Technische Universität Ilmenau
Fakultät für Informatik und Automatisierung
Fachgebiet Telematik/Rechnernetze

Master Thesis

Design and Evaluation of a Rust Framework for Quantum-Resistant PAKE Protocols

Submitted By:	Prateek Banerjee
Submitted On:	10. September 2025
Born On:	
Degree Programme:	M.Sc. in RCSE
Responsible University Professor:	Prof. Dr.-Ing. Günter Schäfer
Scientific Supervisor:	M.Sc. David Schatz

Zusammenfassung

Das Aufkommen von Quantencomputern stellt eine erhebliche Bedrohung für klassische kryptografische Protokolle dar, einschließlich derer, die für Password Authenticated Key-Exchange (PAKE) entwickelt wurden. In dieser Arbeit stellen wir das Design und die Evaluierung eines Rust-basierten Frameworks vor, das drei aktuelle quantenresistente PAKE-Protokolle implementiert, nämlich TK-PAKE, *Modified* OCAKE-PAKE, und KEM-AE-PAKE. Unsere Beiträge umfassen eine qualitative Bewertung der Sicherheitseigenschaften jedes Protokolls sowie eine Leistungsbewertung auf der Grundlage tatsächlicher Implementierungen unter Verwendung verschiedener Post-Quantum-Key-Encapsulation Mechanism (KEM) wie Kyber und Frodo-KEM. Das Framework ermöglicht eine einfache Konfiguration und Implementierung von PAKE-Protokollen für die sichere Erzeugung von Sitzungsschlüsseln. Unsere Ergebnisse zeigen, dass *Modified* OCAKE-PAKE die beste Leistung aufweist und gleichzeitig eine explizite gegenseitige Authentifizierung bietet. Aufgrund des Designs von KEM-AE-PAKE zeigte es die geringste Effizienz und den höchsten Kommunikationsaufwand mit Kyber. Darüber hinaus heben wir hervor, wie die Wahl des KEM die Leistung desselben Protokolls erheblich beeinflusst, wie im Fall von *Modified* OCAKE-PAKE bei der Instanziierung mit Kyber und mit Frodo-KEM zu beobachten ist.

Abstract

The advent of quantum computing poses a significant threat to classical cryptographic protocols, including those designed for Password Authenticated Key-Exchange (PAKE). In this work, we present the design and evaluation of a Rust-based framework implementing three recent quantum-resistant PAKE protocols, namely, TK-PAKE, *Modified* OCAKE-PAKE, and KEM-AE-PAKE. Our contributions include a qualitative evaluation of each protocol's security properties, as well as a performance evaluation based on actual implementations using different post-quantum Key-Encapsulation Mechanisms (KEMs), such as Kyber and Frodo-KEM. The framework allows easy configuration and deployment of PAKE protocols for secure session key generation. Our results show that *Modified* OCAKE-PAKE demonstrates the best performance, while providing explicit mutual authentication. Owing to the design of KEM-AE-PAKE, it demonstrated the least efficient performance and highest communication overhead with Kyber. Furthermore, we highlight how the choice of the KEM substantially impacts the same protocol's performance, as observed in the case of *Modified* OCAKE-PAKE when instantiated with Kyber and with Frodo-KEM.

Contents

Chapter 1

Introduction

More often than not, Public Key Infrastructure (PKI)-based authentication mechanisms are used in a fair majority of network security protocols having a server-client architecture. And in most cases, the server's identity is (securely) authenticated using the PKI certificate, whereas authenticating the client's identity is either not feasible or is not deemed to be necessary. In scenarios where the identities of both parties need to be authenticated mutually, Password Authenticated Key-Exchange (PAKE) protocols come to the rescue [1]. PAKE protocols allow two parties to establish a secure session key based on a (low-entropy and easy-to-remember) password, which enables them to achieve the sought-after mutual authentication [2]. Nowadays, a handful of public key or asymmetric (classical) cryptographic primitives (including multiple PAKE protocols [3], [4]) are constructed based on the Discrete Logarithm Problem (DLP), Elliptic Curve Variant of DLP (ECDLP), etc. The strength of asymmetric cryptography based on the aforementioned mathematical constructs is considered hard to solve as they are primarily one-way functions, i.e., it is computationally infeasible for an adversary (with access to any classical computer) to reverse them within a reasonable amount of time. In simpler terms, we could also say that, using a classical computer, it is next to impossible to solve these hard problems.

To date, there are no publicly known algorithms that are formidable enough to circumvent the security of these asymmetric cryptosystems using a classical computer. But, with the advancements in the field of quantum computing, said cryptosystems are at peril in the face of an adversary possessing a quantum computer. Although such quantum computers are currently bound to their theoretical existence only, progress is being made to realize them. Quantum computers leverage quantum mechanics, and with Shor's algorithm [5], [6], it is fairly easy to break the security of these asymmetric cryptographic primitives. This is because they can efficiently reverse one-way functions such as the factorization of large integers, the DLP, and the ECDLP, which are hard to solve using classical computers [7]. Recently, the National Institute of Standards and Technology (NIST) released their first post-quantum encryption standards [8], which also emphasizes the necessity of deploying post-quantum cryptosystems like a quantum secure PAKE protocol in real-world applications so that they can withstand a cyberattack from a quantum computer.

1.1 About Password Authenticated Key-Exchange (PAKE)

The first PAKE protocol was proposed by Bellovin and Merritt in 1992 [3]. It is considered as the password-authenticated version of the Diffie-Hellman key-exchange [9], and is also referred to as the Encrypted Key-Exchange (EKE) protocol. The fundamental aim of a PAKE protocol is to facilitate two parties, namely a client $\mathcal{C}$ and a server $\mathcal{S}$, in establishing a shared (secure) session key using asymmetric cryptographic primitives while achieving (at least implicit) mutual authentication over an unsecure communication channel. Diffie-Hellman key-exchange, being one of the foundational key-exchange mechanisms, is an unauthenticated (key-exchange) protocol, thereby making it vulnerable to Man-in-the-Middle (MITM) attacks. In PAKE protocols, mutual authentication between $\mathcal{C}$ and $\mathcal{S}$ is achieved through the use of a long-term secret, e.g., a password or some data derived from the password. Along with other design and implementation considerations, the usage of the long-term secret in the authentication process tied to the key-exchange makes PAKE protocols resistant to MITM attacks [10], [11].

There are two categories of PAKE protocols, and the distinction between them primarily arises from the way the long-term secret is used in the protocol execution between the two parties. Considering the long-term secret is a password and the two parties are the client $\mathcal{C}$ and the server $\mathcal{S}$, the categories of PAKE protocols are as follows [1], [11]–[13]:

1. Balanced-PAKE (bPAKE) protocol: In this case, $\mathcal{C}$ generally has to share the (plaintext) password with $\mathcal{S}$, which is to be stored at the server's end, so that $\mathcal{C}$'s identity can be authenticated.

2. Augmented-PAKE (aPAKE) protocol: As mentioned above, in bPAKE protocols, the (plaintext) password is stored with $\mathcal{S}$. This renders a bPAKE protocol vulnerable in case $\mathcal{S}$ is compromised. This allows an adversary $\mathcal{A}$ to steal a client's password and impersonate as the client, not only to this (compromised) server $\mathcal{S}$, but also to other servers, in case $\mathcal{C}$ uses the same password with other servers as well. To prevent this, the concept of aPAKE protocols came into being. In this case, the (plaintext) password is not shared by $\mathcal{C}$ with $\mathcal{S}$. Instead, $\mathcal{S}$ generally stores a transformation of the password, which can also be referred to as a pre-shared verifier or a token. This pre-shared verifier or token cannot be directly linked to the password itself, but can still be used in the PAKE protocol execution to derive the necessary session key and allow $\mathcal{C}$ and $\mathcal{S}$ to achieve mutual authentication.

As is evident from [13], the existence of PAKE protocols is not just limited to theory. Instead, different PAKE protocols have been used for different purposes in a variety of applications, e.g., credential recovery using Secure Remote Password (SRP)-6a in ProtonMail, Apple's iCloud, then Wi-Fi connection establishment using Dragonfly in Wi-Fi Protected Access (WPA)-3, etc.

The Concept of Rounds and Flows in a PAKE Protocol

As PAKE protocols generally involve (at least) two parties, we wish to explicitly shed some light regarding the concept of *rounds* and *flows* in a PAKE protocol execution, which also at times acts as one of the deciding factors regarding the efficiency of the protocol itself. According to the authors in [14]:

- A *flow* in a PAKE protocol execution refers to a unidirectional transfer of a message from one party to another.

- A *round* in a PAKE protocol execution refers to a bidirectional communication, i.e., messages are exchanged by both parties in a request-response manner. In other words, a pair of request-response messages between the two parties constitutes one round in a PAKE protocol. Additionally, a one-round PAKE protocol generally means that the session key is established in just a single round, i.e., just after two (information) flows.

1.2 Our Contribution

We have already observed earlier that implementation and integration of *classical* PAKE protocols for a variety of mainstream applications has already taken place. Over time, multiple renowned classical PAKE protocols have been proposed and realized (as can be seen in [13]), but with the dawn of quantum computing and along with it, (potential) quantum adversaries, realization of quantum-resistant PAKE protocols (which can also be integrated into the real world) is imperative. Indeed, measures have been and are still being taken to address this, but, based on our knowledge, to date, the proposed quantum-resistant PAKE protocols lack clarity in the following:

1. While the authors furnish (detailed) security proofs (at times) to demonstrate their protocol's resilience against certain attacks, it is seldom that the authors definitively state which (security) requirements the proposed PAKE protocol fulfills. This makes it challenging to (quickly) assess and consider one PAKE protocol to be more secure than the other.

2. A noticeable lack of open-source implementation of the (proposed) PAKE protocols is also observed. This prevents us from independently verifying the correlation of the implemented PAKE protocol against what is mentioned in the sources.

Although the authors of [15] indeed try to consolidate and provide a systematization of knowledge about the quantum-resistant PAKE protocols proposed (roughly) over the past decade, we still find a lack of clarity for both of the aforementioned points. To that end, we try to bridge the gap by choosing some of the (recently) proposed quantum-resistant PAKE protocols and provide a concrete opinion regarding the fulfillment of (security) requirements along with their corresponding implementations using the Rust programming language. Furthermore, we evaluate and also provide a comparative performance overview of the implemented quantum-resistant PAKE protocols.

Further Report Organization: The remainder of this work is further organized as follows: In chapter 2, we provide a detailed overview of the different cryptographic building blocks. They act as the key components in the construction of the PAKE protocols mentioned in chapter 3. Additionally, we also mention the different security, functional, and non-functional requirements about the PAKE protocols in chapter 3. In chapter 4, we provide a brief reasoning behind our decision to choose Rust as the programming language for this framework, along with the extensive details of the framework design. Then, chapter 5 includes the concrete details of the performance evaluation of the PAKE protocols, and we provide our concluding comments in chapter 6.

Chapter 2

Background

In this chapter, we describe in detail the different building blocks, namely, Extendable Output Function (XOF) [16], Ideal Cipher (IC) [17], [18], Advanced Encryption Standard (AES) in counter mode [19], [20], Authenticated Encryption (AE) [21], and Key-Encapsulation Mechanisms (KEMs) [22], [23]. Additionally, we also provide a concise overview of the post-quantum security levels classified by National Institute of Standards and Technology (NIST) [24], [25].

2.1 Sponge-Based Extendable Output Function (XOF)

The traditional design of classical hash functions from the Secure Hashing Algorithm (SHA)-2 family [26] allows it to take an input string s of arbitrary length $l_1 \geq 1$ and produce an output string of a *fixed* length l_2 (depending on the choice of the hash function). Upon observing this, Bertoni et al. (2007) [27] proposed a novel approach for designing hash functions which can process an input string s of arbitrary length $l_1 \geq 1$ and produce an output string of *arbitrary* length l_2.These (new) hash functions are referred to as *sponge functions.*

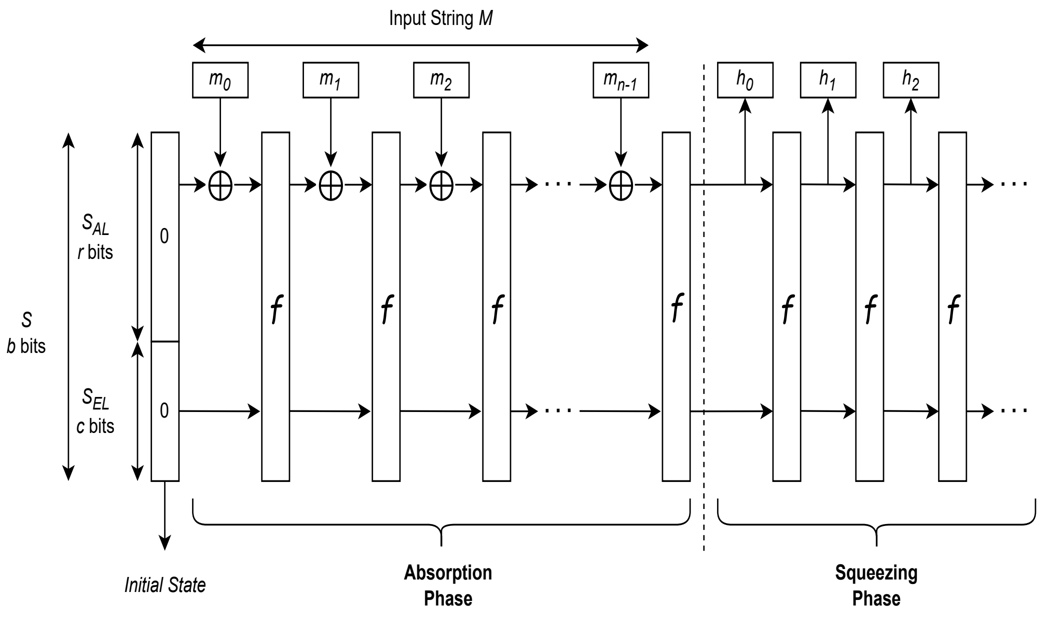

Figure 2.1: Phases of a sponge-based XOF [28].

The authors in [16] referred to this class of hash functions as XOFs, and the most notable implementation of such sponge-based XOFs are observed in the SHA-3 family [29] and the ASCON-XOF in [30]. As illustrated in Figure 2.1, a sponge-based XOF operates in two distinct phases, namely, the *Absorption* and the *Squeezing* phase. For further details regarding these two operations, we redirect a user to [27], [31].

2.2 About the Ideal Cipher Model

It is well known that many cryptosystems use block ciphers as a cryptographic building block. To consider the proposed cryptosystem secure, it is paramount that all the underlying (cryptographic) building blocks are also secure. So, to claim that a block cipher is secure, more often than not, certain (mathematical) assumptions are made, e.g., the input-key K used in the block cipher is a secret value and is chosen completely at random. And, if an adversary $\mathcal{A}$ is unaware of K, then $\mathcal{A}$ cannot successfully distinguish between the output of the block cipher and a truly random value. But, in the real world, it happens that K is not completely random. Now, in order to support the claim that the block cipher (in this larger cryptosystem) is indeed secure, it is assumed to behave like an IC. The preliminary notion of what an (ideal) block cipher should do was first proposed in 1949 by Claude Shannon [17]. The IC model is a theoretical framework that is used in cryptography to furnish security proofs of cryptographic constructions involving the usage of a block cipher, with the assumption that the block cipher behaves like an IC. It is considered an idealised model of computation where an entity (who could also be an adversary) has public access to a block cipher, i.e., anyone (including $\mathcal{A}$) could make encryption and decryption queries to the block cipher. The encryption and decryption functions of the block cipher are (assumed to be) a truly random permutation (or bijection) for every input-key K [18], [32].

If we consider a block cipher which provides encryption (E) and decryption (D) functions, then, for a particular input-key K, E is a random bijection over the plaintext space, and D is the inverse of this random bijection for the same input-key K. In simpler terms, $E(K, P_1) \neq E(K, P_2)$ iff $P_1 \neq P_2$ and $D(K, C_1) \neq D(K, C_2)$ iff $C_1 \neq C_2$. This means that, for a given input-key K, encrypting distinct plaintexts with E will always yield distinct ciphertexts, and decrypting different ciphertexts with D will always map them back to the original distinct plaintexts that were encrypted with E earlier [32].

This indicates that, in an ideal scenario, there are no chances of collisions. But, in the real world, there is nothing to actually prevent this collision completely [32]. Based on our understanding, this restricts ICs to be only theoretical rather than something that can be successfully realized practically. With that being said, [33] realized that it is not at all trivial to instantiate an IC as a block cipher over mathematical constructs like group domains (as shown in [34]), classical finite fields, or post-quantum lattices, etc. This is because working with ICs over different mathematical constructs will likely require us to exclusively design the block cipher to cater to each of them, separately.

Additionally, when assuming that a block cipher behaves like an IC, the encryption function works as $C \leftarrow E(K, P)$. During decryption, if an input key $K' \neq K$ is used, then the decryption function yields $P' \leftarrow D(K', C)$. Here, the generated plaintext $P' \neq P$, but P' is still a valid plaintext from the plaintext space, instead of being some out-of-domain value. This is because every ciphertext maps back to some valid plaintext in the plaintext space via a bijection. The only thing is that, in this case, the decryption function is the inverse of some other random bijection over the plaintext space defined by the input key K', instead of K.

2.3 AES in Counter Mode

In this subsection, we initially provide a very brief overview of the AES encryption scheme [35], [36] and then describe the AES block cipher in counter mode [19].

Overview of the AES Block Cipher: Amongst the propositions mentioned in the Rijndael block-cipher family [35], three were chosen to be formalized by NIST in 2001 [36] as the AES. In AES, encryption and decryption with the secret input-key K can be denoted as $C \leftarrow E(K, P)$ and $P \leftarrow D(K, C)$ respectively, where P is the plaintext message and C is the generated ciphertext [19], [35]. The three different variants of AES are mentioned in Table 2.1 along with their fixed parameter combinations.

Table 2.1: Key-Block-Round combinations in AES [35], [36]

AES Variant	**Key Length in Bits**	**Block Size in Bits**	**Rounds**
AES-128	128	128	10
AES-192	192		12
AES-256	256		14

AES Block Cipher in Counter Mode

A new mode of operation of AES, namely, the counter mode, was proposed by [19] and was also formalized by NIST in 2001 [20]. In this setting, the encryption function E with input-key K is applied on a counter block *ctrblk* of fixed size of 128 bits (16 bytes). The fixed parameter combinations of all three AES variants mentioned in Table 2.1 are also directly applicable for AES in counter mode, and the security of AES block cipher in counter mode is critically dependent on the uniqueness of each counter block *ctrblk*. Each *ctrblk* is generated by concatenating (denoted using $||$) a fixed *nonce* of 96 bits (12 bytes) and a counter value *ctr* of 32 bits (4 bytes), thereby making each *ctrblk* of 128 bits. The initial value of *ctr* is set to 1, and the encryption of a plaintext message P using AES block cipher in counter mode is shown in Figure 2.2.

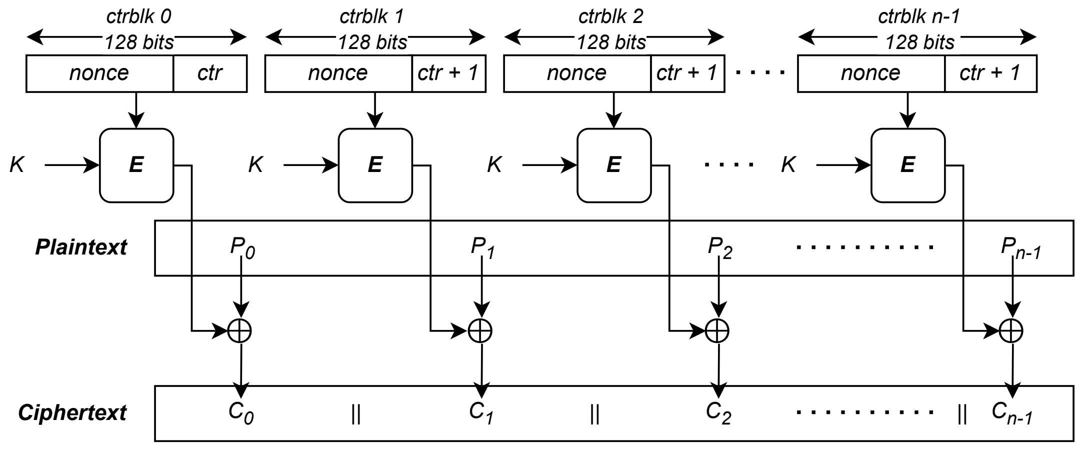

Figure 2.2: Encryption using AES block cipher in counter mode [19], [37].

There is a possibility that the size of the plaintext P is not exactly a multiple of 128. So, the last plaintext block P_{n-1} becomes a partial block of (say) u bits where $u < 128$ and it is handled a bit differently, which can be understood from [20]. The decryption procedure with AES in counter mode works similarly to the encryption, except, in this case, the roles of plaintext and ciphertext blocks are swapped/reversed [19]. Thus, we do not reiterate the details of the decryption procedure here.

2.4 Authenticated Encryption

While symmetric encryption algorithms provide data confidentiality, in most cases, they do not inherently guarantee data integrity. This means there is nothing to stop an adversary $\mathcal{A}$ from tampering with a ciphertext in transit such that it decrypts to a completely different plaintext than what was encrypted in the first place. Thus, AE is essential, as it inherently provides both data integrity and confidentiality, which allows a receiver to know if the received message was modified during transmission or not [38], [39]. The idea of composing an AE scheme was proposed in [21], and it requires two components, namely, a symmetric cipher and a message authentication scheme. For using an AE scheme on a plaintext P, there are three different ways of composing it using a symmetric cipher and a message authentication scheme, which are as follows [21]:

1. The *Encrypt & MAC* (E&M) approach
2. The *MAC then Encrypt* (*MtE*) approach
3. The *Encrypt then MAC* (*EtM*) approach

Irrespective of having three ways of composing an AE scheme, the authors in [38], [40] suggest avoiding computing the Message Authentication Code (MAC) on P because of the *Padding Oracle Attack* discovered in [41]. Additionally, verifying a MAC is faster than decrypting a ciphertext. Both E&M and *MtE* approach compute the MAC on P, so this directs us to align our preferences to the *EtM* approach because neither the MAC is computed on P nor unnecessary computational resources are invested in decrypting the ciphertext at first. In this approach, P is encrypted at first to generate an initial ciphertext C_{init}, and then a MAC of C_{init} as τ is computed. The final ciphertext is generated as $C \leftarrow C_{\text{init}} \| \tau$. During decryption, C is parsed to retrieve C_{init} and τ. Then, we check if the MAC of $C_{\text{init}} \overset{?}{=} \tau$ or not. If the output is 1, then we decrypt C_{init} to obtain P and return it, else $\perp$ is returned.

The Galois/Counter Mode of Operation

Galois/Counter Mode (GCM) was proposed in [42] and is an AE scheme based on the *EtM* approach. The GCM block cipher mode of operation has become extremely popular since its inception, due to its high efficiency and being free of intellectual property restrictions. The GCM block cipher mode of operation uses AES in counter mode as the underlying symmetric cipher and its own GHASH function as the message authentication scheme. It has two algorithms, namely, *authenticated encryption* AE_E and *authenticated decryption* AE_D. We redirect a reader to *Sections 2.3* & *2.4* in [42] and *Sections 7.1* & *7.2* in [43] for the details regarding the pseudocode and the workflow of the algorithm. It is evident from above, in an AE scheme, upon encryption, an authentication tag τ is also generated as output in addition to the ciphertext C. So, the length of the τ can be anything between 64-128 bits, but it must be *at least* 64 bits or *ideally* 128 bits [42], [43].

2.5 About NIST's Post-Quantum Security Levels

We mentioned in chapter 1 that, with the advancements in the field of quantum computing, the security of (asymmetric) classical cryptosystems is at peril in the face of an adversary possessing a quantum computer. So, efforts have indeed been made to realize post-quantum cryptographic primitives and cryptosystems for (at least) close to a decade. However, to be certain of what level of security the said primitive or cryptosystem provides, we needed some kind of classification. NIST provides the necessary classifications, as mentioned in Table 2.2.

Table 2.2: NIST post-quantum security categories and description [24], [25]

Categories	**Equivalent Classical Cryptographic Primitive**	**Equivalent Attack Costs and Description**
Level-I	AES-128	Exhaustive Key Search Attack on a block cipher with 128-bit key.
Level-II	SHA-256	Collision Attack on a 256-bit hash function.
Level-III	AES-192	Exhaustive Key Search Attack on a block cipher with 192-bit key.
Level-IV	SHA-384	Collision Attack on a 384-bit hash function.
Level-V	AES-256	Exhaustive Key Search Attack on a block cipher with 256-bit key.

The security of the AES block cipher scheme lies in the fact that an astronomically large number of trials are required to perform a brute-force/exhaustive input-key search attack using a classical computer. Considering an input-key of size N bits, a classical computer would require performing 2^N trials, but now, Grover's algorithm [44] reduces the following [45]:

- The time needed to perform the attack is reduced to $\sqrt{2^N}$.
- The security provided by the cryptographic primitive in bits is reduced to $\frac{N}{2}$ bits.

Similarly, for hash functions (from both SHA-2 [26] and SHA-3 [29] family), the security is considered based on the resistance to the following attacks [45]:

1. *Preimage Attack*: The resistance against this attack is defined as, for a particular hash digest from a hash function, how difficult it is to find the correct input that produced the output. For a hash function that produces an output of size N bits, to successfully perform a preimage attack using a classical computer, 2^N trials would be needed to find the correct input value.
2. *Collision Attack*: The resistance against this attack is defined by how difficult it is to find two different inputs to a hash function that would produce the same hash digest. For a hash function that produces an output of size N bits, to successfully perform a collision attack using a classical computer, $2^{\frac{N}{2}}$ trials would be needed to find two distinct input values generating the same output.

Now, with Grover's algorithm [44], the number of trials required to perform a preimage attack is reduced to $2^{\frac{N}{2}}$ from 2^N. Grover's algorithm does not exactly speed up the process of performing a collision attack, i.e., even with Grover's algorithm, $2^{\frac{N}{2}}$ trials are still required. But, the algorithm provided by Brassard et al. [46] can successfully perform a collision attack in $2^{\frac{N}{3}}$ trials, which is much faster even compared to performing a preimage attack using Grover's algorithm. The Brassard et al. algorithm is effectively more lethal. For a hash function producing N bits of output, the security provided by the hash function is essentially reduced to $\frac{N}{3}$ bits, and the time needed to perform the attack is reduced to $\sqrt[3]{2^N}$ [45]. Further details regarding the impact of these algorithms on the cryptographic primitives and the equivalent quantum security for the different security categories are provided in Table 2.3.

Table 2.3: Equivalent quantum security provided upon impact of Grover's and Brassard et al.'s algorithm on cryptographic primitives [45]

Cryptographic Primitive	**Potential Attack Trials**		**Classical Security in Bits**	**Equivalent Quantum Security in Bits**	**NIST Security Categories**
	Classical Computer	**Quantum Computer**			
AES-128	2^{128}	2^{64}	128	64	*Level-I*
SHA-256	2^{128}	$\approx 2^{85}$	128	≈ 85	*Level-II*
AES-192	2^{192}	2^{96}	192	96	*Level-III*
SHA-384	2^{192}	2^{128}	192	128	*Level-IV*
AES-256	2^{256}	2^{128}	256	128	*Level-V*

Additional Clarification Regarding the Security Categories: Upon carefully reviewing Table 2.3, we observe that, despite offering the same quantum security of 128 bits, there are two distinct levels, namely, *Level-IV* and *Level-V*. This distinction is based on the computational effort required to perform classical attacks on the cryptographic primitives. A collision attack on SHA-384 requires approximately 2^{192} trials, whereas an exhaustive input-key search attack on AES-256 requires 2^{256} trials. Since $2^{256} \gg 2^{192}$, AES-256 is classified at a higher level than SHA-384, irrespective of providing the same quantum security.

2.6 Key-Encapsulation Mechanism

In this section, we initially provide a brief description of what KEMs are and then we discuss about two specific KEMs, namely, Frodo-KEM [22] and ML-KEM [23]. Before discussing about the particular KEMs, we also provide details about their underlying mathematical constructs, namely, the Learning With Errors (LWE) problem [47] for Frodo-KEM and the Module Learning With Errors (MLWE) problem [48] for ML-KEM.

Brief Introduction to Key-Encapsulation Mechanisms

A KEM is a cryptographic primitive that allows two parties to securely establish a shared secret key over a potentially unsecure communication channel. Any KEM has the following three algorithms [22], [23], [49]–[51]:

1. $(pk, sk) \leftarrow$ KeyGen internally generates the required randomness and returns the KEM key pair as output, namely, the public key pk and the secret key sk.
2. $(C_K, \mathcal{K}_K) \leftarrow \text{Encaps}(pk)$ takes the KEM public key pk as input and internally generates the required randomness. It returns the shared secret key $\mathcal{K}_K$ and the KEM ciphertext C_K, which is an encapsulation of the shared secret as output.
3. $\mathcal{K}'_K \leftarrow \text{Decaps}(C_K, sk)$ takes the KEM ciphertext C_K and the secret key sk as inputs and returns the shared secret key $\mathcal{K}'_K$.

When using a KEM, the sender generates (pk, sk) and sends the pk to the receiver. Then, the receiver generates $\mathcal{K}_K$ and encapsulates it with the sender's pk. This generates the ciphertext C_K, which is then sent to the sender. Then, the sender decapsulates C_K using sk, which yields $\mathcal{K}'_K$. If both parties have honestly executed the protocol, then $\mathcal{K}_K = \mathcal{K}'_K$.

2.6.1 About Learning With Errors

The original LWE problem [47] is one of the most commonly used mathematical constructs in post-quantum cryptography. Akin to an asymmetric encryption scheme involving a public key for encryption and a secret key for decryption, for the LWE problem in the lattice-based paradigm, the secret key $\mathbf{s}$ is represented using a set of linear equations with certain errors $\mathbf{e}$ (also referred to as noise). These errors facilitate masking/hiding the secret key $\mathbf{s}$ and the LWE problem requires solving the set of linear equations with the errors in an attempt to retrieve the secret key $\mathbf{s}$ [22] and this is considered *very* hard to solve with the currently available technology and computational resources. Before diving into defining LWE using mathematical notations, we wish to describe a few of the important notations [22], [47]:

1. q refers to an integer modulus and $q \geq 2$ in almost every case.
2. $\mathbb{Z}_q$ denotes the set or ring of integers modulo q. In case of LWE, it generally represents the (positive) integers between $\{0, 1, \ldots, q-1\}$ inclusive on both ends. One is allowed to perform the following operations in $\mathbb{Z}_q$:
   - Addition: $(a + b) \mod q \in \mathbb{Z}_q$ holds $\forall\ a, b \in \{0, 1, \ldots, q-1\}$
   - Multiplication: $(a \cdot b) \mod q \in \mathbb{Z}_q$ holds $\forall\ a, b \in \{0, 1, \ldots, q-1\}$
3. n refers to the dimensions of an integer vector. In other words, it determines the number of elements/integers in the vector.
4. An integer vector of dimensions n is represented using lowercase bold font like $\mathbf{s}$.
5. A matrix with multiple vectors of integers, each of dimension n, is represented using uppercase bold font such as $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$.
6. The error or noise distribution is represented using $\mathcal{X}$.

7. Any symbol denoted with a T as the superscript such as $\mathbf{A}^T$, $\mathbf{s}^T$, etc. represents the *transpose* of the value.

We wish to explicitly mention that any instance of an LWE problem involves the usage of a public matrix $\mathbf{A}$, a secret vector $\mathbf{s}$, and an error vector $\mathbf{e}$. Given n and q where $q \geq 2$, the LWE problem can be described as follows [22], [47]:

$$\text{Sample } \mathbf{A} \overset{\$}{\leftarrow} \mathbb{Z}_q^{n \times n},\ \mathbf{s} \overset{\$}{\leftarrow} \mathcal{X}^n, \mathbf{e} \overset{\$}{\leftarrow} \mathcal{X}^n,\ \text{where } \mathbf{s}, \mathbf{e} \in \mathbb{Z}_q^n$$
$$\text{Compute } \mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e} \text{ and Return } (\mathbf{A}, \mathbf{t})$$

Based on our understanding, we are going to describe the two versions of the LWE problem in a very concise manner, which are as follows [22]:

1. *Search* Version: Considering that an adversary $\mathcal{A}$ knows $\mathbf{A}$ and $\mathbf{t}$, the idea of the *search* version is to actually find the secret vector $\mathbf{s}$ which was used to compute the (public) value $\mathbf{t}$.

2. *Decision* Version: The idea of the *decision* version is to ensure that it is (extremely) hard for an adversary $\mathcal{A}$ to successfully distinguish between the elements in $\mathbf{t}$ (generated using $\mathbf{s}$) and a completely random vector.

2.6.2 Frodo-KEM

The Frodo-KEM [22] is a key-encapsulation mechanism based on the LWE problem. It has three variants as mentioned in Table 2.4, which are based on different parameter sets for the underlying mathematical construct, i.e., LWE. Frodo-KEM has three core operations, namely, KeyGen, Encaps and Decaps as described earlier. We redirect a reader to *Algorithms 12, 13 & 14* in [22] for further details, as they are described in a consolidated manner. The size of C_K and $\mathcal{K}_K$ for the different variants of Frodo-KEM is mentioned in Table 2.5. During the KeyGen operation in Frodo-KEM, the public matrix $\mathbf{A}$ (of LWE) is generated using either AES-128 or SHAKE-128 XOF [29]. For this work, we are only interested in using SHAKE-128 XOF as the primitive for generating $\mathbf{A}$.

Table 2.4: The approved parameter sets for Frodo-KEM [22]

Variant	Dimensions (n)	Modulus (q)	Classical Security in Bits	NIST Security Categories
Frodo-640	640	2^{15}	128	*Level-I*
Frodo-976	976	2^{16}	192	*Level-III*
Frodo-1344	1,344	2^{16}	256	*Level-V*

2.6.3 About Module Learning With Errors

The LWE problem [47] (as described in subsection 2.6.1) is algebraically *unstructured.* By algebraically unstructured, we mean that it only uses matrices and integer vectors. This makes it easy to use and implement, as it does not require complex algebraic operations. But, it often falls short in terms of performance and efficiency over algebraically *structured* variants like Module Learning With Errors (MLWE) [22]. Owing to these inefficiencies of LWE, the authors of [48] proposed MLWE. The MLWE problem is

Table 2.5: Size of different keys and ciphertext in bytes [22]

Variant	Public Key (pk)	Secret Key (sk)	KEM Ciphertext (C_K)	Shared Secret Key ($\mathcal{K}_K$)
Frodo-640	9,616	19,888	9,720	16
Frodo-976	15,632	31,296	15,744	24
Frodo-1344	21,520	43,088	21,632	32

considered to be a (structured) variant of the LWE problem. Instead of relying only on integer vectors and matrices like LWE, MLWE uses polynomials, polynomial vectors, and matrices. Before diving into defining MLWE using mathematical notations, we wish to describe a few of the important notations [48], [52]:

1. q refers to a *prime* modulus and $q \geq 2$ in almost every case.

2. Polynomials can be considered as mathematical expressions comprising variables and coefficients (referred to as c). Each coefficient multiplied by a variable forms a term of a polynomial. Polynomials can have multiple (but a finite number of) variables and terms, but here we only consider polynomials with one variable (say) x. Here, n refers to the polynomial degree. E.g., a sample polynomial with only one variable x and degree $n = 3$ could be represented as $c_0x^0 + c_1x^1 + c_2x^2 + c_3x^3$ where $c_3 > 0$. Given a polynomial, we could also say that the degree of said polynomial is equal to the largest exponent of the variable x having a non-zero coefficient c [53]. Generally for MLWE, $q > n$ [23].

3. $\mathbb{Z}_q$ denotes the same thing as it does in LWE.

4. $\mathbb{Z}_q[x]$ refers to all the terms or monomials in a polynomial having coefficients in $\mathbb{Z}_q$. In simpler terms, $\mathbb{Z}_q[x]$ represents a whole polynomial having coefficients in $\mathbb{Z}_q$. The polynomial $c_0x^0 + c_1x^1 + c_2x^2 + c_3x^3$ defined earlier has the following terms/monomials c_0x^0, c_1x^1, c_2x^2, and c_3x^3 [54].

5. $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ refers to the ring of integer polynomials modulo $(x^n + 1)$. This means that the overall polynomial $\mathbb{Z}_q[x]$ in this ring (of polynomials) will have coefficients in $\mathbb{Z}_q$ and the degree of the polynomial itself will be at most $n - 1$. It is only possible to perform addition and multiplication operations $\bmod q$ on the coefficients of the polynomial [54].

6. Based on our *very limited* understanding of $x^n + 1$, we believe that this is primarily done to ensure that, if there is a polynomial with a degree k where $k \geq n$, it must be reduced in a manner such that the degree of polynomial is $\leq n - 1$ [54].

7. d refers to the dimensions of a vector of polynomials.

8. A polynomial is represented using lowercase bold italic font such as $\boldsymbol{e} \in R_q$.

9. A vector of (ring of integer) polynomials of dimensions d is represented using lowercase bold font such as $\mathbf{s},\ \mathbf{e} \in R_q^d$.

10. A matrix with multiple vectors of (ring of integer) polynomials, each of dimension d, is represented using uppercase bold font such as $\mathbf{A} \in R_q^{d \times d}$.

11. The symbol for transpose and error distribution is the same as that in LWE.

Given n, d and q where $q \geq 2$, the MLWE problem can be described as follows [23], [48], [52]:

$$\text{Sample } \mathbf{A} \xleftarrow{\$} R_q^{d \times d},\ \mathbf{s} \xleftarrow{\$} \mathcal{X}^d,\ \mathbf{e} \xleftarrow{\$} \mathcal{X}^d \text{ where } \mathbf{s}, \mathbf{e} \in R_q^d$$
$$\text{Compute } \mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e} \text{ and Return } (\mathbf{A}, \mathbf{t})$$

We also have a similar *search* and *decision* version of MLWE [48] like the LWE problem as mentioned in subsection 2.6.1.

2.6.4 ML-KEM

The MLWE-Based KEM (ML-KEM) scheme [23] is also referred to as and is based on CRYSTALS-Kyber [55]. The security of ML-KEM is derived from the hardness assumption of solving the MLWE problem, and it is considered to be the most efficient KEM [23], [56]. In this work, going further, we will refer to ML-KEM as Kyber, as it can be considered as a direct substitute for the term ML-KEM. There are three different variants of Kyber mentioned in Table 2.6 based on different parameter sets, and the sizes of the different values (like ciphertexts and keys) in Kyber are mentioned in Table 2.7. Being a KEM, Kyber [23] has the same three core operations, namely, KeyGen, Encaps and Decaps as described earlier. We redirect a reader to *Algorithms 19, 20 & 21* respectively in [23] for further details.

Table 2.6: The approved parameter sets for Kyber [23]

Variant	**Polynomial Degree (n)**	**Prime Modulus (q)**	**Dimensions (d)**	**Classical Security in Bits**	**NIST Security Categories**
Kyber-512	256	3,329	2	128	*Level-I*
Kyber-768			3	192	*Level-III*
Kyber-1024			4	256	*Level-V*

Table 2.7: Size of different keys and ciphertext in bytes [23]

Variant	**Public Key (pk)**	**Secret Key (sk)**	**KEM Ciphertext (C_K)**	**Shared Secret Key ($\mathcal{K}_K$)**
Kyber-512	800	1,632	768	32
Kyber-768	1,184	2,400	1,088	
Kyber-1024	1,568	3,168	1,568	

The KeyGen Operation in Kyber: $(pk, sk) \leftarrow$ KeyGen internally generates the required randomness and returns the KEM key pair as output. There can be cases like [49], [51], [56] where the authors parametrize the function signature of KeyGen. These parameters mostly refer to the ones from Table 2.6. But, in some cases (like [51]), one could also provide a seed as a parameter to KeyGen. Then, instead of generating the required randomness internally, this seed is used as the randomness source, and this can also be considered as a *seeded* KeyGen operation. In case of a seeded KeyGen operation, the provided seed must be of exactly 64 bytes (as is evident from *Algorithm 19* in [23]), be it one seed of 64 bytes or two seeds of 32 bytes each.

Chapter 3

Requirements and Related Work

In this chapter, we initially state the different requirements of a Password Authenticated Key-Exchange (PAKE) protocol and then discuss the related works. Then, we furnish a detailed reasoning behind choosing some of the PAKE protocols from the ones discussed in the related works, and finally describe the chosen PAKE protocols.

3.1 Requirements of a PAKE Protocol

In order to consider a PAKE protocol secure, it must fulfill the following requirements:

1. Security Requirements: The security requirements that are expected from a PAKE protocol can be categorized as follows:
   - Fundamental Security Requirements:
     a) If an adversary $\mathcal{A}$ just observes (and is not actively involved in) the PAKE protocol execution, $\mathcal{A}$ must not be able to successfully determine the correct password by exhaustively trying random guesses offline in an efficient manner. In simpler terms, the messages exchanged during the PAKE protocol execution must seem computationally independent of the password itself. This is also applicable for low-entropy passwords, and this property can be referred to as *offline dictionary attack resistance* [2], [13].
     b) The adversary $\mathcal{A}$ can be actively involved in the PAKE protocol execution, i.e., $\mathcal{A}$ can run the protocol while arbitrarily guessing different passwords and observing whether the protocol's execution succeeds or fails. According to Ding et al. (2017) [2], this attack is inevitable, so it must be ensured that $\mathcal{A}$ can only test one (randomly guessed) password per PAKE protocol execution. This property can be referred to as *online dictionary attack resistance* [13].
     c) Even if the adversary $\mathcal{A}$ at some later point in time gets to know the long-term secret, e.g., the low-entropy password, $\mathcal{A}$ must not be able to compromise any past session keys solely based on the knowledge of this password. This property can be referred to as *forward secrecy* [13], [49]. As an extension, the compromise of one session key must also not lead to the compromise of other session keys. This property can be referred to as *perfect forward secrecy* [13], [15], [49], [57].
     d) Even if the security, i.e., the hardness of the computational assumption, upon which the PAKE protocol is built, is compromised, an adversary $\mathcal{A}$ must not be able to extract any information about the password itself upon observing the transcript of the PAKE protocol execution. This is achieved by designing the PAKE protocol in a manner such that it seems that any password could have produced the messages exchanged by the parties in the PAKE protocol execution. In other words, the messages exchanged by the parties during the PAKE protocol execution

must seem completely independent of the password. This property can be referred to as *perfect password hiding* [58].

e) A PAKE protocol execution must never leak any information that allows an adversary $\mathcal{A}$ to become aware, whether both parties used the same (high-entropy) password or two different (high-entropy) passwords referred to as a PreShared Key (PSK). As we mentioned in chapter 1 that the passwords used in PAKE protocols are easy-to-remember and of low entropy, we believe that the authors in [58] just refer to a high-entropy password as a PSK. According to [58], this is a strictly weaker property as opposed to *perfect password hiding.* This property can be referred to as PSK*equality hiding* [58].

- Enhanced Security Requirements:

  a) Even if the server $\mathcal{S}$ is compromised, it should not be possible for an adversary $\mathcal{A}$ to directly impersonate as the client $\mathcal{C}$, i.e., $\mathcal{A}$ must need to perform at least an offline dictionary attack to identify the correct password used in the PAKE protocol execution. This property can be referred to as *server compromise resistance* [1], [13]. We wish to explicitly mention that the PAKE protocols fulfilling this property are the ones which are referred to as aPAKE protocols.

  b) A scenario may arise where an adversary $\mathcal{A}$ pre-computes a table of values based on a dictionary of passwords. Even if the server $\mathcal{S}$ gets compromised, $\mathcal{A}$ should not be able to *efficiently* perform an offline dictionary attack and obtain the client's password, and impersonate as the client $\mathcal{C}$, as this diminishes the security benefit for which aPAKE protocols were designed in the first place. This property can be referred to as *precomputation resistance* [12], [13], and PAKE protocols fulfilling this property are referred to as $strong - a$PAKE protocols, like an extension to just aPAKE protocols.

2. Functional Requirements:

   a) The client $\mathcal{C}$ and the server $\mathcal{S}$ must be able to establish a session key upon successful execution of the PAKE protocol. Additionally, the PAKE protocol design must be such that it indeed provides implicit mutual authentication between the two parties involved in the PAKE protocol execution, i.e., the generated session key on both parties' end must be the same iff they used the same password. This requirement can be referred to as *session key establishment.*

   b) The PAKE protocol design must be such that the generated session key is only revealed to parties involved in the PAKE protocol execution and no one else. This requirement can be referred to as *session key secrecy.* If either party involved in the protocol execution explicitly or mistakenly discloses its session key, this *will not* lead to the unfulfillment of this property.

3. Non-Functional Requirements: Although we have not (yet) found any resources that explicitly state any non-functional requirements for PAKE protocols, based on the information provided in [13], we could consider the following as potential non-functional requirements for a PAKE protocol:

   a) Performance: The PAKE protocol must be efficient in terms that it should not incur excessive computation and/or communication overhead, making it suitable for real-world applications and being used in resource-constrained devices.

b) Usability: The PAKE protocol must not be too complex so that it prohibits a user from seamlessly using it or a developer from implementing the same. As is also mentioned in [13], the reason behind the limited adoption of the PAKE protocol in web applications is because of the difficulty faced in integrating it with the existing web authentication systems.

c) Cryptographic Agility: The PAKE protocol must be flexible and agile enough, such that the underlying cryptographic primitives are easily exchangeable with other relevant or newer cryptographic primitives, without having to redesign the whole PAKE protocol itself.

d) Round and/or Flow Optimality: The PAKE protocol must not require an excessive number of rounds or flows to establish the session key and achieve the mutual authentication. In other words, a PAKE protocol with fewer rounds or flows is desirable in practice, because this results in low latency and also makes it suitable to be used in resource-constrained devices.

e) Explicit Mutual Authentication: The PAKE protocol should (ideally) provide and ensure explicit mutual authentication (for both parties) as a part of the PAKE protocol execution.

3.2 Related Work

In this section, we give a detailed overview of the related work. Based on our observation of the recently proposed quantum-resistant PAKE protocols, they could be categorized as follows:

- Generic frameworks or constructions.
- Concrete PAKE protocol instantiations.

3.2.1 Generic Frameworks of Quantum-Resistant PAKE Protocols

Here, we describe some related works where the authors provide generic frameworks or constructions adhering to which quantum-resistant PAKE protocols can be instantiated.

1. *CAKE and OCAKE-PAKE*: The authors of [56] are the first to propose two novel generic constructions of PAKE protocols from a Key-Encapsulation Mechanism (KEM), namely, CAKE-PAKE and OCAKE-PAKE inspired by their classical versions Encrypted Key-Exchange (EKE) [3] and One-Way EKE (OEKE) [59] respectively. In a nutshell, for CAKE-PAKE, the idea is to encrypt the KEM public key pk and the KEM ciphertext C_K using the password of the client $\mathcal{C}$, thereby requiring two instances of the Ideal Cipher (IC). On the other hand, in OCAKE-PAKE, only one instance of the IC is required to encrypt just the pk. And, the C_K in OCAKE-PAKE is directly associated with the authentication (or otherwise referred to as key confirmation) tag of the server $\mathcal{S}$, which provides $\mathcal{S}$'s explicit authentication to $\mathcal{C}$ [15], [56]. The authors here consider the IC and Random Oracle (RO) model and provide the security proofs for both CAKE and OCAKE-PAKE in the Universal Composability (UC) framework, and both of them are bPAKE protocols as well [15]. Upon instantiating CAKE and OCAKE

PAKE with Kyber-1024 [23], they provide 102 and 162 bits of quantum security, respectively. Additionally, upon instantiating OCAKE-PAKE using Kyber-768 [23], it provides 92-bit quantum security [56].

2. *Tight Kyber (TK)-PAKE*: Pan and Zeng (2023) [49] also provide a generic framework using which quantum-resistant PAKE protocols based on KEMs can be instantiated. The authors closely compare TK-PAKE with CAKE-PAKE, and claim to have a tighter security bound. Based on our understanding, this is because the authors claim that any attack on TK-PAKE will be as difficult as having to break the current instance of the underlying KEM. So, it avoids the penalty levied in CAKE-PAKE when considering multiple sessions, decryption oracles, etc., which leads to a less tight security proof. The PAKE protocol resulting from this generic framework is (also) a *b*PAKE protocol like CAKE-PAKE. The authors here also consider the IC and RO model and provide the security proof for TK-PAKE in the Bellare-Pointcheval-Rogaway (BPR) model. Additionally, the authors state that, when TK-PAKE is implemented with Kyber-768 [23], it provides 152-bit security whereas CAKE-PAKE only provides 112-bit security [49]. We are unsure whether TK-PAKE provides 152-bit quantum or classical security, and secondly, as to how the authors of TK-PAKE have obtained this data about the security provided by CAKE-PAKE, as it is not mentioned originally in [56].

3. *No IC Encryption (NICE)-PAKE*: Alnahawi et al. (2024) [33] observed that quantum-resistant PAKE constructions or protocol instantiations based on KEMs such as [34], [49], [50], [56] etc. heavily rely on the IC model for their formal security analysis. In addition to the limitation mentioned in section 2.2 about ICs, the authors of NICE-PAKE also highlighted that no adaptations have been provided yet for the IC model to demonstrate its security against quantum adversaries. So, they proposed a framework named NICE which eliminates the usage of the IC model and provides insights regarding instantiations using Frodo-KEM [22] and Kyber [23], but the parameters have to be tweaked. The authors primarily emphasize the usage of Frodo-KEM over Kyber with appropriate parameter choices. Based on our current understanding, no explicit parameter sets have been defined by the authors, but they indeed recommend resorting to higher noise variance or higher dimensions with respect to the Learning With Errors (LWE) parameters and additionally recommend the usage of an LWE estimator tool [60]. We would also like to emphasize that the authors here consider the RO model and provide the security proof for NICE-PAKE (which is a *b*PAKE protocol [15]) in the BPR model. Nothing is explicitly mentioned regarding the quantum security provided by NICE-PAKE. But, to provide an insight, a correct instantiation of NICE-PAKE with appropriate parameter choices is guaranteed to provide (quantum) security equal to the National Institute of Standards and Technology (NIST) categories 1,3, and 5 with a minimal loss of efficiency [33].

4. *PAKE Combiners*: The authors of [58] acknowledge that, in recent times, significant efforts are being made to develop PAKE protocols which are quantum-resistant. This is generally achieved by implementing PAKE protocols using new post-quantum cryptographic primitives (such as LWE). The transition to post-quantum PAKE protocols based on these new post-quantum cryptographic primitives is relatively untested, so they emphasize the usage of hybrid schemes, i.e., PAKE protocols based on both classical (such as decisional Diffie-Hellman) and post-quantum (such as LWE) primitives. A hybrid approach that combines

a quantum-resistant PAKE with a classical one will allow the protocol to have at least a fallback scenario, in case vulnerabilities are discovered in the newer post-quantum cryptographic primitives. Considering this, the authors proposed two PAKE combiners, namely, a Parallel Combiner (ParComb) and a Sequential Combiner (SeqComb). In a brief overview, according to the authors here, in order to develop a hybrid PAKE using ParComb, it is imperative that both the chosen PAKE protocols must fulfill *perfect password hiding*. But, they have not found a single quantum-resistant PAKE protocol which fulfills this property (based on the information provided in the *Our Contribution* section in [58]), and the ones that indeed fulfill this property are exactly one-round classical PAKE protocols. The authors only provide the workflow of ParComb, considering there is a hypothetical quantum-resistant PAKE protocol which fulfills *perfect password hiding*, but due to the aforementioned reason, they could not definitively state the names of two PAKE protocols that can be used to practically realize a hybrid PAKE protocol using ParComb. Thus, they propose another hybrid PAKE using SeqComb, where they use CPace [61] as the classical one-round protocol which fulfills *perfect password hiding* and Compact Half-IC (HIC) (CHIC)-PAKE [34] as the quantum-resistant protocol fulfilling PSK*equality hiding*. We believe that the generated hybrid PAKE (using SeqComb) is likely to be a *b*PAKE protocol, because, we found that the CPace sub-step (from Figure 4 in Section 3.4 in AuCPace) in [61] depicts a *b*PAKE protocol where the server $\mathcal{S}$ and the client $\mathcal{C}$ share the same Password Related String (PRS) which later leads to the establishment of the session key and CHIC-PAKE is already a *b*PAKE protocol as mentioned later. The authors here consider the RO model and provide their security proofs in the UC framework in the classical setting.

3.2.2 Quantum-Resistant PAKE Protocol Instantiations

Here, we describe some other related works where the authors provide concrete instantiations of quantum-resistant PAKE protocols with information regarding parameter choices, implementation details, performance measurements, etc.

1. *Ring Learning With Errors (RLWE)-SRP*: Gao et al. (2018) [1] acknowledged that the Secure Remote Password (SRP) protocol [4], being an *a*PAKE protocol, has been widely deployed in real-world applications such as 1Password, Blizzard, Transport Layer Security (TLS), etc. With the advent of quantum computers, the security provided by the (classical) SRP protocol is at stake from quantum adversaries. So, they proposed an RLWE variant of SRP referred to as RLWE-SRP, which is a quantum-resistant *a*PAKE protocol. Based on the RLWE parameter choices adopted in RLWE-SRP, it provides 209 bits of security [1]. The authors do not explicitly mention whether RLWE-SRP provides 209 bits of quantum or classical security, but [15] states that this indeed refers to the classical security and not quantum security. Additionally, the authors provide a security proof for RLWE-SRP in the UC framework.

2. *Module Learning With Errors (MLWE)-PAKE*: Ren and Gu (2021) [52] realized that the first post-quantum secure PAKE protocol [2] lacks an efficient implementation, i.e., it has a high computation cost. Even though the authors of [62] were able to reduce the computation cost of [2], it came at the expense of a high communication overhead [52], and these shortcomings arose in these approaches

because of using either LWE or RLWE as the underlying cryptographic primitives and the chosen parameter sets. So, these authors proposed a lightweight and efficient quantum-resistant protocol (claimed to be aPAKE) based on MLWE, because, when designing a lattice-based scheme with multiple security considerations, MLWE proves to be more flexible and straightforward than other primitives like LWE and RLWE [52]. They proposed three variants, namely, LightWeight-PAK, Recommended-PAK, and Paranoid-PAK, providing security of 116, 177, and 239 bits, respectively. We wish to explicitly bring this to the reader's attention that LightWeight-PAK is not considered to be secure against quantum adversaries [52], so the 116-bit security is likely against classical adversaries only (which is also relatively unsecure in our opinion). The authors here only consider the Recommended-PAK and Paranoid-PAK to be indeed quantum-resistant, so going forward, for MLWE-PAKE, we will only consider the Recommended-PAK and Paranoid-PAK variants. The PAKE protocol has three communication flows between the two parties involved in the protocol execution while providing explicit mutual authentication. We wish to mention that the same MLWE-PAKE has also been proposed by Ren et al. (2023) [63], but with a different *paper title* which additionally provides an open-source implementation in C [64], but there is a discrepancy between their papers and their implementation. The authors mentioned in [52], [63] that, for the hash functions H_2, H_3 and H_4 they use SHA3-256 [29], whereas, in their implementation [64], they use SHAKE-256 XOF [29].

3. *Modified OCAKE-PAKE*: The OCAKE-PAKE protocol originally proposed in [56] provides a generic framework to construct PAKE protocols using KEMs (specifically Kyber-768 and Kyber-1024 [23]) and IC as mentioned earlier. The *original* OCAKE-PAKE furnished a security proof in the UC framework, but according to [50], that proof is not against quantum adversaries who can launch quantum attacks. Proving the security against quantum adversaries is considered difficult in the UC framework, especially when components such as ICs or ROs are involved, and this was left as an open problem in [56] (according to [50]). So, the authors slightly *modified* the *original* OCAKE-PAKE protocol and simultaneously addressed this issue by providing a game-based security proof of the *Modified* OCAKE-PAKE protocol in the BPR model. The authors observe that attempts are being made to adapt game-based security proofs to the quantum setting. So, they attempted to lay the groundwork for future propositions that are aimed at achieving *proven post-quantum security* against quantum adversaries, where the security proofs are also easy to formally verify. The modifications to the OCAKE-PAKE protocol compared to the original one are as follows [50], [56]:
   a) The *original* OCAKE-PAKE is a one-round (or exactly two-flow) protocol, whereas the *Modified* OCAKE-PAKE is a three-flow protocol (although the authors in [50] refer to it as a *three-round* protocol).
   b) Session Identifier (SSID) is not used in *Modified* OCAKE-PAKE.
   c) In *Modified* OCAKE-PAKE, a final key confirmation message is added on both the sever $\mathcal{S}$ and the client $\mathcal{C}$'s end, which allows $\mathcal{S}$ and $\mathcal{C}$ to achieve explicit mutual authentication, whereas, in the *original* OCAKE-PAKE, only $\mathcal{S}$ was authenticated by $\mathcal{C}$.

   We wish to explicitly mention that the *Modified* OCAKE-PAKE is still a bPAKE protocol like the *original* one [56]. For experimentation purposes, the authors demonstrated that the *Modified* OCAKE-PAKE can be instantiated with multiple

KEMs, e.g., Frodo-KEM [22], Kyber [23], and more. We believe this to be the reason that nothing is explicitly mentioned regarding the quantum security provided by *Modified* OCAKE-PAKE in [50]. We believe that the authors might want to indicate that the security of the PAKE protocol is equal to the NIST security level provided by the choice of the variant of the underlying KEM.

4. *CHIC-PAKE*: Arriaga et al. (2024) [34] realized that existing KEM based PAKEs (such as CAKE-PAKE [56]) are not efficient, as they use multiple instances of the IC (in this case, two). Other protocols like, EKE-KEM [65] uses only one instance of Half-IC (HIC) and OCAKE-PAKE [56] uses only one instance of IC. To be explicit, HIC (as introduced in [65]) and IC are completely different, even though fundamentally they refer to encryption schemes based on block ciphers. In these cases, the KEM ciphertext C_K is sent in the clear along with a key confirmation (or authentication) tag, which makes the server's message a commitment to a single password only. So, this second design approach with only one instance of HIC or IC is comparatively more practical and efficient. So, the authors here proposed CHIC-PAKE, which is an even compact version of EKE-KEM. They attempt to decrease the communication overhead and increase the computational efficiency by eliminating the need for the HIC abstraction (in EKE-KEM). CHIC-PAKE uses Kyber [23] where they split the KEM public key pk into two parts, one of which is a 32-byte random seed ρ and uses the Modified 2-Round Feistel (M2F) network (from EKE-KEM) directly on the ρ which somehow facilitates in reduction of communication overhead [34]. As opposed to EKE-KEM, a minuscule computation cost is also saved in CHIC-PAKE because, the client does not have to generate 32 bytes of randomness, and the inputs and outputs of the M2F construction is also somehow simplified, which is still unclear to us. CHIC-PAKE is a bPAKE protocol [15]. The authors consider the IC and RO models and provide the security proof for CHIC-PAKE in the UC framework. Although the quantum security provided by CHIC-PAKE is not explicitly mentioned, we believe that the authors do intend to state that the quantum security depends on the variant of Kyber, as described in Table 2.6.

5. *KEM-AE-PAKE*: Lyu et al. (2025) [51] proposed a novel generic compiler to transform a bPAKE into an aPAKE protocol using KEM and Authenticated Encryption (AE) in the RO model. The authors observed that bPAKE to aPAKE compilers do exist, but are bounded by limitations such as relying on signature schemes (which are inherently less efficient than KEMs) or the compiler itself being incompatible with lattice-based cryptographic assumptions. This motivated them to have a bPAKE to aPAKE compiler which is both efficient and instantiable from lattice-based cryptographic primitives. The authors explicitly state that the usage of AE provides explicit mutual authentication and *offline dictionary attack resistance* (from section 3.1). The authors apply their compiler on CAKE-PAKE [56] (which is a bPAKE protocol) to transform it into an aPAKE and instantiate it using Kyber-768 [23], but they do not explicitly mention the quantum security provided by their instantiation. We also wish to highlight a noteworthy point which we believe to be just a typing mistake and not a critical flaw in the protocol description of KEM-AE-PAKE in *Figure 9* in [51]. If we closely observe the usage of the two instances of the IC, i.e., IC_1 and IC_2, we see [51]:

   a) IC_1 uses $H_0(pw)$ as the input-key during encryption, whereas, IC_1 uses rw

as the input-key during decryption.

b) IC_2 uses rw as the input-key during encryption, whereas, IC_2 uses $H_0(pw)$ as the input-key during decryption.

We have established in section 2.2 that ICs are idealized versions of block ciphers, and using a different input-key during decryption (than the one used during encryption) will still result in a valid plaintext, but not the one that was originally encrypted. As the above usage of different input keys for both IC_1 and IC_2 during encryption and decryption will lead to the retrieval of completely different plaintexts than what was actually encrypted, it is not desirable in the case of a PAKE protocol execution. So, the same input key must be used for encryption and decryption.

3.3 Choosing the PAKE Protocols

In this section, we provide a brief methodology of how we are going to choose a PAKE protocol from section 3.2 to proceed further. Then, we do a brief security discussion of the PAKE protocols from section 3.2 while providing some details of our findings and our perspective on that. Then, following the methodology, we choose the PAKE protocols that will be implemented.

3.3.1 Our Methodology

We wish to state that, in this work, we choose a PAKE protocol (from the ones discussed in section 3.2) for further implementation and inclusion in the Rust framework based on the following criteria (ideally) in the exact order mentioned:

1. We evaluate them depending on the fulfillment of the security and functional requirements from section 3.1.

2. We could also consider choosing them based on the exact quantum security (in bits), so that it would be a fair comparison between two (chosen) PAKE protocols of the same category, namely, *b*PAKE and *a*PAKE. But, based on our observation until now, choosing them based on the quantum security in bits is not so easily possible because of either of the following:

    - The sources do not mention anything regarding the (quantum) security provided by the PAKE protocol in bits.
    - The sources do mention that their PAKE protocol instantiation provides a certain security in bits, but do not explicitly clarify whether it is against quantum or classical adversaries.

    So, for this criterion, we intend to consider a protocol depending on the (possibility of assigning a) NIST security level based on the scenarios mentioned in Figure 3.1.

3. Then, if two PAKE protocols (seem to) provide equivalent quantum security, primarily based on the NIST security categories (irrespective of whether it is assigned by the sources or by us), we could proceed as follows:

    a) We could try to evaluate them based on certain non-functional requirements, namely, providing explicit mutual authentication, the number of information flows needed for the PAKE protocol execution, etc., as mentioned in Table 3.3.

b) If the details of the PAKE protocol description have been provided with sufficient clarity in the sources. By sufficient clarity, we expect that we do not have to guess anything by ourselves due to a lack of clarity.

c) If it is observed that multiple PAKE protocols (ideally) use the same underlying (NIST-standardized) quantum-resistant cryptographic primitive, which also has a vetted open-source implementation, then we choose them.

d) We would (ideally try to) avoid resorting to the option where we have to implement the underlying cryptographic primitive ourselves, irrespective of the clarity with which the details of the cryptographic primitives have been provided. This is because, in such cases, the implementation will not undergo sufficient review and vetting for someone to have confidence in it.

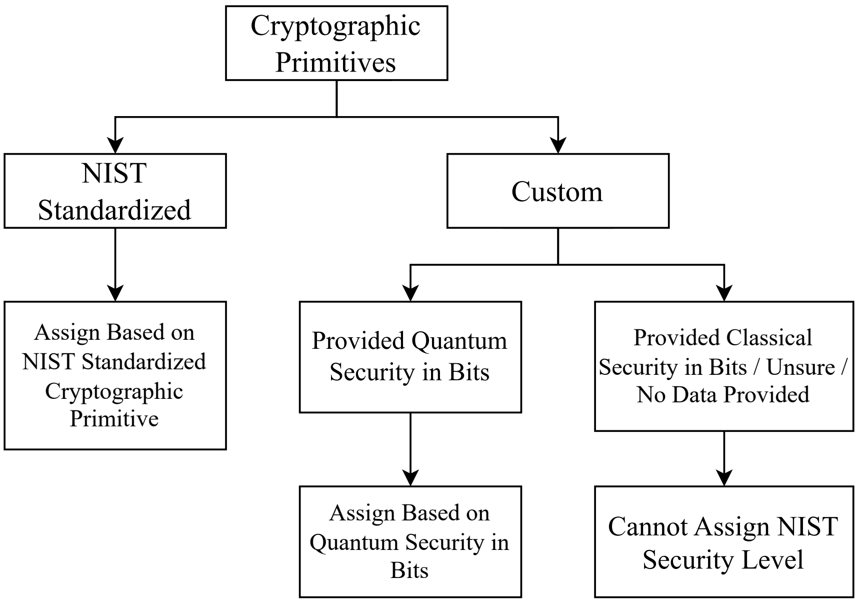
Cryptographic Primitives
NIST Standardized
Custom
Assign Based on NIST Standardized Cryptographic Primitive
Provided Quantum Security in Bits
Provided Classical Security in Bits / Unsure / No Data Provided
Assign Based on Quantum Security in Bits
Cannot Assign NIST Security Level

Figure 3.1: Scenarios when assigning NIST security levels

3.3.2 Discussion of Fulfillment of Requirements by PAKE Protocols

In this subsection, we initially present some clarification regarding the categories of PAKE protocols, and then we proceed with a brief security discussion of the PAKE protocols, which is also consolidated in Table 3.1.

Clarification Regarding Categories of PAKE Protocols

In section 3.1 in *precomputation resistance*, we mentioned that PAKE protocols fulfilling this requirement are considered as $strong - a$PAKE protocols (according to [12]). This means that PAKE protocols which are only able to fulfill *server compromise resistance* and not *precomputation resistance* are *just* aPAKE protocols. But in [13], [15], the authors do not segregate between aPAKE and $strong - a$PAKE, i.e., they only consider a PAKE protocol as aPAKE if it fulfills both *server compromise resistance* and *precomputation resistance* from section 3.1. And, they also consider that there are only two categories of PAKE protocols, namely, bPAKE and aPAKE, and no $strong - a$PAKE (where the bPAKE protocols do not fulfill *server compromise resistance* and *precomputation resistance*). So, in this work, we also adhere to the same analogy. We mentioned in chapter 1 that, for an *a*PAKE protocol, the (plaintext) password must not be stored on the server $\mathcal{S}$ and instead, a pre-shared verifier or token (not directly linked to the password) must be stored. We wish to explicitly clarify that, as opposed to not storing the (plaintext) password on $\mathcal{S}$, one must also *not* store just a hash digest of the password as the pre-shared verifier, as this cannot be considered a secure enough pre-shared verifier [1]. This is because storing just the hash digest of the password does not guarantee *precomputation resistance* to the best of our understanding, thus rendering the PAKE protocol not to be an *a*PAKE protocol.

This clarification regarding the categories of the PAKE protocols seemed imperative because we saw (and mentioned) in subsection 3.2.2 that MLWE-PAKE [52] is claimed to be an *a*PAKE protocol. But, if we carefully observe the PAKE protocol description in *section 4.1* in [52], it clearly states that $\mathcal{S}$ stores just a hash digest of the client $\mathcal{C}$'s password, and this does not guarantee *precomputation resistance* like we mentioned earlier. This leads us to question the validity of the claim that MLWE-PAKE is an *a*PAKE protocol, and surprisingly enough, [15] does not consider MLWE-PAKE to be an *a*PAKE protocol, i.e., according to them, MLWE-PAKE is a *b*PAKE protocol. Although they do not state their reasons for considering MLWE-PAKE as a *b*PAKE protocol, we believe that it must be because of the same reason we mentioned earlier.

We also hold (kind of) a similar opinion regarding KEM-AE-PAKE [51] because, in the protocol description in *Figure 9* in [51], $\mathcal{S}$ stores the tuple (rw, pk), referred to as the *password file* which can also be considered as the verifier. Now, $rw \leftarrow H_0(pw, \mathcal{C}_{id}, \mathcal{S}_{id})$ is basically the hash digest of $\mathcal{C}$'s identity and password (pw) and $\mathcal{S}$'s identity. In our opinion, this rw cannot be considered secure enough because of the following reasons:

1. $\mathcal{S}$'s identity is something that can be assumed to be public.
2. We know from chapter 1 that, in a PAKE protocol, more often than not, the passwords that are used are easy to remember and of low entropy, so they are likely to be weak.

Now, if an adversary $\mathcal{A}$ with fairly sufficient resources already pre-computes a table of values based on a dictionary of (commonly used) passwords, this already increases the chances of $\mathcal{C}$'s password being one from the used dictionary. We know that $\mathcal{A}$ is already aware of $\mathcal{S}_{id}$ and, if $\mathcal{S}$ is compromised, using the precomputed values, it might not take too long for $\mathcal{A}$ to identify the victimized $\mathcal{C}$'s identity and password. This allows $\mathcal{A}$ to completely reconstruct rw. Moreover, $\mathcal{A}$ can now reconstruct $r \leftarrow H_1(pw, \mathcal{C}_{id}, \mathcal{S}_{id})$ as well, which allows it to obtain $(pk, sk) \leftarrow \text{KeyGen}(r)$ and reconstruct the whole *password file*, i.e., (rw, pk) with respect to *Figure 9* in [51]. So, $\mathcal{A}$ can launch an impersonation attack against $\mathcal{C}$ without breaking a sweat. Thus, KEM-AE-PAKE should also *not* be considered as an *a*PAKE protocol. Although according to [15], KEM-AE-PAKE is an *a*PAKE protocol indicating that it fulfills both *server compromise resistance* and *precomputation resistance*. But we believe that KEM-AE-PAKE should also be classified as a *b*PAKE protocol based on our above reasoning.

Since, in both MLWE-PAKE and KEM-AE-PAKE, $\mathcal{S}$ does not directly store the (plaintext) password, it can be considered that they indeed fulfill *server compromise resistance* but not *precomputation resistance*. Additionally, in our opinion, the pre-shared verifier computed in RLWE-SRP [1] could be considered as a secure enough verifier. Even if $\mathcal{S}$ gets compromised, for an adversary $\mathcal{A}$ to be able to extract the client's password, it will be almost as difficult as having to break the RLWE problem. Thus, we acknowledge that, RLWE-SRP is indeed an *a*PAKE protocol fulfilling both *server compromise resistance* and *precomputation resistance* and it seems that our perspective is in accordance with [15] about RLWE-SRP.

Security Discussion

We do not explicitly go into the details of *offline dictionary attack resistance* and *online dictionary attack resistance* as these are the most fundamental security requirements that a PAKE protocol proposition must fulfill, and indeed the PAKE protocols discussed in section 3.2 do fulfill these. With respect to fulfillment of *forward secrecy* or *perfect forward secrecy*, details can be found from *Table 4* in [15] except for KEM-AE-PAKE. But, if we see KEM-AE-PAKE [51], they indeed claim to provide *forward secrecy* based on our (limited) understanding of their security discussion.

Table 3.1: Overview of security requirements fulfilled by PAKE protocols

PAKE Protocol	**Fundamental Security Requirements**					**Enhanced Security Requirements**	
	Offline Dictionary Attack Resistance	**Online Dictionary Attack Resistance**	**(Perfect) Forward Secrecy**	**Perfect Password Hiding**	**PSK Equality Hiding**	**Server Compromise Resistance**	**Precomp-utation Resistance**
CAKE-PAKE [56]	✓	✓	Perfect Forward Secrecy	✗	✓	✗	✗
OCAKE-PAKE [56]	✓	✓	Perfect Forward Secrecy	✗	✓	✗	✗
TK-PAKE [49]	✓	✓	Perfect Forward Secrecy	✗	☆	✗	✗
Modified OCAKE-PAKE [50]	✓	✓	Perfect Forward Secrecy	✗	☆	✗	✗
NICE-PAKE [33]	✓	✓	Perfect Forward Secrecy	✗	☆	✗	✗
CHIC-PAKE [34]	✓	✓	Perfect Forward Secrecy	✗	✓	✗	✗
MLWE-PAKE [52], [63]	✓	✓	Forward Secrecy	✗	☆	✓	✗
KEM-AE-PAKE [51]	✓	✓	Forward Secrecy	✗	⊙	✓	✗
RLWE-SRP [1]	✓	✓	Forward Secrecy	✗	⊙	✓	✓
PAKE Combiners [58]	✪	✪	✪	✪	✪	✪	✪

Legends

✓: Requirement Fulfilled, ✗: Requirement Not Fulfilled, ✪: Depends on Underlying PAKE Protocol, ⊙: No Data is Available, ☆: Unsure

We wish to reiterate that the authors of [58] have mentioned that, to date, none of the quantum-resistant PAKE protocols fulfill the *perfect password hiding* requirement. Furthermore, with respect to PSK*equality hiding*, the authors of [58] claim that, *only the recent quantum-resistant universally composable* PAKE protocols (like [34], [56]) fulfill this property. Although this is a security requirement, we believe that it is primarily applicable for hybrid PAKE protocol schemes as opposed to (stand-alone) PAKE protocols in general. We already provided our clarification regarding fulfillment of *server compromise resistance* and *precomputation resistance* earlier. A consolidated overview of the fulfilment of security requirements is mentioned in Table 3.1.

Based on the data from Table 3.1, we do not think of OCAKE-PAKE [56] to be *more secure* than *Modified* OCAKE-PAKE [50], considering them as stand-alone PAKE protocols (outside of a hybrid scheme). We hold the same ideology for CAKE-PAKE [56] and TK-PAKE [49]. Additionally, to be explicit, all the PAKE protocols from section 3.2 indeed fulfill both the functional requirements from section 3.1.

3.3.3 Assigning NIST Security Level

Based on our methodology from subsection 3.3.1, here we go through every PAKE protocol from section 3.2 and (try to) assign a NIST security level to that. But at first, we briefly recapitulate the information with reference to Table 2.3, Table 2.4, Table 2.6, and Figure 3.1. The consolidated overview of the PAKE protocols with an assigned NIST security level is mentioned in Table 3.2.

Table 3.2: Consolidated overview of PAKE protocols with assigned NIST security levels

Category	**PAKE Protocol**	**Cryptographic Primitive Used**	**Quantum Security in Bits**	**NIST Security Category**
*b*PAKE Protocol	CAKE-PAKE [56]	Kyber-1024 [23]	102	*Level-V*
	OCAKE-PAKE [56]		162	
		Kyber-768 [23]	92	*Level-III*
	TK-PAKE [49]		$152^{?}$	
	Modified OCAKE-PAKE [50]	Kyber [23], Frodo-KEM [22] and More	✪	✘
	CHIC-PAKE [34]	Kyber [23]		
	NICE-PAKE [33]	Kyber [23] and Frodo-KEM [22] ✰	⊙	
	KEM-AE-PAKE [51]	Kyber-768 [23]		*Level-III*
	Recommended MLWE-PAKE [52], [63]	★	177	*Level-V*
	Paranoid MLWE-PAKE [52], [63]		239	
	PAKE Combiners ParComb and SeqComb [58]	✘	✪	✘
*a*PAKE Protocol	RLWE-SRP [1]	★	⊙	

Legends

?: Unclear Whether it is Quantum Security or Not, ⊙: No Data is Available, ✘: Not Applicable, ✪: Depends on Underlying Cryptographic Primitive and/or Parameter Choice, ★: Explicit Parameter Set Provided, ✰: NIST-Standardized Cryptographic Primitive With Modified Parameters

Recapitulation of the Previous References: In a nutshell, Figure 3.1 states that, we can assign a NIST security level to a PAKE protocol in either of the following scenarios:

1. If it uses a NIST standardized cryptographic primitive, irrespective of whether the (quantum) security in bits is mentioned in the original sources or not.
2. Based on the exact quantum security in bits provided by the PAKE protocol as mentioned in the sources.

In other cases, such as the PAKE protocol uses custom cryptographic primitives and/or no details about the (quantum) security provided in bits are mentioned in the sources, then we will not be able to assign a NIST security level to that PAKE protocol. If we are indeed able to assign a NIST security level to a PAKE protocol, they will mainly be *Level-I*, *Level-III*, and *Level-V*. This is because of the following:

1. Majority of the PAKE protocols from section 3.2 use the NIST standardized cryptographic primitive Kyber [23] whose variants only offer *Level-I*, *Level-III*, and *Level-V* as per Table 2.6.
2. Even if they use other primitives like Frodo-KEM [22] according to the authors the different variants of Frodo-KEM also only provide security *Level-I*, *Level-III*, and *Level-V* as per Table 2.4.
3. As per our observation, even if they use completely custom parameter sets, the provided quantum security in bits is mostly claimed to be equivalent to NIST security *Level-V*.

Reasoning for Assigning NIST Security Levels: The NIST security level is assigned to the PAKE protocol as follows:

- CAKE and OCAKE-PAKE [56]: Using Kyber-1024 [23], CAKE and OCAKE-PAKE respectively provide 102 and 162 bits of quantum security, and, using Kyber-768 [23], OCAKE-PAKE provides 92 bits of quantum security. So, based on the cryptographic primitive, we assign NIST *Level-V* to CAKE and OCAKE-PAKE when using Kyber-1024 and NIST *Level-III* to OCAKE-PAKE when using Kyber-768.
- TK-PAKE [49]: Using Kyber-768, TK-PAKE provides 152-bit security. It is unclear whether this security is against quantum or classical adversaries. But, since it uses Kyber-768 which is a NIST standardized cryptographic primitive, we assign NIST *Level-III* to TK-PAKE.
- NICE-PAKE [33]: The authors recommend tweaking the parameters of Frodo-KEM [22] and Kyber [23] for NICE-PAKE. But, no parameter sets were explicitly provided in the sources, and more importantly, no details are provided regarding the quantum security in bits. So, it is not possible for us to assign any NIST security level to NICE-PAKE.
- RLWE-SRP [1]: Using custom parameters for RLWE, RLWE-SRP provides 209-bit of classical security (as per [15]). As it neither uses a NIST standardized cryptographic primitive nor any details regarding the quantum security in bits are mentioned, it is not possible for us to assign any NIST security level to RLWE-SRP.

- MLWE-PAKE [52], [63]: We only consider Recommended-PAK and Paranoid-PAK here, as Lightweight-PAK is not considered quantum-resistant in the sources. Although Recommended-PAK and Paranoid-PAK use custom MLWE parameters, they provide 177 and 239 bits of security against quantum adversaries. So, based on the quantum security (in bits), we assign NIST *Level-V* for both Recommended-PAK and Paranoid-PAK, where Paranoid-PAK can be considered to be even more quantum-resistant than Recommended-PAK.

- *Modified* OCAKE-PAKE [50]: *Modified* OCAKE-PAKE has been instantiated with (variants of) multiple KEMs, like Kyber [23] (which is the only NIST standardized primitive), Frodo-KEM [22] etc. without requiring modifications to the underlying parameters of the KEMs. So, if the *Modified* OCAKE-PAKE is instantiated using Kyber-512, Kyber-768 and Kyber-1024, then the corresponding NIST security level will be *Level-I*, *III*, and *V* respectively. The same holds if it uses Frodo-640, Frodo-976, and Frodo-1344 respectively.

- CHIC-PAKE [34]: A similar explanation like *Modified* OCAKE-PAKE from above holds for CHIC-PAKE as well. As it only uses Kyber, if CHIC-PAKE is instantiated using Kyber-512, Kyber-768 and Kyber-1024, then the corresponding NIST security level will be *Level-I*, *III*, and *V* respectively.

- KEM-AE-PAKE [51]: KEM-AE-PAKE is instantiated using Kyber-768, but nothing is explicitly mentioned regarding the (quantum) security provided in bits. But, since it uses Kyber-768 which is a NIST standardized cryptographic primitive, we assign NIST *Level-III* to KEM-AE-PAKE.

- PAKE Combiners [58]: The security provided by the resulting hybrid PAKE protocol critically depends on the choice of the underlying PAKE protocols for both SeqComb and ParComb. So, we cannot assign any NIST security level to the hybrid PAKE protocol resulting from either SeqComb or ParComb.

3.3.4 Our Reasoning and Choice of the PAKE protocols

Based on subsection 3.3.3, owing to the infeasibility of assigning a NIST security level to NICE-PAKE [33], SeqComb and ParComb from PAKE Combiners [58] and RLWE-SRP [1], we do not consider them further to simplify the decision-making process. So, this leaves us with CAKE and OCAKE-PAKE [56], TK-PAKE [49], *Modified* OCAKE-PAKE [50], CHIC-PAKE [34], MLWE-PAKE [52], and KEM-AE-PAKE [51]. Now, based on our methodology from subsection 3.3.1, considering CAKE-PAKE and TK-PAKE, both of them provide a detailed protocol description in the sources with sufficient clarity and use a NIST standardized cryptographic primitive. Based on our security discussion (from Table 3.1 and subsection 3.3.2) regarding these two PAKE protocols, and the fact that they have the same number of information flows (as shown in Table 3.3), both of them seem equally appealing. But, for this task, we choose to proceed with TK-PAKE as it is claimed to have tighter security bounds than CAKE-PAKE as mentioned in subsection 3.2.1.

Similarly, considering OCAKE-PAKE and *Modified* OCAKE-PAKE, both of them also provide a detailed protocol description in the sources with sufficient clarity and use a NIST standardized cryptographic primitive. Based on the security discussion (from Table 3.1 and subsection 3.3.2), they also seem equally secure. *Modified* OCAKE-PAKE

Table 3.3: Overview of non-functional Requirements fulfilled by PAKE protocols

PAKE Protocol	**Explicit Mutual Authentication**	**Information Flows**
CAKE-PAKE [56]	✕	2
OCAKE-PAKE [56]	☆	
TK-PAKE [49]	✕	
Modified OCAKE-PAKE [50]	✓	3
NICE-PAKE [33]	✕	2
CHIC-PAKE [34]	✓	
Recommended MLWE-PAKE [52], [63]		3
Paranoid MLWE-PAKE [52], [63]		
KEM-AE-PAKE [51]		
RLWE-SRP [1]		2
PAKE Combiners ParComb and SeqComb [58]	✪	

Legends
✓: Requirement Fulfilled, ✕: Requirement Not Fulfilled, ✪: Depends on Underlying PAKE Protocol, ☆: Partially Fulfilled

requires one more information flow than *original* OCAKE-PAKE (as evident from Table 3.3) for achieving explicit mutual authentication. In our opinion, this is a fair trade-off between them, but for this work, we prefer to proceed with *Modified* OCAKE-PAKE, which provides explicit mutual authentication for both parties at the cost of one more information flow. Additionally, in our opinion, *Modified* OCAKE-PAKE is also cryptographically agile as it can be instantiated using different KEMs without requiring any change to the PAKE protocol design itself.

KEM-AE-PAKE also stands out to be a suitable candidate because of a detailed protocol description and usage of a NIST standardized cryptographic primitive, and it also provides *server compromise resistance* as evident from Table 3.1. For CHIC-PAKE and MLWE-PAKE, we would have to implement (part of) the whole cryptographic primitive ourselves. This is especially applicable for CHIC-PAKE, as having to implement the M2F network makes it extremely error-prone due to a lack of (our understanding of the) details regarding the same. So, we would ideally avoid these two protocols from further consideration.

Considering the above arguments, we choose to proceed further with the following protocols:

1. TK-PAKE [49]
2. *Modified* OCAKE-PAKE [50]
3. KEM-AE-PAKE [51]

3.4 Description of Chosen PAKE Protocols

In this section, we provide a detailed description of each of the chosen PAKE protocols from subsection 3.3.4, but first, we briefly provide (and reiterate) some common details below that are applicable to the chosen PAKE protocols:

1. The encryption and decryption functions used in all the chosen PAKE protocols from subsection 3.3.4 are assumed to be an IC.
2. We reiterate (from subsection 2.6.4) that, in different protocol descriptions, the KeyGen method of Kyber [23] might be parametrized or not, and both of them are equally applicable as long as the protocols do not require modifications to the predefined parameters of Kyber from Table 2.6.
3. The authors of TK-PAKE [49] and *Modified* OCAKE-PAKE [50] do not explicitly mention anything regarding a separate client registration phase which precedes the actual PAKE protocol execution. But, based on where PAKE protocols are generally used in the real world, we believe that having a client registration phase goes without saying. Thus, in Figure 3.2 and Figure 3.3 for TK-PAKE and *Modified* OCAKE-PAKE respectively, we have included a client registration phase as well before the actual PAKE protocol execution.
4. Finally, we consider that the server's identity ($\mathcal{S}_{id}$) is public knowledge, so it is known to the clients.

The Client Registration Phase in TK-PAKE and Modified OCAKE-PAKE: We mentioned earlier that, unlike us, the authors (in [49], [50]) do not keep a separate client registration phase. So, in our opinion, this is a one-time client registration phase (in Figure 3.2 and Figure 3.3) that happens any time before the actual PAKE protocol execution takes place. In this phase, the client just sends its identity ($\mathcal{C}_{id}$) and password ($\mathcal{C}_{pw}$), and the server just stores these details for future use. In reality, upon initiation of the PAKE protocol execution, the server would ideally tally the $\mathcal{C}_{id}$ with the ones that were stored earlier, but nothing of such sort has been mentioned in [49], [50]. We believe that this is probably the reason why the authors did not keep a separate client registration phase.

3.4.1 TK-PAKE Protocol

The TK-PAKE protocol [49], as depicted in Figure 3.2, is a two-flow (or one-round) *b*PAKE protocol in a client-server setting. It has two phases, namely, the client registration phase, as mentioned above, and the protocol execution phase, which is described below. Before diving further into the details, we wish to mention that nothing is explicitly mentioned in [49] regarding the choice of H.

PAKE Protocol Execution Phase

The TK-PAKE protocol execution involves three distinct steps which are as follows [49]:

1. $C_1 \leftarrow$ client_init(): In this step, the client $\mathcal{C}$ generates a KEM key pair (pk, sk) and encrypts pk using $\mathcal{C}_{pw}$ as the input-key resulting in the ciphertext C_1, which is subsequently sent to the server $\mathcal{S}$.

2. $C_2 \leftarrow \text{server\_resp}(C_1)$: Upon receiving C_1, $\mathcal{S}$ decrypts it using $\mathcal{C}_{pw}$ as the input-key. Only if $\mathcal{S}$ honestly uses the same $\mathcal{C}_{pw}$ that was received during the registration phase, the decryption will yield a KEM public key such that $pk' = pk$. Then, $\mathcal{S}$ performs an encapsulation using pk' resulting in a tuple of the KEM ciphertext C_K and the shared secret key $\mathcal{K}_K$. Then, $\mathcal{S}$ encrypts C_K using $\mathcal{C}_{pw}$ as the input-key to generate the ciphertext C_2. Again, usage of the correct $\mathcal{C}_{pw}$ is necessary to ensure that C_2 can be properly decrypted by (an honest) $\mathcal{C}$. Finally, $\mathcal{S}$ derives the shared session key $\text{SK}_{\mathcal{S}}$.

3. $\text{client\_ter\_init}(C_2)$: In the final step, $\mathcal{C}$ decrypts C_2 using the password $\mathcal{C}_{pw}$ to recover the KEM ciphertext C'_K. If the same $\mathcal{C}_{pw}$ that was sent during registration is used, then $C'_K = C_K$, generated by $\mathcal{S}$. Then, $\mathcal{C}$ decapsulates C'_K using sk, and if this is the correct sk that was generated earlier, then $\mathcal{K}'_K = \mathcal{K}_K$, which was generated by $\mathcal{S}$. Finally, $\mathcal{C}$ computes the shared session key $\text{SK}_{\mathcal{C}}$.

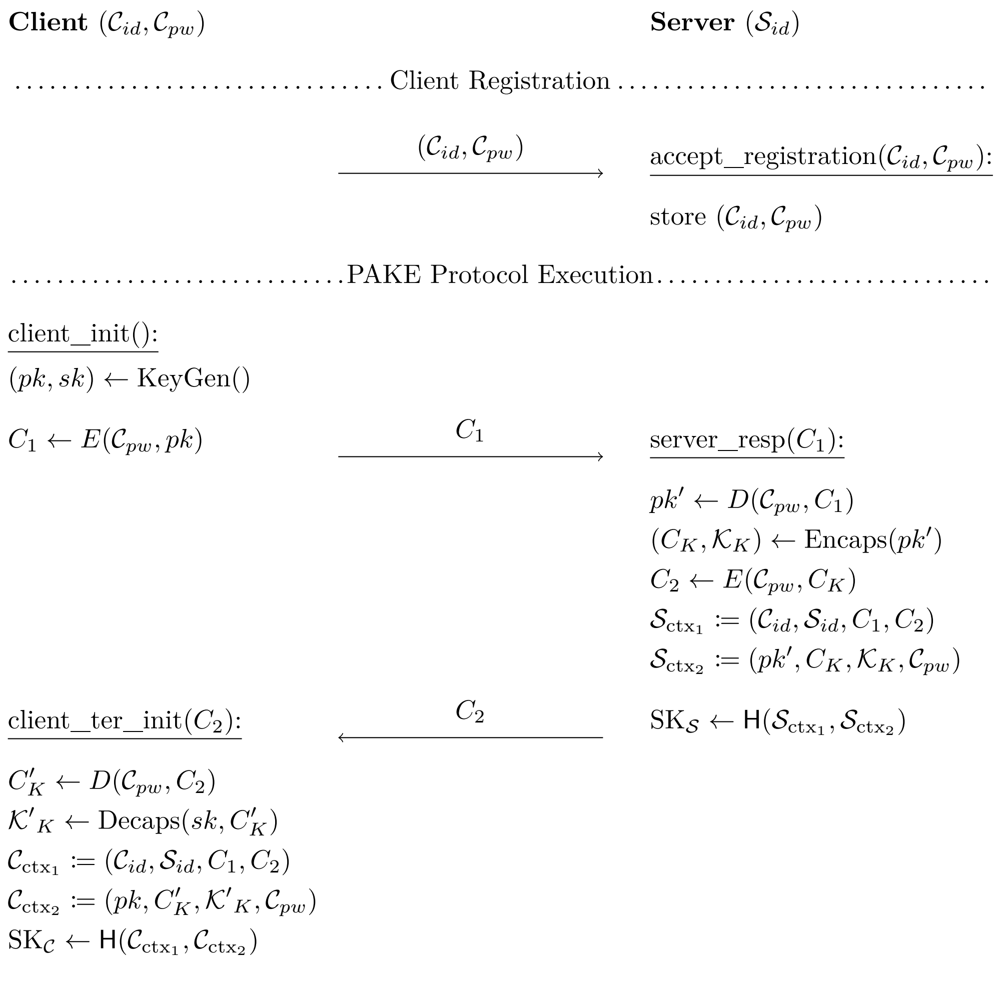
Client $(\mathcal{C}_{id}, \mathcal{C}_{pw})$
Server $(\mathcal{S}_{id})$
Client Registration
$(\mathcal{C}_{id}, \mathcal{C}_{pw})$
$\text{accept\_registration}(\mathcal{C}_{id}, \mathcal{C}_{pw})$:
store $(\mathcal{C}_{id}, \mathcal{C}_{pw})$
PAKE Protocol Execution
client_init():
$(pk, sk) \leftarrow \text{KeyGen}()$
$C_1 \leftarrow E(\mathcal{C}_{pw}, pk)$
C_1
$\text{server\_resp}(C_1)$:
$pk' \leftarrow D(\mathcal{C}_{pw}, C_1)$
$(C_K, \mathcal{K}_K) \leftarrow \text{Encaps}(pk')$
$C_2 \leftarrow E(\mathcal{C}_{pw}, C_K)$
$\mathcal{S}_{\text{ctx}_1} := (\mathcal{C}_{id}, \mathcal{S}_{id}, C_1, C_2)$
$\mathcal{S}_{\text{ctx}_2} := (pk', C_K, \mathcal{K}_K, \mathcal{C}_{pw})$
$\text{SK}_{\mathcal{S}} \leftarrow \mathsf{H}(\mathcal{S}_{\text{ctx}_1}, \mathcal{S}_{\text{ctx}_2})$
C_2
$\text{client\_ter\_init}(C_2)$:
$C'_K \leftarrow D(\mathcal{C}_{pw}, C_2)$
$\mathcal{K}'_K \leftarrow \text{Decaps}(sk, C'_K)$
$\mathcal{C}_{\text{ctx}_1} := (\mathcal{C}_{id}, \mathcal{S}_{id}, C_1, C_2)$
$\mathcal{C}_{\text{ctx}_2} := (pk, C'_K, \mathcal{K}'_K, \mathcal{C}_{pw})$
$\text{SK}_{\mathcal{C}} \leftarrow \mathsf{H}(\mathcal{C}_{\text{ctx}_1}, \mathcal{C}_{\text{ctx}_2})$

Figure 3.2: TK-PAKE protocol description [49]

Iff all the above steps on both parties' end are executed completely honestly, then both the session keys, namely, $\text{SK}_{\mathcal{S}}$ and $\text{SK}_{\mathcal{C}}$ must be identical.

3.4.2 Modified OCAKE-PAKE Protocol

The *Modified* OCAKE-PAKE protocol [50] in Figure 3.3 is a three-flow *b*PAKE protocol in a client-server setting. It has two phases, namely, the client registration phase, which was described earlier, and the protocol execution phase, which is described below. Before diving further into the details, we wish to mention that nothing is explicitly mentioned in [50] regarding the choice of hash function H and SHAKE Extendable Output Function (XOF) G (referred to as *KDF*).

PAKE Protocol Execution Phase

The *Modified* OCAKE-PAKE protocol execution involves four distinct steps which are as follows [50]:

1. $C \leftarrow$ client_init(): In this step, the client $\mathcal{C}$ computes an XOF digest pwd of the password $\mathcal{C}_{pw}$ using G. In our understanding, the purpose of this operation is to transform the (potentially low-entropy) password $\mathcal{C}_{pw}$ into a more suitable value so that it can be used as an input key for encryption and decryption purposes. Next, $\mathcal{C}$ generates a KEM key pair (pk, sk) and encrypts pk using pwd as the input-key. Finally, the ciphertext C is sent to server $\mathcal{S}$.

2. $(C_K, \tau_1) \leftarrow$ server_resp(C): Upon receiving C, $\mathcal{S}$ also computes an XOF digest pwd' of the password $\mathcal{C}_{pw}$ using G, which is then used to decrypt C. Correctness is ensured iff the server uses the same $\mathcal{C}_{pw}$ that was provided during the registration phase. In that case, $pwd' = pwd$, and the decryption yields the correct KEM public key $pk' = pk$ that was originally encrypted by $\mathcal{C}$. Otherwise, an incorrect pk' will inevitably lead to the protocol getting aborted in the further steps. Then, $\mathcal{S}$ performs encapsulation to generate a tuple consisting of the KEM ciphertext C_K and the shared secret key $\mathcal{K}_K$. It also computes an authentication tag τ_1 using the hash function H. Finally, $\mathcal{S}$ sends both C_K and τ_1 to $\mathcal{C}$.

3. $\tau_2 \leftarrow$ client_finish(C_K, τ_1): Here, $\mathcal{C}$ decapsulates C_K to recover the shared secret key $\mathcal{K}'_K$. It is important to note that if $pk' \neq pk$, or if $\mathcal{C}$ uses an incorrect KEM secret key $sk' \neq sk$ or ciphertext $C'_K \neq C_K$, then $\mathcal{K}'_K \neq \mathcal{K}_K$. Then, $\mathcal{C}$ computes two authentication tags, namely, τ_2 and τ'_1, using H. It verifies whether $\tau'_1 \overset{?}{=} \tau_1$, and iff they match, then $\mathcal{C}$ derives the shared session key $\mathrm{SK}_{\mathcal{C}}$ otherwise it aborts. Additionally, $\tau'_1 \overset{?}{=} \tau_1$ also serves as the explicit authentication of $\mathcal{S}$ to $\mathcal{C}$. Finally, τ_2 is sent to $\mathcal{S}$ if $\mathcal{S}$'s authentication is successful on $\mathcal{C}$'s end.

4. server_finish(τ_2): In the final step, $\mathcal{S}$ computes the authentication tag τ'_2 using H and checks whether $\tau'_2 \overset{?}{=} \tau_2$. If this verification succeeds, this serves as the explicit authentication of $\mathcal{C}$ to $\mathcal{S}$. Finally, $\mathcal{S}$ derives the shared session key $\mathrm{SK}_{\mathcal{S}}$.

Iff all the above steps are executed completely honestly, then only both parties are able to derive the session key on their end, which must be identical, i.e., $\mathrm{SK}_{\mathcal{S}} = \mathrm{SK}_{\mathcal{C}}$.

3.4.3 KEM-AE-PAKE Protocol

The KEM-AE-PAKE protocol [51] in Figure 3.4 is a three-flow *b*PAKE protocol in a client-server setting. It has two phases, namely, the client registration phase and the protocol execution phase, which are described below. Before diving into further details, we wish to mention that nothing is explicitly mentioned in [51] regarding the choice of H_0, H_1, H_2, H_3, H_4, and G.

```
Client ($\mathcal{C}_{id}, \mathcal{C}_{pw}$)                                  Server ($\mathcal{S}_{id}$)
..................................Client Registration..................................
                        $\xrightarrow{(\mathcal{C}_{id}, \mathcal{C}_{pw})}$           accept_registration($\mathcal{C}_{id}, \mathcal{C}_{pw}$):
                                                     store ($\mathcal{C}_{id}, \mathcal{C}_{pw}$)
...............................PAKE Protocol Execution...............................
client_init():
$pwd \leftarrow \mathsf{G}(\mathcal{C}_{pw})$
$(pk, sk) \leftarrow \text{KeyGen}()$
$C \leftarrow E(pwd, pk)$           $\xrightarrow{C}$           server_resp($C$):
                                                     $pwd' \leftarrow \mathsf{G}(\mathcal{C}_{pw})$
                                                     $pk' \leftarrow D(pwd', C)$
                                                     $(C_K, \mathcal{K}_K) \leftarrow \text{Encaps}(pk')$
                                                     $\mathcal{S}_{\text{ctx}_1} := (\mathcal{C}_{pw}, C, pk')$
                                                     $\mathcal{S}_{\text{ctx}_2} := (C_K, \mathcal{K}_K, \mathcal{S}_{id})$
client_finish($C_K, \tau_1$):     $\xleftarrow{(C_K, \tau_1)}$     $\tau_1 \leftarrow \mathsf{H}(\mathcal{S}_{\text{ctx}_1}, \mathcal{S}_{\text{ctx}_2})$
$\mathcal{K}'_K \leftarrow \text{Decaps}(sk, C_K)$
$\mathcal{C}_{\text{ctx}_1} := (\mathcal{C}_{pw}, C, pk)$
$\mathcal{C}_{\text{ctx}_2} := (C_K, \mathcal{K}'_K, \mathcal{C}_{id})$
$\tau_2 \leftarrow \mathsf{H}(\mathcal{C}_{\text{ctx}_1}, \mathcal{C}_{\text{ctx}_2})$
$\mathcal{C}_{\text{ctx}_3} := (C_K, \mathcal{K}'_K, \mathcal{S}_{id})$
$\tau'_1 \leftarrow \mathsf{H}(\mathcal{C}_{\text{ctx}_1}, \mathcal{C}_{\text{ctx}_3})$
if $\tau'_1 \neq \tau_1$ then
  abort
else
  $\text{SK}_{\mathcal{C}} \leftarrow \mathsf{G}(\tau_1, \mathcal{K}'_K)$     $\xrightarrow{\tau_2}$     server_finish($\tau_2$):
                                                     $\mathcal{S}_{\text{ctx}_3} := (C_K, \mathcal{K}_K, \mathcal{C}_{id})$
                                                     $\tau'_2 \leftarrow \mathsf{H}(\mathcal{S}_{\text{ctx}_1}, \mathcal{S}_{\text{ctx}_3})$
                                                     if $\tau'_2 \neq \tau_2$ then
                                                       abort
                                                     else
                                                       $\text{SK}_{\mathcal{S}} \leftarrow \mathsf{G}(\tau_1, \mathcal{K}_K)$
```

Figure 3.3: *Modified* OCAKE-PAKE protocol description [50]

Client Registration Phase

This is a one-time client registration phase that occurs prior to the actual execution of the PAKE protocol as shown in Figure 3.4. In this phase, the client $\mathcal{C}$ generates a value rw and a seed r using the hash functions H_0 and H_1 respectively. This seed r is needed for performing the seeded KeyGen operation, specifically $(pk_1, sk_1) \leftarrow \text{KeyGen}(r)$. As previously mentioned in subsection 2.6.4, the seed (r) must be exactly 64 bytes in length. The client assembles the (*not so secure*) *password file* or verifier (rw, pk_1) (as additionally clarified in subsection 3.3.2), which is designed to avoid sending the plaintext password $\mathcal{C}_{pw}$ to the server $\mathcal{S}$. In most cases, KeyGen internally generates the necessary randomness, however, in this case, the seed $r := \mathsf{H}_1(\mathcal{C}_{pw}, \mathcal{C}_{id}, \mathcal{S}_{id})$ is used. We believe the reason for this is to enable the (deterministic) regeneration of the same key pair (pk_1, sk_1) at a later point in time by the legitimate client who possesses the correct values $\mathcal{C}_{pw}, \mathcal{C}_{id}$, and $\mathcal{S}_{id}$. Finally, $\mathcal{C}$ sends the tuple $(\mathcal{C}_{id}, rw, pk_1)$ to the server $\mathcal{S}$. It is worth noting that in *Figure 9* of [51], the authors illustrate only the transmission of rw and pk_1, omitting $\mathcal{C}_{id}$. While this may appear inconsistent, it is plausible that the authors assume $\mathcal{C}_{id}$ is already known to $\mathcal{S}$. Upon receiving $(\mathcal{C}_{id}, rw, pk_1)$, $\mathcal{S}$ simply stores this information for future use.

PAKE Protocol Execution Phase

The KEM-AE-PAKE protocol execution involves four distinct steps which are as follows [51]:

1. $C_1 \leftarrow$ client_init(): In this step, the client $\mathcal{C}$ generates a second KEM key pair (pk_2, sk_2). Unlike the first key pair generated during registration, this is not a seeded KeyGen operation, as it is meant to be ephemeral and generated anew for every invocation of the PAKE protocol execution. The client then encrypts pk_2 using the verifier rw as the input key, resulting in the ciphertext C_1, which is sent to the server $\mathcal{S}$.

2. $(C_2, \psi) \leftarrow$ server_resp(C_1): Upon receiving C_1, $\mathcal{S}$ first performs a KEM encapsulation using pk_1, thereby generating a tuple of the KEM ciphertext C_{K1} and shared secret key $\mathcal{K}_{K1}$. Next, $\mathcal{S}$ decrypts C_1 using the verifier rw to retrieve the second KEM public key pk_2'. This decryption yields the correct value, i.e., $pk_2' = pk_2$, only if the server uses the same rw that was obtained during the client registration phase. Then, $\mathcal{S}$ performs encapsulation using pk_2', producing a tuple of the KEM ciphertext C_{K2} and shared secret key $\mathcal{K}_{K2}$. Then, $\mathcal{S}$ proceeds to encrypt C_{K2} using rw as the input key, resulting in the ciphertext C_2. Now, decryption of C_2 by $\mathcal{C}$ will yield the correct C_{K2} if it also uses the same rw as it had generated during the registration phase. Subsequently, $\mathcal{S}$ generates an authentication tag τ using G, and computes a hash digest of τ using H_2. This hash digest is then used as the input key to encrypt C_{K1} using AE_E, resulting in the ciphertext ψ. Finally, $\mathcal{S}$ sends C_2 and ψ to $\mathcal{C}$. According to the authors, the use of the AE scheme in this step enables explicit mutual authentication and contributes to *offline dictionary attack resistance*. The authors mention that recent KEMs have no *anonymity* under KEM secret key leakage. In other words, if a KEM ciphertext C_K generated under key-pair (pk, sk) is decapsulated using the correct sk, then the resulting values will be close to 0 or close to $q/2$ (where q is the prime modulus of Kyber), whereas, if C_K is decapsulated using some other (incorrect) sk', then the resulting values will neither be close to 0 nor to $q/2$. So, if C_{K1} were transmitted in the clear, an adversary could attempt such an offline (dictionary) attack by guessing different client passwords, regenerating the key pair (pk_1', sk_1') using H_1, and checking whether C_{K1} decapsulates to a value in the expected range or not. Then, with a high probability, the adversary would immediately know

Client $(\mathcal{C}_{id}, \mathcal{C}_{pw})$		**Server** $(\mathcal{S}_{id})$
	.... Client Registration	
generate_registration_details():		
$rw \leftarrow \mathsf{H}_0(\mathcal{C}_{pw}, \mathcal{C}_{id}, \mathcal{S}_{id})$		
$r \leftarrow \mathsf{H}_1(\mathcal{C}_{pw}, \mathcal{C}_{id}, \mathcal{S}_{id})$		
$(pk_1, sk_1) \leftarrow \text{KeyGen}(r)$	$\xrightarrow{(\mathcal{C}_{id}, rw, pk_1)}$	accept_registration($\mathcal{C}_{id}, rw, pk_1$):
		store $(\mathcal{C}_{id}, rw, pk_1)$
	.... PAKE Protocol Execution	
client_init():		
$(pk_2, sk_2) \leftarrow \text{KeyGen}()$		
$C_1 \leftarrow E(rw, pk_2)$	$\xrightarrow{C_1}$	server_resp(C_1):
		$(C_{K1}, \mathcal{K}_{K1}) \leftarrow \text{Encaps}(pk_1)$
		$pk_2' \leftarrow D(rw, C_1)$
		$(C_{K2}, \mathcal{K}_{K2}) \leftarrow \text{Encaps}(pk_2')$
		$C_2 \leftarrow E(rw, C_{K2})$
		$\tau \leftarrow \mathsf{G}(\mathcal{C}_{id}, \mathcal{S}_{id}, C_1, C_2, \mathcal{K}_{K2})$
client_finish(C_2, ψ):	$\xleftarrow{(C_2, \psi)}$	$\psi \leftarrow \text{AE}_E(\mathsf{H}_2(\tau), C_{K1})$
$C_{K'_2} \leftarrow D(rw, C_2)$		
$\mathcal{K}'_{K2} \leftarrow \text{Decaps}(sk_2, C_{K'_2})$		
$\tau \leftarrow \mathsf{G}(\mathcal{C}_{id}, \mathcal{S}_{id}, C_1, C_2, \mathcal{K}'_{K2})$		
if $\text{AE}_D(\mathsf{H}_2(\tau), \psi) \neq \perp$ **then**		
$C_{K'_1} \leftarrow \text{AE}_D(\mathsf{H}_2(\tau), \psi)$		
else		
abort		
$\mathcal{K}'_{K1} \leftarrow \text{Decaps}(sk_1, C_{K'_1})$		
$sid := C_1 \vert C_2$		
$sid_\psi := sid \vert \psi$		
$\sigma \leftarrow \mathsf{H}_3(\tau, \mathcal{K}'_{K1}, sid_\psi)$		
$\text{SK}_\mathcal{C} \leftarrow \mathsf{H}_4(\tau)$	$\xrightarrow{\sigma}$	server_finish(σ):
		$sid := C_1 \vert C_2$
		$sid_\psi := sid \vert \psi$
		if $\mathsf{H}_3(\tau, \mathcal{K}_{K1}, sid_\psi) \neq \sigma$ **then**
		abort
		else
		$\text{SK}_\mathcal{S} \leftarrow \mathsf{H}_4(\tau)$

Figure 3.4: KEM-AE-PAKE protocol description [51]

whether the guessed password was correct or not. Encrypting C_{K1} into ψ using AE_E acts as a protective layer against such attacks where adversaries can exploit such properties of KEMs and quickly determine whether the guessed password was correct or not.

3. $\sigma \leftarrow$ client_finish(C_2, ψ): In this step, $\mathcal{C}$ decrypts C_2 using rw as the input key, retrieving the KEM ciphertext $C_{K'_2}$. If $\mathcal{C}$ correctly uses the same rw value that was generated during the registration phase, then $C_{K'_2} = C_{K2}$, which was generated by $\mathcal{S}$. Then, $\mathcal{C}$ decapsulates $C_{K'_2}$ using the secret key sk_2. If the correct sk_2 is used, then $\mathcal{K}'_{K2} = \mathcal{K}_{K2}$. Next, $\mathcal{C}$ generates the authentication tag τ using G and computes its hash digest using H_2. This hash digest is used as the input key to decrypt ψ using AE_D. Since this is the AE scheme, any modification to ψ would definitely result in a decryption failure, or using an incorrect input-key would also result in a decryption failure most of the time, yielding an error $\perp$ due to authentication tag mismatch (in the AE scheme). If the decryption of ψ succeeds, $\mathcal{C}$ obtains the KEM ciphertext $C_{K'_1}$. Then, $\mathcal{C}$ attempts to decapsulate $C_{K'_1}$ using the secret key sk_1. If the correct sk_1 is used, then $\mathcal{K}'_{K1} = \mathcal{K}_{K1}$, matching the one generated by $\mathcal{S}$. In practice, if the client does not have sk_1 stored locally, this mechanism implicitly ensures that only a legitimate client that knows $\mathcal{C}_{pw}$, $\mathcal{C}_{id}$, and $\mathcal{S}_{id}$ can recompute the same seed r from the registration phase and thus deterministically regenerate the key pair (pk_1, sk_1). Then, $\mathcal{C}$ constructs the session identifier sid by concatenating C_1 and C_2 and then generates sid_ψ by further concatenating ψ with sid. It then computes the hash digest σ using H_3, and derives the shared session key $\mathrm{SK}_{\mathcal{C}}$. Finally, $\mathcal{C}$ sends σ to $\mathcal{S}$. The successful receipt of σ by $\mathcal{S}$ serves as an indication that $\mathcal{C}$ has correctly processed ψ and recovered $\mathcal{K}'_{K1}$, which is ideally equal to $\mathcal{K}_{K1}$ generated on $\mathcal{S}$'s end.

4. server_finish(σ): In this final step, $\mathcal{S}$ reconstructs sid by concatenating C_1 and C_2 and obtains sid_ψ by concatenating sid and ψ. It then computes its own hash digest σ using H_3 and compares it against the σ value received from $\mathcal{C}$. If the two values of σ differ, it indicates that $\mathcal{K}'_{K1} \neq \mathcal{K}_{K1}$, which implies that $\mathcal{C}$ must have used an incorrect secret key sk_1 to decapsulate $C_{K'_1}$. Conversely, if the values match, this serves as explicit authentication of $\mathcal{C}$ to $\mathcal{S}$. Finally, $\mathcal{S}$ derives the shared session key $\mathrm{SK}_{\mathcal{S}}$.

Iff all the above steps are executed completely honestly, then only both parties are able to derive the session key on their end, which must be identical, i.e., $\mathrm{SK}_{\mathcal{S}} = \mathrm{SK}_{\mathcal{C}}$. We wish to explicitly mention that the tag τ from Figure 3.4 is referred to as K_0 in *Figure 9* in [51]. Additionally, in our opinion, every operation relevant to the first instance of the KEM does not prove to be too useful for the PAKE protocol design, as this makes the PAKE protocol computationally intensive. We believe that the authors primarily used it to (try to) strengthen the so-called verifier, i.e., (rw, pk_1), but we already mentioned in the additional clarification in subsection 3.3.2 as to why (rw, pk_1) is still not a secure enough verifier. Indeed, removing the usage of the first KEM instance will require one to redesign the PAKE protocol, but, in our opinion, having this does not contribute *enough* to enhance the security of the PAKE protocol itself.

Summary

In this chapter, we started by describing the different (types of) requirements (in section 3.1) that a PAKE protocol is expected to fulfill. Then, we discussed some of the recent propositions of quantum-resistant PAKE protocols (in section 3.2). We observed that not all the propositions provide concrete PAKE protocol instantiation, and instead, they provide a generic framework adhering to which a user can implement/instantiate a PAKE protocol. After discussing the related works, in order to fulfill our objectives for this task, we defined an evaluation methodology (in subsection 3.3.1). The motive behind defining this evaluation methodology was to ensure that we follow a systematic way that allows us to compare different PAKE protocols as fairly as possible. We wish to explicitly mention that this is a *qualitative* evaluation methodology and not a *quantitative* one. Following the methodology, we briefly discussed the fulfilment of security requirements in subsection 3.3.2, which is also concretely mentioned in Table 3.1. Then, in subsection 3.3.3, we assigned NIST security levels to the discussed PAKE protocols (as mentioned in Table 3.2) so that it facilitates easy comparison of two different protocols. Then, we provided our reasoning in subsection 3.3.4 for choosing some of the PAKE protocols (from section 3.2) for further implementation based on the security discussion and the assigned NIST security levels. Finally, in section 3.4 we provide a detailed description of the chosen PAKE protocols from above. This gives us a blueprint of the exact steps and the information flows that take place in the protocols.

We mentioned in section 1.2 that there is a noticeable lack of (open-source) implementations of the recently proposed PAKE protocols (including the three discussed in section 3.4). Having a working implementation of these PAKE protocols would allow independent verification of the protocols themselves. This is because a theoretical description, no matter how rigorous, might be blind-sided to certain practical considerations, like real-world performance, and/or identification of potential weaknesses, which can only surface when the protocol is actually executed in a live environment. So, going further (in chapter 4), we use the information from section 3.4 to our benefit for actually implementing these PAKE protocols specifically in Rust, because of its inherent safety guarantees while still providing high performance.

Chapter 4

The Rust Framework of Quantum-Resistant PAKE Protocols

In this chapter, we initially provide our reasons for choosing Rust as the programming language for implementing the chosen Password Authenticated Key-Exchange (PAKE) protocols (from section 3.4). Then, we gradually dive into the details of our framework, starting with defining some requirements and then providing a conceptual overview of the framework in terms of designing it.

4.1 Why Choose Rust?

Rust, just like C and C++, is a low-level, compiled programming language. But, Rust stands out when compared to C and C++, because of its following features [66]:

1. *Memory Safety Without Garbage Collection:* Rust enforces a strict ownership system at compile time with different rules for borrowing data items across the different components in the program. This ensures memory safety even without the need for a garbage collector. Based on our very limited understanding of languages such as C or C++, memory management is completely left to the developer of the program, which makes it difficult and error-prone, and for (high-level) languages like Python, garbage collection is provided as a feature, which makes the program (relatively) slow.
2. *Safe Concurrency:* Rust's ownership system, which uses a borrow checker, prohibits code compilation that could cause any memory error. Values owned by one variable can be borrowed and used at other places in the program within a strict set of rules, which leads to the prevention of data races in concurrent code.

In addition to these, Rust's design is inherently focused on preventing security vulnerabilities like buffer overflows or memory leaks, which makes it extremely suitable for building secure applications (like quantum-resistant PAKE protocols). Moreover, as opposed to using return codes and/or exceptions in C and C++ for error handling, which can lead to performance overhead, Rust uses *Result* and *Option* patterns, which can facilitate in creating more reliable and fault-tolerant code with performance overhead that is almost comparable with that of C and C++ [66].

The authors in [67] try to demonstrate the performance of Rust and C++ by building a data storage system that stores key-value pairs, and testing similar operations performed by the server, where the server-side code is written in both languages. They explicitly mention that they use *opt-level* 3 for the C++ implementation, whereas they do not explicitly mention using *dev-build* or *release-build* for their Rust implementation, and no explicit profile configuration is also mentioned in the *Cargo.toml* in [68]. We cannot assume anything about the optimization level of Rust that has been applied here, but based on the performance results, it is evident that, although *safe* Rust seems to be slower than *unsafe* Rust and C++ for single operations, there is a negligible difference in the performance of *safe* Rust for large-scale operations when compared to *unsafe*

Rust and C++. Without a doubt, this indicates that it is wise to accept the (extremely minimal) penalty imposed by (*safe*) Rust over C++, owing to its inherently strong safety guarantees over C++. However, C++ reigns in areas where every single bit of performance matters, but we are in complete accordance with [67] that both languages will continue to be important in the future, each being used in the areas where it makes the most sense. With that being said, we believe that in our case of implementing quantum-resistant PAKE protocols where security is paramount, Rust is definitely the choice to proceed with.

4.2 Conceptual Overview of the Rust Framework

In this section, we initially provide a public API design rationale to determine what we want the framework to have and provide to its users. We state the different requirements of our Rust framework, namely, *qr_pake_protocols*. Afterwards, we highlight some important problem statements that need to be addressed for the implementation, and then provide a high-level overview and design of components of the framework, along with a brief introduction to the low-level design itself.

4.2.1 Public API Design Rationale

We know that our Rust framework *qr_pake_protocols* has to offer certain features to its users, but this has to be provided in a structured manner, so that it allows them to use the framework while the low-level details remain abstracted underneath. From a user's perspective, the framework must provide a client-server application with a public API. We have the liberty of designing the public API and deciding what it could potentially offer. To be explicit, the public API could provide either of the following:

1. It could only provide the core PAKE protocol functionalities. By core functionalities, we mean steps like client_init(), server_resp(), client_finish(), server_finish(), etc. in the different PAKE protocols. In this case, the items exposed by the public API should encapsulate and sufficiently abstract those core functionalities, thus exempting a user from bothering themselves with the lowest-level details. This can be referred to as the *Core-Only* API.
2. If a communication layer using a particular network stack is a part of the framework, then the public API could only provide the methods of the communication layer that abstract the core functionalities (from above). This can be referred to as the *All-In-One* API.
3. Both of the above. This can be referred to as the *Hybrid* API.

We evaluate a public API design approach based on the following criteria (also mentioned in Table 4.1):

1. Flexibility: The scope of further modifications provided by the public API.
2. Explicit User-Effort Needed: The effort needed to make (those) modifications to the framework, when someone chooses to do it.

The three potential design approaches for the *qr_pake_protocols* framework, each with a different public API are as follows:

Table 4.1: Consolidated overview of public API design rationale

Public API Design Approach	Flexibility	Explicit User-Effort Needed
Core-Only API	High	High
All-In-One API	Low	Low
Hybrid API	High	⊙

Legend
⊙: Not Applicable

1. *Core-Only* API: In this design approach, the framework will *only* comprise of the core client and server functionalities described in section 3.4 of the chosen PAKE protocols. The public API will also only expose (items encapsulating) the same. The advantages and disadvantages of this approach are as follows:
   - Advantages:
     a) This provides the maximum flexibility to a user as to how they wish to make the server and the client communicate.
     b) Encourages deep understanding of the actual PAKE protocol itself.
   - Disadvantages:
     a) This approach explicitly requires the highest effort from a user's end because the user has to (correctly) implement the whole communication layer between the server and the client.
     b) It has a steeper learning curve for a user, because one cannot simply run the PAKE protocol by executing the server side and the client side on two different systems via this approach.
2. *All-In-One* API: In this design approach, the framework will comprise the core client and server functionalities described in section 3.4 as well as a predefined implementation of a communication layer between the server and the client. The public API in this case will only expose the communication logic. The advantages and disadvantages of this approach are as follows:
   - Advantage: This requires the least amount of effort from the user's end, as the framework is ready-to-use. Furthermore, a user does not have to bother themselves with the underlying core functionalities of the PAKE protocols.
   - Disadvantage: This approach makes the framework least flexible to extend or modify. This is because the users have extremely limited options in terms of the network stack that they can use to integrate this framework into their own software ecosystem.
3. *Hybrid* API: This design approach comprises the same components as the *All-In-One* API, but the public API in this case is less-restricted because it exposes both the core server and client functionalities and the communication logic. The advantages and disadvantages of this approach are as follows:
   - Advantages:
     a) This approach provides the best of both worlds, which means that it definitely has high flexibility. The effort needed for modification will be

high in case a user chooses to implement their own communication layer, but it is completely optional.

  b) A user could choose to quickly assemble a server and client binary each on a different system and execute the PAKE protocol using the existing communication logic.

- Disadvantages:
  a) The effort needed to modify the framework, especially in terms of implementing a (new) communication layer with a different network stack, will indeed be as high as in the case of the *Core-Only* API, but it is optional.

After carefully weighing the trade-off of the three design approaches based on Table 4.1 and the above description, we choose to adopt the *Hybrid* API approach. This indeed means that our framework will be equipped with a predefined communication layer as well. In practice, a user of our framework could vary a lot, from being an experienced cryptographer who would prefer analyzing the core server and client functionalities to a systems programmer or an end-user who would just prefer to use the PAKE protocol as it comes. By exposing both the core functionalities and the communication logic, the public API of this approach caters to both audiences.

4.2.2 Requirements of the Rust Framework

In this section, we briefly discuss about the availability and usage of the different cryptographic primitives at first and then dive into the details of the framework requirements.

Details of the Available Cryptographic Primitives: We know from Table 3.2 that the authors of the chosen PAKE protocols suggest implementing them using the following cryptographic primitives:

1. TK-PAKE [49] with Kyber-768.
2. *Modified* OCAKE-PAKE [50] with Frodo-KEM [22], the Kyber family [23], and more.
3. KEM-AE-PAKE [51] with Kyber-768.

We see that the *Modified* OCAKE-PAKE protocol can be instantiated with multiple Key-Encapsulation Mechanisms (KEMs) without requiring any further modification to the KEM itself. Owing to the generic construction of TK-PAKE, we believe that it should be possible to instantiate it with other variants of Kyber as well. Although we have categorized KEM-AE-PAKE as a concrete instantiation in subsection 3.2.2 because the authors provide implementation details, performance measurements, etc., we believe that it can also be instantiated using the other remaining variants of Kyber. As it has been established that all the chosen PAKE protocols can be instantiated using all the variants of Kyber, we would like to say that now there are three *protocol variants* for each of the chosen PAKE protocols, depending on the underlying choice of Kyber. Furthermore, in addition to instantiating *Modified* OCAKE-PAKE with all variants of Kyber, we would also like to instantiate it with all variants of Frodo-KEM as well, where the primitive for generating the public matrix $\mathbf{A}$ is the SHAKE-128 XOF [16] (as also mentioned in subsection 2.6.2), like the authors of *Modified* OCAKE-PAKE did.

The Framework Requirements

We wish to mention that, from here onwards, we refer to the PAKE protocol execution phase (from Figure 3.2, Figure 3.3, and Figure 3.4) as the *login* phase. In order to develop (or implement) a Rust framework like *qr_pake_protocols* that comprises the chosen PAKE protocols from section 3.4, which can also be easily used by a user, it must fulfill the following requirements:

1. Functional Requirements
    a) Based on our brief discussion from above regarding cryptographic primitives, the framework should indeed allow a user to execute any of the chosen PAKE protocol while easily switching between all three variants of Kyber. Additionally, it should allow a user to exclusively execute *Modified* OCAKE-PAKE with all the variants of Frodo-KEM while easily switching between them. To be explicit, the framework should indeed support seamless switching between the two choices of KEMs as well, but only for *Modified* OCAKE-PAKE, and the two other chosen protocols should only get instantiated with Kyber.
    b) The framework should provide access to a user to all the (low-level) core functionalities of the chosen PAKE protocols as defined in section 3.4, preferably in an encapsulated manner. This ensures a structured way of representation for the users while having proper separation of concerns.
    c) In addition to the core functionalities of the PAKE protocols, the framework should also include a communication layer between the server and the client while abstracting their core functionalities. This requirement further comes with its own following sub-requirements:
        i. The server should be able to handle multiple clients concurrently in a non-blocking manner. In simpler terms, if one client is waiting for some operation to be over, other clients can be served in the meantime.
        ii. For each of the chosen PAKE protocol, the server should allow a client to register *only once* for a chosen client identifier $\mathcal{C}_{id}$ and protocol variant. In case of *Modified* OCAKE-PAKE exclusively, a client can register *only once* for a chosen $\mathcal{C}_{id}$, KEM, and the protocol variant. But a registered client (for any PAKE protocol) can log in any number of times.
        iii. As every registered client can log in multiple times, the server has to ensure that it correctly stores a mapping of the server instance to each of the client's necessary data received during their registration phase. This is so that the server can extract which instance to use based on the client data received during login.
        iv. *Optionally*, the framework could also support the feature which allows a client to change its password $\mathcal{C}_{pw}$, keeping $\mathcal{C}_{id}$ and the protocol variant the same (and also the chosen KEM in case of *Modified* OCAKE-PAKE).
2. Security Requirements
    a) The server-side of the communication layer for each of the PAKE protocol must be able to prevent an adversary $\mathcal{A}$ (or a malicious client) from performing multiple consecutive incorrect login attempts within a (small) time frame. This is to ensure that the server does not fall prey to a Denial of Service (DoS)

attack, thus making it unavailable to legitimate clients. More importantly, this is also to prevent an adversary from performing an online dictionary attack. Additionally, the login phase should not succeed at all (even) if a client provides an incorrect password during login (by mistake).

   b) According to the optional functional requirement (item 1(c)iv) mentioned above, a client must only be allowed to change their password if he/she is already registered with the server of the respective PAKE protocol and is also logged in.

3. Non-functional Requirement: The communication layer in the framework should be lightweight and efficient, such that it does not introduce significant overhead or negatively impact the performance of the underlying PAKE protocols.

4.2.3 Additional Problems to be Considered

In this section, we primarily describe two problem statements along with their solutions that we have encountered with respect to the implementation of the PAKE protocols.

The Direct Encryption Problem

We know that the chosen PAKE protocols in section 3.4 use Ideal Cipher (IC) for encryption and decryption. As mentioned in section 2.2 that ICs are limited to their theoretical existence only, so we choose to use AES in counter mode (from section 2.3) as an approximation for IC for performing said encryption and decryption.

If we revisit Figure 3.2, Figure 3.3, and Figure 3.4, we can say that AES in counter mode is used as an approximation for IC for encrypting the KEM public key pk and KEM ciphertext C_K using the client's password $\mathcal{C}_{pw}$ or a value generated using $\mathcal{C}_{pw}$ as the input-key. We know that Kyber is based on Module Learning With Errors (MLWE), and from *Algorithm 19* and *20* in [23], we can infer that pk and C_K, respectively, comprise of a *PolynomialVector* made up of K *Polynomials*, each with $n = 256$ coefficients modulo $q = 3,329$. So, if we directly perform IC encryption (using AES in counter mode as its approximation) of pk or C_K (generated using Kyber), it would render the protocol vulnerable. This is because an adversary $\mathcal{A}$ can perform an offline dictionary attack where he tries to decrypt the ciphertext C generated after encrypting pk or C_K by guessing different client passwords (or password-related values) as the input-key. In case the input-key $K \neq \mathcal{C}_{pw}$ or a correct $\mathcal{C}_{pw}$-related value, the plaintext P, obtained after decryption, will have polynomial coefficients $\geq q$ with a high probability. This would quickly indicate $\mathcal{A}$ that the guessed password (related value) is incorrect. Going further, we describe a solution for this obtained from [56].

The Base Encoding and Decoding of the Polynomials: We observed the issue above, when decrypting a (directly encrypted) KEM public key pk or ciphertext C_K with a different input-key (when generated using Kyber). In *Section 5.2* of [56], the authors described a solution to prevent this by encoding the polynomials of pk or C_K into $\{0, \ldots, q^{nK} - 1\}$. This range basically represents an enumeration of all possible KEM public keys or ciphertexts. Based on our understanding, this encoding process implies a mapping of all the polynomials in the given pk or C_K to a unique integer within the previously mentioned range. Then, this unique integer in that range effectively acts as an index for the specific pk or C_K (being encoded) within the set of all possible

KEM public keys or ciphertexts. And, the authors ask us to encrypt this index (in that range) instead of directly encrypting the *pk* or C_K, in which case, it would result in the direct encryption of the underlying polynomials, which would cause the aforementioned issue upon decryption with a different input-key. In our opinion, one way of practically implementing this enumeration is to perform base-encoding of *pk* or C_K. During base-encoding, the following steps happen chronologically:

1. We must deconstruct *pk* and C_K and extract the raw polynomials from them. For deconstructing *pk*, $\mathit{ByteDecode}_{\mathbf{d}}$ is needed (as per *Algorithm 6*) and for C_K, $\mathit{Decompress}_{\mathbf{x}}$ is also additionally required (as per *Equation 4.8*) in [23].
2. Then, we perform a (forward) numerical base-conversion of the raw polynomials, which results in the base-encoded value, namely, $pk_{\mathbf{be}}$ and $C_{K\mathbf{be}}$.

Then, we encrypt this base-encoded value, instead of directly encrypting the original *pk* or C_K. After decryption, as we will obtain $pk_{\mathbf{be}}$ or $C_{K\mathbf{be}}$, we have to perform base-decoding, which comprises of the following steps:

1. We perform a (backward) numerical base-conversion of $pk_{\mathbf{be}}$ and $C_{K\mathbf{be}}$ to obtain the raw polynomials.
2. Then, we must reconstruct *pk* and C_K, for which $\mathit{ByteEncode}_{\mathbf{d}}$ is needed, and for C_K, $\mathit{Compress}_{\mathbf{x}}$ is also additionally required as per *Algorithm 5* and *Equation 4.7* respectively in [23].

Here, $\mathbf{d}$ and $\mathbf{x}$ represent the number of bits per polynomial coefficient during encoding/decoding and compression/decompression, respectively. To be explicit, the value $\mathbf{d}$ is distinct from the dimensions d in subsection 2.6.4 for Kyber. We know (from Table 2.6) that $q = 3,329$ and $n = 256$ in Kyber. This means that all polynomial coefficients (in *pk* or C_K) will be $\in \{0, \dots, q-1\}$, which also means that they will (already) be in base 3,329. Now, to realize the enumeration via base-encoding, one can consider each polynomial in *pk* or C_K as a single large number where each coefficient serves as a single digit (in base 3,329). This means that the whole polynomial itself is one large number in base 3,329. So, we perform a forward base-conversion of all the polynomials in *pk* or C_K from base 3,329 to base 10, considering each polynomial as one large number in base 3,329. The resulting value after base-conversion, i.e., the base-encoded value, is the integer in the range $\{0, \dots q^{nK} - 1\}$ which is the index representing the original *pk* or C_K amongst all possible KEM public keys or ciphertexts. As we need to encrypt this base-encoded value (using AES in counter mode as an approximation for IC), it is wise to serialize this value so that it can be seamlessly used further. The internal forward base-conversion during base-encoding and backward base-conversion during base-decoding are done as follows:

1. *Forward Conversion*: Here we consider each *Polynomial* with its 256 coefficients as a single large number in base 3,329, where each individual coefficient represents a digit in base 3,329 of this large number. For base-conversion, we use the (naively depicted) *convert_base*() function from Algorithm 1 while considering the *src* = 3,329, *targ* = 256 and *digits* = *Polynomial.* This means that the base-encoded polynomial will be in base 256. We have considered the target base to be 256 (instead of base 10), so that the base-encoded polynomial is already in a serialized format, i.e., exactly one byte per base-256 digit in the base-encoded polynomial, which can be seamlessly used further.

2. *Backward Conversion*: This basically reverses the forward base-conversion process. We use the same *convert_base*() function from Algorithm 1 while considering the *src* = 256, *targ* = 3,329 and *digits* = *base-encoded Polynomial.*

Algorithm 1 The base converter

```
function convert_base(src, targ, digits)        ▷ digits: Array of unsigned integers
    digits_copy = trim_leading_zeros(digits)     ▷ Remove leading zeros
    if digits_copy = [ ] then
        return [0]
    end if
    output_digits = [ ]                          ▷ Collect remainders (LSB-first)
    while ! (len(digits_copy) = 1 and digits_copy[0] = 0) do
        quotients = [ ]
        remainder = 0
        for all dig in digits_copy do            ▷ Simulate long-division
            acc = remainder * src + dig
            temp_quots = $\lfloor \text{acc}/targ \rfloor$
            remainder = acc mod targ
            if temp_quots $\neq$ 0 then
                quotients.append(temp_quots)
            end if
        end for
        output_digits.append(remainder)
        digits_copy = quotients
    end while
    reverse(output_digits)                       ▷ Make MSB-first
    if output_digits = [ ] then
        output_digits = [0]
    end if
    return output_digits
end function
function trim_leading_zeros(digits)
    i = 0
    while i < len(digits) and digits[i] = 0 do
        i += 1
    end while
    return digits[i:] or [0]
end function
```

The complete details of how this base-encoding and decoding are applied to *pk* and C_K are mentioned in Algorithm 2 and Algorithm 3, respectively. We also wish to mention that, with respect to Figure 3.2, Figure 3.3, Figure 3.4, and the Algorithm 2 and Algorithm 3, from now onwards, we always the perform base-encoding of *pk* or C_K at first (when generated using Kyber) and then encrypt the base-encoded $pk_{\mathbf{be}}$ or $C_{K\mathbf{be}}$ using AES in counter mode as an approximation for IC. Consequently, after decryption, we perform the base-decoding to restore the original *pk* or C_K.

Expected Outcome from Base Encoding and Decoding: Upon decryption with the *correct* password (related value) as the input-key, it yields the base-encoded $pk_{\mathbf{be}}$ or

$C_{K\mathbf{be}}$, which can be base-decoded to recover the original *pk* or C_K. However, if an adversary $\mathcal{A}$ uses an *incorrect* password (related value) as the input-key, the decrypted output will still resemble a valid-looking pk' or C'_K instead of a value having coefficients which are $\geq q$. This prevents $\mathcal{A}$ from knowing whether the guessed password (related value) that was used as the input-key was correct or not.

Algorithm 2 Base encoder decoder for KEM public key *pk*

```
 1: function KeyGen($d$, $q$) [23]                                  ▷ $d$: Dimension, $q$: Prime modulus from Table 2.6
 2:     $sd_1 \xleftarrow{\$} \{0,1\}^{256}, sd_2 \xleftarrow{\$} \{0,1\}^{256}$       ▷ Two seeds, each of 32 bytes (256 bits)
 3:     $(\rho, \sigma) \leftarrow$ SHA3-512($sd_1 \| d$) $\in \{0,1\}^{512}$            ▷ Two seeds, each of 32 bytes
 4:     $\mathbf{A} \xleftarrow{\$} R_q^{d \times d} := Sam(\rho)$                      ▷ $Sam$: Random sampling algorithm
 5:     $(\mathbf{s}_1, \mathbf{e}_1) \xleftarrow{\$} \mathcal{X}^d \times \mathcal{X}^d := Sam(\sigma)$
 6:     $\mathbf{b} \leftarrow \mathbf{A} \cdot \mathbf{s}_1 + \mathbf{e}_1 \mod q$
 7:     $\mathbf{t} \leftarrow ByteEncode_{\mathbf{d}}(\mathbf{b}), \mathbf{s} \leftarrow ByteEncode_{\mathbf{d}}(\mathbf{s}_1)$   ▷ Here $\mathbf{d}$ = 12 [23]
 8:     $pk \leftarrow (\mathbf{t} \| \rho)$
 9:     $h \leftarrow$ SHA3-256($pk$)
10:     $sk \leftarrow (\mathbf{s} \| pk \| h \| sd_2)$                               ▷ $sd_2$: Acts as implicit rejection value
11:     return $(pk, sk)$
12: end function
13: function $get\_base\_encoded\_pk(pk)$                           ▷ Performs base-encoding
14:     $(\mathbf{t}, \rho) \leftarrow pk$                                                ▷ Parse
15:     $\mathbf{b} \leftarrow ByteDecode_{\mathbf{d}}(\mathbf{t})$                            ▷ Here $\mathbf{d}$ = 12 [23]
16:     polyvec$_{\mathbf{be}}$ = [ ]                                         ▷ Create an empty buffer
17:     for $\boldsymbol{poly}$ in $\mathbf{b}$ do
18:         poly$_{\mathbf{be}}$ = $convert\_base$($q$, 256, $\boldsymbol{poly}$)          ▷ Forward base-conversion
19:         polyvec$_{\mathbf{be}}$.extend(poly$_{\mathbf{be}}$)
20:     end for
21:     $pk_{\mathbf{be}}$ = polyvec$_{\mathbf{be}} \| \rho$                                  ▷ Base encoded $pk$
22:     return $pk_{\mathbf{be}}$
23: end function
24: function $get\_base\_decoded\_pk(pk_{\mathbf{be}})$                     ▷ Performs base-decoding
25:     (polyvec$_{\mathbf{be}}$, $\rho$) $\leftarrow pk_{\mathbf{be}}$                                  ▷ Parse
26:     polyvec$_{\mathbf{bd}}$ = [ ]                          ▷ Create empty buffer of $d$ polynomials
27:     idx = 0
28:     len = $\lceil n \times \log_n(q) \rceil$                  ▷ $n$: Polynomial degree from Table 2.6
29:     for $i$ in 0..$d$ do
30:         poly$_{\mathbf{be}}$ = polyvec$_{\mathbf{be}}$[idx..idx + len]                 ▷ Extract the poly$_{\mathbf{be}}$
31:         poly$_{\mathbf{bd}}$ = $convert\_base$(256, $q$, poly$_{\mathbf{be}}$)        ▷ Backward base-conversion
32:         polyvec$_{\mathbf{bd}}$[$i$] = poly$_{\mathbf{bd}}$
33:         idx += len                                          ▷ To extract the next poly$_{\mathbf{be}}$
34:     end for
35:     $\mathbf{b}$ = polyvec$_{\mathbf{bd}}$
36:     $\mathbf{t} \leftarrow ByteEncode_{\mathbf{d}}(\mathbf{b})$                            ▷ Here $\mathbf{d}$ = 12 [23]
37:     $pk \leftarrow (\mathbf{t} \| \rho)$                         ▷ The $pk$ recovered after base decoding
38: end function
```

Algorithm 3 Base encoder decoder for KEM ciphertext C_K

```
function Encaps($pk$) [23]
    $m \overset{\$}{\leftarrow} \{0,1\}^{256}$    ▷ A random value of 32 bytes in a compressed and encoded form
    $(\mathcal{K}_K, sd_3) \leftarrow$ SHA3-512($m \| $SHA3-256($pk$)) $\in \{0,1\}^{512}$    ▷ Both of 32 bytes
    $(\mathbf{t}, \rho) \leftarrow pk$    ▷ Parse
    $\mathbf{b} \leftarrow ByteDecode_{\mathbf{d}}(\mathbf{t})$
    $\mathbf{A} \leftarrow R_q^{d \times d} := Sam(\rho)$    ▷ Same as the $\mathbf{A}$ in Algorithm 2
    $(\mathbf{s}_2, \mathbf{e}_2) \leftarrow \mathcal{X}^d \times \mathcal{X}^d := Sam(sd_3)$
    $\boldsymbol{e}' \leftarrow \mathcal{X} := Sam(sd_3)$
    $\mathbf{u} \leftarrow \mathbf{A}^T \cdot \mathbf{s}_2 + \mathbf{e}_2 \mod q$
    $\boldsymbol{m}' \leftarrow Decompress_1(ByteDecode_1(m))$
    $\boldsymbol{v} \leftarrow \mathbf{b}^T \cdot \mathbf{s}_2 + \boldsymbol{e}' + \boldsymbol{m}'$    ▷ Encrypting $\boldsymbol{m}'$
    $c_1 \leftarrow ByteEncode_{d_u}(Compress_{d_u}(\mathbf{u}))$    ▷ $d_u$ as per *Table 2* in [23]
    $c_2 \leftarrow ByteEncode_{d_v}(Compress_{d_v}(\boldsymbol{v}))$    ▷ $d_v$ as per *Table 2* in [23]
    $C_K \leftarrow (c_1 \| c_2)$
    return $(C_K, \mathcal{K}_K)$
end function
function $get\_base\_encoded\_kem\_ciphertext(C_K)$    ▷ Performs base-encoding
    $(c_1, c_2) \leftarrow C_K$    ▷ Parse
    $\mathbf{u} \leftarrow Decompress_{d_u}(ByteDecode_{d_u}(c_1))$    ▷ Same as above $d_u$
    $\boldsymbol{v} \leftarrow Decompress_{d_v}(ByteDecode_{d_v}(c_2))$    ▷ Same as above $d_v$
    $C_{K\mathbf{be}} = [\,]$    ▷ Create an empty buffer
    for $\boldsymbol{poly}$ in $\mathbf{u}$ do
        poly_u$_{\mathbf{be}}$ = $convert\_base(q, 256, \boldsymbol{poly})$    ▷ Forward base-conversion
        $C_{K\mathbf{be}}$.extend(poly_u$_{\mathbf{be}}$)
    end for
    poly_v$_{\mathbf{be}}$ = $convert\_base(q, 256, \boldsymbol{v})$    ▷ Forward base-conversion
    $C_{K\mathbf{be}}$.extend(poly_v$_{\mathbf{be}}$)    ▷ Base encoded $C_K$
    return $C_{K\mathbf{be}}$
end function
function $get\_base\_decoded\_kem\_ciphertext(C_{K\mathbf{be}})$    ▷ Performs base-decoding
    polyvec_u$_{\mathbf{bd}}$ = [ ]    ▷ Create empty buffer of $d$ polynomials
    idx = 0
    len = $\lceil n \times \log_n(q) \rceil$    ▷ $n$: Polynomial degree from Table 2.6
    for $i$ in $0..d$ do    ▷ $d$: Dimension from Table 2.6
        poly_u$_{\mathbf{be}}$ = $C_{K\mathbf{be}}$[idx..idx + len]    ▷ Extract the (next) poly_u$_{\mathbf{be}}$
        poly_u$_{\mathbf{bd}}$ = $convert\_base$(256, $q$, poly_u$_{\mathbf{be}}$)    ▷ Backward base-conversion
        polyvec_u$_{\mathbf{bd}}$[$i$] = poly_u$_{\mathbf{bd}}$
        idx += len    ▷ To extract the next poly_u$_{\mathbf{be}}$
    end for
    poly_v$_{\mathbf{be}}$ = $C_{K\mathbf{be}}$[idx..idx + len]    ▷ Extract the poly_v$_{\mathbf{be}}$
    poly_v$_{\mathbf{bd}}$ = $convert\_base$(256, $q$, poly_v$_{\mathbf{be}}$)    ▷ Backward base-conversion
    $\mathbf{u}$ = polyvec_u$_{\mathbf{bd}}$, $\boldsymbol{v}$ = poly_v$_{\mathbf{bd}}$
    $c_1 \leftarrow ByteEncode_{d_u}(Compress_{d_u}(\mathbf{u}))$    ▷ Same as above $d_u$
    $c_2 \leftarrow ByteEncode_{d_v}(Compress_{d_v}(\boldsymbol{v}))$    ▷ Same as above $d_v$
    $C_K \leftarrow (c_1 \| c_2)$    ▷ The $C_K$ recovered after base decoding
end function
```

Additional Clarification: It may seem that the base-encoding adds a communication overhead as opposed to sending the non-base-encoded/original value over a network channel. Based on our understanding, we provide a reasoning behind this in Appendix A. Additionally, to be explicit, this base-encoding and decoding is not required in the case of Frodo-KEM [22] because, it is based on Learning With Errors (LWE), which is algebraically unstructured (unlike Kyber with MLWE), and they use $q \in \{2^{15}, 2^{16}\}$, which means that the valid integers in the vectors or the matrices will be random bit strings of length 15 or 16 bits.

The Lack of Explicit Authentication in TK-PAKE

We know that TK-PAKE is the only protocol amongst the chosen ones that does not provide explicit authentication for either party (as evident from subsection 3.4.1 and Figure 3.2), let alone explicit mutual authentication. As PAKE protocols are likely to be used as a part of a larger (software) ecosystem in the real world, ensuring that indeed the same session key has been established on both parties' ends is imperative. So, we need to add an explicit authentication check after the PAKE protocol execution phase is done from Figure 3.2. This is to ensure that the server is able to decide whether this was a correct login attempt or not, and can handle it accordingly (for the particular client) in case of an incorrect login attempt (using an incorrect password). This prevents a malicious client from overwhelming the server while continuously performing incorrect logins, while the server remains unaware of this fact and also unavailable to other legitimate clients. We provide a solution for this, going further.

```
Client ($\mathcal{C}_{id}, \mathcal{C}_{pw}$)                                   Server ($\mathcal{S}_{id}$)
........................ PAKE Protocol Execution ........................
⋮
                                                           server_resp($C_1$):
                                                           ⋮
client_ter_init($C_2$):                                    $\mathrm{SK}_{\mathcal{S}} \leftarrow \mathsf{H}(\mathcal{S}_{\mathrm{ctx}_1}, \mathcal{S}_{\mathrm{ctx}_2})$
⋮
$\mathrm{SK}_{\mathcal{C}} \leftarrow \mathsf{H}(\mathcal{C}_{\mathrm{ctx}_1}, \mathcal{C}_{\mathrm{ctx}_2})$
.................... Explicit Client Authentication Check ....................
gen_auth_details():
$\mathrm{D}_{\mathrm{SK}_{\mathcal{C}}} \leftarrow \mathsf{H}(\mathrm{SK}_{\mathcal{C}})$      $\xrightarrow{\mathrm{D}_{\mathrm{SK}_{\mathcal{C}}}}$      authenticate_client($\mathrm{D}_{\mathrm{SK}_{\mathcal{C}}}$):
                                                           $\mathrm{D}_{\mathrm{SK}_{\mathcal{S}}} \leftarrow \mathsf{H}(\mathrm{SK}_{\mathcal{S}})$
                                                           if $\mathrm{D}_{\mathrm{SK}_{\mathcal{S}}} \neq \mathrm{D}_{\mathrm{SK}_{\mathcal{C}}}$ then
                                                               handle_incorrect_login()
                                                               abort with $\perp$
```

Figure 4.1: Explicit client authentication check in TK-PAKE

The Client Authentication Check: As the TK-PAKE lacks built-in explicit authentication, we introduce a custom client-to-server authentication step, as shown in Figure 4.1, extending the protocol in Figure 3.2 after session key establishment. This design reflects typical PAKE threat models, where the client is the primary adversary and the server is assumed to be honest-but-curious. All of our chosen PAKE protocols are *b*PAKE protocols and the server already stores the client's password (or a weak verifier in case of KEM-AE-PAKE), requiring implicit trust in the server. Therefore, we do not have a server-to-client authentication in TK-PAKE as we do not consider the server to be someone who is actively malicious.

4.2.4 Framework Components

This section is intended to provide a technology-agnostic picture of how the framework is structured and how its parts fit together without diving into the exact Rust-specific details. The framework adopts a layered modular architecture centred around the following two components:

1. The core functionalities layer
2. The communication layer

The Core Functionalities Layer

This layer primarily comprises the lowest-level functionalities of the client and server for each of the chosen PAKE protocols as described in subsection 3.4.1, subsection 3.4.2, and subsection 3.4.3, along with its other accompanying cryptographic modules. It forms the foundation of the framework, and is completely unaware of and independent from any networking-related things or persistence of data. We could also say that it is completely stateless between different invocations of the registration and login phases of a PAKE protocol. This layer is only focused on consuming inputs like passwords, cryptographic keys, seeds, etc., and producing outputs like ciphertexts, session keys, as it happens in the different steps of a PAKE protocol. We also consider this layer to be responsible for handling the variant selection of (Frodo-KEM [22] and) Kyber [23], which will be used in the current invocation of the PAKE protocol execution based on some configuration parameters.

One could also observe from Figure 3.2, Figure 3.3, and Figure 3.4 that there are some hash functions and Extendable Output Function (XOF) instances used in the different PAKE protocols. For each of the PAKE protocols in section 3.4, we stated that the authors have not explicitly mentioned their choice regarding the hash functions or the XOF, so we mention them below as per our preference, which is used in our implementation:

1. TK-PAKE (from subsection 3.4.1): We have selected H to be SHA-256 [26] with a digest size of 32 bytes. So, the client and the server's session key generated using TK-PAKE are of 32 bytes. The same is applicable for the hash function in Figure 4.1.
2. *Modified* OCAKE-PAKE (from subsection 3.4.2):
    a) We have selected H to be SHA-256 with a digest size of 32 bytes.
    b) We have selected G (or KDF) to be SHAKE-128 [29] as the particular choice of XOF with a digest size of 32 bytes. So, the client and the server's session key generated using *Modified* OCAKE-PAKE are of 32 bytes.

3. KEM-AE-PAKE (from subsection 3.4.3):
   a) We have selected H_1 to be SHA-512 [29] with a digest size of 64 bytes. The reason being, in this protocol, the seeded KeyGen operation of Kyber is used, which can be referred from the client registration phase in subsection 3.4.3.
   b) We have selected all the other hash functions to be SHA-256 with a digest size of 32 bytes. So, the client and the server's session key generated using KEM-AE-PAKE are of 32 bytes.

Additionally, for KEM-AE-PAKE, considering Figure 3.4 and the hash function H_2, which is SHA-256 with an output size of 32 bytes (as mentioned above), for the Authenticated Encryption (AE) scheme in KEM-AE-PAKE, we have selected AES-256 GCM [42]. This is because, in Figure 3.4, the input-key to the AE_E & AE_D function is an output of H_2, which is 32 bytes. We do not go into further details here because the core functionalities of the chosen PAKE protocols are already described in subsection 3.4.1, subsection 3.4.2, and subsection 3.4.3 for TK-PAKE [49], *Modified* OCAKE-PAKE [50], and KEM-AE-PAKE [51] respectively and we are also aware of the different variants of Frodo-KEM and Kyber from subsection 2.6.2 and subsection 2.6.4 respectively. We could also consider this core functionalities layer to be the Trusted Computing Base (TCB) of our Rust framework. Finally, the public API of this layer should expose the following:

1. The different core functionalities of the client and the server, ideally in an encapsulated manner for the different PAKE protocols, so that there is a proper separation of concerns.
2. The configuration parameters that allow a user to decide which Kyber (or Frodo-KEM) variant to use for the current invocation of the PAKE protocol execution.

The Communication Layer

This is a rather higher layer than the core functionalities layer. It is supposed to abstract the core functionalities of the client and the server while providing users a ready-to-use interface that can demonstrate a networked communication between a server and a client where there is a PAKE protocol executing underneath. To achieve this, we must decide multiple things from the design perspective of the framework, as discussed further.

Choosing the I/O Mechanism: In order to fulfill the requirement where the server handles multiple clients concurrently, we need to decide between synchronous and asynchronous I/O. Based on the details mentioned in Table 4.2, in a nutshell, synchronous I/O will not work well in scenarios where multiple clients need to be handled because, while a client waits for some I/O operation to finish, it will block the thread, and nothing useful can happen during the wait. This wastes CPU cycles and memory, whereas with asynchronous I/O, we can start operations and continue working further without blocking any execution, which will allow us to serve multiple clients efficiently and without wasting any system resources. But there is one drawback with asynchronous I/O, i.e., it requires additional constructs such as async/await syntax, schedulers, event loops, etc., which generally introduces boilerplate code that does not exist in synchronous I/O. Although better resource utilization, performance, and scalability are gained with asynchronous I/O for I/O bound systems, it typically comes at the expense of codebase size and architectural overhead. But for our purpose, we do proceed with asynchronous I/O while acknowledging this trade-off of having a relatively large codebase.

Table 4.2: Comparative overview between synchronous and asynchronous I/O [69]

Feature	Synchronous I/O	Asynchronous I/O
Behaviour	Blocking	Non-Blocking
Execution Flow	Strictly Sequential	Concurrent
Resource Utilization	Poor	Efficient
Scalability	Low	High
Development Complexity	Low	Moderate
Codebase Size	(Relatively) Small	Large

Choosing the Networking Protocol: As the client and the server are going to communicate over a network channel, it is imperative to decide upon a networking protocol that the communication layer should use. Upon observing Figure 3.2, Figure 3.3, and Figure 3.4, we see that all the PAKE protocols have a client registration phase preceding the PAKE protocol execution phase. It is known that PAKE protocols allow two parties to establish a shared session key while communicating over a *potentially unsecure* channel, but the registration phase must be done securely, be it an out-of-band registration or using Transport Layer Security (TLS). This is necessary because, during this phase, the client sends its password (or a verifier) to the server. If this registration is done over an unsecure communication channel, then it defeats the purpose of even having a PAKE protocol in the first place. Although not completely desirable, the PAKE protocol execution phase can be done over an unsecure communication channel. The two predominantly used networking protocols are TCP and UDP, and in Table 4.3, we provide a comparative overview between the two based on certain features.

Table 4.3: Comparative overview between TCP and UDP [70]

Feature	TCP	UDP
Reliable	Yes	No
Latency	High	Low
Overhead	High	Low

Based on the data from Table 4.3, it boils down to a simple trade-off between performance and reliability of the two protocols. In our use case of implementing PAKE protocols, we incline towards TCP due to its connection reliability over UDP. Regarding the performance impact of TCP due to higher overhead and latency as compared to UDP, we believe it is definitely within a tolerable limit. This is because TCP (underneath TLS) is the go-to networking protocol for security and reliability-critical tasks like file transfers, online banking, remote access to other systems, etc. [70].

The Storage Handling: The server-side of the communication layer for the PAKE protocols must be (ideally) equipped with a (persistent) storage solution. It is evident from Figure 3.2, Figure 3.3, and Figure 3.4 that, during the client registration phase, the client sends the identifier $\mathcal{C}_{id}$ and the password $\mathcal{C}_{pw}$ (except in KEM-AE-PAKE in Figure 3.4 where it sends (rw, pk_1) instead of the password) as the registration details. As our framework supports multiple variants of Kyber (and Frodo-KEM), during the client registration phase, the client also has to indicate the protocol variant, and in case of *Modified* OCAKE-PAKE, there will be an additional value indicating the choice of KEM. So, the storage solution (according to requirement item 1(c)iii from subsection 4.2.2) must be designed to have a mapping as key-value pairs as follows:

1. Key: A composite key comprising of the $\mathcal{C}_{id}$ and the chosen protocol variant and optionally the KEM choice exclusively for *Modified* OCAKE-PAKE.
2. Value: A server state that holds the server's instance for this current client.

The Incorrect Login Handling: Now, during the login phase, the client has to send the $\mathcal{C}_{id}$, and the protocol variant with which the client had registered (and the KEM choice in case of *Modified* OCAKE-PAKE). Then, the server has to check whether the client with the currently provided details is already registered or not. If the client is registered, then the server proceeds further with the login phase, otherwise, it should inform the client that the login cannot proceed further as the client is not registered. For each PAKE protocol, during the login phase, the client executes its defined steps on its end with its password, and so does the server, but with the stored password (or verifier) that was sent by the client during the registration phase while interactively communicating with one another. There might be two ways this could proceed, depending on whether an explicit mutual authentication check is built into the PAKE protocol or not:

1. *When included*: In this case, if an adversary $\mathcal{A}$ who poses as a client and uses an incorrect password (or verifier), then, when the client authentication fails on the server's end, the server can record this as an incorrect login attempt for this particular registered client. In case of *Modified* OCAKE-PAKE and KEM-AE-PAKE, considering that the server is honest and the client is not, owing to the design of the PAKE protocols, the server's authentication in the client_finish() step on both the protocol's end will fail because the malicious client used an incorrect password. In this case, the client should send an error message to the server, and the server will handle this as an incorrect login attempt. But, at times, the malicious client might not be kind enough to even send an error message, then the server should handle this using a timeout mechanism, i.e., if neither the client's response nor the error message is received within, say, 5 seconds (as the timeout), then the server can record this as an incorrect login attempt.
2. *When not inbuilt*: In this case, we introduce our own explicit authentication check once the steps in the login phase are done and the session key is established on both parties' ends. This would facilitate ensuring whether both parties honestly used the same correct password (or verifier) or not. If this authentication check passes, then it means that the login phase was successful and both parties have arrived at the same session key. But, if it is an adversary $\mathcal{A}$ who poses as a client and uses an incorrect password, and the client's authentication fails on the server's end, then the server can record this as an incorrect login attempt for this particular registered client. This is specifically applicable for TK-PAKE for which we have a solution as per Figure 4.1. This client authentication check should reside in the communication layer to preserve TK-PAKE as a one-round (or two-flow) protocol, as shown in Table 3.3. The previously described timeout mechanism also applies here, in case a malicious client refrains from authenticating, the server should treat it as an incorrect login attempt.

Then, the server makes the client aware by sending a proper error message that the login failed due to an incorrect password, and the client only has a certain number of remaining attempts to correctly log in. The server maintains an incorrect login counter with a particular threshold (say 3) within a particular time window (say 60 seconds). If the same registered client performs multiple login attempts with an incorrect password, the server keeps increasing the counter value for this particular client. If the counter value exceeds the threshold within the given time window (about which the client is unaware), the client with the provided details is blocked forever.

It may also happen that, within the time window, a client performs some incorrect login attempts, but before crossing the threshold, the client provides the correct password. In this case, the server should also allow the client to redeem itself and reset the incorrect login counter value to 0 for this particular registered client. These measures must be taken to cater to the security requirement (item 2a) from subsection 4.2.2.

4.2.5 Brief Overview of Low-Level Design Details

The *qr_pake_protocols* framework is a modular Rust library for developing, executing, and benchmarking quantum-resistant PAKE protocols. The architecture of this framework allows research and experimentation with quantum-resistant PAKE protocols while emphasizing modularity, extensibility, and empirical evaluation. Currently, it comprises of pure Rust implementation of *b*PAKE protocols, namely, TK-PAKE [49], *Modified* OCAKE-PAKE [50] and KEM-AE-PAKE [51] as described in subsection 3.4.1, subsection 3.4.2, and subsection 3.4.3 respectively. We wish to mention that the *qr_pake_protocols* framework can also be referred to as a package. We list the contents of this package below and describe each of them later in more detail:

1. The main library crate, which also holds the same name, i.e., *qr_pake_protocols* is described in section 4.3.
2. A separate workspace crate named *qr_pake_protocol_executors* of whose details are mentioned in section 4.4.
3. The *examples* in section 4.5 and the *benchmarking process* in section 5.3 .

4.3 The qr_pake_protocols Crate

In this section, we provide details of the different modules in the *qr_pake_protocols* crate, which form the core functionalities layer of our framework.

4.3.1 The protocol_variants Module

This module comprises two enumerators, namely, the *AvailableVariants* and the *KemChoice*, along with their associated methods. These elements collectively provide a structured way to select and instantiate the PAKE protocols with different variants of Kyber [23] (and Frodo-KEM [22]). The fields of *KemChoice* are *Kyber* and *Frodo* to indicate the choice of KEM. The *AvailableVariants* can be considered as the configuration parameter that allows us to choose a particular variant of the KEM itself as defined below:

1. *LightWeight*: This corresponds to Kyber-512 or Frodo-640 providing NIST security *Level-I* equivalent to AES-128.

2. *Recommended*: This corresponds to Kyber-768 or Frodo-976 providing NIST security *Level-III* equivalent to AES-192.

3. *Paranoid*: This corresponds to Kyber-1024 or Frodo-1344 providing NIST security *Level-V* equivalent to AES-256.

4.3.2 The mlkem_variant_wrapper Module

The task of easily switching between different variants of Kyber is facilitated by the *AvailableVariants* enum, but it was also necessary to have simultaneous access to all the Kyber variants corresponding to the ones mentioned in the enum. Achieving this was not trivial because of the following reasons:

1. There is not an abundance of open-source pure Rust crates of Kyber. E.g., *pqcrypto-mlkem* [71] is not a pure Rust crate as it is a wrapper around the implementation of *PQClean* [72] which is written in C.

2. Even if there are some open-source pure Rust crates available, it is rare that they allow a user to simultaneously use all three variants of Kyber. E.g., *pqc_kyber* [73] only allows access to one variant at a time. We have a concept of *feature flags* in Rust that allow us to enable or disable parts of the codebase during compile time. The same happens with this crate as well. A developer/user must choose beforehand which (one) Kyber variant to use by specifying it as a feature, and only that selected variant is compiled, and others are excluded. This keeps the binary size small, but also restricts a user from switching between different variants at runtime. Therefore, to use a different variant of Kyber, one must recompile by specifying the other variant as a feature, which may not always be desirable.

Considering the above limitations, we chose to specifically proceed with *ml-kem* [74], as this is the only pure Rust crate that we found (while writing this report) which allows simultaneous access to all three variants of Kyber. This wrapper module comprises the *MlKemDispatcher* struct, which encapsulates the selected variant from *AvailableVariants*. This determines which parameter set of Kyber to use and the respective security level provided (according to Table 2.6). It provides a unified and dynamic interface and serves as the central dispatcher for key generation, encapsulation, and decapsulation operations for different variants of Kyber. Its core components include:

1. The key generation methods, namely, KeyGen and a seeded KeyGen operation. We have already defined the generic KeyGen operation at the beginning of section 2.6. We have additionally used the *deterministic* feature of the *ml-kem* crate [74] to cater to the seeded KeyGen operation of Kyber (as per subsection 2.6.4), as it is needed for the client registration phase of KEM-AE-PAKE [51] as described in subsection 3.4.3.

2. The encapsulation (Encaps) and decapsulation (Decaps) methods as defined in item 2 and item 3 respectively at the beginning of section 2.6.

We have another module called *frodo_variant_wrapper* like this *mlkem_variant_wrapper*, and it also does the same but for Frodo-KEM. In this case, we also have a similar struct called *FrodoDispatcher* which encapsulates the selected variant from *AvailableVariants*, and we use the *frodo-kem-rs* crate [75] for this purpose.

4.3.3 The encode_decode Module

This module provides the methods for performing the $\mathit{ByteEncode}_{\mathbf{d}}, \mathit{ByteDecode}_{\mathbf{d}}$ and $\mathit{Compress}_{\mathbf{x}}, \mathit{Decompress}_{\mathbf{x}}$ operations and representing the following mathematical structures of Kyber [23], [74]:

1. *FieldElement*: It is a wrapper around a 16-bit unsigned integer representing a polynomial coefficient.

2. *Polynomial*: A polynomial is an array of N coefficients, each of type *FieldElement*.

3. *PolynomialVector*: A polynomial vector is a vector comprising of K *Polynomials*.

In comparison with subsection 2.6.4, we wish to explicitly clarify that $N = n$ and $K = d$, but $d \neq \mathbf{d}$. This module has a trait named *ByteEncoderDecoder* which comprises two functions, namely, *byte_encode*() and *byte_decode*(), and the trait is implemented for the type *PolynomialVector*. We already mentioned (the reason) earlier that we need $\mathit{ByteEncode}_{\mathbf{d}}$, $\mathit{ByteDecode}_{\mathbf{d}}$, $\mathit{Compress}_{\mathbf{x}}$, and $\mathit{Decompress}_{\mathbf{x}}$ for accessing the raw *Polynomials* in *pk* and C_K and later reconstructing them. In this module, we have the respective methods, which effectively perform the same according to *Algorithm 5 and 6* and *Equation 4.7 and 4.8* from [23].

4.3.4 The base_converter Module

This module has two functions as mentioned below, which primarily operate on the *Polynomial* type from above, which represents an array of unsigned integer coefficients. The following two functions act as wrappers around the *convert_base*(*src, targ, digits*) from Algorithm 1:

1. *base_encode_poly*(*source, target, poly*): This performs a (forward) base conversion of a *Polynomial* from base 3,329 to 256 and returns the base-encoded *Polynomial*, namely, $\mathit{poly}_{\mathbf{be}}$.

2. *base_decode_poly*(*source, target,* $\mathit{poly}_{\mathbf{be}}$): Fundamentally, this also performs a base conversion. But it reverses the forward base-conversion process so that the base-256 encoded polynomial ($\mathit{poly}_{\mathbf{be}}$) is represented back in base 3,329. It returns the base-decoded/original *Polynomial*, namely, $\mathit{poly}_{\mathbf{bd}} = \mathit{poly}$.

For the purpose of base conversion (inside these wrapper functions), we have used the *convert-base* crate [76], which allows us to easily convert unsigned integers from one numerical base to another.

4.3.5 The overall_common_functionalities Module

This module comprises some functions that are commonly used by all the chosen PAKE protocols, and they are as follows:

1. A method for obtaining the respective instance of Kyber based on the protocol variant from *AvailableVariants*. This allows for dynamic selection of the Kyber variant at runtime, which will be used in the PAKE protocol execution.

2. It comprises of the methods *get_base_encoded_pk*, *get_base_decoded_pk* and *get_base_encoded_kem_ciphertext*, *get_base_decoded_kem_ciphertext* from Algorithm 2 and Algorithm 3, respectively. And, these methods internally use the wrapper functions defined above in subsection 4.3.4 for the base conversion.

3. We also mentioned earlier that we use AES in counter mode as an approximation for IC for all the PAKE protocols. So, here we also have methods for performing encryption and decryption using AES in counter mode. For a plaintext P, ciphertext C, and an input-key K, the method signatures that are used in the implementation are as follows:
   a) For encryption: $(C, nonce) \leftarrow E(K, P)$
   b) For decryption: $P \leftarrow D(K, C, nonce)$

   This basically allows us to handle the *nonce* in a manner such that the plaintext encryption can happen on one machine and the ciphertext decryption on another one. So, whenever we generate a ciphertext using the AES in counter mode as an approximation for IC, in addition to sending the ciphertext C, we also have to send the corresponding *nonce* to the other party (over a network channel) for successful decryption.

4.3.6 The core_protocol_functionalities Module

This module builds the foundation of this Rust framework. It comprises the core client and server functionalities of the chosen PAKE protocols. It is dependent on the other modules discussed earlier. It is further organized as nested submodules, i.e., we have a module *b_pake_protocols* which comprises the TK-PAKE [49], *Modified* OCAKE-PAKE [50], and KEM-AE-PAKE [51] as separate submodules. Each submodule encapsulates the logic for a specific PAKE protocol, which is further divided into client and server modules, as well as modules comprising some shared cryptographic operations. These modules comprise the following named-field structs and their associated functions:

1. TkServer
2. TkClient
3. OcakeServer
4. OcakeClient
5. KemAeServer
6. KemAeClient

The associated functions of the structs are the implementations of steps like client_init(), server_resp(), etc. We mentioned earlier in the *overall_common_functionalities* module, how the encryption and decryption method signatures handle the *nonce*, the same is applicable for the encryption and decryption method signatures in our implementation for KEM-AE-PAKE, where we use AES-256 GCM for AE.

Items Exposed By the Public API: The public API exposed by the *qr_pake_protocols* crate only comprises the client and server structs mentioned earlier and the *AvailableVariants* and *KemChoice* enumerators. This (only) allows a user of the framework to use these fundamental components and additionally implement their own communication layer to be able to execute the server and the client binary on different machines.

4.4 The qr_pake_protocol_executors Crate

This is a separate workspace crate dependent upon the client and server structs from the *core_protocol_functionalities* and the *AvailableVariants* and *KemChoice* exposed by the public API of the *qr_pake_protocols* crate. This forms the communication layer of our framework using the asynchronous runtime *Tokio* [77], [78]. This crate is also organized as nested submodules, whose details are mentioned further. Finally, it is intended to provide high-level executors or wrappers for just running the PAKE protocols.

4.4.1 The protocol_execution_logic Module

This module forms the foundation of the communication layer of our framework. It primarily abstracts the functionalities from the *core_protocol_functionalities* module and acts as a bridge between the low-level protocol details and practical, real-world protocol execution. It is indeed a central component of the framework as it is responsible for correctly orchestrating the execution of the different steps of the different PAKE protocols while robustly handling the message flows between the client and server. A key design feature of the *protocol_execution_logic* module is the usage of the Tokio asynchronous runtime [77], [78]. Tokio is a widely adopted, high-performance async framework for Rust, enabling the realization of scalable and efficient networked applications. PAKE protocols typically involve multiple rounds of message exchanges between the client and the server, typically over a network channel. Tokio provides the primitives and runtime necessary for this style of programming in Rust. We have two nested submodules in the *protocol_execution_logic* module, namely, the *single-client handler* and the *multi-client handler*.

The Multi-Client Handler Module

This module is implemented for the purpose of developing the communication layer that fulfills the framework requirements from subsection 4.2.2 except the optional ones (item 1(c)iii and item 2b) regarding allowing a (registered and logged-in) user to change their password. Before diving into details about how the registration and login phase works in the multi-client handler, we would like to describe three traits that we use, namely, the *ServerHandler*, *ClientHandler*, and the *Storage* trait.

The ServerHandler Trait: This trait provides a common unified interface for handling the server side of the client registration and login phases in our communication layer for the chosen PAKE protocols. It is designed for processing incoming TCP connections during client registration and login phases. Then, it further delegates the actual work to the registration and login handler of the respective server-side of the PAKE protocol. This trait has two associated asynchronous function signatures, which are as follows:

1. *registration_handler*
2. *login_handler*

These asynchronous registration and login handler functions are individually implemented for the structs TkServer, OcakeServer, and KemAeServer. Once the server-side for any of the PAKE protocol is started, it is expected to run indefinitely (except when it is explicitly shut down) while listening for incoming TCP connections and accordingly handling the registration and/or login.

The ClientHandler Trait: This trait provides a common unified interface for handling the client side of the client registration and login phases in our communication layer for the chosen PAKE protocols. It is designed to initiate a TCP connection to the server during client registration and login phases. Then, it further delegates the actual work to the registration and login handler of the respective client-side of the PAKE protocol. This trait has two associated asynchronous function signatures, which are as follows:

1. *register_client*
2. *login*

These asynchronous registration and login functions are individually implemented for the structs TkClient, OcakeClient, and KemAeClient. For all the chosen PAKE protocols, considering that the server-side is running, on the client-side, the *register_client* communicates with the *registration_handler*, and the *login* communicates with the *login_handler* by establishing TCP connections with the server.

The Storage Trait: We mentioned earlier that we need the server of all of the chosen PAKE protocols to be (ideally) equipped with a (persistent) storage solution because it has to handle multiple clients over time, and it must store key-value pairs. It would use a composite key comprising the client identifier $\mathcal{C}_{id}$, the chosen protocol variant, and *optionally* the KEM choice for *Modified* OCAKE-PAKE, and the value would be the server state which holds a server instance for the current client. To that end, we have chosen to keep the communication layer of the framework generic and storage-agnostic by defining a *Storage* trait. This trait comprises multiple associated function signatures, which work with the defined key and are supposed to handle the following things:

1. Inserting server instance into the storage backend for a particular key. This is needed during a client registration, so that the server is able to fetch its stored instance during the same client's login phase.

2. Getting a stored server instance from the storage backend for a particular key. This is needed during a registered client's login phase.

3. Recording incorrect login attempts for a particular key. This is to ensure that a client (or an adversary using the client's details) cannot perform an online dictionary attack and/or DoS attack on the server. We describe this later in detail.

4. Allowing a particular client to redeem itself by resetting failed login attempts. We also describe this later in detail.

5. Blocking a (potentially malicious) client if it performs multiple consecutive incorrect login attempts and goes over the login threshold within a fixed login window.

6. Checking whether a particular client is already blocked or not by the server.

7. Checking whether a client's details already exist in the storage backend or not.

8. Obtaining the number of incorrect login attempts recorded for a particular client.

In our implementation of the communication layer, we use a HashMap as the storage backend encapsulated in a *DefaultStorage* struct, which implements the *Storage* trait. The HashMap stores key-value pairs as follows:

1. The composite key mentioned earlier.

2. Value: A generic struct named *ServerState* so that it can work with any of the servers, namely, TkServer, OcakeServer, and KemAeServer. For each client, *ServerState* encapsulates the server instance, the last incorrect login attempts, and a boolean flag indicating whether the client is blocked or not.

So, if a user wishes to quickly execute a PAKE protocol, they can use this *DefaultStorage* for the server side. This *DefaultStorage* is provided for quick prototyping, and it is not a persistent storage. So, once the server is stopped, all the stored data will be lost. But a user of the framework can choose to implement their own storage solution and implement the *Storage* trait for that.

The Client Registration Phase: We would like to briefly describe the client registration process in the *Modified* OCAKE-PAKE protocol in reference to our above description of the *Storage* trait and Figure 3.3:

1. We assume that the OcakeServer is already running and listening for incoming TCP connections.

2. During the client registration phase, in our implementation, the client sends the identifier $\mathcal{C}_{id}$, password $\mathcal{C}_{pw}$, the chosen KEM from *KemChoice*, and the chosen protocol variant from *AvailableVariants*. This client message is sent over the network channel with a *Registration* prefix to indicate to the server that this is a (new) client registration request, so that it can be delegated to the respective handler, i.e., the registration handler.

3. Upon receiving the client details, the server checks whether the client with the provided $\mathcal{C}_{id}$, KEM choice, and the protocol variant already exists in the *Default-Storage* or not.

4. If it already exists, then the server sends a warning message to the client saying that the client with the provided details already exists, so the server is skipping the registration process. This is an indication to the client that it should either change its identifier or the KEM choice or the protocol variant, or just directly proceed to log in.

5. If no client with the provided details already exists, then the server accepts the client registration. So, it generates an OcakeServer instance that encapsulates these client-specific details and inserts this OcakeServer instance encapsulated in the *ServerState* struct into the *DefaultStorage*.

6. Finally, the server informs the client that the registration is successful.

The registration phase works similarly for the other chosen PAKE protocols as well, so we do not reiterate the details for them. We mentioned in the design decisions of the communication layer in subsection 4.2.4 that, for the client registration phase, we should either opt for an out-of-band registration or use TLS. In this work, we have just used TCP, but we wish to emphasize that, in real-world or any security-critical (production) environments, either an out-of-band registration phase via a secondary communication channel or using TLS is paramount. This is because sending $\mathcal{C}_{pw}$ (or a verifier) over an unencrypted TCP connection is unsecure as it exposes sensitive data and leaves us vulnerable to Man-in-the-Middle (MITM) attacks.

Brief Description of the Login Phase: Considering the registration phase of a client for the *Modified* OCAKE-PAKE protocol is done, keeping the incorrect login handling part into account (from the communication layer in subsection 4.2.4), the login phase, i.e., the *Modified* OCAKE-PAKE protocol execution succeeds and the same session key is generated on both party's end, iff both the client and the server honestly use the same password during the login phase.

The Single-Client Handler Module

We avoid going into details about the *single-client handler* because it is a minimalistic version of the *multi-client handler* approach. It does not incorporate things like handling incorrect login attempts, using any storage backend, etc. It is written for the sole purpose of conducting performance benchmarks of the chosen PAKE protocols while making them communicate over a network channel, which would not have been possible using the implementation in the *multi-client handler* module, especially on the server side. We wish to be explicit that the single-client handler does not fulfill many of the framework requirements defined in subsection 4.2.2 as it is only implemented for evaluation purposes.

4.4.2 The executors Module

The *executors* module provides a user-facing API to run the PAKE protocols implemented in the *protocol_execution_logic* module. While *protocol_execution_logic* handles the step-by-step asynchronous execution of the protocols from the lower level *core_protocol_functionalities*, the *executors* module acts as a wrapper around it, thereby simplifying the execution process for each PAKE protocol. It comprises three different submodules, which are described further.

The bench_api Module: The *bench_api* module, for each of the PAKE protocol, provides a single unified interface for executing the server and the client side from the *single-client handler* in the *protocol_execution_logic* module. This interface is used for conducting the benchmark of the PAKE protocols as mentioned in section 5.3. These benchmarking-relevant methods from the *bench_api* are also exposed by the public API, as it cannot be suppressed. So, we encourage a user of the framework to avoid using the *bench_api* for the purpose of actually using the framework itself, as in this case, one cannot run a server and a client binary on two different machines.

The generic_server Module: This module consists of an asynchronous function named *start_server*, which serves as a wrapper and a generic entry point for running a TCP-based server. It is defined as an asynchronous function that is generic over the types (TkServer, OcakeServer, and KemAeServer) implementing the *ServerHandler* trait. The function is capable of instantiating and managing different server variants, depending on the protocol being executed. Being a top-level interface, *start_server* listens for incoming TCP connections on a specified IP address and port. Upon receiving any connection from a client, it reads the initial prefix (*Register* or *Login*) and determines the type of request and accordingly dispatches it to the respective asynchronous handler function, i.e., registration or login for the corresponding *ServerHandler* implementation. If the request is of some other type, then the server returns an error to the client, mentioning that it is an unknown request type. The handlers (registration or login) invoked are completely protocol-specific, meaning that each server type (TkServer, OcakeServer, and KemAeServer) implements entirely different internal workflows for the different handlers in the communication layer of our framework.

The generic_client Module: This module consists of two asynchronous functions, namely, *register* and *login*, which serve as a wrapper and a generic entry point to the communication layer of the client-side of the PAKE protocols. These two functions are generic over the types (TkClient, OcakeClient, and KemAeClient) implementing the *ClientHandler* trait. They are used to execute the registration and login phases of a PAKE protocol from the client's perspective. The internal implementation of these *register* and *login* functions is completely protocol-specific.

Items Exposed By the Public API: The generic server and client module described above allow the framework to dynamically support the complete registration and login phase of multiple PAKE protocols without modifying the higher-level control flow or interface. Consequently, introducing a new PAKE protocol involves only implementing the *ServerHandler* and *ClientHandler* trait for the new server and client type, respectively, without having to alter the core logic of these two generic modules. So, these generic modules not only simplify the invocation of the client registration and login phases, but also provide a robust, extensible, and maintainable abstraction layer for incorporating PAKE protocols further. Finally, this *qr_pake_protocol_executors* crate publicly exposes the following, which also makes them a part of the public API of the framework as a whole:

1. The *bench_api*, *generic_server* and the *generic_client* module.
2. The *DefaultStorage* struct which uses HashMap.
3. The *ServerHandler*, *ClientHandler* and the *Storage* trait.
4. The following default constants:

   a) IP = 127.0.0.1
   b) Port = 8080
   c) Login_Window = 60
   d) Threshold = 3
   e) Timeout = 5

Reconciling with the Requirements and the Public API Design: Our design and implementation choices enable users to run a PAKE protocol defined in the *qr_pake_protocols* framework without burdening themselves with the low-level details. Circling back to our goal of offering a *Hybrid* API, our design indeed provides users with two options:

1. To be able to use the communication layer of the framework using Tokio [77], [78] based on the *generic_server* and the *generic_client* module and the *AvailableVariants* (and optionally the *KemChoice*) enumerator exposed by the public API.
2. To optionally implement own communication layer based on the client and server structs and the *AvailableVariants* (and optionally the *KemChoice*) enumerator exposed by the public API.

4.5 A Sample Demonstration

The examples demonstrate how to run the *Recommended* variant of *Modified* OCAKE-PAKE with Frodo-KEM, i.e., execute the PAKE protocol with Frodo-976 (as seen in Figure 4.3), which relies on Tokio [77], [78] for asynchronous execution. Users must implement separate client and server binaries, as these run on different machines. While the examples use default values for IP, port, login window, threshold, and timeout, users can customize these as needed. As shown in Figure 4.2 and Figure 4.3, we use *start_server* from the *generic_server* module, and *register* and *login* from the *generic_client* module for running the PAKE protocol. The other two protocols can also be executed in a similar manner, except we have to use the respective server and client structs/types corresponding to the protocol itself.

```
use qr_pake_protocol_executors::{start_server, DefaultStorage,
    Storage, DEFAULT_IP, DEFAULT_LOGIN_THRESHOLD,
        DEFAULT_LOGIN_WINDOW, DEFAULT_PORT, DEEFAULT_TIMEOUT};
use qr_pake_protocols::OcakeServer;
use std::sync::Arc;

#[tokio::main]
async fn main() {
    let storage = Arc::new(DefaultStorage::<OcakeServer>::new())
        as Arc<dyn Storage<_> + Send + Sync>;

    start_server::<OcakeServer>(storage, DEFAULT_IP,
        DEFAULT_PORT, DEFAULT_LOGIN_THRESHOLD,
        DEFAULT_LOGIN_WINDOW,
        DEEFAULT_TIMEOUT).await.expect("Server failed to start");
}
```

Figure 4.2: The server binary

```
use qr_pake_protocol_executors::{login, register, DEFAULT_IP,
    DEFAULT_PORT};
use qr_pake_protocols::{AvailableVariants::Recommended,
    KemChoice::Frodo};

#[tokio::main]
async fn main() {
    let c_id = b"This is a default pake client ID";
    let c_pw = b"This is client default password.";

    let c_inst = register::<OcakeClient>(c_id, c_pw, Recommended,
        Some(Frodo), DEFAULT_IP, DEFAULT_PORT)
        .await.expect("Registration failed");

    let sk_c = login::<OcakeClient>(c_inst, DEFAULT_IP,
        DEFAULT_PORT).await.expect("Login failed");
    println!("Client's Session Key: {:?}",sk_c);
}
```

Figure 4.3: The client binary

Summary

In this chapter, we started by stating our reasons for preferring Rust over any other (similar) low-level programming languages like C/C++ (in section 4.1). We justified that Rust is indeed a *safety-first* language with only a negligible impact on performance, and it is indeed a fair trade-off considering the safety guarantees inherently offered by Rust. Then, we provided a conceptual overview of the framework, starting with the details of the public API design and what it could potentially offer (in subsection 4.2.1). We finally decided that the public API will provide the core protocol functionalities and a predefined communication logic. Then, we discussed about how we want to have access to all variants of Kyber [23] for all the chosen PAKE protocols. Additionally, we also stated that the framework should allow access to the three Frodo-KEM variants [22] with SHAKE-128 XOF as the primitive for generating the public matrix $\mathbf{A}$, exclusively for the *Modified* OCAKE-PAKE protocol. Afterwards, we described the different functional, security, and non-functional requirements of the framework (in subsection 4.2.2). Then, we briefly introduced the direct encryption problem upon encrypting a KEM public key pk or ciphertext C_K generated using Kyber and the lack of explicit authentication in TK-PAKE, along with their detailed solutions in subsection 4.2.3. Based on the requirements, we provided a technology-agnostic overview of the components of the framework in subsection 4.2.4. Here we discussed about the core functionalities and the communication layer of the framework. In the high-level discussion of the communication layer, we mentioned our reasons for choosing TCP over UDP, asynchronous over synchronous I/O, and how the storage backend must be designed.

Then, we dove into the low-level details of the framework, which state exactly how we have implemented the framework based on the requirements and the overview of the framework's components. Starting with the *qr_pake_protocols* crate (in section 4.3), which forms the core functionalities layer, we described the different modules and how they allow us to easily switch between different variants of different KEMs and use them for the chosen protocols in the framework. Based on what is exposed by the public API of the core functionalities layer, we described the *qr_pake_protocol_executors* crate (in section 4.4), which forms the communication layer of our framework. In this case, we particularly emphasized the multi-client handler, which fulfills the different requirements of the framework, allows concurrent handling of multiple clients, while also handling incorrect login attempts. We also mentioned that the single-client handler is only used for benchmarking purposes. Then, we briefly described the *bench_api*, the generic server and client modules. These generic modules provide a unified entry point for starting the server and executing the client side for any of the chosen PAKE protocols, which is also exposed by the public API.

We finally provided a demonstration (in section 4.5) of running the server and client binaries on two different machines for the *Modified* OCAKE-PAKE. Now, we conduct the performance benchmarking of all three PAKE protocols using the *bench_api* module, and provide our results in chapter 5.

Chapter 5

Evaluation

In this chapter, we begin by outlining the key research questions that motivate our benchmarking and performance evaluation efforts. Then, we describe the experimental setup used for benchmarking and detail the methodology followed for conducting the benchmarking itself. This is followed by an analysis and interpretation of the performance results for the implemented Password Authenticated Key-Exchange (PAKE) protocols. Finally, we conclude with a brief discussion on the security aspects of the framework implementation.

5.1 Research Questions

In this section, we briefly state our research questions, which we will address through our setup and evaluation as we proceed. The questions are as follows:

1. What is the codebase size of the framework, including core implementation and dependencies?
2. How do distinct PAKE protocol designs influence computational and communication overhead?
3. How does the number of Ideal Cipher (IC) instances used in a PAKE protocol impact its performance?
4. How do different Key-Encapsulation Mechanisms (KEMs) impact the performance and communication overhead of the same PAKE protocol?

5.2 Experimental Setup

In this section, we initially provide the details of the system on which the benchmark has been conducted. Then, we provide an insight into the lines of code in our framework, along with an estimate of the lines of code in the dependencies that we have used, as this plays a vital role in transparency and ensuring maintainability of the codebase.

System Specifications

The *qr_pake_protocols* framework is implemented in Rust v1.87.0. We have evaluated the PAKE protocols in the framework on a system with the following specifications:

1. *Device*: Dell Inspiron 15 3511 Laptop
2. *Operating System*: Windows 11 Home Single Language v24H2 64-bit
3. *Processor*: 11$^{\text{th}}$ Gen Intel(R) Core(TM) i5-1135G7 @ 2.40 GHz
4. *Installed RAM*: 8.00 GB DDR4 RAM (7.74 GB usable)

Codebase and Dependency Size Analysis

Our *qr_pake_protocols* framework comprises exactly 5,654 lines of (Rust) code, excluding code comments and blank lines, with the following breakdown:

1. The *qr_pake_protocols* crate (or the core functionalities layer): 1,823 lines
   a) 954 lines for the *core_protocol_functionalities* module
   b) 869 lines for the other modules
2. The *qr_pake_protocol_executors* crate (or the communication layer): 2,532 lines
   a) 1,765 lines for the *multi_client_handler, generic_server* and *generic_client* module
   b) 767 lines for the *single_client_handler* and the *bench_api* module
3. *benches* and *examples*: 1,299 lines

In addition to the open-source Rust crates mentioned earlier, we also use the following:

1. *aes* [79]
2. *ctr* [80]
3. *aes-gcm* [81]
4. *sha2* [82]
5. *sha3* [83]
6. *serde* [84]
7. *serde_json* [85]
8. *rand* [86]
9. *rand_core* [87]
10. *anyhow* [88]
11. *strum* [89]
12. *strum_macros* [90]
13. *zeroize* [91]
14. *async-trait* [92]

We have gathered the data from [93] for estimating the lines of code of the aforementioned crates used in our framework as dependencies, which is $\approx 525,142$. This count is after considering the (transitive) dependencies used by these crates themselves. Finally, the total lines of code in the framework, including the aforementioned dependencies, are $\approx 525,142 + 5,654 = 530,796$. We wish to explicitly mention that we have excluded the count of two additional dependencies, namely, the *stats-ci* [94] and *rust_xlsxwriter* [95] crates (which are $\approx 136,500$ and $\approx 254,000$ respectively) from above, as they are only used for benchmarking and evaluation purposes and will not be a part of the finally compiled binary. These details and estimations act as an answer to the research question 1 in section 5.1.

5.3 The Benchmarking Process

We mentioned in subsection 4.4.2 that we use the *bench_api* module for conducting the benchmarking of the implemented PAKE protocols and performing statistical analysis on the gathered data. We describe the benchmarking process for the *Modified* OCAKE-PAKE [50] protocol below. As this is for the *Modified* OCAKE-PAKE, we conduct its benchmark for all three variants of the two KEMs, namely, Kyber [23] and Frodo-KEM [22]. For each variant of each KEM, the following steps are taken:

1. We have the following predefined constants:

a) Iteration counts = 100 b) Confidence level = 95%

2. We initially generate 100 distinct pairs of client identifier ($\mathcal{C}_{id}$) and password ($\mathcal{C}_{pw}$).

3. For each individual client (from above), we perform a one-time client registration phase with the server, which initializes the client and the server instances for this current client. We gather a tuple of the generated client and server instance, and repeat this process for all other clients. Finally, we end up with 100 tuples of distinct client and server instances, which will be used for actually benchmarking the PAKE protocol. As for each client, the registration phase is only executed once (even in the real world), the execution time for gathering these client and server instances is not accounted for in the benchmark.

4. Then, the actual benchmarking is done using the above-generated 100 distinct client and server instances. Here, we run the PAKE protocol execution phase from Figure 3.3 for each distinct tuple of the client and server instance, so the PAKE protocol gets executed 100 times. This also ensures that each iteration of the PAKE protocol execution is completely independent from one another. During each iteration, the communication overhead and the total execution time for the different steps of the client and the server are individually gathered.

5. Once all the data has been gathered, we use the *stats-ci* crate [94] to individually compute the average execution time, its standard deviation, and the 95% confidence intervals for the client and the server.

6. We wish to explicitly mention that the execution times gathered in *step 4* are in seconds, upon which the statistical analysis is performed (according to *step 5*). But, all the final results from *step 5* are multiplied by 1,000 to convert them into milliseconds. Finally, the converted results are printed in a colour-coded human-readable format in the terminal and are also written to an Excel file using the *rust_xlsxwriter* crate [95]. *Steps 2-6* are repeated with the remaining variants of the particular KEM.

The procedure is similar for the other two chosen PAKE protocols, except that the benchmark for those two is only conducted for the different variants using Kyber.

5.4 Interpretation of Performance Evaluation and Communication Overhead

For our evaluation, we only compare the performance of the same variant of two PAKE protocols, as it ensures fairness and that they provide the same NIST security level. We have conducted a benchmark of the following:

1. The performance evaluation mentioned in Table 5.1 represents the average execution time per iteration, in milliseconds, computed over 100 individual iterations of a PAKE protocol execution using Kyber. To be explicit, in Table 5.1, we compare TK-PAKE with the client authentication step with the other two protocols instead of comparing the original one without the client authentication. This is primarily because TK-PAKE with client authentication is the one that actually ensures that the same session key is established on both parties' ends and fulfills our requirements.

2. We know from subsection 3.4.1 and Figure 3.2 that TK-PAKE does not provide explicit (mutual) authentication, so we introduced a client authentication check as per Figure 4.1. So, we conducted the benchmark of TK-PAKE with and without client authentication over 100 iterations, as is evident from Table 5.4.
3. As we have implemented *Modified* OCAKE-PAKE using both Kyber and Frodo-KEM, Table 5.6 provides a comparative performance overview of the protocol over 100 iterations using the two different KEMs.

In addition to the average execution times in Table 5.1, Table 5.4, and Table 5.6, we also provide the standard deviation of the execution times and the respective confidence interval of the average execution time with a 95% confidence level. We wish to explicitly mention that the values in the aforementioned three tables are rounded off to three decimal places, and the average execution times are only for the PAKE protocol execution phase from Figure 3.2 (along with Figure 4.1 for the client authentication check), Figure 3.3, and Figure 3.4. The communication overhead incurred by the client and the server for a single PAKE protocol execution using Kyber is mentioned in Table 5.2. Similarly, a comparative overview of the communication overhead is also mentioned for the following:

1. For TK-PAKE with and without the client authentication in Table 5.5.
2. For *Modified* OCAKE-PAKE using two different KEMs in Table 5.7.

5.4.1 Clarification Regarding Performance Evaluation Using Kyber

Table 5.1: Performance evaluation data using Kyber for 100 iterations

PAKE Protocol Variant		**PAKE Protocols**					
		***Modified* OCAKE-PAKE**		**TK-PAKE With Authentication**		**KEM-AE-PAKE**	
		Client	**Server**	**Client**	**Server**	**Client**	**Server**
LW	**AET**	4.196	2.591	6.549	6.313	7.175	6.469
	SD	0.676	1.365	1.161	1.821	1.624	2.502
	CI	[4.062, 4.330]	[2.320, 2.862]	[6.319, 6.780]	[5.952, 6.674]	[6.853, 7.497]	[5.972, 6.965]
RC	**AET**	5.497	2.813	8.262	7.170	9.789	9.688
	SD	0.966	1.146	0.959	1.585	2.183	3.531
	CI	[5.306, 5.689]	[2.585, 3.040]	[8.072, 8.452]	[6.856, 7.485]	[9.355, 10.222]	[8.987, 10.388]
PR	**AET**	6.325	3.434	9.808	8.067	10.517	9.583
	SD	1.427	0.960	1.401	1.306	2.242	3.461
	CI	[6.042, 6.608]	[3.244, 3.625]	[9.530, 10.086]	[7.808, 8.326]	[10.072, 10.962]	[8.896, 10.270]

Legends

AET: Average Execution Time (in ms), **SD**: Standard Deviation, **CI**: 95% Confidence Intervals, **LW**: LightWeight Variant, **RC**: Recommended Variant, **PR**: Paranoid Variant

Based on the data from Figure 5.1 and Table 5.1, with respect to computational intensiveness, it always seems that *LightWeight* < *Recommended* < *Paranoid*. This is primarily because the PAKE protocols use the different variants of Kyber [23]. The dimensions d (from Table 2.6) dominate the (size and) complexity of the different polynomial operations in the different variants of Kyber. So, the respective Kyber variants, namely, Kyber-512, Kyber-768, and Kyber-1024 mapped to *LightWeight*, *Recommended*, and *Paranoid*, demonstrate that *LightWeight* < *Recommended* < *Paranoid*. Upon careful observation, it also seems that the PAKE protocol description in Figure 3.2 (along with Figure 4.1), Figure 3.3, and Figure 3.4 indicates that the workload (in terms of the number and complexity of steps/operations to perform) is always more on the client's end than the server's. The same is also reflected for every PAKE protocol in Table 5.1 and Figure 5.1 in terms of the average execution time of the clients, which is always more than that of the servers.

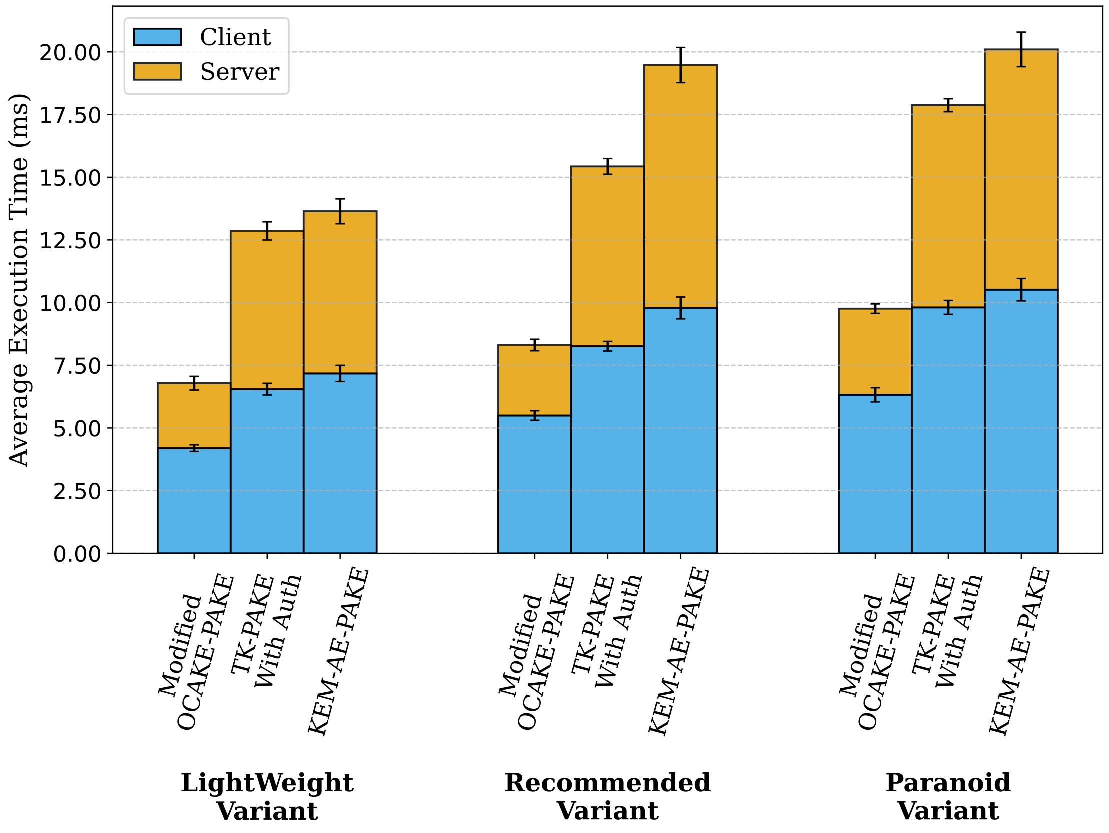

Figure 5.1: Performance evaluation using Kyber

Upon observing the data in Table 5.1 or Figure 5.1, it is evident that the KEM-AE-PAKE is the slowest protocol amongst the three. In our opinion, based on Figure 3.4, this happens because it has multiple time-consuming cryptographic operations (as mentioned below) on both the client and server's end compared to the other two protocols, which contribute to its high latency:

1. It involves two Encaps operations during server_resp() and two Decaps operations in client_finish().

2. Usage of two IC instances and once Authenticated Encryption (AE).

Modified OCAKE-PAKE demonstrates the best performance amongst all three PAKE protocols using Kyber (as evident from Figure 5.1 and Table 5.1). This is because it is the most lightweight protocol amongst the three. The primary reason behind its efficiency is, it only requires one instance of the IC, unlike the other two protocols, which require two IC instances. The overhead incurred due to the base-encoding before the IC encryption (using AES in counter mode as an approximation) and the base-decoding after decryption to uphold the security guarantees comes at a price of increased computational costs. And, the more the number of IC instances needed, the higher the overhead, which was also correctly pointed out in CHIC-PAKE [34] as mentioned in subsection 3.2.2.

5.4.2 Clarification Regarding Communication Overhead Using Kyber

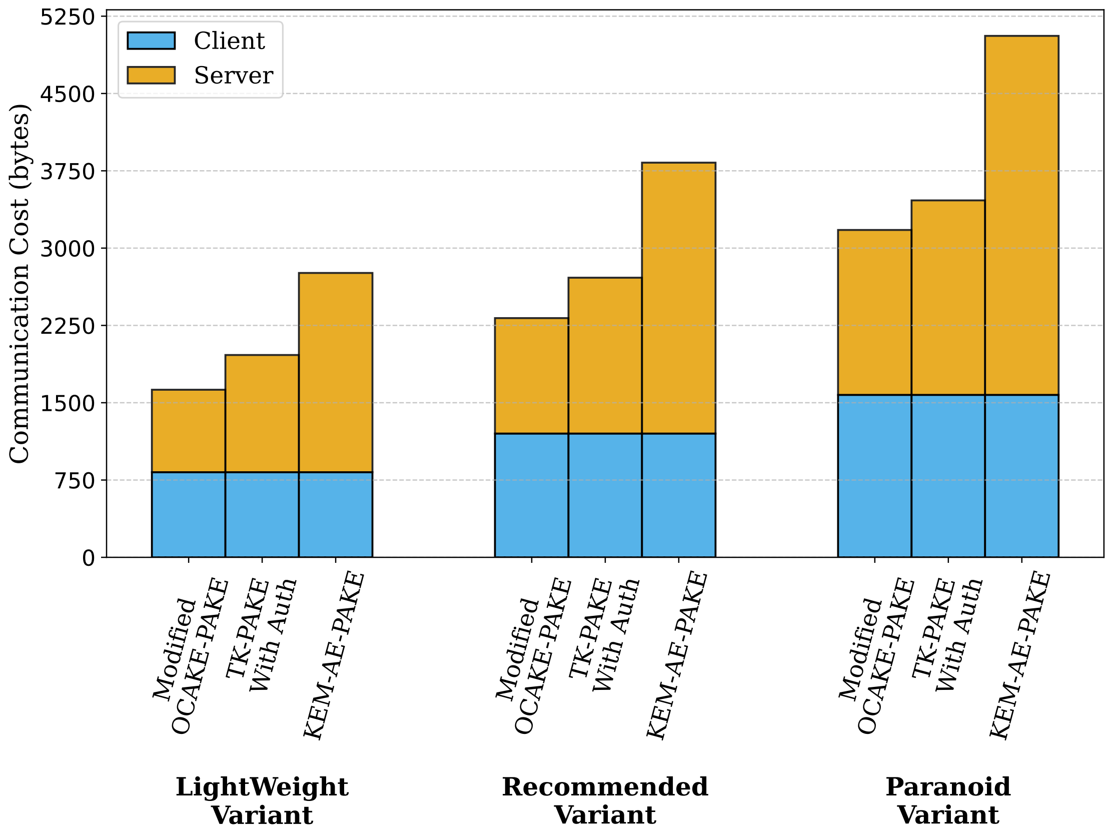

Figure 5.2: Communication overhead using Kyber

Table 5.2 and Figure 5.2 capture the communication overhead incurred by the client and the server for the different variants of the PAKE protocol execution phase. The client's communication cost for all three PAKE protocols is exactly the same. This is because the client sends an IC encrypted ciphertext, i.e., the encrypted, base-encoded KEM public key $pk_{\mathbf{be}}$ and a SHA-256 hash digest of 32 bytes (as per Figure 3.2 (along with Figure 4.1), Figure 3.3, and Figure 3.4). The server's communication cost in KEM-AE-PAKE is the highest because it sends two encrypted KEM ciphertexts, namely, C_2 (which is IC encrypted) and ψ (as per Figure 3.4). The server's communication cost in *Modified* OCAKE-PAKE is even lower than that of TK-PAKE, because, in *Modified* OCAKE-PAKE, C_K is sent in the clear, which is inherently an extremely compressed value, as evident from Appendix A. In contrast, the encrypted, base-encoded KEM ciphertext in TK-PAKE has a certain overhead, also evident from Appendix A.

Table 5.2: Communication overhead of client and server in bytes using Kyber

PAKE Protocol Variant	**PAKE Protocols**					
	***Modified* OCAKE-PAKE**		**TK-PAKE With Authentication**		**KEM-AE-PAKE**	
	Client	**Server**	**Client**	**Server**	**Client**	**Server**
LightWeight	826	800	826	1,137	826	1,933
Recommended	1,201	1,120	1,201	1,512	1,201	2,628
Paranoid	1,576	1,600	1,576	1,887	1,576	3,483

With reference to Table 2.7, the size of $pk > C_K$ for Kyber-512 and Kyber-768. Due to this, in Table 5.2, the client's communication cost in *Modified* OCAKE-PAKE is higher than the server's (for the *LightWeight* and *Recommended* variant). This is because the client sends the encrypted, base-encoded KEM public key $pk_{\mathbf{be}}$, while the server sends the C_K in the clear, which is an extremely compressed value (as per Appendix A). In the *Paranoid* variant (Kyber-1024), for *Modified* OCAKE-PAKE, the server's communication cost exceeds the client's since the original size of $pk = C_K$ as per Table 2.7, and base-encoding the pk slightly reduces its size as per Appendix A while the size of C_K remains the same as it is sent in the clear. In contrast, in the other two PAKE protocols from Table 5.2, the client sends the encrypted, base-encoded $pk_{\mathbf{be}}$ and the server sends the encrypted, base-encoded $C_{K\mathbf{be}}$, where base-encoding adds an overhead to $C_{K\mathbf{be}}$ but reduces the size in case of $pk_{\mathbf{be}}$ (as per Appendix A), leading to consistently higher server-side communication costs. This illustrates that the (original) sizes of pk and C_K, along with base encoding, are the primary contributors to communication overhead. While authentication tags are also exchanged, their impact is minimal and does not (significantly) alter this analysis.

Clarification of Communication Overhead in KEM-AE-PAKE: We know that KEM-AE-PAKE is only implemented using Kyber-768 in [51], but the communication costs depicted in our Table 5.2 for KEM-AE-PAKE with Kyber-768 (i.e., the *Recommended* variant) and the ones mentioned in *Tables 1 and 3* in [51] are different as evident from Table 5.3.

Table 5.3: Comparison of client and server communication costs between our implementation and the original KEM-AE-PAKE [51]

Party	Our Implementation	Original
Client	1,201 Bytes $\approx$ 1.201 KB	0.031 KB
Server	2,628 Bytes $\approx$ 2.628 KB	1.094 KB

From Table 2.7, we know that in Kyber-768, the size of $C_K = 1,088$ bytes. From Figure 3.4, we see that the server sends (C_2, ψ) where C_2 is the IC encryption of the base encoded KEM ciphertext C_{K1} and ψ is the AE_E encryption of C_{K2}, respectively. So, the size of C_2 (including the overhead from base conversion as evident from Appendix A) and its nonce will be 1,500 + 12 = 1,512 bytes, and the size of ψ, which has the authentication tag of the AE and the nonce, will be 1,088 + 16 + 12 = 1,116 bytes. So, the total size of (C_2, ψ) is 1,512 + 1,116 = 2,628 bytes in our implementation. Since [51] neither clarifies the communication cost details nor provides an open-source implementation, we cannot verify their decisions regarding server-client communication costs. However, we believe the authors only account for the difference in communication costs between the KEM-AE-PAKE and executing the Kyber (KEM) protocol. Specifically, as per Figure 3.4, they likely consider only the server-to-client message ψ (1,116 Bytes 1.116 KB), excluding C_2, and the client-to-server hash digest σ (32 Bytes 0.032 KB), excluding C_1. These values roughly align with those mentioned in [51], as seen in our Table 5.3, though minor discrepancies may stem from an unusual rounding method.

5.4.3 Comparison of TK-PAKE With and Without Client Authentication

The Table 5.4 and Table 5.5 respectively depict the comparative performance overview and the communication overhead of TK-PAKE with and without the client authentication step. We observe a slight increase in the average execution time on both the client's and the server's end when performing client authentication, which is due to generating the hash digest of the client's session key and its verification on the server's end. The communication overhead only increases by exactly 32 bytes on the client side because of the SHA-256 digest of the client's session key, which is sent to the server, and the server's communication costs remain intact.

Table 5.4: Performance comparison of TK-PAKE with and without client authentication using Kyber for 100 iterations

Variant		**Without Authentication**		**With Authentication**	
		Client	**Server**	**Client**	**Server**
LW	**AET**	6.474	5.969	6.549	6.313
	SD	1.051	1.548	1.161	1.821
	CI	[6.265, 6.682]	[5.662, 6.276]	[6.319, 6.780]	[5.952, 6.674]
RC	**AET**	8.071	6.960	8.262	7.170
	SD	0.995	1.344	0.959	1.585
	CI	[7.873, 8.268]	[6.694, 7.227]	[8.072, 8.452]	[6.856, 7.485]
PR	**AET**	9.697	7.968	9.808	8.067
	SD	1.400	1.199	1.401	1.306
	CI	[9.419, 9.974]	[7.730, 8.206]	[9.530, 10.086]	[7.808, 8.326]

Legends

AET: Average Execution Time (in ms), **SD**: Standard Deviation, **CI**: 95% Confidence Intervals, **LW**: LightWeight Variant, **RC**: Recommended Variant, **PR**: Paranoid Variant

Table 5.5: Comparison of communication overhead of client and server in bytes in TK-PAKE using Kyber with and without client authentication

Variant	**Without Authentication**		**With Authentication**	
	Client	**Server**	**Client**	**Server**
LightWeight	794	1,137	826	1,137
Recommended	1,169	1,512	1,201	1,512
Paranoid	1,544	1,887	1,576	1,887

Finally, we could say that our detailed interpretation of the performance evaluation and the communication overhead from subsection 5.4.1, subsection 5.4.2, and subsection 5.4.3 acts as an answer to the research question 2 and research question 3 in section 5.1.

5.4.4 Comparison of Modified OCAKE-PAKE Using Different KEMs

As mentioned in subsection 5.4.1, $\textit{LightWeight} < \textit{Recommended} < \textit{Paranoid}$ for variants of Kyber, it holds for variants of Frodo-KEM [22] as well, i.e., Frodo-640, Frodo-976, and Frodo-1344 mapped to *LightWeight*, *Recommended*, and *Paranoid*, demonstrate that $\textit{LightWeight} < \textit{Recommended} < \textit{Paranoid}$. Upon observing Figure 5.3 and Table 5.6, it is evident that the *Modified* OCAKE-PAKE using Kyber outperforms *Modified* OCAKE-PAKE using Frodo-KEM for every variant. The reason behind this is the underlying mathematical construct of the KEMs. Kyber is based on Module Learning With Errors (MLWE) and Frodo-KEM is based on Learning With Errors (LWE).

Table 5.6: Performance comparison of *Modified* OCAKE-PAKE with two different KEMs for 100 iterations

Variant		**KEM**			
		Kyber		**Frodo**	
		Client	**Server**	**Client**	**Server**
LW	**AET**	4.196	2.591	10.640	4.137
	SD	0.676	1.365	1.367	1.028
	CI	[4.062, 4.330]	[2.320, 2.862]	[10.369, 10.911]	[3.933, 4.341]
RC	**AET**	5.497	2.813	16.086	6.067
	SD	0.966	1.146	2.136	0.309
	CI	[5.306, 5.689]	[2.585, 3.040]	[15.662, 16.510]	[6.005, 6.128]
PR	**AET**	6.325	3.434	25.155	10.728
	SD	1.427	0.960	2.160	0.345
	CI	[6.042, 6.608]	[3.244, 3.625]	[24.726, 25.584]	[10.659, 10.796]

Legends

AET: Average Execution Time (in ms), **SD**: Standard Deviation, **CI**: 95% Confidence Intervals, **LW**: LightWeight Variant, **RC**: Recommended Variant, **PR**: Paranoid Variant

We mentioned earlier in subsection 2.6.3 that MLWE is inherently more efficient than LWE. Frodo-KEM based on LWE, which is algebraically unstructured, operates on dimensionally large matrices and integer vectors compared to the polynomial vectors in Kyber, and these are not only computationally intensive operations, but they also incur a large communication overhead. With respect to Figure 5.3, in addition to the average execution time, it also depicts the confidence interval of the average execution time with a 95% confidence level. In case of the *Recommended* and *Paranoid* variant, the confidence interval of the server's average execution time is extremely compact/narrow (as also evident from Table 5.6), so it might not seem explicitly visible in Figure 5.3.

Table 5.7: Comparison of communication overhead of client and server in bytes in *Modified* OCAKE-PAKE with two different KEMs

Variant	**KEM**			
	Kyber		**Frodo**	
	Client	**Server**	**Client**	**Server**
LightWeight	826	800	9,660	9,784
Recommended	1,201	1,120	15,676	15,824
Paranoid	1,576	1,600	21,564	21,728

As for the communication costs from Figure 5.4 and Table 5.7, it does not require much clarification because the difference between the KEM ciphertext and public-key sizes in Frodo-KEM and Kyber is evident from Table 2.5 and Table 2.7, respectively, which are the main contributors to the communication costs. We provided a clarification regarding the impact of the size of pk and C_K of different variants of Kyber on the communication overhead in subsection 5.4.2. When using Frodo-KEM, although there is no concept of base-encoding (and decoding), the communication overhead of the client is always less than that of the server. This is because, with respect to Table 2.5, for all the variants of Frodo-KEM, the size of $pk < C_K$. In Modified OCAKE-PAKE with Frodo-KEM, the client transmits the encrypted pk, incurring no additional overhead aside from the 12-byte *nonce*, while the server sends the C_K in the clear. As a result, the communication overhead is greater on the server side. While authentication tags are also exchanged, their impact is minimal and does not alter this analysis with Frodo-KEM. The aforementioned details act as an answer to the research question 4 in section 5.1.

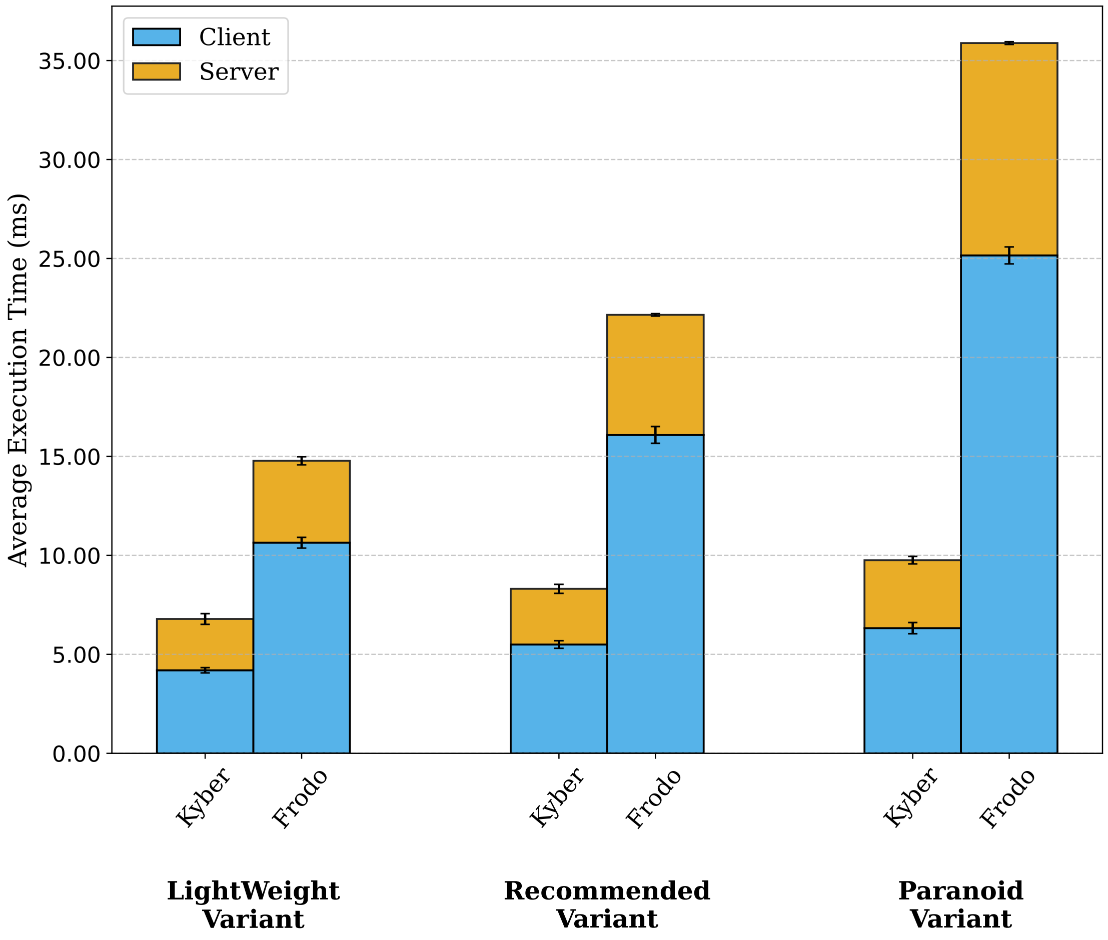

Figure 5.3: Performance evaluation of *Modified* OCAKE-PAKE

5.5 Discussing the Implementation Security

We mentioned in section 4.1 that Rust is inherently a safety-first language, and our *qr_pake_protocols* framework is built on pure Rust. So, we believe that our implementation already benefits from this in terms of being resistant to common implementation-

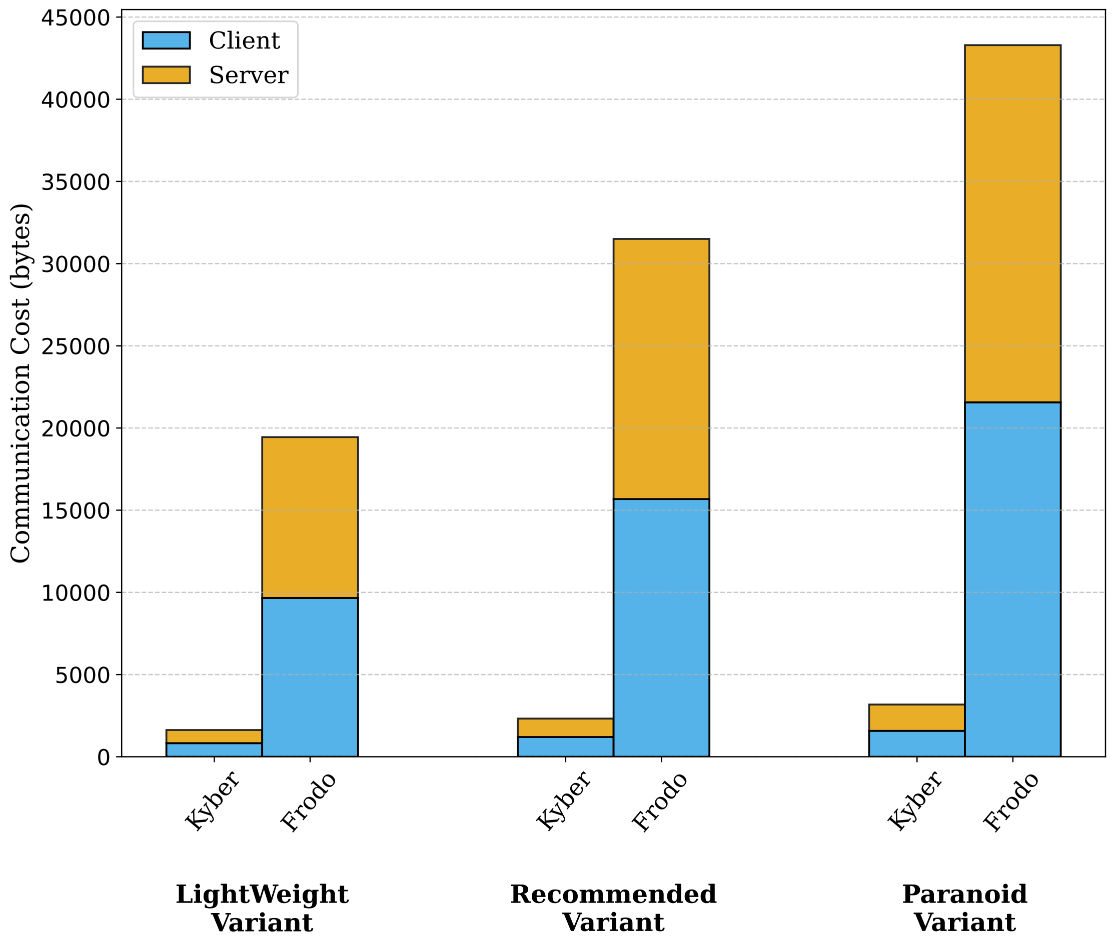

Figure 5.4: Communication cost of *Modified* OCAKE-PAKE

level vulnerabilities. We believe that the below-mentioned points indicate how the implementation fulfills the different framework requirements from subsection 4.2.2:

1. The *mlkem_variant_wrapper* and *frodo_variant_wrapper* modules (as mentioned in subsection 4.3.2) allow simultaneous access to all three variants of Kyber [23] and Frodo-KEM [22] in our framework.

2. The *AvailableVariants* and *KemChoice* enumerators in the *protocol_variants* module (in subsection 4.3.1) facilitates flexible selection among different KEMs and their respective variants during the instantiation of a PAKE protocol at runtime.

3. The public API of the *qr_pake_protocols* crate exposes the client and server structs in the *core_protocol_functionalities* module (as mentioned in subsection 4.3.6). This provides access to a user to all the (low-level) core functionalities of the PAKE protocols in an encapsulated manner, which can be easily used.

4. The *multi-client handler* in the *protocol_execution_logic* module (in subsection 4.4.1) forms the communication layer of the framework. It can handle the registration and/or login phase of multiple clients concurrently, and also prevent malicious clients or adversaries from performing online dictionary attacks and launching a DoS attack on the server by having measures to handle multiple consecutive incorrect login attempts.

Additionally, we believe that there are a few other strong suits of our implementation, which are as follows:

1. No *unsafe* blocks have been used in the codebase, and all the external crates used are also purely Rust crates. This ensures the inherent memory safety guarantees provided by Rust are upheld by the compiler. It also aligns with the industry best practices regarding Rust, which advocate the minimal usage of *unsafe* blocks.
2. All the cryptographic operations are done using established and vetted open-source Rust crates. This provides us with confidence in the following:
   a) As cryptographic operations are inherently complex and prone to subtle errors, using well-regarded, community-maintained Rust crates minimizes the risk of introducing vulnerabilities, especially in security-critical applications like quantum-resistant PAKE protocols.
   b) More often than not, cryptographic operations are computationally intensive, and these open-source crates are optimized for performance. So, it ensures that these operations do not become a bottleneck in our application.
3. It may happen that sensitive data like the client's password $\mathcal{C}_{pw}$, KEM secret key sk, seeds for key generation, etc., are left in the memory even after the completion of code execution. To mitigate this, we have kept the data types of such sensitive values that support zeroization. We use the *zeroize* [91] crate to erase such sensitive data from memory in a reliable manner upon completion of a PAKE protocol execution from both the client's and the server's end. However, certain Rust operations *might* leave copies of the same data in different memory locations, so we cannot guarantee or check for the complete erasure of the sensitive data. This measure is just to demonstrate our approach to uphold the best practices in secure software development using Rust.

Summary

In this chapter, we started by stating some research questions in section 5.1 and then provided the details of our benchmark setup in section 5.2, which comprises information on the system specifications and the analysis of the codebase size. Here, we also saw which external Rust crates were needed to be used to successfully realize our framework. Then, we proceeded with describing our whole benchmarking process in section 5.3 for *Modified* OCAKE-PAKE. Going further, we mentioned the concrete performance evaluation data of all three PAKE protocols using Kyber in Table 5.1 and the communication costs in Table 5.2. For TK-PAKE with and without client authentication, we additionally provided our comparative overview of the performance and communication overhead in Table 5.4 and Table 5.5, respectively. As we had also implemented *Modified* OCAKE-PAKE using Frodo-KEM, we provided the concrete data of *Modified* OCAKE-PAKE using Kyber and Frodo-KEM in Table 5.6 and Table 5.7.

In subsection 5.4.1, we stated that KEM-AE-PAKE demonstrates the slowest performance because it comprises the highest number of computationally intensive cryptographic operations amongst all three PAKE protocols. We also observed that *Modified* OCAKE-PAKE (with Kyber) demonstrates the best performance amongst all three PAKE protocols. We identified that the primary reason behind this is the number of IC instances used in the PAKE protocols. In subsection 5.4.2, we observed that the client-side communication cost is the same for all the PAKE protocols and, due to the design of KEM-AE-PAKE, it has the highest server-side communication cost amongst all. We also saw how the original size of the KEM public key *pk* and ciphertext C_K impacts the communication overhead using Kyber. Additionally, we also provided our clarification regarding the communication costs for KEM-AE-PAKE using Kyber-768 in [51] as they seem different than what we have observed in our evaluation. Then, in subsection 5.4.3, we saw that the client authentication step in TK-PAKE only leads to a slight performance overhead as opposed to without client authentication. Regarding the communication cost, there is only a 32-byte increase on the client-side, whereas the server's communication cost remains intact. Additionally, in subsection 5.4.4, we observed that the instantiation of *Modified* OCAKE-PAKE using Kyber outperforms the instantiation using Frodo-KEM. In our opinion, considering everything, the *Modified* OCAKE-PAKE *using Kyber* is the best choice amongst all three PAKE protocols discussed here.

Finally, we discussed about the implementation security of the framework itself in section 5.5. We mentioned how the implementation fulfills the framework's requirements, and that we do not use any *unsafe* blocks in Rust, only use open-source *pure* Rust crates, and how we try to ensure that sensitive data like a client's password, seeds, etc., are removed from the memory upon completion of the PAKE protocol execution.

Chapter 6

Conclusion

In this work, we have explored and discussed about three recent propositions of quantum-resistant Password Authenticated Key-Exchange (PAKE) protocols, namely, TK-PAKE [49], *Modified* OCAKE-PAKE [50], and KEM-AE-PAKE [51]. We have provided a qualitative evaluation of these PAKE protocols, definitively stated our opinion regarding the fulfilment of certain security requirements, and most importantly, we have an implementation of these quantum-resistant PAKE protocols in Rust, which have undergone performance evaluation.

Now, we certainly have implementations of quantum-resistant PAKE protocols which can be independently verified and can be integrated in other systems for secure generation of shared session keys. The framework allows us to configure the server with our choice of IP, port, login window, login threshold, timeout, and storage backend. A user of the framework will also be able to easily write a server and a client binary, which can be executed on two different machines while they communicate using TCP. We observed in chapter 5 that, when using Kyber, *Modified* OCAKE-PAKE demonstrated the best performance, whereas KEM-AE-PAKE was the least efficient inherently because of its design. Although TK-PAKE without client authentication incurs a slightly lesser overhead than its variant with client authentication, in general, the performance and (roughly the communication overhead) of TK-PAKE with and without client authentication lie in between *Modified* OCAKE-PAKE and KEM-AE-PAKE. Additionally, implementing *Modified* OCAKE-PAKE with Frodo-KEM highlighted how the choice of KEM significantly affects both computation and communication overhead of the same PAKE protocol. Therefore, in latency-sensitive scenarios, using Kyber-based PAKE protocols is preferable over Frodo-KEM-based ones. In a nutshell, *Modified* OCAKE-PAKE using Kyber is indeed the best choice amongst all the PAKE protocols discussed here.

This work not only provides a qualitative and quantitative evaluation of recent quantum-resistant PAKE protocols but also delivers a practical, modular, and easy-to-use Rust framework. It can facilitate both developers and researchers to independently adopt and evaluate the PAKE protocols. As the cryptographic community prepares for a post-quantum future, having reliable, efficient, and verifiable quantum-resistant implementations is paramount. Through this work, we have aimed to contribute to this transition by bridging the gap between theoretical propositions and real-world implementations, offering a foundation for secure password-based key exchange in the upcoming era of quantum computing.

Future Work: It would be worthwhile exploring KEMs other than Kyber as a drop-in replacement to instantiate TK-PAKE and KEM-AE-PAKE. Additionally, implementing MLWE-PAKE [52], [63] could help assess the impact of different MLWE parameter sets on the performance and communication overhead between PAKE protocols using Kyber (which uses MLWE underneath) and the MLWE-PAKE itself.

Appendix A

Clarification Regarding the Communication Overhead During Base Encoding

We mentioned the direct encryption problem of KEM public-key pk and ciphertext C_K in subsection 4.2.3 when generated using Kyber [23]. And, we described a solution obtained from [56] to mitigate this. We also mentioned that, at times, it may seem, sending the base-encoded value to the other party incurs more communication overhead as opposed to sending the non-base-encoded/original value. To that end, we try to provide a clarification as to why this happens, but only for one variant of Kyber, i.e., Kyber-768, and the same analogy is applicable for the other two variants of Kyber.

A.1 For the KEM Ciphertext

The seemingly increased communication overhead is mostly applicable in the case of sending a base-encoded KEM ciphertext ($C_{K\mathbf{be}}$). From Table 2.6 in our work and the d_u and d_v from *Table 2* in [23], we know the following about Kyber-768:

1. $n = 256$
2. $q = 3329$
3. $d = 3$
4. $d_u = 10$
5. $d_v = 4$

Each polynomial coefficient in Kyber can be originally represented using exactly 12 bits because all coefficients are modulo q which is $< 2^{12} = 4,096$ bits. As programming languages do not support native 12-bit integers, they are generally represented using 16 bits in implementation. Based on Algorithm 3, it is evident that, at the lowest level, C_K is composed of a *PolynomialVector* $\mathbf{u}$ and a *Polynomial* $\boldsymbol{v}$. Since, $d = 3$, $\mathbf{u}$ will have 3 polynomials, so altogether, C_K is composed of 4 polynomials (including $\boldsymbol{v}$). Then, $\mathbf{u}$ and $\boldsymbol{v}$ are compressed and byte-encoded based on d_u and d_v, and the respective values are concatenated and returned. Considering the values are not compressed and not byte-encoded, then the original size of $C_K \coloneqq (n \times 12 \times 4)/8 = (256 \times 12 \times 4)/8 = 1,536$ bytes, (where 12 is the number of bits of each polynomial coefficient and 4 is the total number of polynomials in C_K). Now, considering the operation $c_1 \leftarrow \mathit{ByteEncode}_{d_u}(\mathit{Compress}_{d_u}(\mathbf{u}))$, the size of compressed and byte-encoded $\mathbf{u}$ represented as $c_1 = (n \times d_u \times 3)/8 = (256 \times 10 \times 3)/8 = 960$ bytes (where 3 is the number of polynomials in $\mathbf{u}$). Similarly, considering this other operation $c_2 \leftarrow \mathit{ByteEncode}_{d_v}(\mathit{Compress}_{d_v}(\boldsymbol{v}))$, the size of compressed and byte-encoded $\boldsymbol{v}$ represented as $c_2 = (n \times d_v \times 1)/8 = (256 \times 4)/8 = 128$ bytes. So, the total size of $C_K = 960 + 128 = 1,088$ bytes (upon $c_1 \| c_2$), which is consistent with the value in our Table 2.7 for Kyber-768. And, based on our (somewhat limited) understanding, the compression performed here is indeed a lossy compression, which is likely reconcilable during the decapsulation step [23].

It is observed in *get_base_encoded_kem_ciphertext*(C_K) in Algorithm 3 that, during the base-encoding, we $\mathit{ByteDecode}_{d_u}$ and $\mathit{Decompress}_{d_u}$ C_K at first and then perform the *Forward base-conversion* on the raw polynomials in their original size (1,536 bytes). Here, we consider a *Polynomial* with its 256 coefficients as one single large number in base 3,329, where each individual coefficient represents a single digit in base 3,329 of this large number. The forward conversion during base-encoding converts this single large number from base 3,329 to base 256. This means that the base-encoded polynomial will be in base 256. So, the number of digits in each base-encoded polynomial will be $n \times \log_n(q) = 256 \times \log_{256}(3,329) = 256 \times 1.4626 \approx 374.4$, which is ≈ 375 [96]. And, since our target base during forward conversion is 256, each base-256 digit in this base-encoded polynomial is of exactly 1 byte. So, the total size of each base-encoded polynomial will be 375 bytes.

We know from above that the original size of C_K *without* any compression and byte-encoding is 1,536 bytes. Now, after performing the base-encoding, we have altogether 4 base-256 encoded polynomials, i.e., 3 from the polynomial-vector $\mathbf{u}$ and the polynomial $\boldsymbol{v}$. So, the total size of this base-256 encoded $C_{K\mathbf{be}} = 375 \times 4 = 1,500$ bytes, which is a bit more than the size of (extremely) compressed and byte-encoded original C_K, i.e., 1,088 bytes. In our understanding, this is because, in the original compressed and byte-encoded C_K, the coefficients are represented as 10 bits each for $\mathbf{u}$ and 4 bits each for $\boldsymbol{v}$, whereas after base-encoding, each base-256 digit in the base-encoded polynomial requires exactly 1 byte or (8 bits). So, altogether for 4 base-encoded polynomials in $C_{K\mathbf{be}}$ each having 375, 8-bit values/digits tends to be more than the original (compressed and byte encoded) C_K.

A.2 For the KEM Public Key

We saw above how the base-encoding seems to increase the communication overhead as opposed to sending the non-base-encoded C_K. But generally, this does not seem to be the case when base-encoding the KEM public key *pk*. As per Algorithm 2, we know that *pk* is composed of a byte-encoded *PolynomialVector* $\mathbf{t}$ and a 32-byte seed ρ. Keeping the seed ρ aside, we see the operation $\mathbf{t} \leftarrow \mathit{ByteEncode}_{\mathbf{d}}(\mathbf{b})$ which only performs byte-encoding of $\mathbf{b}$ with $\mathbf{d} = 12$ and generates $\mathbf{t}$. This transformation just serializes the polynomial coefficients (in $\mathbf{b}$) into a compact, bit-packed form. The original size of $pk := (n \times 12 \times 3)/8 = (256 \times 12 \times 3)/8 = 1,152 + 32 = 1,184$ bytes (where 12 is the number of bits of each polynomial coefficient and 3 is the total number of polynomials in $\mathbf{b}$) which is consistent with the value in our Table 2.7 for Kyber-768.

As there is no compression involved, the base-encoding is actually performed on the polynomials in their original size (1,184 bytes). We know that there are $d = 3$ polynomials in $\mathbf{b}$ upon which the *Forward base-conversion* is going to be performed. As the base-conversion process is exactly the same (in Algorithm 2), each base-encoded polynomial will be of size 375 bytes. Considering 3 such base-encoded polynomials and the 32-byte seed ρ, the total size of $pk_{\mathbf{be}} = (375 \times 3) + 32 = 1,125 + 32 = 1,157$ bytes. So, the size of this $pk_{\mathbf{be}}$ is only a little bit less than the original *pk* of 1,184 bytes. In our understanding, this is because, in the original *pk*, each polynomial is of size $(n \times 12)/8 = (256 \times 12)/8 = 384$ bytes, whereas, in $pk_{\mathbf{be}}$, each base-encoded polynomial is of 375 bytes, so for each polynomial we save 9 bytes and altogether 1,184 - 1,157 = 27 bytes in $pk_{\mathbf{be}}$ (considering the 32-byte seed ρ). Additionally, in the original byte-encoded *pk*, each coefficient (in base 3,329) is stored using 12 bits, even though $\log_2(3,329) \approx 11.7$, resulting in a ≈ 0.3-bit wastage per coefficient. In contrast, base encoding only introduces at most 7 bits of wastage per polynomial in the MSB due to rounding $374.4 \approx 375$ to full bytes, leading to a total wastage of at most 21 bits across all the polynomials.

Bibliography

[1] X. Gao, J. Ding, J. Liu, and L. Li, “Post-quantum secure remote password protocol from rlwe problem,” in *Information Security and Cryptology: 13th International Conference, Inscrypt 2017, Xi’an, China, November 3–5, 2017, Revised Selected Papers 13*, Springer, 2018, pp. 99–116.

[2] J. Ding, S. Alsayigh, J. Lancrenon, S. Rv, and M. Snook, “Provably secure password authenticated key exchange based on rlwe for the post-quantum world,” in *Cryptographers’ Track at the RSA conference*, Springer, 2017, pp. 183–204.

[3] S. M. Bellovin and M. Merritt, “Encrypted key exchange: Password-based protocols secure against dictionary attacks,” 1992.

[4] T. D. Wu *et al.*, “The secure remote password protocol.,” in *NDSS*, Citeseer, vol. 98, 1998, p. 111.

[5] P. W. Shor, “Algorithms for quantum computation: Discrete logarithms and factoring,” in *Proceedings 35th annual symposium on foundations of computer science*, Ieee, 1994, pp. 124–134.

[6] P. SHOR, “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer,” *SIAM journal on computing (Print)*, vol. 26, no. 5, pp. 1484–1509, 1997.

[7] T. Laing and T. Charles, *Anticipating the Quantum Threat to Cryptography*, Feb. 2024. [Online]. Available: `https://threatresearch.ext.hp.com/anticipating-the-quantum-threat-to-cryptography/` (visited on 09/03/2025).

[8] National Institute of Standards and Technology, *NIST Releases First 3 Finalized Post-Quantum Encryption Standards*, Feb. 2025. [Online]. Available: `https://www.nist.gov/news-events/news/2024/08/nist-releases-first-3-finalized-post-quantum-encryption-standards` (visited on 09/03/2025).

[9] W. Diffie and M. E. Hellman, “New directions in cryptography,” *IEEE Transactions on Information Theory*, vol. 22, no. 6, 1976.

[10] M. Rossberg and M. Theil, “Secure enrollment of certificates using short pins,” in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, 2017, pp. 1–9.

[11] M. Green, *Let’s talk about PAKE*, Oct. 2018. [Online]. Available: `https://blog.cryptographyengineering.com/2018/10/19/lets-talk-about-pake/` (visited on 09/03/2025).

[12] S. Jarecki, H. Krawczyk, and J. Xu, “Opaque: An asymmetric pake protocol secure against pre-computation attacks,” in *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part III 37*, Springer, 2018, pp. 456–486.

[13] F. Hao and P. C. van Oorschot, “Sok: Password-authenticated key exchange–theory, practice, standardization and real-world lessons,” in *Proceedings of the 2022 ACM on Asia conference on computer and communications security*, 2022, pp. 697–711.

[14] Z. Li and D. Wang, "Achieving one-round password-based authenticated key exchange over lattices," *IEEE transactions on services computing*, vol. 15, no. 1, pp. 308–321, 2019.

[15] N. Alnahawi, D. Haas, E. Mauß, and A. Wiesmaier, "Sok: Pqc pakes-cryptographic primitives, design and security," *Cryptology ePrint Archive*, 2025.

[16] M. J. Dworkin, "Sha-3 standard: Permutation-based hash and extendable-output functions," 2015.

[17] C. E. Shannon, "Communication theory of secrecy systems," *The Bell System Technical Journal*, vol. 28, no. 4, pp. 656–715, 1949. DOI: `10.1002/j.1538-7305.1949.tb00928.x`.

[18] J.-S. Coron, J. Patarin, and Y. Seurin, "The random oracle model and the ideal cipher model are equivalent," in *Advances in Cryptology–CRYPTO 2008: 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings 28*, Springer, 2008, pp. 1–20.

[19] H. Lipmaa, P. Rogaway, and D. Wagner, "Ctr-mode encryption," in *First NIST Workshop on Modes of Operation*, Citeseer. MD, vol. 39, 2000.

[20] M. Dworkin, "Recommendation for block cipher modes of operation: Methods and techniques," *NIST Special Publication*, vol. 800, 38A, 2001.

[21] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer, 2000, pp. 531–545.

[22] E. Alkim, J. W. Bos, L. Ducas, *et al.*, *FrodoKEM Learning With Errors Key Encapsulation*, Jun. 2021. [Online]. Available: `https://frodokem.org/files/FrodoKEM-specification-20210604.pdf` (visited on 09/03/2025).

[23] F. I. P. S. Publication 203, *Module-Lattice-Based Key-Encapsulation Mechanism Standard*, Aug. 2024. [Online]. Available: `https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.203.pdf` (visited on 09/03/2025).

[24] National Institute of Standards and Technology, *Post-Quantum Cryptography Security (Evaluation Criteria)*, Apr. 2025. [Online]. Available: `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/evaluation-criteria/security-(evaluation-criteria)` (visited on 09/03/2025).

[25] D. Moody, *Let's Get Ready to Rumble. The NIST PQC "Competition"*, Apr. 2018. [Online]. Available: `https://csrc.nist.gov/CSRC/media/Presentations/Let-s-Get-Ready-to-Rumble-The-NIST-PQC-Competiti/images-media/PQCrypto-April2018_Moody.pdf` (visited on 09/03/2025).

[26] F. I. P. S. Publication 180-4, *Secure Hash Standard (SHS)*, Aug. 2015. [Online]. Available: `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf` (visited on 09/03/2025).

[27] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Sponge functions," in *ECRYPT hash workshop*, vol. 2007, 2007.

[28] T. Lange, *Sponge functions*, Sep. 2021. [Online]. Available: `https://hyperelliptic.org/tanja/teaching/crypto21/hash-5.pdf` (visited on 09/03/2025).

[29] F. I. P. S. Publication 202, *SHA-3 Standard: Permutation-Based Hash and Extend -able-Output Functions*, Aug. 2015. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf (visited on 09/03/2025).

[30] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon v1.2: Lightweight authenticated encryption and hashing," *J. Cryptol.*, vol. 34, no. 3, p. 33, 2021. DOI: 10.1007/S00145-021-09398-9. [Online]. Available: https://doi.org/10.1007/s00145-021-09398-9.

[31] B. Guido, D. Joan, P. Michaël, and V. Gilles, "Cryptographic sponge functions," *2011-STMicroelectronics NXP Semiconductors, Version 0.1 January 14*, 2011.

[32] Matthew Green, *The Ideal Cipher Model (wonky)*, Apr. 2013. [Online]. Available: https://blog.cryptographyengineering.com/2013/04/11/wonkery-mailbag-ideal-ciphers/ (visited on 09/03/2025).

[33] N. Alnahawi, J. Alperin-Sheriff, D. Apon, and A. Wiesmaier, "Nice-pake: On the security of kem-based pake constructions without ideal ciphers," *Cryptology ePrint Archive*, 2024.

[34] A. Arriaga, M. Barbosa, S. Jarecki, and M. Skrobot, *C'est très CHIC: A compact password-authenticated key exchange from lattice-based KEM*, Cryptology ePrint Archive, Paper 2024/308, 2024. [Online]. Available: https://eprint.iacr.org/2024/308.

[35] V. Rijmen and J. Daemen, "Advanced encryption standard," *Proceedings of federal information processing standards publications, national institute of standards and technology*, vol. 19, p. 22, 2001.

[36] F. I. P. S. Publication 197, *Advanced Encryption Standard (AES)*, May 2023. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf (visited on 09/03/2025).

[37] S. Wang, *The difference in five modes in the AES encryption algorithm*, Aug. 2019. [Online]. Available: https://www.highgo.ca/2019/08/08/the-difference-in-five-modes-in-the-aes-encryption-algorithm/ (visited on 09/03/2025).

[38] Matthew Green, *How to choose an Authenticated Encryption mode*, May 2012. [Online]. Available: https://blog.cryptographyengineering.com/2012/05/19/how-to-choose-authenticated-encryption/ (visited on 09/03/2025).

[39] A. Corbellini, *Authenticated encryption: Why you need it and how it works*, Mar. 2023. [Online]. Available: https://andrea.corbellini.name/2023/03/09/authenticated-encryption/#block-ciphers (visited on 09/03/2025).

[40] A. Malek, *Authenticated Encryption*, Apr. 2022. [Online]. Available: https://superkogito.github.io/blog/2019/05/01/authenticated_encryption.html#id1 (visited on 09/03/2025).

[41] S. Vaudenay, "Security flaws induced by cbc padding—applications to ssl, ipsec, wtls...," in *International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2002, pp. 534–545.

[42] D. McGrew and J. Viega, "The galois/counter mode of operation (gcm)," *submission to NIST Modes of Operation Process*, vol. 20, pp. 0278–0070, 2004.

[43] N. S. P. 800-38D, *Recommendation for block cipher modes of operation: Galois/-counter mode (gcm) and gmac*, Nov. 2007. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf (visited on 09/03/2025).

[44] L. K. Grover, “A fast quantum mechanical algorithm for database search,” in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 212–219.

[45] M. Ivezic, *Grover’s Algorithm and Its Impact on Cybersecurity*, Aug. 2017. [Online]. Available: `https://postquantum.com/post-quantum/grovers-algorithm/?` (visited on 09/03/2025).

[46] G. Brassard, P. Hoyer, and A. Tapp, “Quantum algorithm for the collision problem,” *arXiv preprint quant-ph/9705002*, 1997.

[47] O. Regev, “On lattices, learning with errors, random linear codes, and cryptography,” in *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, 2005, pp. 84–93. DOI: `10.1145/1060590.1060603`. [Online]. Available: `https://doi.org/10.1145/1060590.1060603`.

[48] A. Langlois and D. Stehlé, “Worst-case to average-case reductions for module lattices,” *Designs, Codes and Cryptography*, vol. 75, no. 3, pp. 565–599, 2015.

[49] J. Pan and R. Zeng, “A generic construction of tightly secure password-based authenticated key exchange,” in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer, 2023, pp. 143–175.

[50] N. Alnahawi, K. Hövelmanns, A. Hülsing, and S. Ritsch, “Towards post-quantum secure pake-a tight security proof for ocake in the bpr model,” in *International Conference on Cryptology and Network Security*, Springer, 2024, pp. 191–212.

[51] Y. Lyu, S. Liu, and S. Han, “Efficient asymmetric pake compiler from kem and ae,” in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer, 2025, pp. 34–65.

[52] P. Ren and X. Gu, “Practical post-quantum password-authenticated key exchange based-on module-lattice,” in *International Conference on Information Security and Cryptology*, Springer, 2021, pp. 137–156.

[53] R. Pierce, *"Polynomials" Math Is Fun*, Apr. 2024. [Online]. Available: `https://www.mathsisfun.com/algebra/polynomials.html` (visited on 09/03/2025).

[54] W. authors, *Quotient ring*, Jan. 2025. [Online]. Available: `https://en.wikipedia.org/wiki/Quotient_ring` (visited on 09/03/2025).

[55] J. Bos, L. Ducas, E. Kiltz, *et al.*, “Crystals - kyber: A cca-secure module-lattice-based kem,” in *2018 IEEE European Symposium on Security and Privacy (EuroSP)*, 2018, pp. 353–367. DOI: `10.1109/EuroSP.2018.00032`.

[56] H. Beguinet, C. Chevalier, D. Pointcheval, T. Ricosset, and M. Rossi, “Get a cake: Generic transformations from key encaspulation mechanisms to password authenticated key exchanges,” in *International Conference on Applied Cryptography and Network Security*, Springer, 2023, pp. 516–538.

[57] J. Becerra, D. Ostrev, and M. Škrobot, “Forward secrecy of spake2,” in *Provable Security: 12th International Conference, ProvSec 2018, Jeju, South Korea, October 25-28, 2018, Proceedings 12*, Springer, 2018, pp. 366–384.

[58] J. Hesse and M. Rosenberg, “Pake combiners and efficient post-quantum instantiations,” *Cryptology ePrint Archive*, 2024.

[59] E. Bresson, O. Chevassut, and D. Pointcheval, “Security proofs for an efficient password-based key exchange,” in *Proceedings of the 10th ACM conference on Computer and communications security*, 2003, pp. 241–250.

[60] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," *Journal of Mathematical Cryptology*, vol. 9, no. 3, pp. 169–203, 2015.

[61] B. Haase and B. Labrique, "AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 1–48, 2019.

[62] X. Gao, J. Ding, L. Li, R. Saraswathy, and J. Liu, "Efficient implementation of password-based authenticated key exchange from rlwe and post-quantum tls," *Cryptology ePrint Archive*, 2017.

[63] P. Ren, X. Gu, and Z. Wang, "Efficient module learning with errors-based post-quantum password-authenticated key exchange," *IET Information Security*, vol. 17, no. 1, pp. 3–17, 2023.

[64] L. Suifeng, *Mlwe-pake*, Apr. 2022. [Online]. Available: `https://github.com/lala-suifeng/mlwe-pake` (visited on 09/03/2025).

[65] B. F. D. Santos, Y. Gu, and S. Jarecki, "Randomized half-ideal cipher on groups with applications to uc (a) pake," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2023, pp. 128–156.

[66] J. Erolin, *Rust Vs C++ Performance*, Jun. 2021. [Online]. Available: `https://www.bairesdev.com/blog/when-speed-matters-comparing-rust-and-c/` (visited on 09/03/2025).

[67] A. Yadav, *Rust vs. c++ performance: Analyzing safe and unsafe implementations in system programming*, Feb. 2025. [Online]. Available: `https://www.researchgate.net/publication/389282759_Rust_vs_C_Performance_Analyzing_Safe_and_Unsafe_Implementations_in_System_Programming` (visited on 09/03/2025).

[68] A. Yadav, *C-vs-rust-*, Feb. 2025. [Online]. Available: `https://github.com/aditya0yadav/c-vs-rust-` (visited on 09/03/2025).

[69] M. Chojnowska, *Asynchronous vs. synchronous programming*, Sep. 2023. [Online]. Available: `https://sunscrapers.com/blog/programming-async-vs-sync-best-approach/` (visited on 09/03/2025).

[70] D. Samba, *Tcp vs udp protocols*, Jan. 2024. [Online]. Available: `https://www.digitalsamba.com/blog/tcp-and-udp-protocols` (visited on 09/03/2025).

[71] T. Wiggers, *pqcrypto-mlkem v0.1.0*, Mar. 2025. [Online]. Available: `https://crates.io/crates/pqcrypto-mlkem/0.1.0` (visited on 09/03/2025).

[72] T. Wiggers and D. Stebila, *PQClean*, Apr. 2025. [Online]. Available: `https://github.com/PQClean/PQClean/` (visited on 09/03/2025).

[73] M. Berry, *pqc_kyber v0.7.1*, Aug. 2023. [Online]. Available: `https://crates.io/crates/pqc_kyber/0.7.1` (visited on 09/03/2025).

[74] T. Ariceri, *ml-kem v0.2.1*, Apr. 2025. [Online]. Available: `https://crates.io/crates/ml-kem/0.2.1` (visited on 09/03/2025).

[75] M. Lodder, *frodo-kem-rs v0.5.0*, Feb. 2025. [Online]. Available: `https://crates.io/crates/frodo-kem-rs/0.5.0` (visited on 09/03/2025).

[76] J. Halliday, *convert-base v1.1.2*, Oct. 2021. [Online]. Available: `https://crates.io/crates/convert-base/1.1.2` (visited on 09/03/2025).

[77] C. Lerche, *Announcing tokio 1.0*, Dec. 2020. [Online]. Available: `https://tokio.rs/blog/2020-12-tokio-1-0` (visited on 09/03/2025).

[78] C. Lerche and A. Ryhl, *tokio v1.45.0*, May 2025. [Online]. Available: `https://crates.io/crates/tokio/1.45.0` (visited on 09/03/2025).

[79] A. Pavlov, *aes v0.8.4*, Jun. 2025. [Online]. Available: `https://crates.io/crates/aes/0.8.4` (visited on 09/03/2025).

[80] A. Pavlov, *ctr v0.9.2*, Jun. 2025. [Online]. Available: `https://crates.io/crates/ctr/0.9.2` (visited on 09/03/2025).

[81] T. Ariceri, *aes-gcm v0.10.3*, Jun. 2025. [Online]. Available: `https://crates.io/crates/aes-gcm/0.10.3` (visited on 09/03/2025).

[82] T. Ariceri and A. Pavlov, *sha2 v0.10.9*, Jun. 2025. [Online]. Available: `https://crates.io/crates/sha2/0.10.9` (visited on 09/03/2025).

[83] A. Pavlov, *sha3 v0.10.8*, Jun. 2025. [Online]. Available: `https://crates.io/crates/sha3/0.10.8` (visited on 09/03/2025).

[84] D. Tolnay, *serde v1.0.219*, Jun. 2025. [Online]. Available: `https://crates.io/crates/serde/1.0.219` (visited on 09/03/2025).

[85] D. Tolnay, *serde_json v1.0.140*, Jun. 2025. [Online]. Available: `https://crates.io/crates/serde_json/1.0.140` (visited on 09/03/2025).

[86] D. Hardy, *rand v0.8.5*, May 2025. [Online]. Available: `https://crates.io/crates/rand/0.8.5` (visited on 09/03/2025).

[87] D. Hardy, *rand_core v0.9.3*, May 2025. [Online]. Available: `https://crates.io/crates/rand_core/0.9.3` (visited on 09/03/2025).

[88] D. Tolnay, *anyhow v1.0.98*, Jul. 2025. [Online]. Available: `https://crates.io/crates/anyhow/1.0.98` (visited on 09/03/2025).

[89] P. Glotfelty, *strum v0.27.1*, Feb. 2025. [Online]. Available: `https://crates.io/crates/strum/0.27.1` (visited on 09/03/2025).

[90] P. Glotfelty, *strum_macros v0.27.1*, Feb. 2025. [Online]. Available: `https://crates.io/crates/strum_macros/0.27.1` (visited on 09/03/2025).

[91] T. Ariceri, *zeroize v1.8.1*, Apr. 2025. [Online]. Available: `https://crates.io/crates/zeroize/1.8.1` (visited on 09/03/2025).

[92] D. Tolnay, *async-trait v0.1.88*, Jul. 2025. [Online]. Available: `https://crates.io/crates/async-trait/0.1.88` (visited on 09/03/2025).

[93] K. Lesiński, *Lib.rs*, Jul. 2025. [Online]. Available: `https://lib.rs/` (visited on 09/03/2025).

[94] X. Defago, *stats-ci v0.0.7*, Dec. 2024. [Online]. Available: `https://crates.io/crates/stats-ci/0.0.7` (visited on 09/03/2025).

[95] J. McNamara, *rust_xlsxwriter v0.89.1*, Jul. 2025. [Online]. Available: `https://crates.io/crates/rust_xlsxwriter/0.89.1` (visited on 09/03/2025).

[96] GeeksforGeeks, *Given a number n in decimal base, find number of its digits in any base (base b)*, Feb. 2023. [Online]. Available: `https://www.geeksforgeeks.org/dsa/given-number-n-decimal-base-find-number-digits-base-base-b/` (visited on 09/03/2025).

List of Abbreviations

AES	Advanced Encryption Standard
NIST	National Institute of Standards and Technology
PKI	Public Key Infrastructure
PAKE	Password Authenticated Key-Exchange
DLP	Discrete Logarithm Problem
ECDLP	Elliptic Curve Variant of Discrete Logarithm Problem (DLP)
EKE	Encrypted Key-Exchange
OEKE	One-Way Encrypted Key-Exchange (EKE)
MITM	Man-in-the-Middle
SRP	Secure Remote Password
KEM	Key-Encapsulation Mechanism
IC	Ideal Cipher
RO	Random Oracle
UC	Universal Composability
BPR	Bellare-Pointcheval-Rogaway
NICE	No IC Encryption
LWE	Learning With Errors
MLWE	Module Learning With Errors
RLWE	Ring Learning With Errors
PRS	Password Related String
AE	Authenticated Encryption
TLS	Transport Layer Security
TK	Tight Kyber
HIC	Half-IC
CHIC	Compact Half-IC (HIC)
M2F	Modified 2-Round Feistel
SSID	Session Identifier
SHA	Secure Hashing Algorithm
XOF	Extendable Output Function
MAC	Message Authentication Code
GCM	Galois/Counter Mode
PSK	PreShared Key
ML-KEM	MLWE-Based KEM
TCB	Trusted Computing Base
DoS	Denial of Service

Versicherung über selbstständige Anfertigung

Hiermit versichere ich diese Masterarbeit eigenhändig und selbstständig verfasst, keine anderen als die angegebenen Hilfsmittel und Quellen verwendet und direkt oder indirekt übernommene Gedanken als solche gekennzeichnet zu haben. Diese Arbeit wurde bisher in dieser oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt oder anderweitig veröffentlicht.

Ilmenau, 10. September 2025

Prateek Banerjee