Design and Implementation of a Post-Quantum Group Authenticated Key Exchange Protocol With the LibOQS Library: A Comparative Performance Analysis From Classic McEliece, Kyber, NTRU, and Saber

José Ignacio Escribano Pablos, Misael Enrique Marrigga, and Ángel L. Pérez del Pozo

ABSTRACT
Group authenticated key exchange protocols (GAKE) are cryptographic tools enabling a group of several users communicating through an insecure channel to securely establish a common shared high-entropy key. In the last years, the need to design cryptographic tools which provide security in the presence of attackers with access to quantum resources has become unquestionable; the field dealing with these types of protocols is usually referred to as Post-Quantum Cryptography. The U.S. National Institute for Standards and Technology (NIST) launched in 2017 an open call to find suitable post-quantum public-key algorithms for standardization. In this work, we design a GAKE that can be instantiated with any key encapsulation mechanism (KEM) that satisfies the strong security notion IND-CCA, matching NIST’s requirements for this primitive. We have implemented our GAKE with the four finalist KEMs from the NIST process: Classic McEliece, Kyber, NTRU, and Saber, making use of the open-source library LibOQS where these algorithms are provided. We have conducted a detailed comparative performance analysis of the resulting GAKE protocols, taking into account all the parameter sets proposed in the submissions. We have also made a performance analysis of all the involved building pieces, including the four finalist KEMs. Finally, we also compare our GAKE with a previous proposal implemented with Kyber.

INDEX TERMS
Cryptography, cryptographic protocols, system implementation, post-quantum cryptography, public key cryptography.

I. INTRODUCTION
Group authenticated key exchange (GAKE) protocols are cryptographic constructions that allow a group of $n \geq 2$ users or parties, communicating through an insecure network, to agree on common session keys. These keys are then typically used to provide security guarantees, such as confidentiality, integrity, and/or authentication, for further communication among the group members.

In the last years, we have seen a growing concern about the threat that quantum computation presents to the security of many existing cryptographic primitives based on mathematical problems related to integer factorization or computation of discrete logarithms. This led the U.S. National Institute of Standards and Technology (NIST) to launch an open call in 2017 asking for proposals of post-quantum algorithms
that could be subsequently standardized. The term “post-quantum” in this context refers to algorithms that could be considered to offer security against attackers that have access to quantum computational resources. In the NIST call for proposals, two types of cryptographic primitives were allowed: Key Encapsulations Mechanisms (KEMs) and digital signatures. At the beginning of 2022, there were three rounds of announcements from NIST stating which candidates advanced in the process. After round 3, there were 7 finalists: 4 KEMs and 3 signature schemes. NIST also proposed a list of alternate candidates for further study and future consideration; it was composed of 5 KEMs and 4 signature schemes. During the revision process of this paper, NIST announced the algorithms selected for standardization, namely Kyber as the KEM and Dilithium, Falcon and SPHINC+ as digital signatures. In this work, we focus on the four finalist KEMs, as they are known to be a basic building block from which GAKE protocols can be constructed.

A. RELATED WORK

There have been several recent proposals of group key exchange protocols that provide some kind of resistance against quantum attacks (see Table 1 for a comparison between the main GAKE protocols). The protocol presented in [1] by Fujioka et al. is based on the problem of finding isogeny mappings between two supersingular elliptic curves with the same number of points. In the same line, Hougaard and Miyaji presented in [2] several designs based on isogenies. The authenticated protocols are named A-SIT and A-P2P-SIT, with the latter being the peer-to-peer version of A-SIT, which means that it reduces the protocol complexity in terms of communication and memory. Both are authenticated protocols, resistant to active attacks, and achieve authentication through a signature scheme. Apon et al. ([3]) constructed an unauthenticated protocol proven secure under the ring learning with errors (RLWE) assumption. This scheme may be transformed into an authenticated one by using the Katz and Yung compiler ([12]), that adds a signature scheme and an additional round to the original protocol. The protocols from Choi et al. ([4], [5]) are also based in the same problem; the authors build on [12] and propose three different protocols: the first is unauthenticated, the second (STAG) adds authentication, and the third is, in addition, dynamic (meaning that users may join or leave the group at any time). Choi et al. [6] proposed a generic GAKE also relying on the RLWE assumption, built on a tree structure in the dynamic setting. In more detail, this protocol has been instantiated with NewHope [13]: a KEM submitted to the NIST standardization process, but which has not been selected as a finalist in Round 3. Takashima constructed in [7] two different families of GAKEs based on static lattice and isogeny assumptions respectively, where static means that the size of the computational problem does not depend on the number of participants in the group.

There also exist protocols, like the one we propose in this work, that use compilers, which produce a quantum-resistant GAKE from simpler post-quantum primitives. In this line, the protocol from Persichetti et al. ([8]) was constructed from a KEM and a signature scheme. González Vasco et al. ([9]) introduced a protocol derived from a KEM and a Message Authentication Code (MAC). However, this construction cannot be considered completely post-quantum; security holds in the future-quantum scenario, where adversaries do not have access to quantum resources during the protocol execution but only later. Escobedo et al. ([11]) used the compiler from Abdalla et al. ([14]) to obtain a GAKE from the IND-CPA Public Key Encryption (PKE) scheme included in the Kyber suite ([15]) and the $\text{FO}_{\text{AKE}}$ transformation, and proved it to be secure in the Quantum Random Oracle Model (QROM). The compiler introduced in [10] allows to obtain a GAKER protocol using any two-party key exchange, being a generalization of the Burmester and Desmedt [16] protocol in the G-CK+ security model. Two versions of the compiler have been proposed: the original version known as GKE-C and the peer-to-peer version (P2P-GKE-C). The latter reduces the resources consumption (memory and communication) compared to the original compiler.

B. OUR CONTRIBUTION

In this work, we propose a generic post-quantum GAKER protocol in the same line of [11]. We rely only on three primitives: an IND-CCA secure KEM, a one-time symmetric encryption scheme, and a cryptographic hash function. The $\text{FSXY}$ transformation by Fujioka et al. ([17]) provides a two-party authenticated key exchange $\text{2AKE}$ from the KEM. Then we use Abdalla et al.’s compiler ([14]) to obtain the GAKER from the $\text{2AKE}$. The compiler also requires a commitment scheme satisfying certain properties, but we show that it can be obtained from the same KEM used for the $\text{FSXY}$ transformation. Whereas this construction may be seen as a generalization of [11] (which also builds on Abdalla et al.’s compiler and a generic transformation from KEM to AKE), this is not exactly so, as here the $\text{FSXY}$ transformation is used instead of the $\text{FO}_{\text{AKE}}$ (see [18] and [19]), which is used in [11].

Our aim is to design a generic protocol that may be implemented with any of the four Round 3 KEM finalists from the NIST post-quantum standardization process, namely Classic McEliece, Kyber, NTRU, and Saber. In fact, our design can be implemented with any IND-CCA KEM (yet its final security against quantum adversaries is of course not guaranteed if the KEM is not post-quantum). We would like to point out that the $\text{FO}_{\text{AKE}}$ transformation offers some advantages over $\text{FSXY}$, such as having a security proof in the Quantum Random Oracle Model (QROM) and being simpler. However, it cannot be directly applied to the public key encryption schemes described in the Classic McEliece and NTRU submissions, as they are deterministic and cannot satisfy the IND-CPA requirement from [18] and [19]. Therefore, our rationale for choosing $\text{FSXY}$ is that it is
TABLE 1. Main features of the GAKE protocols claimed to be quantum-resistant.

| Name                  | Type     | Assumption | Authenticated | Static or Dynamic? | Compiled design? | Scenario       | Users signatures | Instanciated with NIST KEM finalist? |
|-----------------------|----------|------------|---------------|--------------------|------------------|---------------|----------------|-------------------------------------|
| Fujisaki et al. [1]   | Protocol | Isogeny    | ✔️            | Static             | ✗                | Post-Quantum   | ✗             | ✗                                   |
| Hougaard and Miyaji [2]| Protocol | Isogeny    | ✔️            | Static             | ✔️               | Post-Quantum   | ✔️            | ✗                                   |
| Apon et al. [3]       | Protocol | Lattice (RLWE) | ✗            | Static             | ✗                | Post-Quantum   | ✗             | ✗                                   |
| Choi et al. [4], [5]  | Protocol | Lattice (RLWE) | ✔️            | Dynamic            | ✔️               | Post-Quantum   | ✔️            | ✗                                   |
| Choi et al. [6]       | Protocol | Lattice (RLWE) | ✗            | Dynamic            | ✔️               | Post-Quantum   | ✗             | ✗                                   |
| Takahshima [7]        | Protocol | Lattice (RLWE), Isogeny | ✔️            | Static             | ✔️               | Post-Quantum   | ✔️            | ✗                                   |
| Presicetti et al. [8] | Compiler | Inherited from KEM | ✔️            | Static             | —                | Post-Quantum   | ✔️            | ✗                                   |
| Gonzalez et al. [9]   | Compiler | Inherited from KEM | ✔️            | Static             | —                | Post-Quantum   | ✗             | ✗                                   |
| Hougaard and Miyaji [10]| Compiler | Inherited from 2AKE | ✔️            | Static             | —                | Post-Quantum   | ✗             | No                                  |
| Escrivano et al. [11] | Protocol | Lattice (Modular LWE) | ✔️            | Static             | ✔️               | Kyber          | ✗             | ✗                                   |

As every NIST finalist must include a KEM, this allows us to provide full and working implementations of our GAKE with all the finalists.

We have instantiated and implemented our GAKE protocol with the aforementioned four finalists from the NIST competition. Our implementations make use of the open-source library LibOQS and they cover the four KEMs and all the different parameter sets proposed for each one. We have conducted a performance analysis of the whole GAKE protocol and compared the different versions.

In addition, we have independently studied the performance of the different building blocks, including each of the KEMs. We consider this comparative performance analysis of the Round 3 finalists to be an interesting additional and independent contribution.

Finally, we provide performance figures comparing our GAKE implemented with Kyber to the GAKE presented in [11], which is also Kyber based but uses the \( F^{\text{NKE}} \) transformation ([18], [19]) to obtain the 2AKE. The GAKE in [11] is the only one in the previous literature, as far as we know, to have been implemented with one of the four KEM finalists from NIST competition.

C. PAPER ROADMAP

We start by providing some preliminaries in Section II, which will help the reader understand our GAKE design, subsequently depicted in Section IV. The security model we are considering is described in Section III and we provide a security proof for our protocol in Section IV. Section V describes the different implementation possibilities and gives a detailed explanation of our comparative experiments, which results are further analyzed in Section VI. We finalize with a brief summary of our conclusions in Section VII, which is followed by two appendices. Appendix A depicts a complete run of the GAKE protocol using Classic McEliece as a building block, whereas Appendix B shows some numerical results (linked to the graphics from Section VI).

II. PRELIMINARIES

A. ABDALLA ET AL.’S COMPILER: FROM 2-PARTY AKE TO GAKE

Here, we briefly describe the compiler due to Abdalla et al. ([14]), which derives a group authenticated key exchange protocol GAKE from an arbitrary 2-party key exchange protocol 2AKE. Abdalla et al.’s compiler only adds 2 additional rounds of communication to 2AKE, i.e., if 2AKE needs in \( r \) rounds to run, GAKE requires \( r + 2 \) rounds. Moreover, the compiler does not require any authentication method beyond the ones required by 2AKE. It only assumes that participants in GAKE are distributed on a ring, i.e., user \( U_i \) is aware of the identity of its left neighbour \( U_{i-1} \) and its right neighbour \( U_{i+1} \).

We denote with \( P \) the set of users that can participate in the protocol GAKE and with \( G \) the subset \( \{U_0, U_1, \ldots, U_{n-1}\} \subset \)
$\mathcal{P}$ of $n \geq 2$ users that want to agree on a session key. A user $U_i$ can run a polynomial number of (parallel) instances of GAKE. 2AKE assumes long-term authentication keys that have been established in a trusted authentication phase. It allows a pair of public/secret keys for each user $U_i$, a high entropy symmetric key, or a low entropy symmetric key, shared for each pair of users, and a common secret for all users.

The compiler depends on the following cryptographic tools: A non-interactive non-malleable commitment scheme $\mathcal{C}$ that is perfectly binding and achieves non-malleability for multiple commitments, a collision-resistant pseudorandom function family $\mathcal{F}$, and a hash function $\mathcal{H}$ selected from a family of universal hash functions.

We briefly describe the compiler (see details in [14]): in Round 1 $\sim r$, each user $U_i$ runs $2\text{AKE}$ with $U_i$ and $U_{i+1}$, obtaining two keys $\overrightarrow{K_i}$ and $\overleftarrow{K_i}$, shared with $U_{i+1}$ and $U_{i-1}$. In Round $r + 1$, each user $U_i$ computes a commitment $C_i = C(i, X_i, r_i)$, where $X_i = \overrightarrow{K_i} \oplus \overleftarrow{K_i}$ and $r_i$ is chosen at random. $U_i$ broadcast $M_i^1 = (U_i, C_i)$. Finally, in Round $r + 2$, each user $U_j$ broadcasts $M_i^2 = (U_j, X_i, r_i)$, checks that $\bigoplus_{i=0}^{n-1} X_i = 0$ and the correctness of the commitments $C_i$. If any one of last two conditions fails, then user $U_i$ ends the protocol at this point. Then, $U_i$ computes the master key $K = (K_0, K_1, \ldots, K_{n-1}, \mathcal{G})$, where $K_{j,i} = \overrightarrow{K_i} \oplus X_{i-1} \oplus \cdots \oplus X_{i-j}$, $j = 1, 2, \ldots, n-1$.

$U_i$ sets the session key $sk_i$ and the session identifier $sid_i$, derived from $\mathcal{F}$ and $\mathcal{H}$, respectively.

### B. POST-QUANTUM KEMs

We instantiate both the $2\text{AKE}$ and the commitment scheme $\mathcal{C}$ (needed for the compiler described in Section IV) from a post-quantum KEM. Next we recall the formal definition of a KEM: it is a triple of algorithms $\text{KEM} = (\text{KeyGen}, \text{EncCap}, \text{DeCap})$ such that:

- The probabilistic key generation algorithm $\text{KeyGen}(1^\ell)$ takes as input the security parameter $\ell$ and outputs a key pair $(dk, ek)$.
- The probabilistic encapsulation algorithm $\text{EncCap}(ek; r)$ takes as input a public encapsulation key $ek$ and outputs a ciphertext $c$ and a key $k$. The value $r$ corresponds to the random coins used by $\text{EncCap}$. We include it as an explicit input as we will need to refer to it in the description of our GAKE.

- The deterministic decapsulation algorithm $\text{DeCap}(dk, c)$ takes as input a secret decapsulation key $dk$ and a ciphertext $c$ and outputs a key $k$ or $\perp$ (meaning decryption failure).

We will consider the four Round 3 KEM finalists from NIST’s Post-Quantum standardization process. All of them target the IND-CCA2 security notion as required by NIST in its call for proposals, which is also usually named just IND-CCA; we will use the latter denomination throughout this paper. A KEM is considered to be IND-CCA secure if, given an encapsulated ciphertext and a key which is either the encapsulated key or a random one, an adversary (modeled as a probabilistic polynomial-time algorithm) with access to a decapsulation oracle is unable to distinguish between these two options with a probability non-negligibly better than a random guess. For a more formal definition see, for instance, [20].

Concerning the practical security strength of the candidates, NIST establishes 5 security levels ([21]). These security levels ask for resistance against attacks that use computer resources comparable to or greater than those required for key search against a block cipher or collision search for a certain hash function. More precisely, the security levels are summarized in Table 2.

Next, we briefly overview the four finalists KEMs. The full description of all the algorithms submissions to Round 3 can be found in the NIST webpage [22].

1) CLASSIC McEliece

Classic McEliece [23] is the only candidate based on codes. The KEM is built from an OW-CPA deterministic PKE, namely Niederreiter’s dual version of McEliece’s PKE, which uses binary Goppa codes ([24]). The Round 3 submission of McEliece comes with 5 different parameter sets, each one with two versions, depending on whether the parity check matrix of the code is reduced to systematic or semi-systematic form. The names of the parameter sets and their claimed security levels are shown in Table 3.

2) CRYSTALS-KYBER

Kyber [25], like two other finalists, NTRU and Saber, bases its security on a lattice problem, in this case, the Module learning with errors (MLWE) problem. The proposal is based on an IND-CPA PKE that allows decryption failures to occur with a negligible probability. Then, a modification of the Fujisaki-Okamoto transformation is used to obtain an IND-CCA KEM. The submission includes 3 different parameter sets pointing at NIST security levels 1, 3, and 5, respectively, depicted in Table 4.

3) NTRU

The NTRU submission for Round 3 is a merger of two different previous submissions, namely NTRUEncrypt and NTRU-HRSS-KEM ([26]), both based in the NTRU cryptosystem ([27]). Although the original NTRU was a partially correct probabilistic PKE, this submission starts
by defining a correct and deterministic PKE, which is assumed to be OW-CPA. Then an IND-CCA KEM is obtained from it by making small changes to the Saito-Xagawa-Yamakawa variant of NTRU-HRSS-KEM ([28]). The Round 3 submission proposes parameter sets which are shown in Table 5. The authors of the NTRU submission make two different estimations for the security level of their parameter sets, depending on whether the computation model is non-local or local. Details about these models and the motivation for differentiating the security levels depending on them can be found in the submission ([22]).

4) SABER
Saber (first proposed in [29]) is similar to Kyber, in the sense that the authors present in their Round 3 submission an IND-CPA PKE, and then they use a Fujisaki-Okamoto-like transformation to obtain an IND-CCA KEM. In addition, the PKE also comes with a negligible decryption failure probability, and the security is reduced to a lattice problem, in this case, the Module Learning With Rounding (MLWR) problem. The authors propose three different parameter sets for the KEM which are shown in Table 6 together with their claimed security levels.

C. FSXY: A GENERIC CONSTRUCTION FROM KEM TO POST-QUANTUM AKE

Generic transformations that convert secure KEMs into AKEs have been proposed in the standard model in [30] and [31]. These transformations give AKE protocols from IND-CCA secure KEM schemes using pseudorandom functions (PRFs). The resulting AKEs are proven secure in widely accepted security models, CK [32] and CK+ [31], respectively. Unfortunately, KEM schemes secure in the standard model are computationally inefficient for both classical and post-quantum communications.

In [17], Fujioka et al., proposed an efficient generic construction of AKE protocols from OW-CCA secure KEM schemes (which we denote by FSXY) by relaxing the security model to the Random Oracle Model (ROM). The resulting AKE protocols were proved to be CK+ secure in the ROM. Moreover, it was shown that the (ring-)LWE, McEliece one-way, NTRU one-way (among others) post-quantum assumptions can be used to construct secure AKE protocols. In addition, it was shown that by adapting the ROM in the security proof of the FSXY construction, the AKE protocols obtained from each post-quantum assumption become efficient on the communication cost.

The FSXY construction is as follows. Let $KEM_1 = (\text{KeyGen}_1, \text{EnCap}_1, \text{DeCap}_1)$ be a OW-CCA secure KEM and $KEM_2 = (\text{KeyGen}_2, \text{EnCap}_2, \text{DeCap}_2)$ be a OW-CPA secure KEM. Let $\ell$ be the security parameter $H_1: \{0, 1\}^\ell \rightarrow \text{RSE}$ and $H_2: \{0, 1\}^\ell \rightarrow \{0, 1\}^f$ be hash functions modeled as random oracles, where $\text{RSE}$ is a randomness space. The random values $r$ and $r_1$ are chosen from $\{0, 1\}^f(\ell)$, where $f$ is a polynomial function of the security parameter. The two-pass key exchange protocol involving users $U_A$ (the initiator) and $U_B$ (the responder) is shown in Fig. 1.

The session state of a session owned by $U_A$ contains an ephemeral secret key $r$ and a KEM key $K_A$. Similarly, the session state of a session owned by $U_B$ contains ephemeral secret keys $(r_1, r_2)$ and KEM keys $(K_{B,1}, K_{B,2})$.

It was shown in [17] that if $KEM_1$ is OW-CCA secure, and if $KEM_2$ is OW-CPA secure, then the FSXY transformation is CK+ secure under the Random Oracle Model.

D. BUILDING THE COMMITMENT SCHEME FROM THE KEM

The compiler described in Section II-A requires as a building block, a non-interactive non-malleable commitment scheme that is perfectly binding and achieves non-malleability for multiple commitments. Such a commitment scheme is realized by applying the transformation proposed in [33] to a KEM scheme (in particular, any Post-Quantum KEM described in Section II-B) to obtain an IND-CCA PKE from the KEM. As pointed out in [14], the commitment scheme with the required security properties follows readily from the PKE.

Let $KEM = (\text{KeyGen}, \text{Encap}, \text{Decap})$ be a key encapsulation mechanism and let $\text{SKE} = (\text{Enc}, \text{Dec})$ be a one-time symmetric key encryption scheme (as defined in Section 7.2 of [33]). The key lengths of both primitives must be the same for any value of the security parameter $\ell$. Then, a PKE scheme $\text{PKE}$ is obtained as follows.

The key generation algorithm for $\text{PKE}$ is the same as that of $\text{KEM}$, and, hence, the secret and public keys for $\text{PKE}$ are the same as those of $\text{KEM}$. That is, $\text{PKE}$ runs $\text{KeyGen}$ and obtains $(sk, pk)$, where $sk$ and $pk$ are the secret and public key, respectively.
The encryption algorithm for $PKE$ runs as follows. Upon receiving a message $m$, $PKE$ runs the encapsulation algorithm of $KEM$ and obtains $(k, \xi) \leftarrow \text{EnCap}(pk; r)$, where $k$ is a symmetric key, $r$ are random coins, and $\xi$ is a ciphertext encrypting $k$. The message $m$ is encrypted using the key $k$ and the encryption algorithm of $SKE$, $\nu \leftarrow \text{Enc}(k, m)$. The output of the encryption algorithm is $c = (\xi, \nu)$.

The decryption algorithm is defined as follows. Given a ciphertext $c = (\xi, \nu)$, $PKE$ runs the decapsulation algorithm of $KEM$ and obtains $k = \text{DeCap}(sk, \xi)$, and then runs the decryption algorithm of $SKE$ with the key $k$ to obtain $m = \text{Dec}(k, \nu)$. The output of the decryption algorithm is the plaintext $m$.

As shown in [33], the IND-CCA security of $PKE$ is inherited from the IND-CCA security of $KEM$ and $SKE$. As pointed out in [14], it is known that in the CRS model with a common reference string $\rho$, the required commitment schemes depending on $\rho$ can be constructed from any public-key encryption scheme that is non-malleable and secure for multiple encryptions (in particular, from any IND-CCA secure public-key encryption scheme).

The approach in this section is usually known as the \textit{KEM-DEM paradigm}, where DEM stands for data encapsulation mechanism. Here the algorithm $SKE$ plays the latter role, so we will usually refer to it as the DEM.

\section{Security Model}

In this section we present the security model under which our protocol is proven to be secure. The model is taken from [14] which is in turn based on the one from Bellare et al. [34]. We assume a fully connected communication network, that is, each pair of users are able to communicate through a point-to-point channel. We consider an active adversary who is in full control of the network: it has the power to eavesdrop, delay, insert or delete messages in communication flow at will.

\subsection{Protocol Instances}

Let $U_0, U_1, U_2, \ldots, U_{n-1}$ be the set of participants. Each of them may run any polynomial number of protocol instances in parallel. Given $i \in \{0, 1, \ldots, n - 1\}$ and $s_i \in \mathbb{N}$ we denote with $\Pi_{i}^{s_i}$ the $s_i$-th instance vinculated to user $U_i$. Each instance $\Pi_{i}^{s_i}$ has seven variables associated with it:
First we need a definition of several oracles during a security game (described later). The adversarial power is modelled by providing A

Definition 1: Instances $\Pi_i^A$ and $\Pi_j^B$ are partnered if $\text{pid}_i = \text{pid}_j$, $\text{sid}_i = \text{sid}_j$, $\text{sk}_i = \text{sk}_j$ and also $\text{acc}_i = \text{acc}_j = \text{true}$.

Against a passive adversary which does not interfere with protocol execution, all involved users should accept and end with the same session key. This is captured in the definition of correctness.

Definition 2: A group key establishment protocol is correct if, in the presence of a passive adversary $A$ (that is, $A$ does not have access to the $\text{Send}$ and $\text{Corrupt}$ oracles), the following condition holds: for all $i,j$ with $\text{sid}_i = \text{sid}_j$ and $\text{acc}_i = \text{acc}_j = \text{true}$, we have $\text{sk}_i = \text{sk}_j \neq \text{null}$ and $\text{pid}_i = \text{pid}_j$.

The notion of integrity, introduced in [35], ensures that, even with adversarial intervention, honest users (meaning that $\text{Corrupt}$ has not been queried on them) have some guarantees of holding the same key.

Definition 3: A correct group key establishment protocol is said to have integrity if, with overwhelming probability, all instances of honest participants that have accepted with the same session identifier $\text{sid}_i$ hold the same session key $\text{sk}_i$ and partner identifier $\text{pid}_i$.

Before providing the definition of a secure protocol we need to limit when a query to the $\text{Test}$ oracle, to avoid trivial attacks from the adversary.

Definition 4: A $\text{Test}$ query should only be allowed to instances holding a key that is not for trivial reasons known to the adversary. To achieve this, an instance $\Pi_i^A$ is called fresh if none of the following condition holds:

- For some $U_j \in \text{pid}_i^A$ a query $\text{Send}(U_k, s_k, M)$ after a query $\text{Corrupt}(U_j)$.
- The adversary has queried $\text{Reveal}(U_j, s_j)$ with $\Pi_i^A$ and $\Pi_i^A$ being partnered.

The last notion we need before defining a secure group key establishment protocol is adversarial advantage.

Definition 5: Given a security parameter $\ell$ and an adversary $A$, the advantage $\text{Adv}_A(\ell)$ in attacking the protocol is a function in $\ell$, defined as

$$\text{Adv}_A(\ell) = |2 \cdot \text{Succ} - 1|$$

where $\text{Succ}$ is the probability that the adversary queries $\text{Test}$ only on fresh instances and outputs correctly the bit $b$ used by the $\text{Test}$ oracle (without later breaking the freshness of those instances queried with $\text{Test}$).

Definition 6: We say that an authenticated group key establishment protocol is secure if for every adversary $A$ we have that

$$\text{Adv}_A(\ell) \leq \text{negl}(\ell)$$

where negl is a negligible function.

IV. OUR GAKE CONSTRUCTION

In this section we describe our GAKE protocol for $n \geq 2$ users or parties $U_0, U_1, U_2, \ldots, U_{n-1}$. They are organized in a cycle: each user $U_i$ has as his left neighbour $U_{i-1}$ and as his right neighbour $U_{i+1}$. The indices are taken modulo $n$, so $U_n$...
means $U_0$ and $U_{-1}$ means $U_{n-1}$. We assume that each user is aware of his index and the rest of the indices identifying the other users of the protocol.

For the construction of the GAKE we use the following primitives:

- **KEM** = (KeyGen, EnCap, DeCap) is an IND-CCA KEM.
- **SKE** = (Enc, Dec) is an one-time symmetric key encryption scheme used as a DEM.
- **$H$** is a hash function (theoretically modeled as a random oracle).

To obtain the GAKE, we feed the tools described in Sections II-A, II-C and II-D with these primitives (Fig. 2). First we instantiate a 2-party AKE with the FSXY transformation from Section II-C, using KEM as both KEM₁ and KEM₂. Note that the security notion IND-CCA is well known to imply both the OW-CPA and OW-CCA requirements for KEM₁ and KEM₂. The resulting 2AKE satisfies the strong security notion CK⁺ ([31]), which is enough for the compiler described in Section II-A. We would like to stress that, as pointed out in [35], an integrity property is also needed for 2AKE in order to attain the claimed security notion. It is a straightforward conformation that the 2AKE obtained from FSXY has integrity because of the way session identifiers are computed. The other ingredient needed for the compiler is a commitment scheme, which is also obtained from KEM as described in Section II-D. Note that all the KEMs enumerated in Section II-B fulfill the IND-CCA security and can be used in our construction. Finally, the hash function $H$ is used to derive session identifiers and keys, both in the 2AKE and the final step of the protocol. It is worth pointing out that the resulting GAKE achieves security in the model described in Section II-A, which covers strong adversaries that are in full control of the communication network and may delay, eavesdrop, insert, and delete messages at will.

Next, we describe the resulting GAKE protocol which is composed of 4 rounds of communication:

**Init**: Each $U_i$ is assumed to hold a pair $(dk_i, ek_i)$ generated with KeyGen. Here $ek_i$ is the long-term public encapsulation key for $U_i$ and is assumed to be certified and known by the rest of the users, whereas $dk_i$ is the long-term secret decapsulation key for $U_i$.

**Round 1-2**: For each $i \in \{0, 1, \ldots, n-1\}$ the 2AKE is run between $U_i$ and $U_{i+1}$. The two rounds are as follows:

**Round 1**: Each $U_i$ follows these steps:
- Generates randomness $\overrightarrow{r_i} \leftarrow \{0, 1\}^{|\ell|}$.
- Generates encapsulated key $\overrightarrow{(C_i, \overrightarrow{k_i})} \leftarrow \text{EnCap}(ek_{i+1}; H(\overrightarrow{r_i}, dk_i))$.
- Generates an ephemeral key pair $(dk_i, ek_i) \leftarrow \text{KeyGen}()$.
- Sends $(U_i, U_{i+1}, \overrightarrow{C_i}, ek_i)$ to $U_{i+1}$.

**Round 2**: Each $U_i$ follows these steps:
- Generates randomness $\overrightarrow{r_i} \leftarrow \{0, 1\}^{|\ell|}$ and $\overrightarrow{\rho_i} \leftarrow \text{RSE}$.
- Generates encapsulated key $\overrightarrow{(C_i, \overrightarrow{k_i})} \leftarrow \text{EnCap}(ek_{i-1}; H(\overrightarrow{r_i}, dk_i))$.
- Generates another encapsulated key $\overrightarrow{(\overrightarrow{T_i}, \overrightarrow{\lambda_i})} \leftarrow \text{EnCap}(ek_{i-1}; \overrightarrow{\rho_i})$.
- Sends $(U_{i-1}, U_i, \overrightarrow{C_i}, \overrightarrow{T_i})$ to $U_{i-1}$.

After receiving Round 2 message from $U_{i+1}$ each $U_i$:
- Decapsulates the keys $\overrightarrow{\kappa_{i-1}}, \overrightarrow{\kappa_{i+1}}, \overrightarrow{\lambda_{i+1}}$ from received messages.
- Sets $\overrightarrow{\text{sid}} = (U_{i-1}, U_i, ek_{i-1}, ek_i, \overrightarrow{C_{i-1}}, \overrightarrow{C_i}, \overrightarrow{T_i})$.
- Computes key (shared with $U_{i-1}$) $\overrightarrow{\kappa_i} = H(\overrightarrow{\kappa_{i-1}}, \overrightarrow{\kappa_i}, \overrightarrow{\lambda_i}, \overrightarrow{\text{sid}})$
- Sets $\overrightarrow{\text{sid}} = (U_i, U_{i+1}, ek_i, ek_{i+1}, \overrightarrow{C_i}, \overrightarrow{C_{i+1}}, \overrightarrow{T_{i+1}})$. 
Computes key (shared with $U_{i+1}$)

$$\overline{K}_i = H(\overline{k}_i, \overline{K}_{i+1}, \overline{X}_{i+1}, \overline{sid}).$$

**Round 3:** Each $U_i$ follows these steps:
- Computes $X_i = \overline{K}_i \oplus K_i$.
- Generates randomness $r_i' \leftarrow RSE$ for KEM and a random $IV_i$ for DEM.
- Generates encapsulated key $(C_i, \kappa_i) \leftarrow \text{EnCap}(ek_i; r_i')$.
- Sets commitment to $com_i = (C_i, C_i)$ and stores randomness $r_i = (IV_i, r_i')$.
- Broadcasts $M_{i}^{0} = (U_i, com_i)$.

**Round 4:** Each $U_i$ follows these steps:
- Broadcasts $M_{i}^{1} = (U_i, X_i, r_i)$.
- Checks that $X_0 \oplus X_1 \oplus \cdots \oplus X_{n-1} = 0$ and the correctness of the commitments. If any of these conditions fails, then $U_i$ ends the protocol execution.
- Computes the $n-1$ values $K_j$ for $j = 0, 1, 2, \ldots, n-1$ with $j \neq i$,

$$K_{i-j} = \overline{K}_i \oplus X_{i-1} \oplus \cdots \oplus X_{i-j}.$$

- Defines session key $sk_i$ and session identifier $sid_i$ as

$$(sk_i||sid_i) = H(K_0, \ldots, K_{n-1}, U_0, \ldots, U_{n-1}).$$

Next, we provide some comments pointing out differences between the compiler described in Section II-A and our implementation and also explaining some design choices and protocol steps in more detail:

- The input of KeyGen() in the asymptotic description of the KEM is the security parameter. In our implementation, KeyGen() outputs keys of fixed length for each KEM and parameter set, so it has no input. The sets $\{0, 1\}^{||}$ and RSE are the randomness spaces described in Section II-C.

- To compute the commitment $com_i$ to $X_i$, first an encapsulated key $k_i$ is generated with EnCap. Then the message $i||X_i$ (where $||$ denotes concatenation) is encrypted with Enc using key $k_i$. The randomness $r_i$ used by EnCap needs to be stored by $U_i$ to open the commitment in the next round. Therefore we needed to modify EnCap for each KEM in our implementation to make the randomness an explicit output of the algorithm instead of being generated by the algorithm itself.

- The verification of the commitments in Round 4 is done by recovering the key $k_i$ from $r_i$ with EnCap, then generating a new commitment $com_i'$ with $X_i$ and $r_i$ and checking that $com_i$ and $com_i'$ are equal.

- In the original compiler, the final key and session identifier derivation is done with a collision-resistant pseudorandom function family. The reason for using this tool is that, if the 2AKE has a security proof in the standard model, the compiled GAKE is also secure in the standard model. As the FSXY transformation already uses a hash function, our GAKE is only secure in the random oracle model, so we have chosen to simplify the compiler and use the same hash function for key and session identifier derivation.

- The hash function we have chosen in our implementation has output length which is double of the GAKE session key length. So the hash value $H(\overline{K}_0, \overline{K}_1, \ldots, \overline{K}_{n-1}, U_0, U_1, \ldots, U_{n-1})$ is computed and $sk_i$ is set to be the first half of this value and $sid_i$ is set to be the second one.

### A. SECURITY OF OUR PROPOSED GAKE PROTOCOL

Next we prove a security result for our protocol under the security model described in Section III.

**Theorem 1:** In the random oracle model, the protocol presented in Section IV is a correct and secure authenticated group key establishment protocol fulfilling integrity.

**Proof:** We follow the security proof of Theorem 1 in [14].

**Correctness.** It is easily verified that in an honest execution of the protocol, all participating users will terminate by accepting and computing the same session identifier and session key.

**Integrity.** As a consequence of the collision-resistance of the random oracle $H$, all oracles that accept with the same session identifiers also hold, with overwhelming probability, identical session keys $K_0, \ldots, K_{n-1}$ and associated these keys with the same participants $U_0, \ldots, U_{n-1}$.

**Key secrecy.** The proof of the secrecy is organized in a sequence of games, starting with a real attack of an adversary $A$ against the key secrecy of the GAKE protocol and ending in a game in which the advantage of the adversary is negligible. The idea is that we can bound the difference of the adversary’s advantage between any two consecutive games. We denote the advantage of the adversary in Game $i$, as usual, by $\text{Adv}(A, G_i)$. For the sake of clarity, we classify the Send queries into three categories depending on the stage of the protocol to which the query is associated. More precisely, $\text{Send}$ denotes the Send query associated with round $t$.

The first three games of this proof coincide with the same as those in Theorem 1 of [14]. Here we summarize the bounding of the adversary’s advantage and refer the interested reader to the original paper for the details.

**Game 0.** In this game, a real attack is performed by the adversary $A$, in which all the parameters such as the public parameters and the long-term secrets of each user are chosen as in the actual scheme. By definition we have $\text{Adv}(A, G_0) = \text{Adv}(A)$.

**Game 1.** For $i = 0, 1, \ldots, n-1$, we modify the simulation of the $\text{Send}$ and $\text{Execute}$ oracles so that whenever an instance $\Pi_i^0$ is still considered fresh at the end of Round 2, the keys $\overline{K}_i$ and $\overline{K}_i$ that it shares with instances $\Pi_{i-1}$ and $\Pi_{i+1}$, respectively, are replaced with random values from the range of the random oracle $H$.

It is not difficult to see that the difference between the advantage of this game and the previous one is bounded by the probability that the adversary breaches the security of any of the underlying 2AKE protocols executions. Therefore,
we have
\[ |\text{Adv}(A, G_1) - \text{Adv}(A, G_0)| \leq 2 \cdot \text{Adv}_{2\text{AKE}}(\ell, 2 \cdot q_{\text{send}}), \]
where \( q_{\text{send}} \) denotes the number of distinct protocol instances in Send queries.

**Game 2.** Here, the simulation of the Send oracle is modified so that a fresh instance \( \Pi_j^0 \) does not accept in Round 4 whenever one commitment \( \text{com}_j, j \neq i \) it receives in Round 3 was generated by the simulator but not generated by the respective instance \( \Pi_i^0 \) in the same session.

If the adversary \( A \) replays a commitment that should have let to acceptance in Round 4 in Game 1, then \( A \) detects the difference between this and the previous games. Therefore,
\[ |\text{Adv}(A, G_2) - \text{Adv}(A, G_1)| \leq \text{negl}(\ell). \]

**Game 3.** In this game, the simulation of the Send oracle changes so that a fresh instance \( \Pi_j^0 \) does not accept in Round 4 whenever one commitment \( \text{com}_j, j \neq i \) it receives in Round 3 was generated by the adversary. The advantage of the adversary differs from the previous game by a negligible amount, that is,
\[ |\text{Adv}(A, G_3) - \text{Adv}(A, G_1)| \leq \text{negl}(\ell). \]

**Game 4.** Here the simulations of the Execute and Send oracles are modified at the point of computing the session key. On one hand, in this game, all session keys are chosen uniformly at random and the adversary has no advantage. Hence,
\[ \text{Adv}(A, G_4) = 0. \]

On the other hand, the simulator keeps a list of strings \((K_0, \ldots, K_{n-1}, U_0, \ldots, U_{n-1})\) and once an instance receives the last Send-4 query, the simulator computes \( K_0, \ldots, K_{n-1} \) and checks if for the corresponding string \((K_0, \ldots, K_{n-1}, U_0, \ldots, U_{n-1})\) has already been used. If this is the case, the simulator assigns the corresponding string to the instance. If no such strings exist, the simulator assigns a session key \( sk_j^0 \in \{0, 1\}^l \) uniformly at random. Note that even if the messages from Round 4 are sent out, the list of strings still contains sufficient entropy so that the output of the random oracle \( H \) is indistinguishable from a random \( sk_j^0 \) with overwhelming probability. Consequently,
\[ |\text{Adv}(A, G_4) - \text{Adv}(A, G_3)| \leq \text{negl}(\ell). \]

Together, all the bounds obtained in the games imply that
\[ \text{Adv}(A) \leq 2 \cdot \text{Adv}_{2\text{AKE}}(\ell, 2 \cdot q_{\text{send}}) + \text{negl}(\ell). \]

\[ \Box \]

**V. IMPLEMENTING THE GAKEs**

In the following, we describe the implementation of the GAKE protocol, which is publicly available at https://github.com/jiep/pq-gake-fsxy. To do so, we describe separately each of the building blocks that make up the protocol.

**A. BUILDING BLOCKS**

1) KEM

The KEMs are taken from the open-source library LibOQS ([37]). It provides all the finalist implementations submitted to the NIST standardization process.\(^2\) It has been developed by the Open Quantum Safe project, which aims at prototyping and experimenting with post-quantum cryptography, but as of today, it is not production-ready. It is written in C99 and its advantages include:

- Dynamic management of the KEMs, making it possible to exchange one for the other without the need to modify the code of the protocol.
- Building the library with only the KEM implementations that are needed in the application.
- Easy cross-compilation.
- Provides common functions (e.g. hash and random bits functions, among others).

LibOQS provides 10 parameter sets for Classic McEliece, 3 for Kyber,\(^3\) 4 for NTRU, and 3 for Saber. Table 7 shows the key sizes (public and secret), the size of the shared secret and ciphertext, as well as the claimed security level and security model of all parameter sets in LibOQS. It can be noted that the public key size of Classic McEliece is several orders of magnitude larger than the other KEMs. On the other hand, the ciphertext size is smaller than the other finalists in the standardization process.

Two different implementations come from each parameter set: the reference implementation (called ref, clean, or vec by the KEMs) and the optimized implementation (named avx2 or avx). All information on these implementations can be found in Table 8. It can be noted that the reference implementations do not present any architecture or operating system limitation, whereas the optimized implementation runs only on the x86_64 architecture for macOS and Linux operating systems. It is noteworthy that the Classic McEliece implementations have large stack usage and may cause failures when run on threads or in constrained environments ([36]).

The KEM is the basic building block on which the subsequent ones depend.

2) 2-PARTY AKE

The 2AKE has been implemented by following the FSXY transformation of Fig. 1. It has been split into three algorithms:

- \( \text{Init} \): the algorithm that runs \( U_A \) at the beginning of the protocol and outputs message \( M \).
- \( \text{AlgB} \): the algorithm that runs \( U_B \) by taking the message \( M \) and outputs the message \( \overline{M} \) and the session key \( sk \).

\(^2\) It also provides all the implementations of the finalist digital signatures, but for the implementation of this protocol, they are not required.

\(^3\) It provides 6 parameter sets, but the so-called 90s variants will not be considered because they are intended for legacy hardware and do not support SHA-3, and our implementations depends on it.
AlgA: the algorithm that runs $U_A$ at the end of the protocol by taking as input the message $\overrightarrow{M}$ and outputs the session key $SK$.

Note that Init corresponds to Round 1 of our GAKE description in Section IV whereas AlgB and AlgA constitute Round 2.

In the implementation, $KEM_1 = KEM_2$ and the hash functions $H_1$ and $H_2$ are SHA3-256 provided by LibOQS.

### 3) COMMITMENT SCHEME

The commitment scheme has been implemented as an IND-CCA PKE with the KEM/DEM approach (see details in Section II-D), with the KEM being any of those implemented in LibOQS and the DEM being set to AES256-GCM imported from OpenSSL 1.1.1f ([38]). The commitment is given by the ciphertext of the KEM and the tag of the DEM. The randomness $r_i$ is given by the coins of the KEM and the tag of the DEM. The randomness $r_i$ is given by the coins of the KEM and the tag of the DEM. Note that in Round 4 of the GAKE protocol, $r_i$ is broadcast, so it was required to modify all LibOQS implementations (see Table 8) to make the KEM deterministic to preserve the randomness.

Three algorithms are implemented:

- Init allocates space for KEM and DEM ciphertexts.
- Commit creates a commitment as described in Section II-D.
- Check creates a commitment and checks if it is equal to a commitment created previously.

### B. GAKE PROTOCOL

The GAKE protocol has been implemented using the aforementioned building blocks. All hash functions come from the SHA-3 hash functions implemented in LibOQS. In addition, the implementation assumes a zero-delay communications network.

The protocol allows for a polynomial number of instances running in parallel. Hence, certain variables are required to keep the state of the instance. These are inherited from the Abdalla et al.’s compiler ([14]):

- public_key contains authentication public key.
- secret_key contains authentication secret key.
- pid contains the user identifiers $U_i$ that are involved in the protocol instance.
- $sk$ is the session key. Its size is 32 bytes. By default, its value is set to $0^{256}$.
- $sid$ is the public identifier for $sk$. Its size is 32 bytes.
- $term$ is a boolean variable that indicates whether an instance has terminated. In the implementation, 0 indicates false and 1, true.
- $acc$ is a boolean variable that indicates whether an instance has been accepted.
- Other variables that contain all the needed values for the protocol (e.g. $\overrightarrow{K_i}$, $\overrightarrow{K_j}$, $X_i$, $r_j$, $K$, etc.).

#### 1) INIT

During the Init phase, all parties generate their long-term authentication keys, and all the public keys are assumed to be known by the rest of the users.

#### 2) ROUND 1-2

During Round 1-2, two types of messages are exchanged:

- Message $\overrightarrow{M}$ contains a public key, a ciphertext, and $U_A$ and $U_B$. The size of $U_A$ and $U_B$ is set to 20 bytes.
- Message $\overrightarrow{M}$ contains two ciphertexts, as well as $U_A$ and $U_B$. Their size depends on the KEM in use (see Table 7).
3) ROUND 3
During Round 3, the messages $M^i_1, i = 0, \ldots, n - 1$, are broadcast to all other users. The message $M^i_1$ contains the user identifier $U_i$, the KEM ciphertext, and the DEM ciphertext and tag. The randomness $r_i$ keeps the coins of the KEM and the IV of the DEM. The size of $U_i$ is set to 20 bytes and to encrypt $i || X_i$, 36 bytes are needed ($|X_i| = 32$ and $|i| = 4$). The tag size is 16 bytes.
4) ROUND 4
In Round 4, \( n \) messages \( M_i^2 \) are broadcast. The message contains \( U_i \) and the randomness \( r_i \) (coins of the KEM and IV of the DEM). The size of \( X_i \) is 32 bytes and the IV is 12 bytes. The size of coins depends on the KEM being used (see Table 7).

The session key \( sk \) and \( sid \) is generated from master key \( K \) with SHA3-512, where first 32 bytes are set to be the \( sk \) and last 32 bytes are set to be the \( sid \).

Fig. 3 shows a run of GAKE protocol for Kyber1024 and Classic-McEliece-8192128f with 100 parties. See Appendix A for a complete run of the protocol.

C. BENCHMARKING ENVIRONMENT

A workflow has been developed on GitHub Actions that allows reproducing the experiments in an isolated environment. The workflow is described in Fig. 4 and includes all the required steps from building the GAKE protocol binaries to executing them and obtaining the experimental results. The workflow runs on an Ubuntu 20.04 runner hosted on GitHub Actions and consists of 4 steps:

1) Build: It builds all the binaries and the libraries they depend on. Fig. 5 shows the complete process. A custom library is built from the LibOQS v0.7.0 library ([37]). In it, all the KEM implementations have been modified to be deterministic to keep the randomness of the commitment scheme. Its building has been automated with CMake by enabling the options \(-\text{DOQS\_DIST\_BUILD} = \text{ON} \) and \(-\text{DOQS\_MINIMAL\_BUILD} = "\{\text{ENABLED\_ALGS}\}"\), where \text{ENABLED\_ALGS} is an array that enables desired KEMs (see details in [36]). It enables only the algorithms specified in the LibOQS library in Table 7. In addition, OpenSSL 1.1.1f is statically linked, which is a dependency required by LibOQS. With the options enabled, a static library is built and gcc has been used as the C compiler.

B The GAKE protocol code is built with CMake and gcc as in the previous step. The latter uses the \(-\text{O3} \) and \(-\text{fwrapv} \) options. The custom LibOQS library and OpenSSL 1.1.1f are statically linked. The latter is used to implement AES256-GCM in the commitment scheme. A series of tests with ctest is launched to guarantee that the generated binaries work properly. These include the correct functioning of all the building blocks that integrate the GAKE protocol: AES256-GCM, the AKE, the commitment scheme, and the implementation of the GAKE protocol itself for each of the KEM implementations in LibOQS.

2) Run tests: This step measures the performance of each of the building blocks of the GAKE protocol. Performance is measured in terms of the number of CPU cycles and execution time. For this purpose, we used the LibOQS header ds_benchmark.h available at https://github.com/open-quantum-safe/liboqs/blob/0.7.0/tests/ds_benchmark.h. It implements two macros to measure performance:

- \text{TIME\_OPERATION\_ITERATIONS}: It executes a piece of code for a given number of iterations.
- \text{TIME\_OPERATION\_SECONDS}: It executes a piece of code for a given number of seconds.

All experiments are run with the former macro. This step is run on a self-hosted runner with Ubuntu 20.04 on WSL2 ([39]) under Windows 10 on with specifications given in Table 9. This was done with this approach because runners hosted on GitHub Actions can only run for a maximum of 6 hours ([40]), which is not enough time to run all the necessary experiments.

The tests defined in this step are:

- \text{test\_speed\_kem}: This test measures the performance (in CPU cycles and execution time) of each KEM implemented in LibOQS (see Table 7). Key generation, encapsulation, and decapsulation are measured separately. The result of this test is an average of 10000 iterations.
- \text{test\_speed\_ake}: It measures the performance of the FSXY transformation for each of the KEMs implemented in LibOQS. Each algorithm of the transformation is measured independently.
The result is an average of the execution of 10,000 iterations.

- **test_speed_commitment**: It measures the performance of the commitment scheme for each of the KEMs implemented in LibOQS. The DEM is always fixed to AES256-GCM. 10,000 iterations are run to measure the performance of key generation, generate a commitment and check it.

- **test_speed_gake**: It measures the performance of the GAKE protocol for each of the KEMs implemented in LibOQS based on the number of parties, \( n \), running the protocol. It is run for \( n = 2, 2^2, \ldots, 2^{11} = 2048 \). In addition, the performance of each round of the protocol is measured separately.

All of the tests above generate tables that are converted to CSV format to be processed in the subsequent steps.

3) **Generate graphics**: This step generates the graphics of Section VI. The graphics are plotted with Python and the seaborn visualization library ([41]). All graphics are saved in png format.

4) **Release**: It creates a new release on GitHub and uploads all the data that has been generated during the workflow, which was stored in Artifacts on GitHub Actions ([42]): binaries, graphics, and CSV files.

### VI. EXPERIMENTAL RESULTS: COMPARISON AMONG THE FOUR KEMs

In this section, we compare the experimental results achieved from the aforementioned tests. More precisely, we present the results of the tests described in Section V-C. For each of the four KEMs and each security level, we compare the performance of all the cryptographic primitives involved, including all the underlying operations (algorithms) of each of them. Namely, KEM, the two-party AKE, the commitment scheme and, finally, the GAKE protocol. Note that we only compare the optimized implementation of each parameter set (see Table 8). Numerical results can be found in Appendix B.

Fig. 6 shows the performance of each KEM operation for each security level. It can be observed that, for all security levels, the **KeyGen** algorithm is significantly slower on Classic McEliece than on the other KEMs. This is caused by the huge size of the keys in Classic McEliece (see Table 7). The **Encaps** algorithm does not show significant differences, whereas the **Decap** algorithm does show this difference Classic McEliece vs. other KEMs, but it is not as meaningful as in the case of the **KeyGen** algorithm.

Fig. 7 shows the performance of the AKE achieved from the FSXY transformation (Fig. 1). It can be seen that, for...
each security level and each AKE algorithm, the performance difference between Classic McEliece and the rest of the schemes is significant. This is because the AKE is KEM dependent, with the KeyGen and Decap algorithms being slower in Classic McEliece than in the other KEMs.

Concerning the commitment scheme (Fig. 8), Classic McEliece is faster during the Init algorithm. This is mainly because Classic McEliece ciphertexts are smaller than the rest (see Table 7). The Commit and Check algorithms perform better with NTRU, at any security level.

Fig. 9 shows the performance of the GAKE protocol in each round. The Init round initializes the structure and variables needed to store the state of the protocol instance. It can be observed that Kyber is noticeably more efficient than the rest of the schemes, at any security level. In Round 1-2 (AKE) and Round 3 (commitment generation), the same applies: Kyber parameter sets offer the best performance. Finally, in Round 4 (commitment checking, master key derivation, and session key generation) the performance at security level 1 is very similar among the parameter sets. The most efficient is NTRU-HPS-2048-677 for level 3 and NTRU-HPS-4096-821 for level 5.

Fig. 10 shows the performance of the GAKE protocol as a function of the number of parties participating in the protocol. It can be noted that, at all security levels, Classic McEliece is significantly less efficient than the rest of the KEMs and this is found to worsen as the number of parties in the protocol increases. The most efficient at security level 1 is Kyber 512, at level 3 is NTRU-HRSS-701, and at level 5 is Kyber1024.
In this section, we compare experimentally the performance of the FSXY transformation described in Section II-C with the FOAKE transformation ([18], [19]) used in [11] on Kyber. The latter is a novel transformation analogous to FSXY, i.e., it derives a secure two-party AKE from another cryptographic primitive (in this case, from an IND-CPA public-key novel encryption scheme). FOAKE is proved to be secure in the QROM, but it cannot be applied to just any KEM, only to those that satisfy several properties (see details in [19] and [11]). It is shown in [11] that it can be applied to Kyber. Table 10 shows a theoretical comparison between FSXY and FOAKE transformations. The FOAKE transformation consists of 2 messages in the same way as the FSXY transformation, but the former sends messages $\overrightarrow{M}$ and $\overleftarrow{M}$ that do not contain $U_A$ and $U_B$, which, consequently, produces messages of a smaller size. As in Section VI, 

### Table 10. Theoretical comparison between FSXY and FOAKE transformations.

| Property                        | FSXY              | FOAKE             |
|---------------------------------|-------------------|-------------------|
| Security model                  | ROM               | QROM              |
| Initial cryptographic primitive | IND-CCA KEM       | IND-CCA public-key encryption scheme |
| Transformation result           | Secure 2-party AKE| Secure 2-party AKE|
| Additional hypothesis on the initial primitive? | No | Yes. Disjoint Simulatability [18], [19] |
| Sent messages                   | 2                 | 2, but shorter    |

In this section, we provide experimental comparisons between the performance of the the FSXY transformation described in Section II-C with the FOAKE transformation ([18], [19]) used in [11] on Kyber.
we compare the performance of all the involved operations of each security level of Kyber.

Fig. 11 shows the comparison between both transformations with Kyber. It can be seen that the \( FO_{AKE} \) transformation performs better than \( FSXY \). The results show that, if Kyber is applied as KEM on this GAKE protocol, \( FO_{AKE} \) should be considered instead of \( FSXY \) providing, in addition, a higher level of security by being secure in the QROM.

**VII. CONCLUSION**

This paper shows the performance of a post-quantum key authenticated key exchange (GAKE) protocol constructed by applying the generic \( FSXY \) transformation to the all NIST finalist post-quantum KEMs. The protocol has been implemented with LibOQS, an open-source library that provides all the finalist KEMs of the NIST standardization process. We show experimentally that Classic McEliece is not suitable in this GAKE because it is significantly slower than the other KEMs. The most appropriate KEM for security level 1 is Kyber 512, for level 3 is Kyber768 and NTRU-HRSS-701, and, for level 5 is Kyber1024. In addition, the \( FO_{AKE} \) transformation is compared against \( FSXY \) on Kyber, showing that the latter is significantly faster than \( FSXY \) and provides a higher level of security by being QROM secure. This last result is especially noteworthy considering that Kyber is the first post-quantum KEM that will be standardized by NIST.

**APPENDIX A**

**COMPLETE RUN OF THE GAKE PROTOCOL**

Here, we show a complete run of GAKE protocol with Classic-McEliece-8192128f for 3 parties (for brevity).
J. I. Escribano Pablos et al.: Design and Implementation of a Post-Quantum CAKE With the LibOQS Library
Commitments:
c0: 00000000000000000000000000000000000000000000000000
cl: 00000000000000000000000000000000000000000000000000
c2: 00000000000000000000000000000000000000000000000000
Master Key:
k0: 00000000000000000000000000000000000000000000000000
kl: 00000000000000000000000000000000000000000000000000
x2: 00000000000000000000000000000000000000000000000000
Pids:
pid0: Party 0
pid1: Party 1
pid2: Party 2
Accepted: 0
Terminated: 0
Party 2
Public key: 82ba5d6099df021b00932...354e329a95e3ed5d87
Secret key: ffe16d5a10b18b3b2c8c...37e996644c8de670c2b
Left key: 2f055bd8d8e460e5358...7a007e4e7d118e59f
Right key: 9b9d417f0d79771b3a...14a4fda239588e9c66
Session id: 00000000000000000000000000000000000000000000000000
Session key: 00000000000000000000000000000000000000000000000000
X:
x0: 00000000000000000000000000000000000000000000000000
x1: 00000000000000000000000000000000000000000000000000
x2: 00000000000000000000000000000000000000000000000000
Coins:
r0: 00000000000000000000000000000000000000000000000000
r1: 00000000000000000000000000000000000000000000000000
r2: 00000000000000000000000000000000000000000000000000
Commitments:
c0: 00000000000000000000000000000000000000000000000000
cl: 00000000000000000000000000000000000000000000000000
x2: 00000000000000000000000000000000000000000000000000
Master Key:
k0: 00000000000000000000000000000000000000000000000000
kl: 00000000000000000000000000000000000000000000000000
k2: 00000000000000000000000000000000000000000000000000
Pids:
pid0: Party 0
pid1: Party 1
pid2: Party 2
Accepted: 0
Terminated: 0
Round 3
Party 0
Public key: ef36ce9975ef95084a00...3af8707e36c6f28625c
Secret key: 7786f299723d3f6b93e7...1851e6954bab7493627
Left key: 981a6417d6b407971a3...1a4a9fda239588e9c66
Right key: 2a88fa68d1035adebe37d...226727415e4563d113
Session id: 00000000000000000000000000000000000000000000000000
Session key: 00000000000000000000000000000000000000000000000000
X:
x0: 3395e2930eb1a9a2bd2e...3bec3b897d1107238955
x1: 84899b5bcb4cd4c5d9252...a8e720a519e9e23d40c
x2: b7c1c3f8b237657222f...930498364f489005a9
Coins:
r0: 00000000000000000000000000000000000000000000000000
r1: 00000000000000000000000000000000000000000000000000
r2: 00000000000000000000000000000000000000000000000000
Commitments:
c0: 00000000000000000000000000000000000000000000000000
cl: 00000000000000000000000000000000000000000000000000
cl: 00000000000000000000000000000000000000000000000000
Master Key:
k0: 00000000000000000000000000000000000000000000000000
kl: 00000000000000000000000000000000000000000000000000
k2: 00000000000000000000000000000000000000000000000000
Pids:
pid0: Party 0
pid1: Party 1
pid2: Party 2
Accepted: 0
Terminated: 0
Party 1
Public key: Daa123ba6c84a03f87b...ba9a4a03b57074e575f
Secret key: e2e2de6632a67b5...2e2d230945e3e2eb6
Left key: ab88fa68d1035adebe37d...226727415e4563d113
Right key: 2f065d46d8e60e5358...7a007e4e7d118e59f
Session id: 00000000000000000000000000000000000000000000000000
Session key: 00000000000000000000000000000000000000000000000000
X:
J. I. Escribano Pablos et al.: Design and Implementation of a Post-Quantum GAKE With the LibOQS Library

Round 1-2
Party 0
Public key: c63c3c95f5e9508d00...3f98778e3c628e25c
Secret key: 77d6d9c3d5e6b39e7...18d1e69f54bab7439627
Left key: 983a6417d9b9407971a3...14a49fda239588e9c6
Right key: ab8f568da035abc37d...2f6727415e485f2d1193
Session id: 0000000000000000000000000000000000000000
Session key: e0000000000000000000000000000000000000000
X:
X0: 0000000000000000000000000000000000000000
X1: 0000000000000000000000000000000000000000
X2: 0000000000000000000000000000000000000000
Coins:
r0: 0000000000000000000000000000000000000000
r1: 0000000000000000000000000000000000000000
r2: 0000000000000000000000000000000000000000
Commissions:
c0: 0000000000000000000000000000000000000000
C1: 0000000000000000000000000000000000000000
C2: 0000000000000000000000000000000000000000
Master Key:
k0: 0000000000000000000000000000000000000000
k1: 0000000000000000000000000000000000000000
k2: 0000000000000000000000000000000000000000
Pids:
pid0: Party 0
pid1: Party 1
pid2: Party 2
Accepted: 0
Terminated: 0

Round 1-2
Party 1
Public key: 0a9129ba6c64803f87b...baa4a03b57075e755fe
Secret key: 6b8ae20e2e4632ae67b5...2ed2399845ca526ba6
Left key: ab8f6544d1035abc37d...2f6727415e485f2d1193
Right key: 2f605b9d706e5358...87a074476d18ec59f
Session id: 0000000000000000000000000000000000000000
Session key: e0000000000000000000000000000000000000000
X:
X0: 0000000000000000000000000000000000000000
X1: 0000000000000000000000000000000000000000
X2: 0000000000000000000000000000000000000000
Coins:
r0: 0000000000000000000000000000000000000000
r1: 0000000000000000000000000000000000000000
r2: 0000000000000000000000000000000000000000
Commissions:
c0: 0000000000000000000000000000000000000000
C1: 0000000000000000000000000000000000000000
C2: 0000000000000000000000000000000000000000
Master Key:
k0: 0000000000000000000000000000000000000000
k1: 0000000000000000000000000000000000000000
k2: 0000000000000000000000000000000000000000
Pids:
pid0: Party 0
pid1: Party 1
pid2: Party 2
Accepted: 0
Terminated: 0
APPENDIX B

NUMERICAL RESULTS OF TESTS

Tables 11–18 show numerical results for each graphic shown in Section VI.

### TABLE 11. CPU cycles for each operation on KEM.

| Parameter set              | decaps | encaps | keygen  |
|----------------------------|--------|--------|---------|
| Classic-McEliece-348864    | 123636 | 39442  | 35020985 |
| Classic-McEliece-460896    | 123316 | 39679  | 42829701 |
| Classic-McEliece-460897    | 357779 | 75205  | 10163568 |
| Classic-McEliece-668812    | 351315 | 75432  | 77796873 |
| Classic-McEliece-668813    | 382416 | 124901 | 142221367 |
| Classic-McEliece-668818    | 384330 | 180361 | 222641603 |
| Classic-McEliece-690119    | 349284 | 122261 | 131269058 |
| Classic-McEliece-690128    | 347672 | 123114 | 89402713 |
| Classic-McEliece-812128    | 380697 | 136072 | 134511827 |
| Classic-McEliece-812128    | 392884 | 136633 | 973372927 |
| FireSaber-KEM              | 94797  | 98948  | 91801   |
| Kyber1024                  | 51014  | 70850  | 57722   |
| Kyber512                   | 22730  | 33506  | 28819   |
| Kyber68                    | 35993  | 53185  | 43089   |
| LightSaber-KEM             | 39431  | 43959  | 42212   |
| NTRU-HPS-2048-509          | 29134  | 31251  | 134086  |
| NTRU-HPS-2048-677          | 42862  | 42405  | 217540  |
| NTRU-HPS-4096-821          | 53396  | 48300  | 799511  |
| NTRU-HRSS-701              | 45244  | 30143  | 209976  |
| Saber-KEM                  | 63544  | 68726  | 63409   |

### TABLE 12. CPU cycles for each operation on AKE.

| Parameter set              | algA  | algB  | init  |
|----------------------------|-------|-------|-------|
| Classic-McEliece-348864    | 858059| 858736| 32393667 |
| Classic-McEliece-348864    | 8569345| 8612752| 243414602 |
| Classic-McEliece-460896    | 17700937 | 17730416 | 931398068 |
| Classic-McEliece-460896    | 19521974 | 20018038 | 767502668 |
| Classic-McEliece-668812    | 41542425 | 40623605 | 1476796001 |
| Classic-McEliece-668812    | 21021351 | 37142750 | 1063334356 |
| Classic-McEliece-690119    | 35253033 | 35449125 | 130841572 |
| Classic-McEliece-690119    | 35693068 | 35588478 | 9586935355 |
| Classic-McEliece-812128    | 47284442 | 44764384 | 1346369051 |
| Classic-McEliece-812128    | 46678824 | 40703072 | 958258184 |
| FireSaber-KEM              | 252384 | 382050 | 220898 |
| Kyber1024                  | 157702 | 275832 | 152475 |
| Kyber512                   | 74852 | 132841 | 75748 |
| Kyber68                    | 116833 | 208207 | 112594 |
| LightSaber-KEM             | 106402 | 171794 | 102198 |
| NTRU-HPS-2048-509          | 85414 | 132512 | 176955 |
| NTRU-HPS-2048-677          | 125943 | 184092 | 276924 |
| NTRU-HPS-4096-821          | 156025 | 214513 | 364665 |
| NTRU-HRSS-701              | 137014 | 166123 | 265199 |
| Saber-KEM                  | 173708 | 269199 | 156085 |
### TABLE 13. CPU cycles for each operation on the commitment scheme.

| Parameter set          | check  | commit | init  |
|------------------------|--------|--------|-------|
| Classic-McEliece-348864| 29197  | 26110  | 504   |
| Classic-McEliece-348864f| 27409  | 26154  | 507   |
| Classic-McEliece-460896| 46800  | 46610  | 579   |
| Classic-McEliece-460896f| 44406  | 44034  | 631   |
| Classic-McEliece-6688128| 83966  | 83803  | 721   |
| Classic-McEliece-6688128f| 87638  | 79644  | 730   |
| Classic-McEliece-6960119| 87490  | 87024  | 529   |
| Classic-McEliece-6960119f| 87350  | 87096  | 685   |
| Classic-McEliece-8192128| 99463  | 106419 | 522   |
| Classic-McEliece-8192128f| 116952 | 108657 | 752   |
| FireSaber-KEM          | 95846  | 97015  | 3376  |
| Kyber1024              | 64164  | 63844  | 3499  |
| Kyber512               | 29839  | 29204  | 1531  |
| Kyber768               | 50658  | 50824  | 2653  |
| LightSaber-KEM         | 39516  | 39006  | 1710  |
| NTRU-HPS-2048-509      | 18166  | 17824  | 1740  |
| NTRU-HPS-2048-677      | 26137  | 25647  | 2118  |
| NTRU-HPS-4096-821      | 29673  | 29643  | 2850  |
| NTRU-HRSS-701          | 21728  | 21212  | 2669  |
| Saber-KEM              | 64943  | 64084  | 2503  |

### TABLE 14. Time (in us) for each round on GAKE protocol.

| Parameter set          | \(n\) | init | round12 | round3 | round4 |
|------------------------|------|------|---------|--------|--------|
| Classic-McEliece-348864| 2    | 615431324 | 1696747143 | 214368 | 118846 |
| Classic-McEliece-348864| 4    | 1583603828 | 2869072109 | 490660 | 78044 |
| Classic-McEliece-348864| 8    | 2367403159 | 850125364 | 719104 | 195208 |
| Classic-McEliece-348864| 16   | 858591986 | 1920440006 | 1279658 | 8055160 |
| Classic-McEliece-348864| 32   | 1554079507 | 1355677857 | 263843 | 4554024 |
| Classic-McEliece-348864| 64   | 2818234753 | 679532571 | 5516502 | 20208740 |
| Classic-McEliece-348864| 128  | 4010991978 | 2435017956 | 12636566 | 834462290 |
| Classic-McEliece-348864| 256  | 3911923429 | 3232859700 | 36204592 | 3417393020 |
| Classic-McEliece-348864| 512  | 1004661476 | 803441999 | 286293883 | 1669956478 |
| Classic-McEliece-348864| 1024 | 564876830 | 2321618938 | 494598324 | 711534243 |
| Classic-McEliece-348864| 2048 | 2225555345 | 1254861332 | 243603057 | 572706871 |
| Classic-McEliece-348864f| 2    | 488943030 | 107269180 | 231969 | 117684 |
| Classic-McEliece-348864f| 4    | 975847422 | 2069043740 | 364754 | 481170 |
| Classic-McEliece-348864f| 8    | 1946928856 | 4143706238 | 695710 | 1910756 |
| Classic-McEliece-348864f| 16   | 3871425313 | 3975704069 | 1291402 | 782339 |
| Classic-McEliece-348864f| 32   | 3459567823 | 3645043765 | 2628756 | 43810472 |
| Classic-McEliece-348864f| 64   | 2634568672 | 3270145154 | 5745522 | 201125972 |
| Classic-McEliece-348864f| 128  | 922596822 | 1816343482 | 13297466 | 83517934 |
| Classic-McEliece-348864f| 256  | 2119659048 | 4180954250 | 32626524 | 3438590182 |
| Classic-McEliece-348864f| 512  | 338168335 | 1679142040 | 113538214 | 1387490533 |
| Classic-McEliece-348864f| 1024 | 1217596450 | 889263199 | 421270974 | 963786826 |
| Classic-McEliece-348864f| 2048 | 4166382218 | 721092034 | 1925740795 | 3960384406 |
| Classic-McEliece-460896| 2    | 1912024768 | 3921837646 | 362890 | 200634 |
| Classic-McEliece-460896| 4    | 3415015131 | 2835097176 | 653708 | 87246 |
| Classic-McEliece-460896| 8    | 3927924184 | 2779169586 | 1205896 | 5070148 |
| Classic-McEliece-460896| 16   | 890839526 | 4016145298 | 2228700 | 19678514 |
| Classic-McEliece-460896| 32   | 1089735463 | 2424453689 | 5206710 | 108774642 |
| Classic-McEliece-460896| 64   | 2411407737 | 4113391356 | 995120 | 373476180 |
| Classic-McEliece-460896| 128  | 397558273 | 2035324370 | 21195804 | 1515298786 |
TABLE 14. (Continued.) Time (in us) for each round on GAKE protocol.

| Protocol                | Time (in us) |
|-------------------------|--------------|
| Classic-McEliece-460896 | 120976       |
| Classic-McEliece-460896 | 512          |
| Classic-McEliece-460896 | 1024         |
| Classic-McEliece-460896 | 2048         |
| Classic-McEliece-460896f| 2            |
| Classic-McEliece-460896f| 4            |
| Classic-McEliece-460896f| 8            |
| Classic-McEliece-460896f| 16           |
| Classic-McEliece-6688128 | 32           |
| Classic-McEliece-6688128 | 4            |
| Classic-McEliece-6688128 | 8            |
| Classic-McEliece-6688128 | 16           |
| Classic-McEliece-6688128 | 32           |
| Classic-McEliece-6688128 | 64           |
| Classic-McEliece-6688128 | 128          |
| Classic-McEliece-6688128 | 256          |
| Classic-McEliece-6688128 | 512          |
| Classic-McEliece-6688128 | 1024         |
| Classic-McEliece-6688128 | 2048         |
| Classic-McEliece-6688128f| 2            |
| Classic-McEliece-6688128f| 4            |
| Classic-McEliece-6688128f| 8            |
| Classic-McEliece-6688128f| 16           |
| Classic-McEliece-6688128f| 32           |
| Classic-McEliece-6688128f| 64           |
| Classic-McEliece-6688128f| 128          |
| Classic-McEliece-6688128f| 256          |
| Classic-McEliece-6688128f| 512          |
| Classic-McEliece-6688128f| 1024         |
| Classic-McEliece-6688128f| 2048         |

J. I. Escribano Pablos et al.: Design and Implementation of a Post-Quantum GAKE With the LibOQS Library
| Protocol                  | Time (in us)  |
|--------------------------|--------------|
| Classic-McEliece-6960119f | 1362125798   |
| Classic-McEliece-6960119f | 128          |
| Classic-McEliece-6960119f | 2397190546   |
| Classic-McEliece-6960119f | 1543625822   |
| Classic-McEliece-6960119f | 256          |
| Classic-McEliece-6960119f | 2011115315   |
| Classic-McEliece-6960119f | 1279615086   |
| Classic-McEliece-6960119f | 2048         |
| Classic-McEliece-8192128  | 2407516062   |
| Classic-McEliece-8192128  | 4            |
| Classic-McEliece-8192128  | 2520870316   |
| Classic-McEliece-8192128  | 16           |
| Classic-McEliece-8192128  | 32           |
| Classic-McEliece-8192128  | 64           |
| Classic-McEliece-8192128  | 128          |
| Classic-McEliece-8192128  | 256          |
| Classic-McEliece-8192128  | 512          |
| Classic-McEliece-8192128  | 1024         |
| Classic-McEliece-8192128  | 2048         |
| Classic-McEliece-8192128f | 2            |
| Classic-McEliece-8192128f | 4            |
| Classic-McEliece-8192128f | 3897540184   |
| Classic-McEliece-8192128f | 8            |
| Classic-McEliece-8192128f | 345283495    |
| Classic-McEliece-8192128f | 16           |
| Classic-McEliece-8192128f | 32           |
| Classic-McEliece-8192128f | 64           |
| Classic-McEliece-8192128f | 128          |
| Classic-McEliece-8192128f | 256          |
| Classic-McEliece-8192128f | 512          |
| Classic-McEliece-8192128f | 1024         |
| Classic-McEliece-8192128f | 2048         |
| FireSaber-KEM            | 607290       |
| FireSaber-KEM            | 4            |
| FireSaber-KEM            | 8            |
| FireSaber-KEM            | 16           |
| FireSaber-KEM            | 32           |
| FireSaber-KEM            | 64           |
| FireSaber-KEM            | 128          |
| FireSaber-KEM            | 256          |
| FireSaber-KEM            | 512          |
| FireSaber-KEM            | 1024         |
| FireSaber-KEM            | 2048         |
| Kyber1024                | 569818       |
| Kyber1024                | 4            |
| Kyber1024                | 8            |
| Kyber1024                | 16           |
| Kyber1024                | 32           |
| Kyber1024                | 64           |
| Kyber1024                | 128          |
| Kyber1024                | 256          |
| Kyber1024                | 512          |
| Kyber1024                | 1024         |
| Kyber1024                | 2048         |
| Kyber12                  | 498940       |
| Kyber12                  | 4            |
| Kyber12                  | 8            |
| Kyber12                  | 2            |
| Kyber12                  | 4            |
| Kyber12                  | 8            |

(Continued.) Time (in us) for each round on GAKE protocol.
### TABLE 14. (Continued.) Time (in us) for each round on GAKE protocol.

| Protocol      | Round | Time (in us) |
|---------------|-------|--------------|
| Kyber512      | 16    | 1857542      |
| Kyber512      | 32    | 4882138      |
| Kyber512      | 64    | 16119412     |
| Kyber512      | 128   | 56363002     |
| Kyber512      | 256   | 229336298    |
| Kyber512      | 512   | 839032092    |
| Kyber512      | 1024  | 3444274237   |
| Kyber512      | 2048  | 371370860    |
| Kyber768      | 2     | 577860       |
| Kyber768      | 4     | 761194       |
| Kyber768      | 8     | 1230752      |
| Kyber768      | 16    | 2316806      |
| Kyber768      | 32    | 6308418      |
| Kyber768      | 64    | 2085368      |
| Kyber768      | 128   | 75010626     |
| Kyber768      | 256   | 270613704    |
| Kyber768      | 512   | 1046487012   |
| Kyber768      | 1024  | 48885311     |
| Kyber768      | 2048  | 3548206715   |
| LightSaber-KEM| 2     | 496816       |
| LightSaber-KEM| 4     | 629846       |
| LightSaber-KEM| 8     | 961000       |
| LightSaber-KEM| 16    | 1887810      |
| LightSaber-KEM| 32    | 5124908      |
| LightSaber-KEM| 64    | 16425678     |
| LightSaber-KEM| 128   | 57217068     |
| LightSaber-KEM| 256   | 225489147    |
| LightSaber-KEM| 512   | 847092706    |
| LightSaber-KEM| 1024  | 3389600373   |
| LightSaber-KEM| 2048  | 211324268    |
| NTRU-HPS-2048-509| 2 | 981000       |
| NTRU-HPS-2048-509| 4 | 1245528      |
| NTRU-HPS-2048-509| 8 | 2446550      |
| NTRU-HPS-2048-509| 16| 4932194      |
| NTRU-HPS-2048-509| 32| 14072786     |
| NTRU-HPS-2048-509| 64| 44271612     |
| NTRU-HPS-2048-509| 128| 175005570    |
| NTRU-HPS-2048-509| 256| 600702206    |
| NTRU-HPS-2048-509| 512| 2349192856   |
| NTRU-HPS-2048-509| 1024| 1539923810   |
| NTRU-HPS-2048-509| 2048| 2590192843   |
| NTRU-HPS-2048-677| 2 | 1091314      |
| NTRU-HPS-2048-677| 4 | 1562378      |
| NTRU-HPS-2048-677| 8 | 3215126      |
| NTRU-HPS-2048-677| 16| 6950574      |
| NTRU-HPS-2048-677| 32| 18987826     |
| NTRU-HPS-2048-677| 64| 57640154     |
| NTRU-HPS-2048-677| 128| 208377538    |
| NTRU-HPS-2048-677| 256| 792694984    |
| NTRU-HPS-2048-677| 512| 298360364    |
| NTRU-HPS-2048-677| 1024| 1068468532   |
| NTRU-HPS-2048-677| 2048| 3842053389   |
| NTRU-HPS-4096-821| 2 | 1323266      |
| NTRU-HPS-4096-821| 4 | 2123426      |
### TABLE 14. (Continued.) Time (in us) for each round on GAKE protocol.

| Parameter set | 4 | 8 | 16 | 32 | 64 |
|---------------|---|---|----|----|----|
| Classic-McEliece-348644 | 92788.0 | 178346.0 | 289386.0 | 460470.0 | 1274000.0 |
| Classic-McEliece-400K | 60287.0 | 117905.0 | 230679.0 | 468796.0 | 936692.0 |
| Classic-McEliece-40896 | 225094.0 | 460915.0 | 921878.0 | 1861680.0 | 3639741.0 |
| Classic-McEliece-408964 | 1754855.0 | 3601946.0 | 7195912.0 | 1482067.0 | 2953638.0 |
| Classic-McEliece-648128 | 3066597.0 | 7000197.0 | 1508732.0 | 2669367.0 | 5362399.0 |
| Classic-McEliece-648128F | 2250711.0 | 4572701.0 | 9004607.0 | 1729725.0 | 3601801.0 |
| Classic-McEliece-648119 | 3603944.0 | 6396253.0 | 12348363.0 | 25451437.0 | 5234633.0 |
| Classic-McEliece-648119F | 2313379.0 | 4358284.0 | 8684243.0 | 1734679.0 | 3464277.0 |
| Classic-McEliece-819228 | 3447378.0 | 6520993.0 | 1331885.0 | 2583188.0 | 5366245.0 |
| Classic-McEliece-819228F | 2377773.0 | 4759007.0 | 9531537.0 | 1912292.0 | 3820634.0 |
| FireSaber-KEM | 1911.0 | 3795.0 | 8784.0 | 21540.0 | 43840.0 |
| KyBer/204 | 15897.0 | 2870.0 | 6289.0 | 15308.0 | 30317.0 |
| KyBer/262 | 948.0 | 1550.0 | 3522.0 | 7600.0 | 15200.0 |
| LightSaber-KEM | 1029.0 | 1935.0 | 4177.0 | 9541.0 | 19094.0 |
| NTRU-HPS-2048-909 | 1221.0 | 2311.0 | 4201.0 | 9588.0 | 19589.0 |
| NTRU-HPS-2048-677 | 1541.0 | 2678.0 | 5014.0 | 11016.0 | 22432.0 |
| NTRU-HPS-6254-8121 | 1859.0 | 3406.0 | 7496.0 | 16530.0 | 33060.0 |
| NTRU-HRSS-701 | 1544.0 | 2908.0 | 5464.0 | 12351.0 | 25465.0 |
| Saber-KEM | 1392.0 | 2814.0 | 5109.0 | 11548.0 | 23094.0 |

### TABLE 15. Total running time (in us) of the GAKE protocol for each number of parties.

| Parameter set | 2 | 4 | 8 | 16 | 32 | 64 |
|---------------|---|---|---|----|----|----|
| Classic-McEliece-348644 | 892788.0 | 178346.0 | 289386.0 | 460470.0 | 1274000.0 | 2462760.0 |
| Classic-McEliece-400K | 60287.0 | 117905.0 | 230679.0 | 468796.0 | 936692.0 | 1873384.0 |
| Classic-McEliece-40896 | 225094.0 | 460915.0 | 921878.0 | 1861680.0 | 3639741.0 | 7279482.0 |
| Classic-McEliece-408964 | 1754855.0 | 3601946.0 | 7195912.0 | 1482067.0 | 2953638.0 | 5907276.0 |
| Classic-McEliece-648128 | 3066597.0 | 7000197.0 | 1508732.0 | 2669367.0 | 5362399.0 | 10724798.0 |
| Classic-McEliece-648128F | 2250711.0 | 4572701.0 | 9004607.0 | 1729725.0 | 3601801.0 | 7203602.0 |
| Classic-McEliece-648119 | 3603944.0 | 6396253.0 | 12348363.0 | 25451437.0 | 5234633.0 | 10469266.0 |
| Classic-McEliece-648119F | 2313379.0 | 4358284.0 | 8684243.0 | 1734679.0 | 3464277.0 | 6928554.0 |
| Classic-McEliece-819228 | 3447378.0 | 6520993.0 | 1331885.0 | 2583188.0 | 5366245.0 | 10732370.0 |
| Classic-McEliece-819228F | 2377773.0 | 4759007.0 | 9531537.0 | 1912292.0 | 3820634.0 | 7641267.0 |
| FireSaber-KEM | 1911.0 | 3795.0 | 8784.0 | 21540.0 | 43840.0 | 87680.0 |
| KyBer/204 | 15897.0 | 2870.0 | 6289.0 | 15308.0 | 30317.0 | 60634.0 |
| KyBer/262 | 948.0 | 1550.0 | 3522.0 | 7600.0 | 15200.0 | 30317.0 |
| LightSaber-KEM | 1029.0 | 1935.0 | 4177.0 | 9541.0 | 19094.0 | 38188.0 |
| NTRU-HPS-2048-909 | 1221.0 | 2311.0 | 4201.0 | 9588.0 | 19589.0 | 39178.0 |
| NTRU-HPS-2048-677 | 1541.0 | 2678.0 | 5014.0 | 11016.0 | 22432.0 | 44864.0 |
| NTRU-HPS-6254-8121 | 1859.0 | 3406.0 | 7496.0 | 16530.0 | 33060.0 | 66120.0 |
| NTRU-HRSS-701 | 1544.0 | 2908.0 | 5464.0 | 12351.0 | 25465.0 | 50930.0 |
| Saber-KEM | 1392.0 | 2814.0 | 5109.0 | 11548.0 | 23094.0 | 46188.0 |
### TABLE 16. CPU cycles for each operation on AKE scheme between FSXY and FOAKE transformations.

| Parameter set | type   | algA   | algB   | init  |
|---------------|--------|--------|--------|-------|
| Kyber1024     | fo-ake | 61272  | 164921 | 103966|
| Kyber1024     | fsxy   | 157702 | 275832 | 152475|
| Kyber512      | fo-ake | 24129  | 64225  | 40741 |
| Kyber512      | fsxy   | 74852  | 132814 | 75748 |
| Kyber768      | fo-ake | 42554  | 114265 | 74408 |
| Kyber768      | fsxy   | 116833 | 208207 | 112594|

### TABLE 17. CPU cycles for each operation on GAKE protocol between FSXY and FOAKE transformations.

| Parameter set | type   | n     | init     | round12 | round3 | round4 |
|---------------|--------|-------|----------|---------|--------|--------|
| Kyber1024     | fo-ake | 2     | 352728   | 1455104 | 190636 | 240810|
| Kyber1024     | fo-ake | 4     | 435276   | 2878382 | 322330 | 932940|
| Kyber1024     | fo-ake | 8     | 957022   | 5556222 | 562612 | 3529734|
| Kyber1024     | fo-ake | 16    | 2115684 | 11209454 | 1106828 | 141031608 |
| Kyber1024     | fo-ake | 32    | 6124772 | 21877848 | 2261222 | 55572826 |
| Kyber1024     | fo-ake | 64    | 20678658 | 44261601 | 6373372 | 225249026 |
| Kyber1024     | fo-ake | 128   | 77548204 | 88040096 | 15884218 | 899885456 |
| Kyber1024     | fo-ake | 256   | 285147300 | 165981610 | 35437206 | 3596092132 |
| Kyber1024     | fo-ake | 512   | 1094344170 | 342508865 | 128393108 | 1828211446 |
| Kyber1024     | fo-ake | 1024  | 4281287051 | 672156610 | 504682447 | 1344193494 |
| Kyber1024     | fo-ake | 2048  | 4063233753 | 1387462198 | 3724341955 | 3413607674 |
| Kyber1024     | fsxy   | 2     | 569818   | 2996082 | 249696 | 294682 |
| Kyber1024     | fsxy   | 4     | 803884   | 5137618 | 389362 | 1109024 |
| Kyber1024     | fsxy   | 8     | 1362792  | 9885126 | 685716 | 436241 |
| Kyber1024     | fsxy   | 16    | 2754268 | 19360662 | 1316186 | 17536258 |
| Kyber1024     | fsxy   | 32    | 7859682  | 38739572 | 2498038 | 67571944 |
| Kyber1024     | fsxy   | 64    | 26334516 | 75688836 | 5218650 | 272631328 |
| Kyber1024     | fsxy   | 128   | 90258852 | 150989577 | 11764600 | 1125447114 |
| Kyber1024     | fsxy   | 256   | 347379710 | 30596352 | 30854094 | 203874448 |
| Kyber1024     | fsxy   | 512   | 1362064436 | 626489624 | 118268879 | 1450552719 |
| Kyber1024     | fsxy   | 1024  | 1306379099 | 1201640936 | 376335071 | 463196417 |
| Kyber1024     | fsxy   | 2048  | 143981347 | 2418133628 | 1432070771 | 1939460493 |
| Kyber512      | fo-ake | 2     | 172880   | 597112 | 140126 | 91336 |
| Kyber512      | fo-ake | 4     | 244948   | 1108004 | 265502 | 349942 |
| Kyber512      | fo-ake | 8     | 457206   | 2493920 | 293888 | 1389898 |
| Kyber512      | fo-ake | 16    | 1126550  | 4318414 | 557998 | 7792834 |
| Kyber512      | fo-ake | 32    | 3547112  | 8893216 | 1156686 | 2306240 |
| Kyber512      | fo-ake | 64    | 11532644 | 17044954 | 2675020 | 87354510 |
| Kyber512      | fo-ake | 128   | 42822396 | 33423604 | 7119092 | 358312410 |
| Kyber512      | fo-ake | 256   | 162999452 | 64532826 | 21795546 | 1447317430 |
| Kyber512      | fo-ake | 512   | 642910434 | 134759080 | 89890960 | 1794799431 |
| Kyber512      | fo-ake | 1024  | 26826064673 | 260979316 | 342297686 | 1200990103 |
| Kyber512      | fo-ake | 2048  | 1555164747 | 527148304 | 1803526899 | 3942602428 |
| Kyber512      | fsxy   | 2     | 498940   | 1391742 | 169900 | 133814 |
| Kyber512      | fsxy   | 4     | 614774   | 2540226 | 238504 | 566142 |
| Kyber512      | fsxy   | 8     | 971834   | 5323738 | 412224 | 2415210 |
| Kyber512      | fsxy   | 16    | 1857542  | 9684790 | 696384 | 7663526 |
| Kyber512      | fsxy   | 32    | 4882138  | 19159176 | 1431924 | 30344334 |
| Kyber512      | fsxy   | 64    | 16119412 | 36680564 | 2909726 | 119957652 |
| Kyber512      | fsxy   | 128   | 56363002 | 73662228 | 6650430 | 496642586 |
| Kyber512      | fsxy   | 256   | 229336298 | 145675628 | 17999734 | 2023890528 |
| Kyber512      | fsxy   | 512   | 839032092 | 308417712 | 95626794 | 4235688616 |
### TABLE 17. (Continued.) CPU cycles for each operation on GAKE protocol between FSXY and FOAKE transformations.

| Protocol | Operation | CPU Cycles |
|----------|-----------|------------|
| Kyber512 | fsxy      | 1024       |
| Kyber512 | fsxy      | 2048       |
| Kyber768 | fo-ake    | 2          |
| Kyber768 | fo-ake    | 4          |
| Kyber768 | fo-ake    | 8          |
| Kyber768 | fo-ake    | 16         |
| Kyber768 | fo-ake    | 32         |
| Kyber768 | fo-ake    | 64         |
| Kyber768 | fo-ake    | 128        |
| Kyber768 | fo-ake    | 256        |
| Kyber768 | fo-ake    | 512        |
| Kyber768 | fo-ake    | 1024       |
| Kyber768 | fo-ake    | 2048       |
| Kyber768 | fsxy      | 2          |
| Kyber768 | fsxy      | 4          |
| Kyber768 | fsxy      | 8          |
| Kyber768 | fsxy      | 16         |
| Kyber768 | fsxy      | 32         |
| Kyber768 | fsxy      | 64         |
| Kyber768 | fsxy      | 128        |
| Kyber768 | fsxy      | 256        |
| Kyber768 | fsxy      | 512        |
| Kyber768 | fsxy      | 1024       |

### TABLE 18. Mean running time that runs every party (in us) of the GAKE protocol as a function of the number of parties.

| Parameter set | Parties | fo-ake | fsxy |
|---------------|---------|--------|------|
| Kyber1024     | 2       | 433.0  | 793.5|
| Kyber1024     | 4       | 441.25 | 717.5|
| Kyber1024     | 8       | 511.625| 786.125|
| Kyber1024     | 16      | 688.875| 988.0|
| Kyber1024     | 32      | 1034.96875| 1406.65625|
| Kyber1024     | 64      | 1787.796875| 2289.96875|
| Kyber1024     | 128     | 3259.359375| 4154.8203125|
| Kyber1024     | 256     | 6152.765625| 7809.76953125|
| Kyber1024     | 512     | 12266.849609375| 15625.9609375|
| Kyber1024     | 1024    | 25217.29296875| 32006.654296875|
| Kyber1024     | 2048    | 54962.12158203125| 69890.1923828125|
| Kyber512      | 2       | 194.0  | 424.0|
| Kyber512      | 4       | 190.25 | 382.5|
| Kyber512      | 8       | 223.625| 440.25|
| Kyber512      | 16      | 332.8125| 480.0|
| Kyber512      | 32      | 442.03125| 673.4375|
| Kyber512      | 64      | 715.015625| 1059.0|
| Kyber512      | 128     | 1331.2890625| 1908.8984375|
| Kyber512      | 256     | 2556.94140625| 3642.390625|
| Kyber512      | 512     | 5242.50390625| 7364.724609375|
| Kyber512      | 1024    | 11399.52734375| 16064.0166015625|
| Kyber512      | 2048    | 26556.4296875| 38339.2978515625|
| Kyber768      | 2       | 305.5  | 584.0|
| Kyber768      | 4       | 321.75 | 589.5|
TABLE 18. (Continued.) Mean running time that runs every party (in us) of the GAKE protocol as a function of the number of parties.

| Kyber768 | 8     | 364.625 | 596.625 |
|---------|-------|---------|---------|
| Kyber768 | 16    | 598.25 | 753.5625 |
| Kyber768 | 32    | 746.96875 | 1083.34375 |
| Kyber768 | 64    | 1234.828125 | 1732.203125 |
| Kyber768 | 128   | 2315.6640625 | 3081.3125 |
| Kyber768 | 256   | 4355.17578125 | 5854.5078125 |
| Kyber768 | 512   | 8726.408203125 | 11618.892578125 |
| Kyber768 | 1024  | 18668.8976490375 | 24514.66796875 |
| Kyber768 | 2048  | 40742.357421875 | 54436.67626953125 |

ACKNOWLEDGMENT
The authors would like to thank María I. González Vasco for her help during the process of elaboration of the manuscript, through fruitful discussions and useful comments and suggestions.

REFERENCES

[1] A. Fujioka, K. Takashima, and K. Yoneyama, “One-round authenticated group key exchange from isogenies,” in Proc. ProvSec, in Lecture Notes in Computer Science, vol. 11821. Cham, Switzerland: Springer, 2019, pp. 330–338.

[2] H. B. Hougaard and A. Miyaji, “Authenticated logarithmic-order supersingular isogeny group key exchange,” Int. J. Inf. Secur., vol. 21, pp. 207–221, May 2021.

[3] D. Apon, D. Dachman-Soled, H. Gong, and J. Katz, “Constant-round group key exchange from the ring-LWE assumption,” in PQCrypt, in Lecture Notes in Computer Science, vol. 11505. Cham, Switzerland: Springer, 2019, pp. 189–205.

[4] R. Choi, D. Hong, and K. Kim, “Constant-round dynamic group key exchange from RLWE assumption,” Cryptol. ePrint Arch., Paper 2020/035, vol. 2020, p. 35, 2020.

[5] R. Choi, D. Hong, S. Han, S. Baek, W. Kang, and K. Kim, “Design and implementation of constant-round dynamic group key exchange from RLWE,” IEEE Access, vol. 8, pp. 94610–94630, 2020.

[6] R. Choi, D. Hong, and K. Kim, “Implementation of tree-based dynamic group key exchange with newhope,” in Proc. Int. Conf. Netw. Syst. Secur., vol. 2020, p. 35, 2020.

[7] K. Takashima, “Post-quantum constant-round group key exchange from static assumptions,” in Proc. Int. Symp. Math., Quantum Theory, Cryptogen: Singapore: Springer, 2021, p. 251.

[8] E. Persichetti, R. Steinwandt, and A. S. Corona, “From key encapsulation to authenticated group key establishment—A compiler for post-quantum primitives,” Entropy, vol. 21, no. 12, p. 1183, Nov. 2019.

[9] M. I. G. Vasco, L. A. P. D. Pozo, and R. Steinwandt, “Group key establishment in a quantum-future scenario,” Informatica, vol. 31, no. 4, pp. 751–768, 2020.

[10] H. B. Hougaard and A. Miyaji, “Group key compiler extractors from generic key exchanges,” in Proc. Int. Conf. Netw. Syst. Secur. Cham, Switzerland: Springer, 2021, pp. 162–184.

[11] J. I. E. Pablos, M. I. G. Vasco, M. E. Marriaga, and Á. L. P. D. Pozo, “Compiled constructions towards post-quantum group key exchange: A design from kyber,” Mathematics, vol. 8, no. 10, p. 1853, Oct. 2020.

[12] J. Katz and M. Yung, “Scalable protocols for authenticated group key exchange,” in Advances in Cryptology—CRYPTO 2003, vol. 2729, D. Boneh, Ed. Santa Barbara, CA, USA: Springer, Aug. 2003, pp. 110–125.

[13] E. Alkim, L. Ducas, T. Poppelmann, and P. Schwabe, “Post-quantum key exchange—a new hope,” in Proc. 25th USENIX Secur. Symp. (USENIX Security), 2016, pp. 327–343.

[14] M. Abdalla, J. Bohli, M. I. G. Vasco, and R. Steinwandt, “(Password) authenticated key establishment: From 2-party to group,” in Proc. TCC, in Lecture Notes in Computer Science, vol. 4392. Berlin, Germany: Springer, 2007, pp. 499–514.

[15] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, “CRYSTALS—Kyber: ACCA-secure module-lattice-based KEM,” in Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&IP), Apr. 2018, pp. 353–367.

[16] M. Burmester and Y. Desmedt, “A secure and efficient conference key distribution system,” in Proc. EUROCRYPT, in Lecture Notes in Computer Science, vol. 950. Berlin, Germany: Springer, 1994, pp. 275–286.

[17] A. Fujioka, K. Suzuki, K. Xagawa, and K. Yoneyama, “Practical and post-quantum authenticated key exchange from one-way secure key encapsulation mechanism,” in Proc. 5th ACM SIGSAC Symp. Inf., Comput. Commun. Secur., 2013, pp. 83–94.

[18] K. Hövelmanns, E. Kiltz, S. Schäge, and D. Unruh, “Generic authenticated key exchange in the quantum random Oracle model,” Cryptol. ePrint Arch., Paper 2018/928, vol. 2018, p. 928, 2018.

[19] K. Hövelmanns, E. Kiltz, S. Schäge, and D. Unruh, “Generic authenticated key exchange in the quantum random Oracle model,” in Public-Key Cryptography—PKC 2020, A. Kiayias, M. Kohlweiss, P. Wallden, and V. Zikas, Eds. Cham, Switzerland: Springer, 2020, pp. 389–422.

[20] T. Saito, K. Xagawa, and T. Yamakawa, “Tightly-secure key-encapsulation mechanism in the quantum random Oracle model,” Cryptol. ePrint Arch., Paper 2017/1005, 2017.

[21] Post-Quantum Cryptography. Security (Evaluation Criteria), NIST, Gaithersburg, MD, USA. [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/evaluation-criteria/security-evaluation-criteria

[22] Post-Quantum Cryptography. Round 3 Submissions, NIST, Gaithersburg, MD, USA. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography-post-quantum-cryptography-standardization/round-3-submissions

[23] M. R. Albrecht, D. J. Bernstein, T. Chou, C. Cid, J. Gilch, T. Lange, V. Maram, J. V. Maurich, R. Misoczki, R. Niederhagen, K. G. Paterson, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier, J. Szefer, C. J. Thyai, M. Tomlinson, and W. Wang, “Classic McEliece: Conservative code-based cryptography,” NIST, Tech. Rep., 2020. [Online]. Available: https://classic.mceliece.org/nist/mceliece-postquantum-cryptography.pdf

[24] H. Niederreiter, “Knapsack-type cryptosystems and algebraic coding theory,” Problems Control Inf. Theory, vol. 15, no. 2, pp. 157–166, 1960.

[25] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, “CRYSTALS-kycer algorithm specifications and supporting documentation,” NIST, Tech. Rep., 2021. [Online]. Available: https://pq-cry.stals.org/kyber/data/kyber-specification-round3-20210804.pdf

[26] Post-Quantum Cryptography. Round 1 Submissions, NIST, Gaithersburg, MD, USA.

[27] J. Hoffstein, J. Pipher, and J. H. Silverman, “NTRU: A ring-based public key cryptosystem,” in Algorithmic Number Theory (Lecture Notes in Computer Science), vol. 1423. J. Buhler, Ed. Portland, OR, USA: Springer, Jun. 1998, pp. 267–288.
[28] T. Saito, K. Xagawa, and T. Yamakawa, “Tightly-secure key-encapsulation mechanism in the quantum random Oracle model,” in Proc. Annu. Int. Conf. Theory Appl. Cryptograph. Techn. Cham, Switzerland: Springer, 2018, pp. 520–551.

[29] J.-P. D’Anvers, A. Karmakar, S. S. Roy, and F. Vercauteren, “Saber: Module-LWR based key exchange. CPA-secure encryption and CCA-secure KEM.” Cryptol. ePrint Arch., Paper 2018/230, p. 230, 2018.

[30] C. Boyd, Y. Cliff, J. G. Nieto, and K. G. Paterson, “Efficient one-round key exchange in the standard model,” in Proc. ACISP, in Lecture Notes in Computer Science, vol. 5107. Berlin, Germany: Springer, 2008, pp. 69–83.

[31] A. Fujioka, K. Suzuki, K. Xagawa, and K. Yoneyama, “Strongly secure authenticated key exchange from factoring, codes, and lattices,” Des., Codes Cryptogr., vol. 76, no. 3, pp. 469–504, Sep. 2015.

[32] R. Canetti and H. Krawczyk, “Analysis of key-exchange protocols and their use for building secure channels,” in Proc. EUROCRYPT, in Lecture Notes in Computer Science, vol. 2045. Berlin, Germany: Springer, 2001, pp. 453–474.

[33] R. Cramer and V. Shoup, “Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack,” SIAM J. Comput., vol. 33, no. 1, pp. 167–226, 2003.

[34] M. Bellare, D. Pointcheval, and P. Rogaway, “Authenticated key exchange secure against dictionary attacks,” in Advances in Cryptology—EUROCRYPT 2000, vol. 1807, B. Preneel, Ed. Bruges, Belgium: Springer, May 2000, pp. 139–155.

[35] J.-M. Bohli, M. I. G. Vasco, and R. Steinwandt, “Secure group key establishment revisited,” Int. J. Inf. Secur., vol. 6, no. 4, pp. 243–254, Jun. 2007.

[36] Algorithms in LibOQS, Open Quantum Safe, 2022.

[37] D. Stebila and M. Mosca, “Post-quantum key exchange for the internet and the open quantum safe project,” in Proc. SAC, in Lecture Notes in Computer Science, vol. 10532. Cham, Switzerland: Springer, 2016, pp. 14–37.

[38] Cryptography and SSL/TLS Toolkit, OpenSSL, 2022.

[39] What is the Windows Subsystem for Linux?, Microsoft, 2022.

[40] Usage Limits, Billing, and Administration, GitHub Actions, 2022.

[41] M. L. Waskom, “Seaborn: Statistical data visualization,” J. Open Source Softw., vol. 6, no. 60, p. 3021, 2021.

[42] Storing Workflow Data as Artifacts, GitHub Actions, 2022.