve clarity and grammatical accuracy in the writing using prompts similar to the following:

"Correct this text using British English: for clarity, consistency, and grammatical accuracy."

followed by the paragraph of text to be revised.

The following text at the end of Implementing ML-KEM in the final evaluation and conclusions section was revised using the prompt above:

Looking at table 5 the data supports our assessment that the APIs under review, with the exception of wolfCrypt were easy to implement. Even though the maintainability index is on the lower end for all code examples, the code still have a good maintainability score (20 - 100 represents good maintainability). This can be attributed to the fact that key encapsulation and decapsulation are one-line function calls.

Into the following:

As shown in Table 5, the data support our assessment that the APIs under review—with the exception of wolfCrypt—were straightforward to implement. Although the maintainability index is relatively low across all code examples, the scores still fall within the range considered to indicate good maintainability (20–100). This can be attributed to the fact that key encapsulation and decapsulation are implemented as one-line function calls.