When Security Meets Usability: An Empirical Investigation of Post-Quantum Cryptography APIs

Marthin Toruan
Royal Melbourne Institute of Technology
s4075803@student.rmit.edu.au

R.D.N. Shakya
University of Moratuwa
shakyardn.26@uom.lk

Samuel Tseitkin
ExeQuantum
sam@exequantum.com

Raymond K. Zhao
ExeQuantum
raymond@exequantum.com

Nalin Arachchilage
Royal Melbourne Institute of Technology
nalin.arachchilage@rmit.edu.au

***Abstract*—Advances in quantum computing increasingly threaten the security and privacy of data protected by current cryptosystems, particularly those relying on public-key cryptography. In response, the international cybersecurity community has prioritized the implementation of Post-Quantum Cryptography (PQC), a new cryptographic standard designed to resist quantum attacks while operating on classical computers. The National Institute of Standards and Technology (NIST) has already standardized several PQC algorithms and plans to deprecate classical asymmetric schemes, such as RSA and ECDSA, by 2035. Despite this urgency, PQC adoption remains slow, often due to limited developer expertise. Application Programming Interfaces (APIs) are intended to bridge this gap, yet prior research on classical security APIs demonstrates that poor usability of cryptographic APIs can lead developers to introduce vulnerabilities during implementation of the applications, a risk amplified by the novelty and complexity of PQC. To date, the usability of PQC APIs has not been systematically studied. This research presents an empirical evaluation of the usability of the PQC APIs, observing how developers interact with APIs and documentation during software development tasks. The study identifies cognitive factors that influence the developer's performance when working with PQC primitives with minimal onboarding. The findings highlight opportunities across the PQC ecosystem to improve developer-facing guidance, terminology alignment, and workflow examples to better support non-specialists.**

I. Introduction

Quantum computing (QC) presents a paradigm shift in computational power, most notably in its ability to solve cryptographic problems that are computationally infeasible for classical machines [1]. Central to this advantage is Shor's algorithm [2], [3], which poses a direct threat to widely used public-key encryption standards like Rivest-Shamir-Adleman (RSA) [4], [5]. To understand the magnitude of this threat, one can compare a classical supercomputer to a burglar who attempts to crack a lock by checking one number at a time–a process that could take billions of years for 2048-bit encryption. In contrast, a quantum computer takes advantage of the principles of quantum mechanics to evaluate vast combinations simultaneously, theoretically picking the same lock in a matter of hours [6].

This danger has moved beyond abstract theory, as active research continues to reduce the resources required to mount such attacks. For example, recent optimizations of windowed arithmetic circuits have reduced the total gate count for factoring by approximately 3.4% [7]. Although this may seem small, it certainly lowers the barrier to practical quantum cryptanalysis, indicating that widely deployed algorithms will eventually become vulnerable [8].

The implications of this vulnerability extend far beyond digital data theft; they introduce serious physical risks. If encryption is broken, attackers could bypass security protocols in Operational Technology (OT) and National Critical Infrastructure. In a practical scenario, this capability would allow adversaries to manipulate valve controls in water treatment facilities, trigger shutdowns in national energy grids, disrupt railway transportation networks, or sabotage manufacturing production lines [9].

Compounding this physical risk is the immediate strategic threat known as "Harvest Now, Decrypt Later" (HNDL) [10], [11]. Even before a fully capable quantum computer exists, attackers are already intercepting and storing large volumes of encrypted traffic from critical infrastructure systems such as water, energy, healthcare, transportation, and manufacturing, with the explicit intention of decrypting it retrospectively once quantum capabilities mature. For example, an adversary monitoring encrypted telemetry from a national power grid can archive operational logs and configuration updates that are currently protected by classical public-key cryptography. When a cryptographically capable quantum computer emerges, the attacker could decrypt the stored data, reconstruct the state of the system, and uncover structural weaknesses. Thus, the impact of HNDL attacks materializes not during interception but when future decryption becomes technically possible.

In response to these emerging threats, the National Institute of Standards and Technology (NIST) initiated its Post-

Symposium on Usable Security and Privacy (USEC) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-07-7
https://dx.doi.org/10.14722/usec.2026.23076
www.ndss-symposium.org, https://www.usablesecurity.net/USEC/

Quantum Cryptography (PQC) effort in 2016, calling for the development of quantum-resistant classical algorithms. By 2025, this process had produced five standardized schemes [12]. In addition, NIST's PQC transition plan [13] specifies a phased deprecation schedule for vulnerable algorithms such as RSA and the Elliptic Curve Diffie-Hellman, including a prohibition on their use after 2035. Beyond the NIST transition plan, several international initiatives further underscore the urgency of migrating to post-quantum cryptography, including the European Union's coordinated implementation roadmap[14], which outlines a phased, cross-sector transition strategy and highlights the practical challenges of deploying PQC in real-world software systems.

Substantial work has also been undertaken to facilitate migration to PQC. This includes a systematic literature review on migration to PQC by Näther et al. [15], studies on network adoption rates [16], and the creation of PQC libraries for developers such as PQClean [17] and Liboqs [18]. However, while PQC research has made substantial strides in algorithmic performance and protocol integration, limited research has investigated whether general software developers can correctly and securely implement these algorithms without security knowledge or cryptographic expertise.

Previous studies demonstrate that the usability of cryptographic libraries has a major influence on the security outcomes of applications [19], [20], [21]. When documentation is poor or APIs are unintuitive, developers are prone to misusing standard tools like Secure Sockets Layer (SSL), authentication mechanisms, and symmetric-key encryption. These usability issues often result in misuse in the application, such as the use of unsafe default settings or incorrect parameter configuration in classical cryptographic algorithms. To address these issues, Green [22] and Schmüser [23] have emphasized the importance of human-centered cryptography. However, at the time of writing, no study has empirically analyzed the usability of PQC APIs from a developer experience perspective.

Consequently, there is an urgent need for a systematic examination of the usability of the PQC API. Enhancing usability is critical for minimizing implementation errors of applications and promoting secure software development practices.

To address this gap, this research employed a moderated remote usability testing protocol facilitated via video conferencing software (specifically, Microsoft Teams). This format allowed participants to operate within their natural development environments while sharing their screens for real-time observation. Using the Cognitive Dimensions Framework (CDF) [24], [25] as an analytical lens, the study evaluated the interaction patterns of a diverse group of developers. The study compared two accessible APIs that represent distinct architectural models. Participants were tasked with implementing PQC algorithms in a simulated client-server environment, followed by a mixed-methods analysis to isolate the root causes of usability friction.

This study examines the behavior of non-specialized developers and the experience of implementation under minimal manual intervention. It does not evaluate the security guarantees, cryptographic correctness, operational posture, or maturity of the deployment of the APIs examined. All findings relate solely to the experience of the developers and the usability of the APIs under experimental conditions.

Specifically, this study seeks to explore key aspects of developer interaction with Post-Quantum Cryptography APIs by addressing the following research questions:

RQ1: What common misuses and usability barriers do developers encounter when utilizing PQC APIs?

RQ2: How do the integration challenges differ between endpoint-based PQC APIs and local library PQC API?

RQ3: What usability and guidance improvements are needed to support secure implementation?

The rest of the paper is outlined as follows: Section II, "Literature Review", where we analyze previous studies relevant to our research questions. Section III, "Research Methodology" details our data collection and analysis methods. Section IV, "Results", where findings are presented with the aid of graphs and tables. Section V, "Discussion and Evaluation", which interprets and discusses the findings. In the concluding Section VII, "Conclusion", we summarize the insights gained from the experiment and suggest ways they could enhance future research.

II. Literature Review

This literature review first establishes the urgent need to migrate to PQC. Then it examines the challenges involved in this migration and identifies the stakeholders responsible for addressing them. Then, the review investigates how usability and documentation problems in PQC APIs can lead to insecure implementation behavior during application development. Finally, it explores the evaluation methodologies employed in related work to identify these issues and improve API design, including documentation, code examples, abstraction, and the inputs and outputs of functions.

A. Importance of migrating to PQC

Information security is entering a period of major change because a powerful quantum computer could break the classical encryption systems that protect today's digital world. Specifically, Shor's algorithm [2], [3], a quantum computing algorithm, is theoretically capable of solving integer factorization and discrete logarithm problems in polynomial time. This capability would render the most widely used public-key cryptography algorithms, such as Elliptic Curve Diffie-Hellman (ECDH), Elliptic Curve Menezes-Qu-Vanstone (ECMQV), and Rivest-Shamir-Adleman (RSA), completely insecure [8], [13]. Consequently, the NIST transition plan mandates that these algorithms will be deprecated by 2030 and fully disallowed after 2035 [13].

This threat is made more urgent by HNDL attacks, where encrypted data are harvested today to be decrypted by a future quantum computer [10], [11]. This is no longer a purely theoretical concern, as active research is consistently reducing the practical resource requirements for such an attack. For instance, recent optimizations of the windowed arithmetic

circuits required for Shor's algorithm [2], [3] have been shown to reduce the total gate count for factoring by up to 3.4%, incrementally lowering the barrier to a practical attack on gate-based quantum computers [7].

To mitigate this threat, the NIST standardization of algorithms like ML-KEM (FIPS 203)[26] introduces a paradigm shift based on the 'KEM/DEM' framework [27], [28]. In this hybrid approach, the KEM asymmetrically derives a shared secret, which is then utilized by a Data Encryption Mechanism (DEM) for symmetric message encryption. Consequently, the integration of these primitives shifts the burden from understanding mathematical theory to mastering practical software implementation [29].

According to Näther et al. [15], this migration involves four key phases: diagnosis, planning, execution, and maintenance. While various roles such as Migration Managers and Security Experts are involved, the software developer emerges as the pivotal figure, holding primary responsibility for the critical execution and maintenance phases.

However, the central role of the developer is complicated by the intersection of limited domain expertise and the inherent complexity of the PQC algorithms themselves [15]. Implementing PQC is a non-trivial task; as detailed in NIST guidelines SP.800.227 and FIPS 203 [30], [26], a secure implementation must account for numerous sophisticated factors. These include managing secure handshake protocols, ensuring cryptographically secure random key generation, handling lattice set problems, and preventing side-channel attacks. Although addressing these factors is essential for compliance, this increased complexity often negatively impacts usability [31]. This tension between robust security and API usability remains an open research problem [32], [33], creating substantial friction for developers tasked with securing the next generation of digital systems.

B. Usability Issues of Security API

Historically, cryptographic APIs that exhibit limited usability have demonstrated substantial vulnerability to misuse by developers, resulting in major security flaws in applications [19], [20], [21]. This underlines the urgent requirement for empirical evaluation of the developer experience with emerging PQC APIs. Without usability evaluation, there will be risks of repeating past mistakes, where poor usability undermines the very security the migration aims to provide [34], [35], [36], [37], [38].

Although the mathematical integrity of the new PQC standards is critical, history has demonstrated that mathematical security alone is insufficient to ensure real-world security [39], [40]. The cryptographic community has repeatedly learned this lesson from the failures of the classical cryptography API [39]. The major vulnerabilities, such as the "Heartbleed" bug in OpenSSL [40] or the Apple "goto fail" vulnerability [39], were not failures of the cryptographic algorithms themselves, but rather API implementation failures caused by developer error, complex code, and poor documentation [39].

Research in the usable security domain has consistently demonstrated that if an API is difficult to use, developers will make critical security errors in applications [22], [20], [21]. Developers, who are often not security experts, may misconfigure parameters, mishandle sensitive data like secret keys, or fail to implement necessary procedures such as error handling or signature verification [22].

However, these issues do not emerge as isolated single incidents; rather, they reflect recurring and well-documented misuse patterns. Frequent errors include the selection of inappropriate parameters, such as insecure modes such as ECB [41], [42] and improper key handling practices, including hard-coding secrets or relying on unsafe default configurations [20]. Developers also commonly omit essential security steps, such as certificate verification [43], [19]. For example, confusion and ambiguity in API usage have contributed to the widespread disabling of SSL / TLS verification in mobile applications. These failures often arise from inadequate documentation, the absence of robust secure-by-example guidance, an increased cognitive burden during implementation, and a reliance on insecure code snippets found through online forums and repositories [22], [44], [38], [45].

As the cryptography field advances toward PQC, these established misuse patterns pose even more risks. PQC introduces additional complexity that may further strain developer comprehension, including large key sizes, a novel key exchange model, inconsistent terminology, and the need for crypto-agility [46]. The need to implement secure hybrid schemes that combine classic encryption with PQC also introduces new risks of broken mechanisms or incorrect encapsulation logic [47], [48].

Furthermore, existing research highlights how even well-designed algorithms can fail in practice [22], [24], most usability studies have focused on traditional libraries like OpenSSL [49], BouncyCastle [50], and various Java or Python APIs [51], [20]. This situation creates a critical research gap, as there has been little to no empirical evaluation of post-quantum-specific APIs. Therefore, this research aims to fill this gap by applying established usability evaluation methods to PQC APIs to identify usability issues that could lead to security vulnerabilities in applications.

C. Related Work

Research into the usability of cryptographic APIs has evolved from casual observation to structured and empirical evaluation. Acar et al. [20] conducted a landmark quantitative study involving 256 Python developers to compare five libraries (PyCrypto, M2Crypto, cryptography.io, Keyczar and PyNaCl). Their methodology combined a controlled experiment with functional analysis, revealing a complex relationship between API design and security. Although comprehensive documentation facilitated functional correctness, it often led to insecure implementations, whereas overly simple APIs caused functional failures. Crucially, their demographic analysis noted that general programming familiarity did not correlate with

security success; rather, specific security knowledge was the determining factor.

Complementing this quantitative approach, Wijayarathna and Arachchilage argue that identifying the root causes of insecure utilization of API requires qualitative depth. Through systematic literature reviews [52], [24], they evaluated various methodologies–including heuristic evaluations and API walkthrough–and concluded that empirical user studies are essential to reveal real-world developer experiences (DevX). To standardize this analysis, they adapted Clarke's CDF [37], expanding it from 12 to 15 dimensions to specifically address security contexts [53].

Applying this adapted CDF methodology, Wijayarathna and Arachchilage further investigated specific usability flaws across multiple environments. In their evaluation of the BouncyCastle API, they utilized the think-aloud protocol to identify that low-level parameters in the `SCrypt.generate()` method confused non-experts [50]. Similarly, their assessment of the Google Authentication API revealed that misleading abstraction levels forced developers to rely on insecure third-party code snippets [21]. Furthermore, their study of the Java Secure Socket Extension (JSSE) API linked low penetrability and uninformative error messages to vulnerable TLS implementations [51]. These studies collectively demonstrate that when secure APIs are difficult to learn, developers inevitably revert to simpler, less secure alternatives.

In the domain of PQC, usability challenges are compounded by new cryptographic primitives. Zeier et al. [54] addressed the complexity of stateful hash-based signature schemes (e.g., XMSS) by proposing "EasySigner", a crypto-agile API designed to abstract state management. Although their user study demonstrated high functional success, it highlighted a "transparency paradox": the effective abstraction left participants unaware that they were using a stateful scheme. This lack of awareness poses a risk, as developers might inadvertently compromise keys through external actions such as virtual machine cloning, underscoring the need for evaluation methods that assess both API usability and developer awareness.

III. Research Methodology

This research aims to identify security vulnerabilities in applications that arise from usability issues that developers encounter when implementing PQC algorithms. As discussed in Section II-C, we adopted an empirical user study methodology based on the CDF as summarized in Fig. 1 as modified by Wijayarathna and Arachchilage [24]. This approach was selected to facilitate a detailed analysis of PQC APIs and to identify specific cognitive hurdles that hinder secure implementation. Aligning with observations by Acar et al. [20] with respect to security experience and Näther et al.'s insights on PQC migration [15], we recruited a diverse set of participants ranging from software engineers and developers to IT students. This diversity allows the study to explore how varying levels of expertise influence security-relevant outcomes. Special emphasis is placed on developers without specialized cybersecurity training, as they are more likely to introduce implementation flaws in applications when interacting with complex or insufficiently supportive PQC APIs and libraries.

Fig. 1. Cognitive Dimensions Framework.

To ensure that the findings reflect real-world programming scenarios, we employed a moderated remote usability study design. This approach maximized ecological validity by allowing participants to use their familiar Integrated Development Environments (IDEs) and external resources, such as search engines and AI assistants (e.g., ChatGPT[1], Stack Overflow[2]). This was an intentional design choice to preserve ecological validity and to observe realistic developer–API interaction in natural development settings. To ensure consistency across sessions and to minimize environment-induced variability, all participants used the same programming language, identical task instructions, standardized skeleton code, and fixed versions of the evaluated PQC APIs. Environment requirements and dependencies were communicated in advance and verified at the start of each session, while live screen sharing and moderation allowed immediate resolution of environment-related issues. Consequently, observed differences are attributable to API usability and developer behavior rather than tooling or configuration discrepancies.

Data from pilot tests revealed that the cognitive load was excessive for a within-subjects approach. Therefore, the study was refined to a between-subjects model [55] where participants engaged with only one API. This design isolated the quantitative and qualitative analysis to a single interaction, removing the risk of learning bias.

The core assessment tasks were designed around the practical scenario of building a secure client-server communication protocol in compliance with NIST recommendations. Participants were required to implement two foundational PQC functions: a KEM, such as ML-KEM, and a DSA, such as ML-DSA. These tasks were presented sequentially to construct a layered security model: beginning with KEM to establish

[1]https://chatgpt.com/
[2]https://stackoverflow.com/

confidentiality, followed by symmetric-key encryption, and finally introducing DSA to prevent on-path attackers and ensure end-to-end integrity in the application.

The overall study procedure is illustrated in Fig. 2. Before data collection, the study protocol was reviewed and approved by the university ethics committee. Following this approval, the process went on to recruit software engineers, developers, and IT students, who underwent a screening questionnaire to determine eligibility. Qualified participants received an informed consent form via email and were scheduled for a remote session. The session started with a briefing on general knowledge, followed by specific task guidelines(see appendix B). The participants then executed the programming task using the "think-aloud" protocol [56]. Once the moderator verified the completion of the task, the session concluded with a post-task questionnaire adapted from Wijayarathna et al. [24]. All data collected was then synthesized and mapped to the 15 Cognitive Dimensions to systematically categorize how API design features impact developer usability and software security.

A. Programming Language and Chosen PQC APIs

To minimize cognitive load during these tasks, the study utilized Python, selected for its readability and popularity in introductory programming [57]. Participants were provided with a skeleton script containing helper functions for network socket communication, allowing them to focus strictly on the cryptographic implementation. Two PQC libraries– the endpoint-based PQ-Sandbox [58] and the local library QuantCrypt [59]–were selected solely based on public availability, PyPI accessibility, and completeness of public documentation. No API was selected due to institutional affiliation or endorsement. These criteria minimized installation barriers, serving as a methodological control to isolate usability issues inherent in the API design rather than environmental configuration. The PQ-Sandbox API [58] is a research prototype provided solely for academic evaluation. It is distinct from any production system and is intentionally simplified for experimentation. The API exposes only the cryptographic functions necessary for the study and should not be interpreted as a commercial or deployment-ready environment.

B. Task design

The task design was focused on the core functions of the PQC algorithms, namely KEM and DSA. We follow the NIST recommendations for secure KEM implementations [30]. These guidelines served as the standard for task design and implementation evaluation. Participants were given a set of materials to complete the task: two Python scripts containing skeleton code, API documentation, and a task instruction sheet.

The scenario placed the participant in the role of a software engineer at a financial technology company. They were assigned the task of developing a secure chat application project to prevent quantum computer attacks. The two scripts represented the server and client components of this application, which were assumed to operate on an insecure network. The participant's objective was to implement the PQ-Sandbox API and QuantCrypt API into this skeleton code to enable secure communication between the server and client on the insecure channel.

Task completion was measured against several goals:

- **KEM**: The first task required participants to use ML-KEM to establish a shared secret between the server and the client. The fundamental security goal was to achieve confidentiality against a passive eavesdropper. By design, the KEM allows both parties to agree on a secret value without ever transmitting that secret directly across the network, thus protecting it from being intercepted.
- **Symmetric-key Encryption**: In the second task, participants were instructed to use the shared secret generated in Task 1 as a session key for a symmetric-key encryption algorithm such as the Advanced Encryption Standard (AES). This task demonstrates the "hybrid encryption" model, where the computationally expensive KEM is used only to establish the key, and the efficient symmetric-key cipher is used to protect the bulk of the application data.
- **DSA for Handshake Authentication**: The third task directly addressed the on-path attackers vulnerability by introducing ML-DSA to provide server authentication. Participants had to modify the initial KEM handshake, requiring the server to use its long-term private key to sign a key component of the key exchange (such as the KEM ciphertext it generates). By verifying this signature with the server's trusted public key, the client can cryptographically confirm that it is communicating with the genuine server, not an impostor. This step adds the crucial security properties of authentication (proving the server's identity) and non-repudiation (providing undeniable proof that the server participated in that specific handshake).
- **DSA for Message Exchange**: the final task was to implement message integrity and mutual authentication for the data-in-transit. Participants were asked to use ML-DSA on both the server and the client to sign every application message exchanged over the encrypted channel. This ensures that no message is altered by an attacker after it has been signed and confirms that both parties are who they claim to be throughout the entire session. Although Task 3 secured the handshake, it did not protect subsequent application data from being tampered with (even if it is encrypted).

C. Participant recruitment

Participants were recruited via posters with a QR code linking to a page outlining the study's objectives and tasks, distributed through the researchers' professional networks. Recruitment primarily attracted university students, staff, and professional developers from diverse sectors. Interested individuals completed a screening questionnaire assessing web API knowledge and programming proficiency to ensure foundational skills needed to prevent data noise. Eligible participants provided contact information to schedule sessions, while

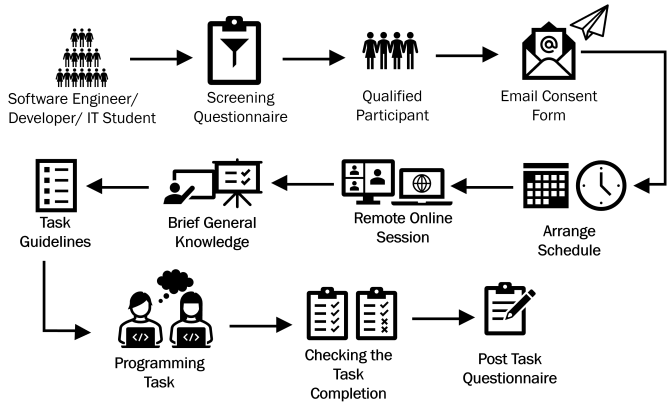

Fig. 2. Study Procedure Diagram.

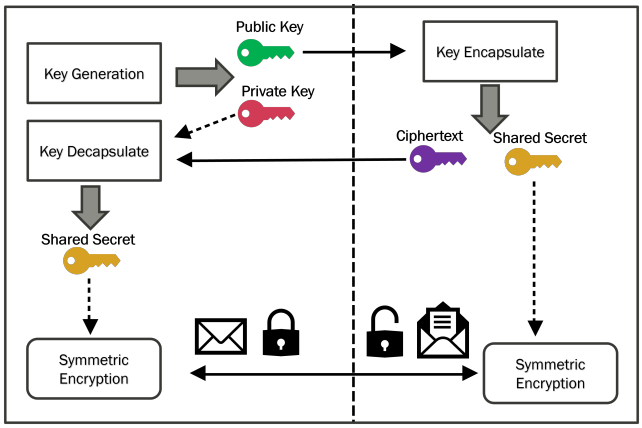

Fig. 3. Key Encapsulation Mechanism and Symmetric Encryption Workflow.

ineligible individuals were not asked for further details, protecting privacy and providing a clear rationale for ineligibility.

Eligible participants then received a Participant Information and Consent Form via email and, upon agreement, a researcher scheduled the experimental session based on their availability. A Microsoft Teams link confirmed the session.

D. The pilot study

Before the main study, a preliminary pilot study was conducted with three participants. This small-scale trial was essential to ensure that the study ran smoothly and focused on testing the task design, rather than obtaining the final results. This process helped us determine the practicality, estimated duration, cost, and any unforeseen issues. These practical insights allowed us to refine our methods and proceed with the main study with greater confidence in our plan.

The pilot study revealed two critical challenges: technical setup hurdles and time constraints. First, to address difficulties with IDE configuration and data transfer, we refined the skeleton code by providing pre-built classes and included detailed environment requirements in the preparation email. This allowed participants to focus on core logic rather than troubleshooting. Second, because the average task duration exceeded 90 minutes, we assigned only one API per participant. Additionally, if a session surpassed two hours, we explicitly asked participants if they wished to continue. These strategies mitigated participant fatigue and preserved the quality of our research results.

E. Study procedure

Following the pilot study and task refinements, we conducted the main usability study. The participants received a setup guide and a short video introducing the think-aloud method [56].

The study workflow is shown in Fig. 2. The protocol was approved by the University Ethics Committee before data collection began. The recruitment targeted software engineers, developers, lecturers, and IT students. Eligible participants, identified through a screening questionnaire, received an informed consent form and were scheduled for a remote session.

Each session opened with a brief introduction to the purpose of the study, background concepts, and task instructions. The participants then completed a programming task while verbalizing their thoughts using the think-aloud method [56]. After the moderator confirmed the completion of the task, participants completed a post-task questionnaire adapted from Wijayarathna et al. [24]. All data were then mapped to the 15 Cognitive Dimensions to evaluate how API design influences usability and software security.

We conducted the study remotely, recording screen and audio to capture detailed navigation and reasoning processes often missed by surveys alone [60], [55]. Sessions began with a briefing on KEM and DSA concepts (e.g., Fig. 3) to ensure a common baseline before participants received the skeleton code. Participants performed tasks using a think-aloud protocol under passive moderation, followed by a post-study questionnaire. We prioritized participant comfort, explicitly allowing pauses or withdrawal at any time.

F. Evaluation of PQC Usability

The evaluation was based on the data collected, which included performance metrics, screen recordings, transcripts of the participants' "thinking-aloud" verbalization, and the post-task questionnaire. Quantitative analysis focused on performance metrics, including task completion rate, task completion time, and error rate of participants' implementations. The completion rate of the task was calculated as the percentage of participants who completed each task. Among the participants' implementations, the error rate was determined by counting the number of incorrect outputs, API misuse events, and runtime failures for each task. To compare performance differences between the two API groups (PQ-Sandbox and QuantCrypt), unpaired two-sample t-tests were used [61].

The qualitative data were analyzed using a thematic coding approach guided by the CDF [25]. The first author performed the primary coding of all qualitative materials, including think-aloud transcripts and post-task questionnaire responses. A second author independently reviewed the coded data and the evolving codebook to validate interpretations and to identify

TABLE I
PARTICIPANT DEMOGRAPHICS.

Background	Category	N
Software Development Experience (SDE)	No Experience	3
	IT student (Entry Level Developer)	4
	<3 years (Beginner Developer)	3
	≥3 years (Expert Developer)	6
Python Experience (PE)	No experience	0
	Less than 1 Year	5
	1 - 3 Years	8
	3 - 5 Years	0
	> 5 Years	3
Cybersecurity Experience (CE)	No	2
	Basic Knowledge	11
	Expert Cybersecurity	3

TABLE II
TASK PERFORMANCE TIME METRICS.

API	Mean (min)	SD (min)	n	Comp. Rate
Task 1				
QuantCrypt	39.38	15.52	8	100%
PQ-Sandbox	65.38	18.02	8	100%
Task 2				
QuantCrypt	41.43	6.88	7	87.5%
PQ-Sandbox	25.43	7.72	7	87.5%
Task 3				
QuantCrypt	25.60	5.77	5	62.5%
PQ-Sandbox	23.25	6.02	4	50%
Task 4				
QuantCrypt	–	–	–	0%
PQ-Sandbox	–	–	–	0%

potential inconsistencies. Any discrepancies were discussed and resolved through consensus, resulting in iterative refinement of code definitions and thematic boundaries. This validation process was employed to mitigate individual researcher bias and to strengthen analytical rigor. To preserve fidelity to participants' perspectives, all quoted statements are reported verbatim, including original grammatical errors and typographical inconsistencies.

IV. RESULTS

A. Participant Demographics

A total of 16 participants were recruited for the study and randomly assigned to two independent, between-subjects groups: PQ-Sandbox ($N = 8$) and QuantCrypt ($N = 8$). One participant in the PQ-Sandbox group did not complete the post-task questionnaire. As summarized in Table I, the participants possessed varying levels of technical expertise. Regarding software development, the group included both beginners and professionals; while 7 participants were students or had no prior experience, the majority (9) were professional developers, with 6 possessing over three years of experience. Furthermore, all participants had at least some familiarity with Python, with half of the group ($N = 8$) falling into the 1–3 year experience range. In terms of knowledge in the cybersecurity domain, the sample consisted predominantly of individuals with basic knowledge ($N = 11$), while only three identified as experts and two reported no previous qualification or experience.

B. Task Performance

Table II summarizes the performance data from participants' initial API interactions. Results are grouped by PQC API and include the mean and standard deviation (in minutes), as well as the completion rate. While most participants successfully completed Task 1, failure rates increased for Tasks 2 and 3. Notably, no participants were able to complete Task 4 within the allotted time.

TABLE III
PARTICIPANT FINAL CODE ANALYSIS.

API	ID	KEM Decap Leak	DSA Priv Leak	Shared Sec Leak	Missing Encap Check	Missing Decap Check	DSA Verify	Key Not Destroyed
QC	P1				✗	✗	✓	✗
	P2				✗	✗		✗
	P5	●		●	✗	✗	✗	✗
	P6				✗	✗	✗	✗
	P9				✗	✗	✗	✗
	P11				✗	✗		✗
	P13				✗	✗	✗	✗
	P16				✗	✗	✗	✗
	Total	**1**	**0**	**1**	**8**	**8**	**5**	**8**
PQS	P3				✗	✗		✗
	P4			●	✗	✗		✗
	P7			●	✗	✗		✗
	P8				✗	✗		✗
	P10		●		✗	✗	✗	✗
	P12	●			✗	✗	✗	✗
	P14			●	✗	✗	✗	✗
	P15	●			✗	✗		✗
	Total	**2**	**1**	**3**	**8**	**8**	**3**	**8**

Key: ●=Leak Found, ✓=Feature Implemented, ✗=Missing Handling/Check

To evaluate performance on Task 1, an analysis was conducted to compare the average completion time (in minutes) between the QuantCrypt and PQ-Sandbox groups. The results revealed a clear and statistically significant difference, indicating that the QuantCrypt group was substantially faster. Participants using QuantCrypt finished the task in an average of 39.37 minutes, while the PQ-Sandbox group took considerably longer, averaging 65.38 minutes–a difference of 26 minutes. This conclusion is supported by an unpaired t-test ($t(14) = 3.09$, $p = 0.0079$), which shows that the probability that this large difference occurs by random chance is very low. Furthermore, we are 95% confident that the true average advantage for the QuantCrypt group is between 7.97 and 44.03 minutes. Because this confidence interval does not include zero, it confirms that the observed performance gap is a genuine finding and not a statistical fluke.

For Task 2, a similar unpaired t-test was used. It revealed a difference, but with the opposite result ($t(12) = 4.09$, $p = 0.0015$). In this second task, the participant in the PQ-Sandbox group was significantly faster, with a mean completion time of 25.43 minutes, compared to the participant in the QuantCrypt group, who took significantly longer, a mean time of 41.43 minutes. On average, the PQ-Sandbox group took 16 minutes less to complete the second task. The 95% confidence interval for this difference (7.49 to 24.51 minutes) again does not contain zero, confirming that the slower time observed in the QuantCrypt group is statistically significant.

Finally, in the analysis on Task 3 completion time, we found no statistically significant differences between the two groups ($t(7) = 0.596$, $p = 0.5700$). Descriptively, the mean time for the QuantCrypt group (N=5) was 25.60 minutes, and the mean of the PQ-Sandbox group's ($N = 4$) was 23.25 minutes. This small difference of 2.35 minutes is likely due to random chance, a conclusion supported by the 95% confidence interval, which ranged from -6.98 to 11.68 minutes. As this interval contains 0, it confirms the lack of a statistically significant difference in the implementation of ML-DSA using both APIs.

Within this primary data set (see appendix C), Table III details the security vulnerabilities identified in the participants' implementations and reveals distinct error patterns of the applications between the two groups. A notable and concerning observation was the complete absence of defensive programming across all participants' implementations. In their applications, irrespective of the library employed, every participant failed to incorporate error handling for KEM encapsulation or decapsulation, and none implemented explicit destruction of unused cryptographic keys. Despite this uniform lack of hygiene, differences emerged in the exposure to critical data from insecure applications written by participants. Participants using PQ-Sandbox demonstrated a higher rate of writing insecure applications, including shared-secret exposure and mishandling of key material. In contrast, the QuantCrypt group resulted in fewer data exposures in their applications, with only two leaks combined. However, this group struggled with the implementation logic, recording five instances of incorrect DSA verification handling in their applications compared to three among PQ-Sandbox users.

These outcomes highlight how introducing PQC to generalist developers remains inherently challenging. In the constrained study setting, documentation written for a cryptography-aware audience did not fully bridge the domain-knowledge gap, an issue mirrored across the broader PQC ecosystem.

C. CDF Questionnaire

Table IV presents the analysis of the CDF questionnaire results, mapping problematic dimensions to their corresponding themes. Dimensions for which participants reported no difficulties are listed but not mapped to a specific theme, as they did not contribute to the identified usability barriers.

TABLE IV
MAPPING OF COGNITIVE DIMENSIONS PROBLEMS TO IDENTIFIED THEMES.

Cognitive Dimension	Identified Theme
The Abstraction Level	API Complexity and Granularity
Learning Style Penetrability	Documentation Deficiencies and Learning Barriers
The Working Framework The Work-step Unit Premature Commitment	Sequencing and Flow Dependency
API Elaboration Consistency Role Expressiveness	Naming, Data Handling, and Consistency Issues
Domain Correspondence	Importance of Prior Security Knowledge
Error Proneness End-user Protection Testability	Security Dependence and Lack of Testability Guidance
Progressive Evaluation API Viscosity	*No Problem Found* *No Problem Found*

1) API Complexity and Granularity: Participants generally perceived the API as complex due to its low-level granularity, forcing them to manually combine separate cryptographic components. A majority (71%) reported that multiple classes were needed to implement core functionality, contradicting the expectations of 59% who anticipated a single-class solution. P8 (CE - Expert, SDE - Beginner) highlighted this discrepancy, noting that they *"didn't expect that I would need to use and integrate multiple classes"* but rather assumed *"a single entry point"* would handle the process. Instead, developers had to manually assemble the KEM, Key Derivation Function (KDF), and symmetric-key cipher. Although P8 felt that this assembly was achievable with structured guidance, P15(SDE - Expert) cautioned that *"stitching them together... requires careful handling"* of endpoints and parameters, even if the modular roles were clear.

This requirement for manual assembly contributed to the consensus that the abstraction level was too low, exposing developers to *"too many nuts and bolts"*. P5 (SDE - Expert) argued that developers expect to *"minimize friction by just worrying about the relevant data"* rather than navigating internal cryptography jargon, while P1 (SDE - Expert) noted that manual key handling differed from typical key exchange mechanisms. Consequently, the volume of code required became a major friction point. Describing the workload for simple tasks as *"unsustainable"*, P5 emphasized that code is a *"liability in the long run"*, and that a developer typically expects to execute a task in *"one liner or 2–3 lines max"* without managing the underlying interworking.

2) Documentation Deficiencies and Learning Barriers: Feedback highlighted hurdles in the learning process, particularly with respect to the depth and clarity of the documentation. Nearly half of the participants (44%) felt that there

was insufficient information, with P5 noting that code samples were often *"unclear [or] misleading"* and P13 (SDE - Entry Level) finding the material unsuitable for those with *"limited experience"*. This confusion was compounded by a lack of context; although the documentation detailed isolated function calls, it failed to illustrate the relationships between them. P2 (PE - $<$ 1 year) criticized this approach, observing that, despite being extensive, the documentation lacked a *"clear picture overview"* of the core mechanism required to get the system running.

The presentation style further alienated developers by relying heavily on academic terminology. P5 remarked that the *"too many jargons"* meant a standard developer would struggle with half the content, leading P12 (CE - Expert) to suggest that *"visual aids and a reduction in technical security jargon"* would make the concepts more accessible. Faced with these barriers, 63% of the participants relied on copying code and *"trial and error"* to understand the API. As P8 explained, the strict sequence of operations was not initially obvious, forcing them to derive the correct data flow by *"ensuring outputs matched expectations"* rather than through clear instruction.

3) Importance of Prior Security Knowledge: The participants overwhelmingly agreed that the existing security knowledge was a critical factor in mitigating the difficulty of the API, with 88% stating that prior experience would have facilitated the process. This created a distinct divide in user experience based on background. While P1 noted that the library is *"hard to use... without prior knowledge"* due to the documentation lacking a clear flow overview, those with a foundational understanding fared considerably better. P16 (CE - Basic) reported that familiarity with concepts like public/private keys provided a *"clearer picture of how the encryption and key exchange process actually works"*, preventing the disorientation felt by novices.

Specific technical competencies were often required to bridge the gap between documentation and the task. Experienced participants like P8 cited the need for *"hands-on knowledge of ML-KEM and ML-DSA"* alongside general API integration skills. Consequently, the reliance on such specialized domains led some to perceive a mismatch in the intended audience. P6, identifying as a frontend developer, argued that the API was *"mainly targeted for someone who has some experience in security"*, suggesting that without this specific context, the implementation barrier remains high for generalist software engineers.

4) Sequencing and Flow Dependency: The API workflow was defined by a rigid sequential structure, with 88% of the participants reporting that the system forced them to think ahead and prioritize specific decisions. This workflow dictated a strict execution order–key generation $\rightarrow$ encapsulation / decapsulation $\rightarrow$ deriving shared secret $\rightarrow$ encryption / signing–which required developers to preemptively plan their architecture. P15 emphasized the cognitive load of this planning, stating that success required understanding *"who generates keys, when to send the public key, and when to encapsulate/decapsulate"* before implementation. However, this dependency chain was not immediately intuitive; 63% of participants admitted to identifying these necessary advanced decisions through trial and error, often struggling to determine the correct operational order for complex components like encapsulation and encryption.

5) Naming, Data Handling, and Consistency Issues: Participants encountered friction with respect to ambiguous naming conventions and unclear data requirements. The use of abbreviations like "`pk`", "`sk`", and "`data`" was criticized as non-intuitive, with P10 (SDE - Expert) arguing that these were *"confusing"* and poor conventions. Similarly, P5 felt that method names such as "`keygen`" sounded unprofessional–likening it to *"pirated software"*–and suggested more standard alternatives like "`generateKey()`". This lack of clarity extended to data definition; P15 noted that functions such as `encrypt_text` did not specify necessary input properties, while P16 reported that the data types for the parameters and the return values were generally *"difficult to find"*.

Beyond terminology, manual data manipulation and functional consistency presented challenges. P16 highlighted a specific hurdle where the API produced a 32-byte key while the `Krypton` class required a 64-byte key, forcing them to *"look for external resources"* to handle the conversion. Users also reported initial confusion regarding the similarity of certain functions. P2 noted that *"some functions looked similar at first because their names were not very clear"*, requiring documentation checks to distinguish them. However, P8 offered a counterpoint, observing that while KEM and KDF had similar goals, they were *"clearly separated by role"*–one for exchange and one for shaping secrets–which helped clarify the distinction within the workflow.

6) Security Dependence and Lack of Testability Guidance: Participants acknowledged that security was highly dependent on their correct implementation, yet they often lacked the necessary guidance to verify it. A majority (80%) understood that end-user security depended on both the API's guarantees and their own code, specifically regarding key handling and sequencing. P8 articulated this distinction, noting that while the API provided primitives like ML-KEM, the *"actual security outcome"* relied on the developer ensuring *"correct sequencing, validation, and correct handling of keys"* during the key exchange. However, maintaining this security was complicated by opaque error reporting. P8 noted that most issues, such as decapsulation failures, had to be handled at the program level, while P5 expressed frustration that generic error messages like `CipherVerifyError` provided *"not enough info as to 'why' "*, forcing them to *"ask AI to help debug"* rather than relying on API feedback.

Despite these risks, the majority of participants (67%) did not test the security of their applications, citing time constraints and lack of instructional support. P2 admitted that they skipped testing because they were *"wasn't sure how to do it properly"* and required specific examples. This uncertainty reflected a broader gap in documentation; 57% were unsure if testing guidance existed and 36% stated that it was absent.

P8 criticized the lack of context regarding how the algorithms matched *"NIST PQC standards"* or regulatory requirements, while P16 noted that the documentation failed to explain the *"key exchange process and why a 64-byte symmetric key was required"*, forcing developers to rely on external sources to validate their security posture.

V. Discussion

A. RQ1: What common misuses and usability barriers do developers encounter when utilizing PQC APIs?

The evaluation revealed recurring usability patterns that align with previous research on classical cryptographic APIs. These patterns highlight how introducing PQC to generalist developers without expert onboarding or contextual guidance creates predictable friction. The findings reflect the natural gap between rapidly evolving PQC standards and the mental models of developers who are encountering these primitives for the first time.

Many participants mishandled key material, for example, sending decapsulation keys or sharing secrets across the network. These behaviors stemmed from a missing conceptual understanding of typical KEM / DSA workflows when working without onboarding, as well as documentation gaps. In particular, many participants assumed that any value output by a function must be transmitted, a common behavior documented in past usability research. Clarifying examples in future documentation, especially emphasizing that shared secrets never leave local memory, would reduce such misunderstandings.

Both APIs used established cryptographic abbreviations such as (`pk` / `sk`). Although these are standard within the cryptography community, several participants unfamiliar with such conventions found them difficult to interpret. This challenge was amplified by the fact that NIST released new educational guidance, such as the Encapsulation / Decapsulation Key terminology in SP 800-227–after the API documentation used in this study had already been written. As a result, the documentation and the newer NIST teaching examples diverged slightly in terminology, leaving participants without the contextual anchors they would normally rely on in a production setting. This mismatch reflects the evolution of the natural ecosystem and highlights the importance of aligning terminology across the PQC ecosystem as standards mature.

Minor documentation inconsistencies (e.g., typos, missing brackets) were found in both APIs. Specifically, neither API provides comprehensive explanations of variable types or the nature of the outputs returned by their functions. This lack of detailed guidance limits non-specialized developers' ability to correctly interpret API behavior, increasing the potential for implementation errors and reinforcing usability-related security risks in the applications. There are also some typos, such as incorrect variable names (see Fig. 5) or missing brackets (see Fig. 4) in the syntax. Although these seem like minor issues for experienced developers, for someone who lacks experience in programming, they might not know where the error is. Inconsistency in variable naming and error output also became a pain point for participants, as they were confused by changing terminology, and the error output was not explained in the documentation.

A further observation concerned the developer's handling of transient key material at the application layer. Participants did not destroy intermediate keys or ephemeral secrets, as recommended in NIST SP 800-227[30]. Neither API is designed to manage memory erasure, and Python's memory model does not provide built-in, high-assurance primitives to securely erase sensitive data; object lifetime and copies are managed by the interpreter and garbage collector, so this behavior was not a functionality flaw or non-compliance of either API with NIST's guideline. However, the unfamiliarity of PQC workflows may argue that documentation across the broader PQC ecosystem may benefit from clearer conceptual guidance (beyond the documentation on the functionality alone) on application-layer key-lifecycle practices, particularly for non-specialist developers.

The evaluation revealed testability issues in both APIs, as most participants did not write tests for their cryptographic applications. Although time constraints contributed to this behavior, but unclear documentation and lack of testing examples were also factors. Participants often overlooked error-handling mechanisms, highlighting the need for documentation to provide explicit guidance on best practices. These findings indicate an opportunity for the PQC ecosystem to improve support for verifying correctness when using novel cryptographic primitives.

B. RQ2: How do the integration challenges differ between endpoint-based PQC APIs and local library PQC API?

The results and analysis that informed the RQ2 reveal differences and clear trade-off between the QuantCrypt API and the PQ-Sandbox API design choices. Participant performance and perceived usability varied between the two APIs, reflecting the distinct cognitive and operational demands imposed by each. These variations are expected, given the fundamentally different design principles and levels of abstraction underlying the two APIs, which shape both the ease of use and the types of errors developers are likely to encounter.

The QuantCrypt API provided a direct KEM implementation, but concealed several underlying complexities. A major usability challenge emerged when participants attempted to integrate the shared secret generated by KEM with a symmetric-key cipher. Most participants did not recognize that the KEM output was 32 bytes and was required to process through a KDF to produce a 64-byte key suitable for subsequent AES encryption. This gap illustrates how low-level abstractions, while flexible, can impose substantial cognitive overhead and increase the risk of implementation errors in applications.

For context, ML-KEM outputs a 32-byte shared secret by design. QuantCrypt lets developers to apply their own Key Derivation Function (KDF) when a 64-byte key is needed for AES or similar ciphers. In contrast, the PQ-Sandbox prototype applies a KDF internally and returns a 64-byte symmetric key directly. This architectural difference explains much of the divergence in task completion times for Task 2.

Several participants misinterpreted example snippets because they lacked context on how the example differed from their task scenario. This reflects the limitations of the scope of the documentation under experimental conditions. Participants often did not know that the source of the problem was in their initial use of a random key instead of the KEM-generated shared secret. Resolving the issue required line-by-line backtracking to identify the origin of the error, increasing the cognitive load, and could lead to further developer errors. This confusion directly explained the reasons for the longer task completion times observed for Task 2 among the QuantCrypt group.

In contrast, the PQ-Sandbox API operates at a different level of abstraction for the KEM function. It integrates the KDF step directly within the KEM operation. As a result, the shared key produced by the KEM was already 64 bytes and could be used immediately for symmetric-key encryption, eliminating the confusion observed with the QuantCrypt API. Nevertheless, PQ-Sandbox introduced its own usability challenges, primarily related to its endpoint-based design. Participants were required to create custom objects or methods to access API functions and frequently struggled with implementing authentication headers and tokens to use the API, which proved confusing. This initial setup overhead contributed to slower completion times for Task 1 among PQ-Sandbox users compared to QuantCrypt. However, once participants became familiar with these procedures, the consistency and similarity of subsequent function calls facilitated the implementation of ML-DSA in Task 2, making the process more straightforward than for QuantCrypt users.

Both APIs leave the key rotation policy to the application layer. Developers unfamiliar with such practices may benefit from higher-level documentation guidance or examples. QuantCrypt exposed a KDF that could be leveraged to derive multiple keys from the same shared secret. With PQ-Sandbox, generating a new key required repeating the entire KEM process, a time-consuming procedure that could unintentionally lead to insecure utilizations in the application when lacking clear documentation or examples, such as reusing keys across multiple sessions. This design choice highlights a trade-off between abstraction convenience and the flexibility required for secure key management.

Furthermore, the design of the PQ-Sandbox, which prioritizes security and privacy, further ensures that the generated keys are not stored on the server and are instead output directly to the users. However, this approach leads to developers receiving multiple keys for different purposes during the DSA signing process, which confuses non-experts. The DSA signing step returns a key pair in the prototype environment. In the constrained-study setting, several participants were unfamiliar with how signature schemes typically separate long-term identity keys from ephemeral signing keys. Without explicit onboarding explaining these roles, some participants incorrectly assumed that all returned fields needed to be transmitted. The observed misunderstandings were attributable to gaps in participant familiarity with DSA workflows and the limited conceptual guidance provided in the study material.

C. RQ3: What usability and guidance improvements are needed to support secure implementation?

The research attempted to identify potential solutions to the observed issues through the third research question. The findings indicate that the security vulnerabilities uncovered in participants' implementations stem not merely from individual developer mistakes but from structural usability obscurities embedded within the architectures of APIs themselves. Addressing these shortcomings, therefore, requires a developer-centric design philosophy that guides non-expert users toward secure practices in the API designs and documentation. Drawing on insights into mental models and workflows of developers, the proposed recommendations focus on improving usability and developer performance while ensuring that secure development practices are reinforced by default in the applications.

1) Adopt a Secure-by-Default Design: The absence of implemented key destruction and error handling in the applications highlights that developers cannot be expected to understand that they need to manage such tasks manually, particularly when documentation offers no guidance or examples. In addition, the confusion surrounding different key lengths and the necessity of KDF usage highlights a broader need for improved API design. One sound solution is to offer additional high-level functions that encapsulate common cryptographic workflows. Such an approach will reduce developer misuse and cognitive load while still allowing experts to access low-level primitives for advanced users who require greater control.

- **Automate Key Life Cycle**: To reduce the risk of secret-key exposure in the application, we recommend that cryptographic APIs manage the full key life cycle, including timely memory cleanup. Ideally, this is achieved through native automation, for example, binding private keys to constructs like Python's "with" statement so that keys are wiped when they leave scope. If such mechanisms cannot be built in, such as due to compliance constraints (i.e., highly regulated environments), we recommend that the cryptographic APIs should still guide developers by integrating secure-memory libraries or, at a minimum, providing clear documentation and examples that demonstrate proper key-destruction procedures and explain their security rationale.
- **Provide High-Level, Secure-by-Default Functions:** We argue that cryptographic APIs should, by default, encapsulate complex security operations, particularly for developers without specialized cybersecurity expertise. A high-level function, such as `establish_secure_channel()`, should set up the full sequence of operations, including KEM, KDF, and handshake authentication internally, and these different cryptographic components should be used and assembled following the best security practice by default. The low-level functions should remain accessible, but explicitly marked for expert use, and should be used with caution.

If such high-level functions are not built into the cryptographic API, then the documentation must compensate by providing complete, secure, and executable example workflows that demonstrate best practices and prevent misuse.
- **Enforce Error Handling**: Furthermore, cryptographic functions should not return boolean or status codes that developers can ignore. Failures in decapsulation or signature verification are security-critical and should raise explicit exceptions by default, forcing developers to handle them properly.

2) Prioritize Developer-Centric Documentation: Documentation has consistently emerged as a central usability barrier. It should be revised from a developer-oriented perspective rather than that of a security expert designing the API. Effective documentation is essential to help developers build secure applications while reducing the unnecessary cognitive burden.

- **Create "Flow" Documentation:** The documentation should evolve from a function-by-function reference into a comprehensive developer guide. It should present high-level use cases and include diagrams that illustrate the complete client–server workflow from start to finish. Examples should align with trusted guidelines, such as NIST, and demonstrate correct usage, common pitfalls, and how to apply the code in real-world applications.
- **Provide Runnable, Secure-by-Default Examples:** The documentation should include complete "copy-and-paste" examples that demonstrate the full secure workflow, for example, KEM and DSA with correct error handling and secure key management. Such examples serve as practical best-practice models, a need emphasized by participants who noted that the provided skeleton code was essential for understanding how to proceed. These examples must also be contextualized, reflecting realistic scenarios, such as establishing a secure communication channel rather than presenting isolated function calls.
- **Use Clear and Standardized Terminology:** APIs and their documentation should employ intuitive, self-explanatory, and unambiguous names for functions and variables to improve readability and help developers understand expected behaviors. To minimize confusion, APIs should adopt the NIST recommended terminology, using "encapsulation key" and "decapsulation key" instead of generic "public key" and "secret key", and avoid unclear abbreviations such as "`sk`", "`pk`" and "`enc`", which increase cognitive load. Recognizing that not all developers are cybersecurity experts, these clear names should be accompanied by explicit explanations and descriptions in the documentation. Following NIST naming conventions would standardize terminology across PQC APIs and make them more accessible to general developers.

VI. Limitations

This study examined general developer behavior under limited documentation and limited onboarding, reflecting a low-support environment rather than real-world operational deployments or cryptographic experts' deployment. The findings are based on a small sample of participants, which limits quantitative generalizability but provides strong qualitative insight into early-stage usability patterns. Additionally, the study focuses on the developer experience rather than the security, performance, or production readiness of the assessed APIs.

VII. Conclusions

This study examined the usability of two Post-Quantum Cryptography (PQC) APIs, QuantCrypt, a local library, and PQ-Sandbox, an endpoint-based API, among developers without deep cybersecurity expertise. Both APIs posed challenges in integrating PQC algorithms, with design differences affecting developer interaction. Usability limitations contributed to implementation errors, showing that strong mathematical security alone is insufficient without developer-friendly tools.

Results revealed a performance-usability trade-off, while highlighting mismatches between API design assumptions and developer expectations. Participants struggled with jargon-heavy documentation, domain knowledge gaps, and abstraction challenges requiring trial-and-error assembly of cryptographic components. These findings emphasize that simply providing PQC algorithms in an API is insufficient for application security; a developer-centric focus on usability is critical to prevent insecure implementations of applications.

Based on these insights, we recommend that PQC API designers should prioritize usability, providing secure-by-default, high-level functions, NIST-compliant naming, and clear, illustrated documentation with executable examples. Future work should involve larger studies and the development of a high-usability PQC API addressing the failures identified here, ensuring that cryptographic tools effectively shield developers from the application vulnerabilities they aim to prevent.

Acknowledgment

The authors thank Associate Professor Nalin Arachchilage for his supervision and guidance. We gratefully acknowledge Samuel Tseitkin and ExeQuantum for their technical support with the APIs and Sandbox environment. We also extend our appreciation to the study participants for their time and valuable contributions. This work was supported by the Australia Awards Scholarship (AAS).

References

[1] S. S. Gill, A. Kumar, H. Singh, M. Singh, K. Kaur, M. Usman, and R. Buyya, "Quantum computing: A taxonomy, systematic review and future directions," *Software: Practice and Experience*, vol. 52, no. 1, pp. 66–114, 2022. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3039

[2] P. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134.

[3] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, 1997. [Online]. Available: https://doi.org/10.1137/S0097539795293172

[4] K. K. Soni and A. Rasool, "Cryptographic attack possibilities over rsa algorithm through classical and quantum computation," in *2018 International Conference on Smart Systems and Inventive Technology (ICSSIT)*, 2018, pp. 11–15.

[5] D. Handa, K. K. Nikhil, S. Duvarakanath, and M. K, "Quantum-driven big data processing," in *2024 International Conference on Advances in Modern Age Technologies for Health and Engineering Science (AMATHE)*, 2024, pp. 1–7.

[6] C. Gidney and M. Ekerå, "How to factor 2048 bit rsa integers in 8 hours using 20 million noisy qubits," *Quantum*, vol. 5, p. 433, Apr. 2021. [Online]. Available: http://dx.doi.org/10.22331/q-2021-04-15-433

[7] A. Luongo, V. Narasimhachar, and A. Sireesh, "Optimizing windowed arithmetic for quantum attacks against rsa-2048," in *2025 62nd ACM/IEEE Design Automation Conference (DAC)*, 2025, pp. 1–7.

[8] D. J. Bernstein, *Introduction to post-quantum cryptography*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, ch. 1, pp. 1–14. [Online]. Available: https://doi.org/10.1007/978-3-540-88702-7_1

[9] Cybersecurity and Infrastructure Security Agency (CISA), "Post-quantum cryptography initiative: Critical infrastructure security," U.S. Department of Homeland Security, Tech. Rep., 2022. [Online]. Available: https://www.cisa.gov/quantum

[10] H. Singh, *Managing the Quantum Cybersecurity Threat*, 1st ed., ser. Harvest Now, Decrypt Later. CRC Press, 2024, ch. 9, pp. 142–158. [Online]. Available: https://doi.org/10.1201/9781003475286-9

[11] A. Ali, "A pragmatic analysis of pre- and post-quantum cyber security scenarios," in *2021 International Bhurban Conference on Applied Sciences and Technologies (IBCAST)*, 2021, pp. 686–692.

[12] National Institute of Standards and Technology, "Selected algorithms for post-quantum cryptography," 2025, accessed: 2025-04-19. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms

[13] D. Moody, A. Regenscheid, R. Perlner, A. Robinson, and D. Cooper, "Transition to post-quantum cryptography standards," National Institute of Standards and Technology, Tech. Rep., 2024. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2024/NIST.IR.8547.ipd.pdf

[14] EU PQC Workstream, "A coordinated implementation roadmap for the transition to post-quantum cryptography, part 1, version 1.1," Online, European Union, Jun. 2025, version 1.1. [Online]. Available: https://digital-strategy.ec.europa.eu/en/library/coordinated-implementation-roadmap-transition-post-quantum-cryptography

[15] C. Näther, D. Herzinger, S.-L. Gazdag, J.-P. Steghöfer, S. Daum, and D. Loebenberger, "Migrating software systems toward post-quantum cryptography-a systematic literature review," *IEEE Access*, vol. 12, pp. 132 107–132 126, 2024.

[16] J. Sowa, B. Hoang, A. Yeluru, S. Qie, A. Nikolich, R. Iyer, and P. Cao, "Post-quantum cryptography (pqc) network instrument: Measuring pqc adoption rates and identifying migration pathways," in *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, vol. 01, 2024, pp. 1835–1846.

[17] M. J. Kannwischer, P. Schwabe, D. Stebila, and T. Wiggers, "Improving software quality in cryptography standardization projects," in *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, June 2022, pp. 19–30.

[18] D. Stebila and M. Mosca, "Post-quantum key exchange for the internet and the open quantum safe project," in *Selected Areas in Cryptography – SAC 2016*, R. Avanzi and H. Heys, Eds. Cham: Springer International Publishing, 2017, pp. 14–37.

[19] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why eve and mallory love android: an analysis of android ssl (in)security," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 50–61. [Online]. Available: https://doi.org/10.1145/2382196.2382205

[20] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the usability of cryptographic apis," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 154–171.

[21] C. Wijayarathna and N. A. G. Arachchilage, "An empirical usability analysis of the google authentication api," in *Proceedings of the 23rd International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 268–274. [Online]. Available: https://doi.org/10.1145/3319008.3319350

[22] M. Green and M. Smith, "Developers are not the enemy!: The need for usable security apis," *IEEE Security & Privacy*, vol. 14, no. 5, pp. 40–46, 2016.

[23] J. Schmüser, P. Klostermeyer, K. Friedrich, and S. Fahl, " "I'm pretty expert and I still screw it up": Qualitative Insights into Experiences and Challenges of Designing and Implementing Cryptographic Library APIs ," in *2025 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 2322–2340. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00026

[24] C. Wijayarathna and N. A. G. Arachchilage, "Using cognitive dimensions to evaluate the usability of security apis: An empirical investigation," *Information and Software Technology*, vol. 115, pp. 5–19, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584919301624

[25] C. Wijayarathna, M. Grobler, and N. A. G. Arachchilage, "Software developers need help too! developing a methodology to analyse cognitive dimension-based feedback on usability," *Behaviour & Information Technology*, vol. 40, no. 6, pp. 506–527, 2021. [Online]. Available: https://doi.org/10.1080/0144929X.2019.1705393

[26] National Institute of Standards and Technology, "Module-Lattice-Based Key-Encapsulation Mechanism Standard," Department of Commerce, Washington, D.C., Federal Information Processing Standards Publication NIST FIPS 203, 2024. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.203

[27] R. Cramer and V. Shoup, "Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack," *SIAM Journal on Computing*, vol. 33, no. 1, pp. 167–226, 2003. [Online]. Available: https://doi.org/10.1137/S0097539702403773

[28] V. Shoup, "A proposal for an ISO standard for public key encryption," Cryptology ePrint Archive, Paper 2001/112, 2001. [Online]. Available: https://eprint.iacr.org/2001/112

[29] K. Cherkaoui Dekkaki, I. Tasic, and M.-D. Cano, "Exploring Post-Quantum cryptography: Review and directions for the transition process," *Technologies (Basel)*, vol. 12, no. 12, p. 241, Nov. 2024.

[30] G. Alagic, E. Barker, L. Chen, D. Moody, A. Robinson, H. Silberg, and N. Waller, "Recommendations for Key-Encapsulation Mechanisms," National Institute of Standards and Technology, Special Publication 800-227, Sep. 2020. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-227.ipd

[31] L. Zapata, A. M. Moreno, and E. Fernandez-Medina, "Is usability an obstacle for information systems security?" in *Proceedings of the 10th International Workshop on Security in Information Systems - Volume 1: WOSIS, (ICEIS 2013)*, INSTICC. SciTePress, 2013, pp. 53–65.

[32] C. Burns, J. Ferreira, T. D. Hellmann, and F. Maurer, "Usable results from the field of api usability: A systematic mapping and further analysis," in *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2012, pp. 179–182.

[33] L. Murphy, M. B. Kery, O. Alliyu, A. Macvean, and B. A. Myers, "Api designers in the field: Design practices and challenges for creating usable apis," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2018, pp. 249–258.

[34] J. Stylos and S. Clarke, "Usability implications of requiring parameters in objects' constructors," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. USA: IEEE Computer Society, 2007, p. 529–539. [Online]. Available: https://doi.org/10.1109/ICSE.2007.92

[35] U. Dekel and J. D. Herbsleb, "Improving api documentation usability with knowledge pushing," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 320–330.

[36] U. Farooq, L. Welicki, and D. Zirkler, "Api usability peer reviews: a method for evaluating the usability of application programming interfaces," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 2327–2336. [Online]. Available: https://doi.org/10.1145/1753326.1753677

[37] S. Clarke and Steven, "Measuring api usability," *Dr. Dobb's Journal*, vol. 29, pp. S6–, 05 2004.

[38] B. A. Myers and J. Stylos, "Improving api usability," *Commun. ACM*, vol. 59, no. 6, p. 62–69, May 2016. [Online]. Available: https://doi.org/10.1145/2896587

[39] H. A. Boyes, P. Norris, I. Bryant, and T. Watson, "Trustworthy software: lessons from 'goto fail' & heartbleed bugs," in *9th IET International Conference on System Safety and Cyber Security (2014)*. Institution of Engineering and Technology, 2014.
[40] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman, "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ser. IMC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 475–488. [Online]. Available: https://doi.org/10.1145/2663716.2663755
[41] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 73–84. [Online]. Available: https://doi.org/10.1145/2508859.2516693
[42] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, ""jumping through hoops": Why do java developers struggle with cryptography apis?" in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 935–946.
[43] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating ssl certificates in non-browser software," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 38–49. [Online]. Available: https://doi.org/10.1145/2382196.2382204
[44] P. L. Gorski and L. Lo Iacono, "Towards the usability evaluation of security apis," in *Clarke, Furnell (Eds.): Tenth International Symposium on Human Aspects of Information Security & Assurance (HAISA 2016), Frankfurt, Germany, July 19-21, 2016*, 2016, pp. 252 – 265. [Online]. Available: https://www.cscan.org/?page=openaccess&eid=17&id=287
[45] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky, "You get where you're looking for: The impact of information sources on code security," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 289–305.
[46] D. Ott, C. Peikert, and other workshop participants, "Identifying research challenges in post quantum cryptography migration and cryptographic agility," 2019. [Online]. Available: https://arxiv.org/abs/1909.07353
[47] E. Crockett, C. Paquin, and D. Stebila, "Prototyping post-quantum and hybrid key exchange and authentication in tls and ssh," *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 858, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:198925340
[48] N. Bindel, J. Brendel, M. Fischlin, B. Goncalves, and D. Stebila, "Hybrid key encapsulation mechanisms and authenticated key exchange," in *Post-Quantum Cryptography*, J. Ding and R. Steinwandt, Eds. Cham: Springer International Publishing, 2019, pp. 206–226.
[49] M. Ukrop and V. Matyas, "Why johnny the developer can't work with public key certificates," in *Topics in Cryptology – CT-RSA 2018*, N. P. Smart, Ed. Cham: Springer International Publishing, 2018, pp. 45–64.
[50] C. Wijayarathna and N. A. G. Arachchilage, "Why johnny can't store passwords securely? a usability evaluation of bouncycastle password hashing," in *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, ser. EASE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 205–210. [Online]. Available: https://doi.org/10.1145/3210459.3210483
[51] ——, "Why johnny can't develop a secure application? a usability analysis of java secure socket extension api," *Computers & Security*, vol. 80, pp. 54–73, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404818304887
[52] ——, "A methodology to evaluate the usability of security apis," in *2018 IEEE International Conference on Information and Automation for Sustainability (ICIAfS)*, 2018, pp. 1–6.
[53] C. Wijayarathna, N. A. G. Arachchilage, and J. Slay, "A generic cognitive dimensions questionnaire to evaluate the usability of security apis," in *Human Aspects of Information Security, Privacy and Trust: 5th International Conference, HAS 2017, Held as Part of HCI International 2017, Vancouver, BC, Canada, July 9-14, 2017, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2017, p. 160–173. [Online]. Available: https://doi.org/10.1007/978-3-319-58460-7_11
[54] A. Zeier, A. Wiesmaier, and A. Heinemann, "Api usability of stateful signature schemes," in *Advances in Information and Computer Security*, N. Attrapadung and T. Yagi, Eds. Cham: Springer International Publishing, 2019, pp. 221–240.
[55] B. Albert, T. Tullis, and D. Tedesco, "Chapter 2 - planning the study," in *Beyond the Usability Lab*, B. Albert, T. Tullis, and D. Tedesco, Eds. Boston: Morgan Kaufmann, 2010, pp. 17–47. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123748928000028
[56] M. Van Someren, Y. F. Barnard, and J. A. Sandberg, *The Think Aloud Method - A Practical Guide to Modelling CognitiveProcesses*. Academic Press, 01 1994.
[57] Simon, R. Mason, T. Crick, J. H. Davenport, and E. Murphy, "Language choice in introductory programming courses at australasian and uk universities," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 852–857. [Online]. Available: https://doi.org/10.1145/3159450.3159547
[58] Exequantum, "Quickstart - exequantum docs," https://exequantum.gitbook.io/exequantum-docs/documentations/quickstart, 2025, accessed: November 9, 2025.
[59] M. Aabmets, "Quantcrypt wiki," https://github.com/aabmets/quantcrypt/wiki, 2024, accessed: November 9, 2025.
[60] A. J. Ko, T. D. LaToza, and M. M. Burnett, "A practical guide to controlled experiments of software engineering tools with human participants," *Empir. Softw. Eng.*, vol. 20, no. 1, pp. 110–141, Feb. 2015.
[61] GraphPad Software, LLC, "T test calculator — GraphPad QuickCalcs," https://www.graphpad.com/quickcalcs/ttest1/, GraphPad Software, LLC, 2025, accessed: October 26, 2025.

APPENDIX A
EXAMPLE OF MISTAKE ON DOCUMENTATION

Fig. 4. Mistake on PQ-Sandbox Documentation (Source: [58]).

Fig. 5. Mistake on Quantcrypt Documentation (Source: [59]).

APPENDIX B
TASK GUIDELINES

The purpose of these tasks is to evaluate the usability issue that might arise when developer implement Post-Quantum Cryptography (PQC) algorithm on software using PQC APIs. Based on the implementation of API, it is categorised by two types of APIs: Web-based PQC API (accessed via HTTP requests) and traditional PQC API (accessed via a local library/package). You will be performing real-world–inspired

programming tasks that simulate integrating PQC API into a simple software application.

A. Introduction (2 minutes):

- "Thank you for participating in this usability test. We're evaluating the usability of new cryptographic algorithms, specifically Post-Quantum Cryptography (PQC), which includes the Key Encapsulation Mechanism (KEM) and Digital Signature Algorithm (DSA). These algorithms are designed to be secure against future quantum computers, but there might be some usability issues for developers when they implement them using an API."
- "Your task is to implement a simplified secure communication channel between Server and Client, using these algorithms. Please think aloud as you work, explaining your steps and any challenges you encounter."
- "You are free to consult documentation if needed, but we'd like to see how intuitive the APIs are initially. There are no right or wrong answers; we are testing the technology, not you."
- "Do you have any questions before we begin?"

B. Goal and Framing (2 minutes):

- "Your objective is to Integrate the PQC API into a provided skeleton program to send a secure message between two server and client."
- "Imagine you are a software engineer at a mid-sized financial technology company. Your team is preparing for the future where traditional cryptography may no longer be secure against quantum computers. To ensure customer data remains protected, your manager has asked you to prototype the use of Post-Quantum Cryptography (PQC) algorithm on the software using PQC API. You need to generate the shared secret (using Key Encapsulation Mechanism) between the server and client. This shared secret will be used as key on symmetric encryption, so server and client will exchange information in encrypted way."

C. Instruction (3 minutes):

- "You will be given two python script, server.py and client.py. you need to open these scripts on your IDE and you need to runt the server first before running the client"
- "You are free to use any tools that you usually use when developing software. But please share the screen when you are doing this and think out loud when you use this tools"
- "While performing the tasks, you need to talk aloud what you are thinking, so it can be recorded with screen recording."
- "Before starting, make sure your microphone is unmute and choose share the whole screen. If you work using two or more monitor please make sure what you work only on one screen so what you read and work could be captured on share screen."
- "Do you have any questions?"

D. Task Explanation (1 minutes):

- "You will be given to four tasks"
- "First Key Encapsulation and Decapsulation Mechanism"
- "Second Symmetric encryption and decryption"
- "Third Digital Signature Algorithm for Handshake Authentication Protocol"
- "Fourth Digital Signature Algorithm for Message Exchange"

E. Task 1: Key Encapsulation and Decapsulation (15-20 minutes):

- "Now, let's simulate Server and Client establishing a shared secret using ML-KEM."
- "Your first step is to make Client generate key pairs (Public and Private Key) for ML-KEM."
- "Then Client will have to send client public key to the server."
- "Server will receive client public key and use it to generate ciphertext and shared secret."
- "Server will send this ciphertext to Client."
- "Client should then use his ML-KEM private key to decapsulate the ciphertext and recover the shared secret."
- "Finally, please implement a check to verify that the shared secrets generated by Server and Client are identical."

F. Task 2: Symmetric Encryption (15-20 minutes):

- "Now, after server and Client got shared secret, this shared secret need to be used for symmetric encryption."
- "Then Server will have to use this key to encrypt the message and send this message to client."
- "Client will receive server encrypted message and decrypt it."
- "Show the decrypted message on client."
- "Next Client will encrypt the message and send this encrypted message to server."
- "Server should decrypt the message from client and show it on server side."

G. Task 3: Digital Signature Algorithm and Verification for Handshake Authentication Protocol (15-20 minutes):

- "Next, we'll focus on authentication using ML-DSA."
- "Now, let's combine these pieces into a simplified secure handshake. Imagine Client initiating a secure communication with Server."
- "Client generates an ML-KEM key pair and sends its public key to Server."
- "Server then uses Client's public key to encapsulate a shared secret and sends the resulting ciphertext."
- "Crucially, Server also signs the entire message its sends to Client (which includes the ML-KEM ciphertext and his own public key) using its ML-DSA private key."
- "Client receives this message (ciphertext, Server Public Key, and Server Signature). Client should verify Server's signature using Server public key, then if it is true client

proceed the process to decapsulate the shared secret using its ML-KEM private key."
- "The goal is to have a functional handshake where Client has a shared secret with Server and can be confident that the message (including the encapsulated secret and Server's identity) came from Server."

H. Task 4: Digital Signature Algorithm and Verification for Message Exchange (15-20 minutes):

- "Next, we'll focus on authentication using ML-DSA."
- "Server has a message or document she wants to send to Client securely. Please implement a process where Server signs this message using its ML-DSA private key."
- "Client should then implement a way to verify Server's signature using Server ML-DSA public key."

I. Post Task Questionnaire (20-30 minutes):

- "After you finish all the task or finalized the experiment you could scan the qr code or click the link on this task guideline to fill the post task questionnaire."

APPENDIX C
TASK PEFORMANCE RAW DATA

TABLE V
PARTICIPANT TASK COMPLETION TIME (MINUTES)

API	ID	T1	T2	T3	T4
QC	P1	23	43	–	–
	P2	27	38	–	–
	P5	40	42	31	–
	P6	21	31	18	–
	P9	41	42	21	–
	P11	64	–	–	–
	P13	42	40	28	–
	P16	57	54	30	–
PQS	P3	82	23	–	–
	P4	98	–	–	–
	P7	48	29	23	–
	P8	65	32	–	–
	P10	46	10	29	–
	P12	73	31	15	–
	P14	55	30	26	–
	P15	56	23	–	–