Hybrid HPKE

November 19, 2021

1 Introduction

The use of public key cryptography ensures the secure data exchange between two communication parties. However, the structure of the asymmetric (public key) schemes and their high computational cost does not make them practical for encryption of large amount of exchanged data. For instance, the highly deployed RSA-encrypt scheme features high computational cost and imposes a limitation of the plaintext length, bounded by the modulus value, where it is further decreased depending on the selected padding mode. The encryption of large data requires multiple executions of the RSA-encrypt procedure, thus, converts the encryption into a bottle-neck for the secure data transmission. Therefore, the use of hybrid cryptographic protocols which make use of both - asymmetric and symmetric primitives, becomes the most preferable option for secure data exchange. The asymmetric schemes ensure the secure data communication aiming at obtaining a shared secret, which is used to derive a key for symmetric low-cost encryption schemes. The symmetric schemes ensure the low computational cost while executing the message encryption and decryption with symmetric cryptographic primitives.

The use of hybrid public key encryption scheme is not a new cryptographic approach and has been undergoing several different design implementations. There are three approaches for the communication parties to obtain a shared secret value, discussed briefly below.

The original idea behind the hybrid public key encryption schemes is to obtain a symmetric key, encrypt arbitrary-sized plaintexts via symmetric scheme and **encrypt** the key using an asymmetric primitive (as shown in Figure 1). This approach is used in hybrid public key encryption schemes based on RSA-encrypt asymmetric primitive, where the symmetric key is generated by one of the communication parties, used for encryption, encrypted under the public key of the recipient, and then send along with the ciphertext. On the other side, the recipient decrypts the symmetric key using his/her secret key and use the result to reveal the original content of the message.

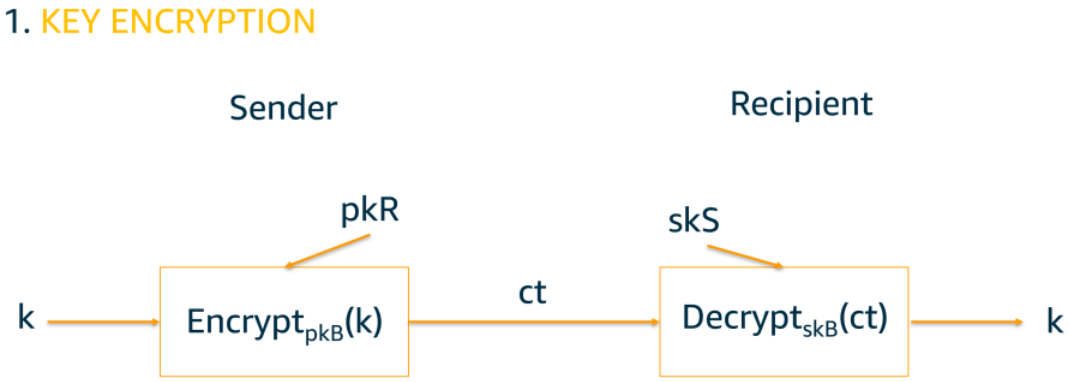


Figure 1: Approach 1: Key encryption.

Another approach for implementing a public key encryption scheme is to use a **key agreement** primitive (as shown in Figure 2) (such as Diffie-Hellman [17]) to agree on a shared secret and use it as a symmetric key value for inexpensive encryption/decryption. The widely used Elliptic Curve Integrated Encryption System (ECIES) [23] is based on key agreement underlying protocols. However, due to the popularity of the ECIES scheme, it lacks a single standard specification (ANSI X9.63(ECIES)- [3], IEEE 1363a [12], ISO/IEC 18033-2 [24], SECG SEC 1 [18]) where some of the standards allow the use of weekend (or even broken: ANSI-X9.63-KDF with SHA-1) underlying primitives. Moreover, the key agreement schemes lack proof of IND-CCA2 (attack under adaptive chosen ciphertext) security. Finally, the ECIES scheme does not allow easy integration of the highly demanded post-quantum algorithms, restraining the protocol extension of use

cases.

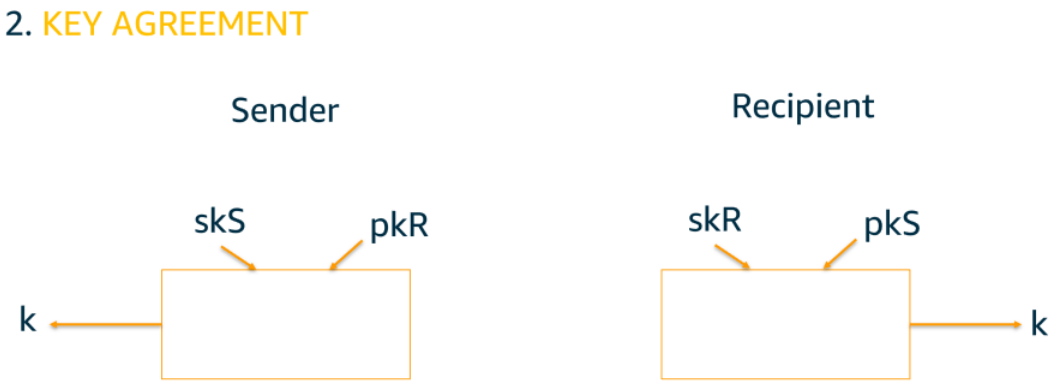


Figure 2: Approach 2: Key Agreement.

The steps required for the execution of a given key agreement based protocol are summarized as follow (Assuming Alice is the Sender and Bob is the receiver from Figure 2):

- Alice generates a pseudo-random symmetric key of fixed length.
- Alice uses this symmetric key to encrypt the plaintext message that she want to communicate to Bob.
- Alice encapsulate the symmetric key using Bob's public key values.
- Bob receives the encapsulated symmetric key along with a ciphertext message(s). He decapsulates the value of the symmetric key.
- Bob uses the decapsulated key to decrypt the value of the communicated message.

To overcome the security and implementation lacks of the ECIES, there has been a newly standardized scheme called Hybrid Public Key Encryption (HPKE) [6]. The Hybrid Public Key Encryption standardizes the integration of a Key Encapsulation Mechanism (KEM) along with a Key Derivation Function (KDF), and Authenticated Encryption with Additional Data (AEAD) scheme into a protocol that combines the use of symmetric and asymmetric cryptographic primitives. The newly defined hybrid public key encryption (HPKE) standard uses the "generate the symmetric key and its encapsulation with the public key" (as shown in Figure 3) approach to generate the symmetric key and encrypt the data using it. Based on IND-CCA2 robust KEM schemes, offering easy integration of post-quantumness of the protocol, allowing arbitrary-size plaintext consumption and ensuring a single specification, the HPKE protocol has become focus of several academia and industry teams, aiming to update the hybrid public key encryption schemes used so far. The integration of Key Encapsulation Mechanism allows the combination of (a)symmetric schemes, ensuring efficient performance, rises the robustness of the protocol adding IND-CCA2 security and removes the challenge of adding PQ algorithms.

The execution flow of the HPKE standard, base of this work, is described as follow (Assuming Alice is the Sender and Bob is the Recipient from Figure 3):

- Alice uses a KEM (as specified later in this work) to generate and encapsulate shared secret based on her secret key and Bob's public key.
- Alice applies a KDF to the shared secret to obtain the symmetric key and uses it to encrypt the plaintext messages.
- Bob receives the encapsulated shared secret along with Alice's public key and uses them to obtain and verify the shared secret.
- Bob applies the same KDF to the shared secret to obtain the value of the symmetric key and uses it to decrypt the ciphertext messages.

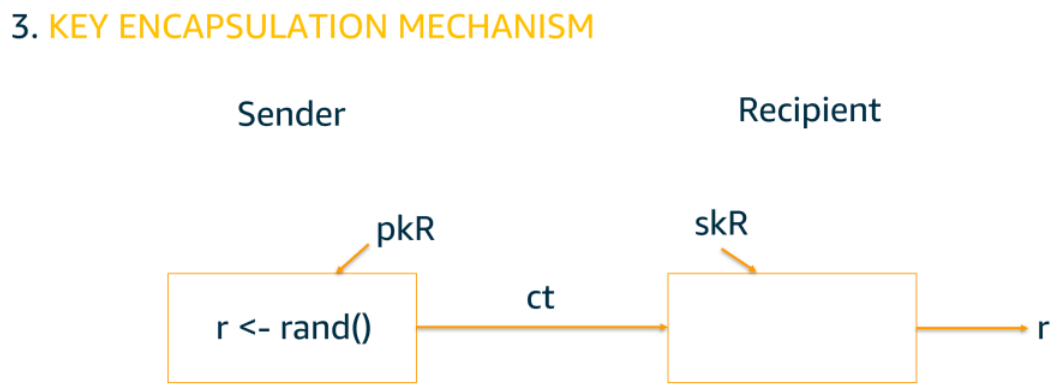


Figure 3: Approach 3: Key encapsulation.

The recent HPKE standard has been discussed in [1] and [14] where the authors present a description and analysis of the HPKE standard and its operation modes. Detailed description of the HPKE algorithm in its base mode is illustrated in Figure 4. The complete execution flow of the HPKE protocol, using one of the elliptic curve family as an underlying primitive is described as follow:

- Encapsulate
  - Alice generates an ephemeral key pair.
  - Alice uses her secret key and Bob's public key (applying a KEM) to obtain a shared secret.
  - Alice applied a KDF to obtain a common value `zz`.
- Key Schedule
  - Alice uses the common value `zz` to extract a fixed length pseudo-random key (prk) value.
  - Alice expands the prk to a desired length and obtain a symmetric key and nonce for the seal (encryption) step.
- Seal
  - Alice applies an encryption AEAD function to encrypt the plaintext messages using the symmetric key and the nonce.
- Expand
  - Alice expands again the value of the symmetric key to obtain new common value, stored into the AEAD context variable.

On the other side of the communication Bob performs the mirrored sequence of operations to decapsulate Alice's public key, obtain the shared secret, the symmetric key and decrypt the ciphertext messages. His steps are described as follow:

- Decapsulate
  - Bob uses his secret key and Alice's public key (applying a KEM protocol) to obtain a shared secret.
  - Bob applies a KDF to obtain a common value `zz`.
- Key Schedule
  - Bob uses the common value `zz` to extract a fixed length pseudo-random key (prk) value.
  - Bob expands the prk to a desired length and obtain secret symmetric key and nonce for the open (decryption) step.
- Open

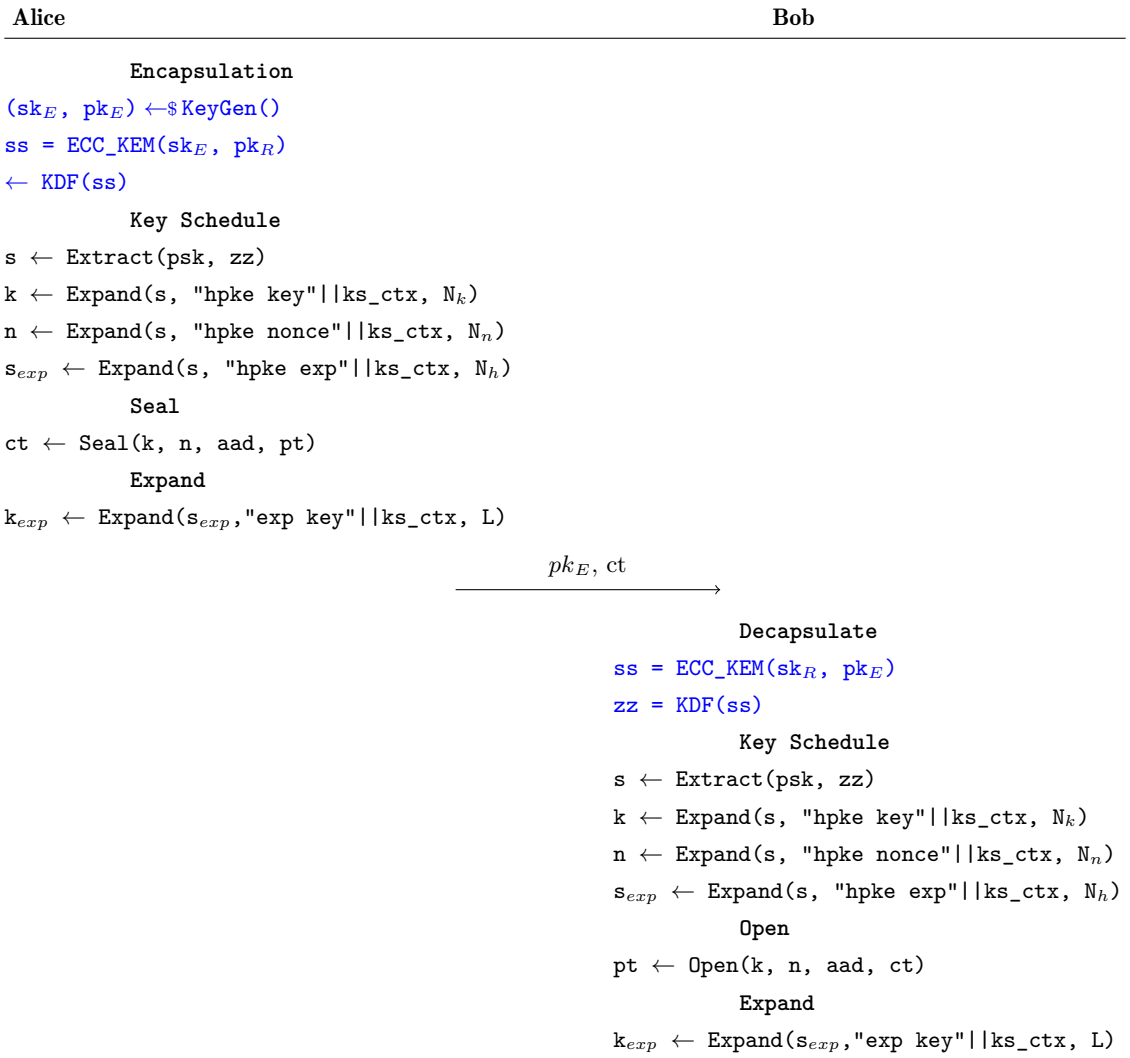


Figure 4: Detailed description of the Hybrid Public Key Encryption (HPKE) protocol [14].

  - Bob applies a decryption AEAD function to decrypt the ciphertext messages using the symmetric key and the nonce.
- Expand
  - Bob expands again the value of the symmetric key to obtain new common value, stored into the AEAD context variable.

The HPKE [5], standardized by the Crypto Forum Research Group (CFRG) within the Internet Research Task Force (IRTF), integrates three main cryptographic primitives to ensure the functionality of the protocol: Key Encapsulation Mechanism (KEM), Key Derivation Function (KDF) and Authenticated Encryption with Additional Data (AEAD). The current implementation of the HPKE within AWS-LC is based on the ECDHKEM using Curve25519.

The Marshal and Unmarshal functions, from [14], are defined as SerializePublicKey(pk) and DeserializePublicKey(enc) in [6], where it produce a byte string encoding the public key. These functions are

omitted in the implementation design based on Curve25519 and Curve448 since these schemes already use fixed-length byte strings for public keys.

1.1 Transform ECDH to a KEM

The ECDH protocol has a symmetric shape, where both communication parties execute equivalent set of operations. In Figure 5, the execution flow of the ECDH key exchange is represented. Specifically, for the execution of the ECDH protocol the sender - Alice and the recipient - Bob perform the same operation (point multiplication) on a different input data. First, Alice and Bob generate set of public and private keys. First, they generate their secret data (sk_A and sk_B, respectively) as pseudo-random numbers. Later, they obtain their public keys (pk_A and pk_B, respectively) performing point multiplication among their own secret key value and a public parameter generator point $\mathtt{G}$. For obtaining a shared secret Alice and Bob use their secret key along with the other party public key and perform another point multiplication. In particular, Alice computes Equation 1 and Bob computes Equation 2.

$$[\mathsf{sk}_A] \cdot \mathsf{pk}_B = [\mathsf{sk}_A] \cdot [\mathsf{sk}_B] \cdot \mathtt{G} \tag{1}$$

$$[\mathsf{sk}_B] \cdot \mathsf{pk}_A = [\mathsf{sk}_B] \cdot [\mathsf{sk}_A] \cdot \mathtt{G} \tag{2}$$

$$[\mathsf{sk}_A] \cdot [\mathsf{sk}_B] \cdot \mathtt{G} = [\mathsf{sk}_B] \cdot [\mathsf{sk}_A] \cdot \mathtt{G} \tag{3}$$

Based on Equation 3 they reach final value as a shared secret for further performance efficient symmetrically encrypted communication. In the scope of AWS-LC the ECDH KEM protocol is performed using Curve25519 where the elliptic curve used is defined as $E : y^2 = (x^3 + 48666x^2 + x)$ over the finite filed F_p with $p = 2^{255} - 19$.

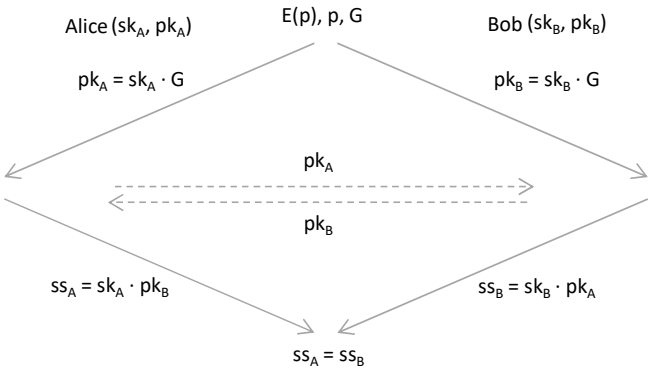


Figure 5: Graphical representation of ECDH algorithm.

The integration of an ECDH scheme into the HPKE protocol requires to first convert it into a KEM, where the ECDH KEM is illustrated in Figure 6 and is defined by the underlying elliptic curve, base of the ECDH, and a Key Derivation Function (KDF), used to derive fixed-length symmetric key k.

The RSA, widely deployed, and (EC)DH KEM, performance and memory efficient, however, are shown to be vulnerable to quantum computer attacks. Specifically, Shor's algorithm [22] shows that the hard mathematical problems, factorization and (EC) discrete logarithm, underlying the today's asymmetric cryptography schemes, can be broken in polynomial time when powerful quantum computer with enough q-bits is built. Thus, the use of quantum-safe algorithms and their integration into the used standards becomes

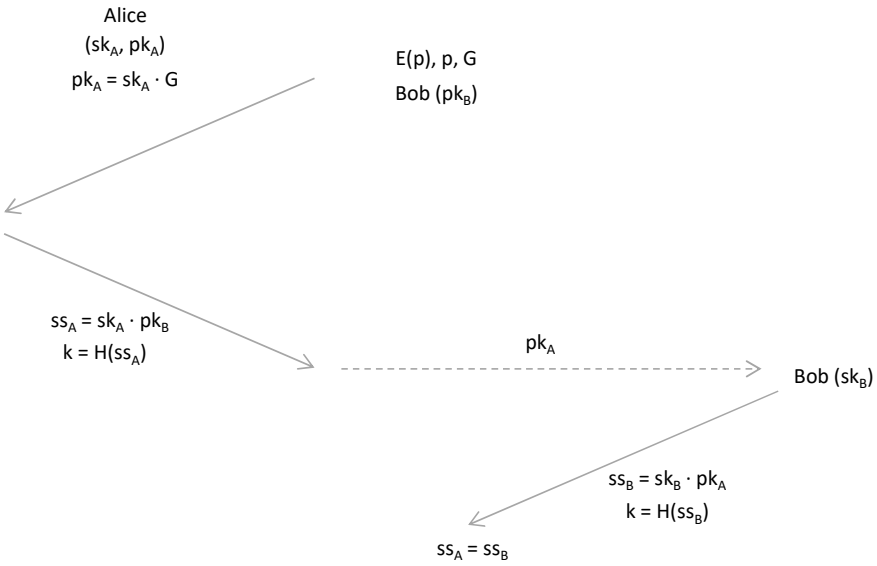


Figure 6: Graphical representation of ECDH KEM algorithm.

urgent and is focus of several researches and engineers. The transition to post-quantum algorithms, underlying the crypto protocols, and the flexible HPKE standard, allowing integration of new schemes, is part of the main motivation for the work presented in this article.

The use of solely post-quantum cryptography, however, is a risk that should not be undertaken before their robustness against classical and quantum computer attacks is studied in more details. Therefore, the goal of cryptographers is to "combine" the classical schemes with potentially post-quantum secure schemes, where this ensures further evaluation of the PQ primitives, their security and performance, where it excludes the risk that potential weakness of the PQ schemes may break the security of the scheme since well known classical crypto schemes are used in parallel. The aim of this work is, indeed, to integrate a post-quantum secure primitive into the Hybrid Public Key Encryption (HPKE) scheme converting into classical- and PQ-robust standard.

First, we have aimed at integrating the isogeny-based algorithm SIKE, which security relies on pseudo-random walks among isogeny graphs. It features the smallest key sizes among the NIST post-quantum candidates and favors well-studied elliptic curve elements. Later, we have integrated the lattice-based Kyber algorithm which main advantage is the optimal performance. We describe the design strategies for integrating these crypto schemes under HPKE. Additionally, we implement a hybrid version of the standard, where the ECC primitive is "combined" with (any) post-quantum scheme. We have focused on the integration of these two PQ-secure schemes and their execution in parallel with a classical ECC scheme, however, our design allows easy integration of any other classical or PQ protocol into the HPKE.

2 Post-Quantum Primitives

The increasing capabilities of the quantum computers and the thread that they represent for the classical cryptographic schemes motivate the transition from classical cryptography to algorithms that are believed to be quantum safe. Driven by the fast technological advances the National Institute of Standards and Technology (NIST) [16] has initiated a standardization process of the quantum resistant candidates, where it has already gone through Rounds 1 and 2, announcing four finalists in the group of Key Encapsulation

Mechanisms (KEMs) and another three finalists in the group of Digital Signature Algorithms (DSAs). Another 8 schemes has been classified as alternate candidates - 5 KEMs and 3 DSAs, which are likely to go through another round of optimizations and evaluations and after that may form part of the finalist group of PQ schemes.

2.1 Supersingular Isogeny Key Encapsulation

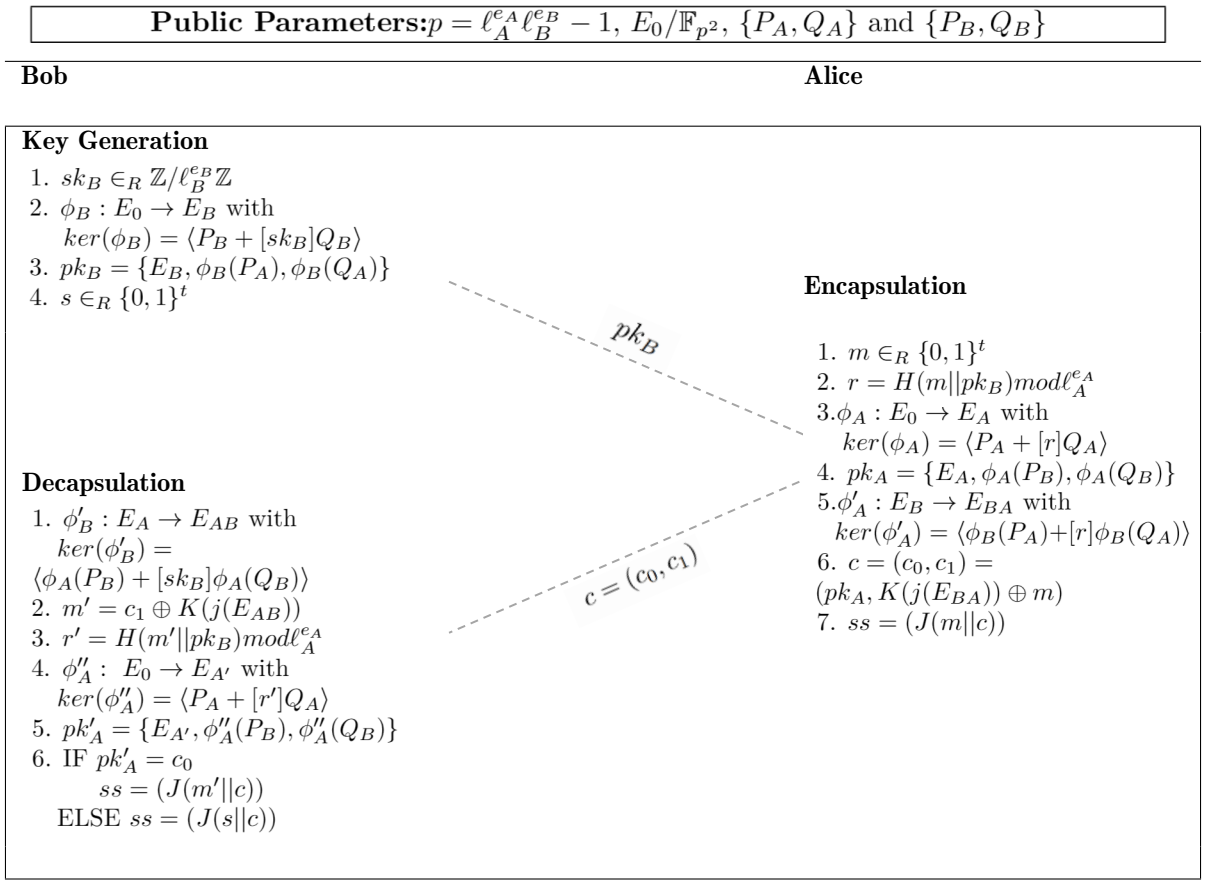


Figure 7: Supersingular Isogeny Key Encapsulation (SIKE) mechanism [7].

The Supersingular Isogeny Key Encapsulation [7] mechanism is the only post-quantum algorithm based on isogeny maps between supersingular elliptic curves. It forms part of the alternate candidates during the third round of the NIST post-quantum standardization process, where it still goes through several optimizations improving its performance. Based on the well known elliptic curve structures and on well studied mathematical background, the SIKE mechanism represents the post-quantum safe candidate with the smallest key sizes, which ensures low communication latency, specifically important in scenarios with limited bandwidth. This is the reason for several researchers ot continue optimizing the scheme for different ARM-based platforms [19], [2], [20] and FPGA platforms [8], [9], [13] and even GPUs [21] . In 2016 *Azarderakhsh et. al* proposed even further reduction of the key sizes [4], which however, introduces a significant performance overhead, thus, is not focus of this work.

The Supersingular Isogeny Diffie-Hellman (SIDH) protocol was introduced in 2011 by David Jao and Luca

de Feo in [11]. The protocol is based on the Diffie-Hellman key exchange method, however, it lays on more sophisticated mathematical concepts to add quantum security. The SIDH algorithm, however, is vulnerable to a static key attacks, thus, the Supersingular Key Encapsulation mechanism, applying the Fujisaki-Okamoto (FO) transformation [10], converting the CPA-secure Public Key Encryption (PKE) scheme into CCA secure algorithm. The mathematical base of both protocols, however, remains the same. The difficulty of the protocols is based on isogeny maps among supersingular elliptic curves. In particular, Alice and Bob compute secret isogenies, which lead them to curves that feature common characteristic, specifically their j-invariant which they use towards the computation of the shared secret, used later as a common symmetric key.

3 HPKE and its Operation Modes

The Hybrid Public Key Encryption integrates the use of a Key Encapsulation Mechanism (KEM), a Key Derivation Function (KDF) and an Authenticated Encryption with Associated Data (AEAD) allowing to integration of any instance of the before-mentioned algorithms. The proposal for HPKE is described in details in [6]. The standard defines the use of symmetric and asymmetric cryptographic schemes to provide security, mainly data integrity and privacy, and efficiency. The main advantages of HPKE over different hybrid public key encryption protocols as mentioned before are the easy integration of different emerging schemes (such as PQ KEMs), the single standard specification for using asymmetric and symmetric schemes and the definition of the more sophisticated authentication operation modes.

3.1 Base Mode HPKE

This work is centered on the implementation and development of PQ HPKE operating in its base mode. On completion of this step we may continue our work on other operation modes, however, this will be an extension of the project and will be additionally addressed if designed.
The base mode HPKE integrates any IND-CCA secure scheme (along with a KDF) and an AEAD algorithm where the former is used for shared secret and the latter for plaintext message encryption. Specifically, the base mode of the HPKE standard defines a KEM for negotiation of a shared secret which is used for a KDF extract and expand function for the derivation of a common key. It is later passed to a the Key Schedule routine outputting a fixed-length symmetric key and a nonce used for the following symmetric communication performed by the "seal" and "open" functions. The base operation mode does not authenticate the sender, thus, the only pair of keys the sender uses are pk_E and sk_E where "E" denotes the ephemeral nature of the keys, which are generated in each execution of the protocol.

3.2 Authentication Mode HPKE

For authenticating the sender there are two different methods used in the scope of the HPKE. The authentication mode "$\text{HPKE}_{\text{Auth}}$" is the most appealing and innovative operation mode of the HPKE protocol. For the purpose of authentication, the sender has to posses a pair of static keys pk_S and sk_S where we need to differentiate them from the ephemeral key pair.

The authentication of the sender (based on ECC) is performed by executing another round of the ECDH, where the Sender uses his/her secret key sk_S and the public key of the recipient pk_R to obtain another point multiplication. The resulting value is appended to the shared secret and leads to a common symmetric key value only if the recipient can obtain and concatenate the same value after the second round of the ECDH using pk_S and sk_R. The description of the sender (Alice) steps (Equation 4) and recipient (Bob) (Equation 5) lead to common share secret, input for KDF execution and derivation of the symmetric key (Equation 6).

$$DH(\mathsf{sk}_E, \mathsf{pk}_R)||DH(\mathsf{sk}_S, \mathsf{pk}_R) \tag{4}$$

$$DH(\mathsf{sk}_R, \mathsf{pk}_E)||DH(\mathsf{sk}_R, \mathsf{pk}_S) \tag{5}$$

$$[\mathsf{sk}_E] \cdot [\mathsf{sk}_R] \cdot G||[\mathsf{sk}_S] \cdot [\mathsf{sk}_R] \cdot G = [\mathsf{sk}_R] \cdot [\mathsf{sk}_E] \cdot G||[\mathsf{sk}_R] \cdot [\mathsf{sk}_S] \cdot G \tag{6}$$

The sender (Alice) is then authenticated by the recipient (Bob) since the derivation of a common symmetric key is only possible if Alice is the actual sender. A failure in the authentication of Alice will lead to different symmetric keys for both parties and will prevent further communication. Otherwise, they derive a shared secret which is later passed to the Key Schedule subroutine to be expanded and applied to the "seal" and "open" functions as a symmetric key.

3.3 PSK Mode HPKE

The HPKE offers another execution mode, where the authentication of the sender is performed by a Pre-Shared Key (PSK) which is used in the Key Schedule subroutine to form the value of the symmetric key. Many real world scenarios and protocols execution allow PSK authentication using data established during previous connection between the sender and the recipient. The value of the PSK is used along with the shared secret, derived during the ECDH protocol execution, for the formation of the symmetric key and the nonce. Specifically, it is used in the extract function, part of the KDF, where a common value is derived and used for the expand functions. Thus, the symmetric keys of both parties coincide and allow further communication only if the PSK values used are equivalent.

The main advantage of the PSK operation mode of HPKE is the lack of particular protocol integration for authentication. Therefore, when execution the protocol in post-quantum mode (using PQ KEM), there is no need of a classical algorithm (such as DH or ECDH as specified in the standard) for the authentication step. Therefore, HPKE PSK mode allows authentication in classical, post-quantum or hybrid mode, independently.

The main goal of the project is the integration of SIKE into the base operation mode. However, upon completion of the given task, the implementation of the PSK mode of operation may become a new objective for this work.

3.4 Authentication PSK Mode HPKE

Finally, the HPKE standard offers a combination of the two authentication modes, where Alice and Bob append the value of the ECDH result to the shared secret and later use the value of a PSK as an Input Key Material (IKM) for the KDF during the Key Schedule routine.

4 PQ HPKE

The integration of PQ protocols into standards is becoming necessary and composes the main goal of this work - integration of post-quantum secure schemes into the HPKE standard, allowing hybrid execution of classical and PQ algorithms.

The HPKE protocol allows integration of any KEM for the derivation of the shared secret. Therefore, we modify the design of the HPKE structures and functions to allow execution of both - classical and PQ algorithms. Furthermore, as discussed previously in this article, we aim at implementing a hybrid version of the HPKE, which we refer to as PQ HPKE to avoid repetition or confusion due to use of the term "hybrid", where with PQ we refer to the hybrid operation between classical and post-quantum KEMs since (hybrid)

HPKE refers to the use of both - asymmetric and symmetric cryptographic schemes.

In the process of work we have followed the idea of *replace* and then *bring-back* strategy, where we first *replace* the ECC cryptographic primitive used for the shared secret computation with a quantum secure scheme, test and verify the functionality using only PQ scheme, and then *bring-back* the ECC scheme to the standard.

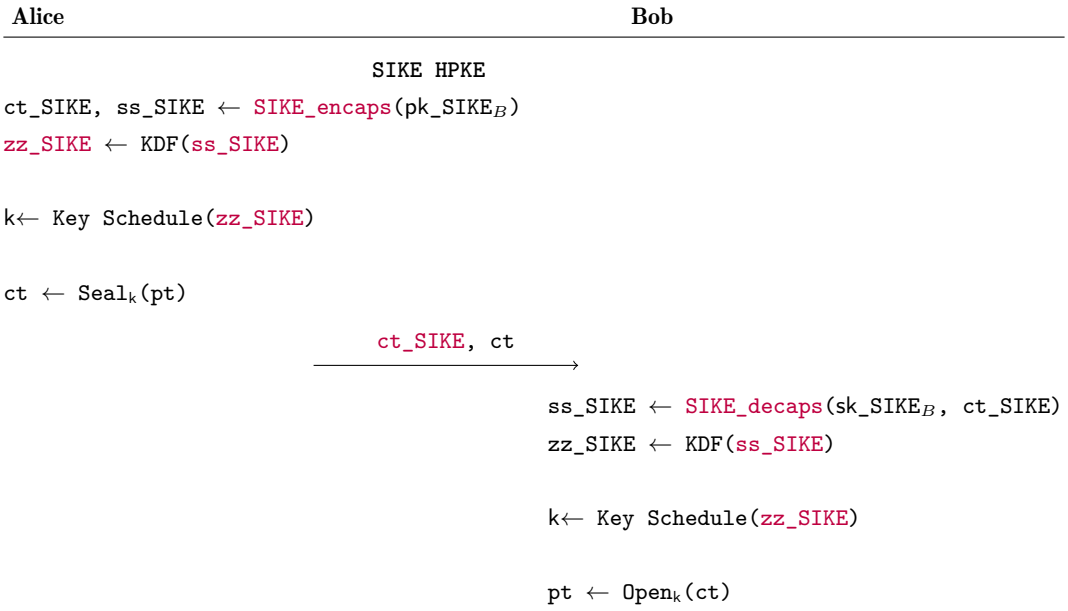


Figure 8: SIKE HPKE

The followed development strategy allowed a step-by-step progress, where we could integrate, test and debug separately the classical and post-quantum KEMs. Furthermore, the separate integration of post-quantum protocols allows the execution of HPKE in a pure post-quantum mode (where, as specified later, Kyber can reach even better performance results than ECC and, if not risky, can be executed solely). In Figure 8, we present the *replacement* of the new Key Encapsulation Mechanism (KEM) into the protocol where we first use one of the NIST post-quantum candidates - SIKE. However, the similar construction of the PQ KEMs due to the applied FO transformation to the PKEs schemes will allow easy integration of other post-quantum schemes upon careful designing of the PQ integration techniques. The invocation of the key encapsulation and the key decapsulation functions on both ends of the communication, as shown in Figure 8, allows both sides to reach a shared secret `ss`, which we denote as `ss_SIKE` in Figure 8 and mark with purple color for illustrative purposes. The nature of the PQ KEMs requires Alice to send Bob a ciphertext message, consisting of her public key and masked value of a random number, used for the derivation of the secret secret. Due to the double use of the term "ciphertext" in the context PQ HPKE (due to symmetric encryption schemes and the use of PQ KEMs) we will refer the post-quantum ciphertext value as `ct_PQ` (or SIKE in the case of specifically the SIKE PQ scheme) and the encryption of the plaintext value as `ct`.

Additionally, the value of the `ct_PQ` is sent directly to the recipient without applying a marshal to it. The marshal function is defined in the HPKE draft standard as a Serialization Function (as explained earlier in this article) and is used in the case of ECC instances based on P-256, P-384 and P-521. In the case of X25519, X448 and PQ KEMs it is omitted.

The integration of both classical and post-quantum key encapsulation schemes operation with both - classical and post-quantum key pairs. In our design we use one long array and store both values appended. However, in the documentation we refer them separately for simplicity and better illustration of the PQ HPKE implementation.

The execution flow for the PQ (hybrid) HPKE is represented in Figure 9, keeping the coloring for illustration purposes. We follow the *bring-back* ECC scheme strategy, denoting it in blue color in Figure 9. For the completion of the ECC key generation on the sender side a pseudo-random value is issued for the value of the secret key. The sender performs an ECDH point multiplication deriving his/her public key. The sender then performs another round of point multiplication and derives an ECC shared secret. The nature of the PQ KEMs allows the implementation of the algorithms without the need of secret key and public key generation on the sender side (ephemeral keys are generated as part of the PQ KEM encapsulation function). The PQ KEM generates a `ct_PQ` containing the PQ public key and a masked random number used as a secret key for the sender. The values are concatenated and are sent to the recipient. PQ KEM encapsulation function also outputs a shared secret which, appended to the ECC share secret, is used to derive the symmetric key. The step on the sender side (Alice) are shown in Figure 9.

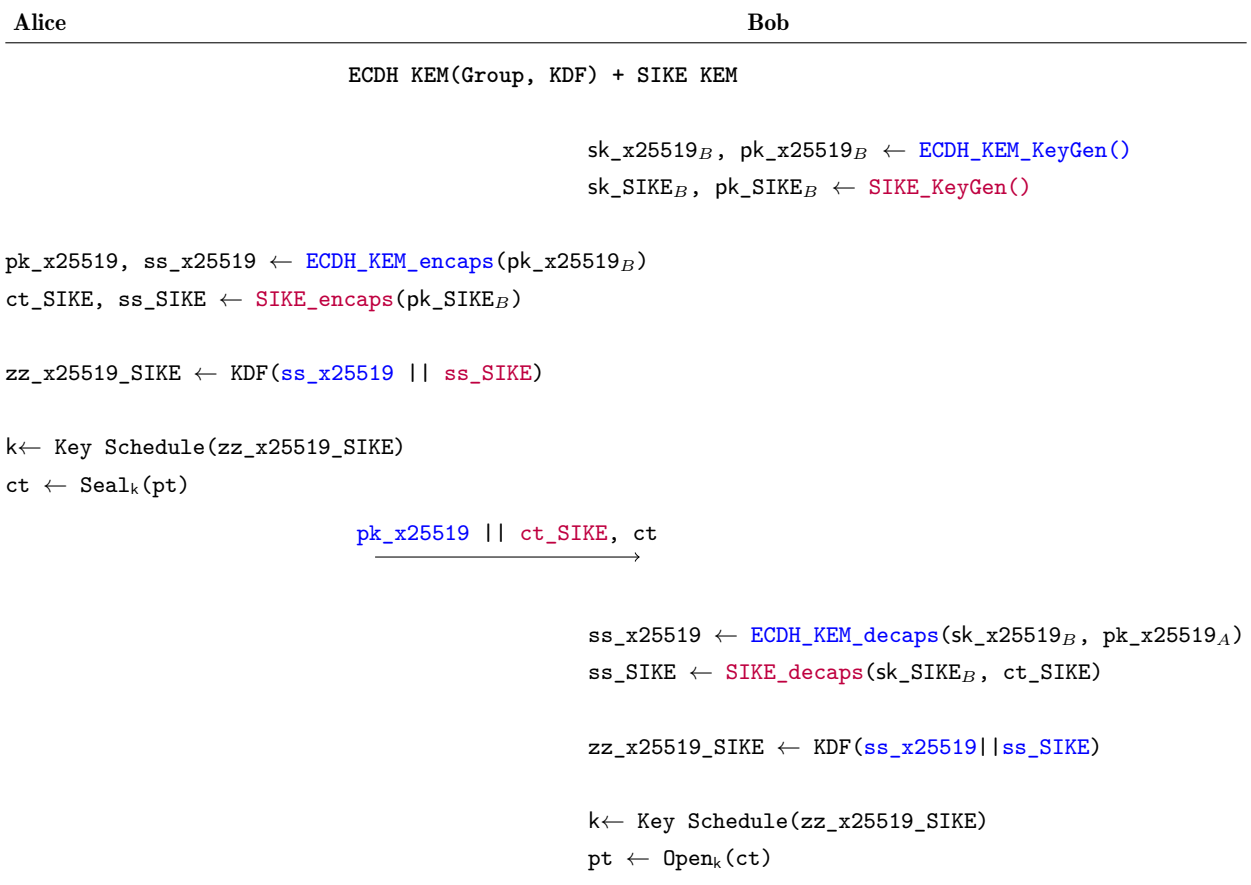


Figure 9: PQ (hybrid) HPKE.

Respectively, the recipient executes the ECDH KEM shared secret key derivation using point multiplication and the PQ KEM decapsulation, where he/she obtains a post-quantum shared secret value. The

recipient (Bob in Figure 9) uses the concatenation of these shared secret values to obtain a symmetric key, where the fault in any of the algorithms will prevent further communication.

In Figure 9 we present and overview of the entire PQ (hybrid) HPKE implementation design. The details about the decisions taken during the development and the reasoning behind them is presented in the following sections.

5 Design

In the scope of this project, we have taken into consideration two different implementation designs, considering the directory organization and the source files, and have compared them to decide which one better serves the needs of AWS-LC. We compare them mainly based on portability since the goal of the project is to allow easy integration of any new scheme (PQ or classical) into AWS-LC.

5.1 Notation

In Figure 10 and Figure 11 we represent the `aws-lc` directory and its content. We denote the directories by a red horizontal line containing the name of the given folder above it, where all the lower level line represent sub-directories. Therefore, the designs described in Figure 10 and Figure 11 illustrate the hierarchy of the files inside the AWS-LC library. Each file is denoted as a box where the name and the type of the file are specified. The light orange boxes represent files that already exist and form part of the AWS-LC. The dark orange boxes represent the changes that need to be made to a given file (or its creation if it does not exist). The gray boxes contain future additional changes to the directories, such as integration of other post-quantum schemes into the AWS-LC and the HPKE standard. The red circles along with the red arrows specify the source file used for the instantiation of a specific structure or the invocation of a certain function.

5.2 Design #1

The first implementation design considered is shown on Figure 10. The design is based on reusing the post-quantum schemes (so far only SIKE) integrated into AWS-LC and on the use of the low level PQ KEMs source files (obtained from the NIST Round 3 submissions) for the execution of the protocol within the HPKE. In the given design the HPKE file will serve as a high level wrapper for the classical and PQ low level APIs. The high level EVP APIs, frequently used to absorb the cryptographic primitives from the AWS-LC, will *not* be used in this design.

Avoiding the EVP APIs by using the actual low-level PQ KEM subroutines, such as `crypto_kem_keygen`, `crypto_kem_enc` and `crypto_kem_dec` ensures independence of the EVP high level APIs which are still subject of modifications.

In Figure 10 we reuse the original NIST SIKE submission Round 3 files (we developed also SIKE integration into HPKE using the `s2n` SIKE files where some minor changes are additionally applied to the PQ code). The source files are placed into a folder called `pq-crypto`. We should highlight that we moved the location of the `pq-crypto` folder since we consider that it better follows the hierarchy structure of the AWS-LC when placed under the `crypto`. Similar to the `curve25519` folder, consisting of a classical elliptic curve primitive, we place `SIKE` (and later `Kyber`) under the `pq-crypto` folder. The design and the structural configuration of the files allow the integration of any post-quantum KEM (such as Kyber, marked in grey in Figure 10, which was later on integrated to the HPKE design as well).

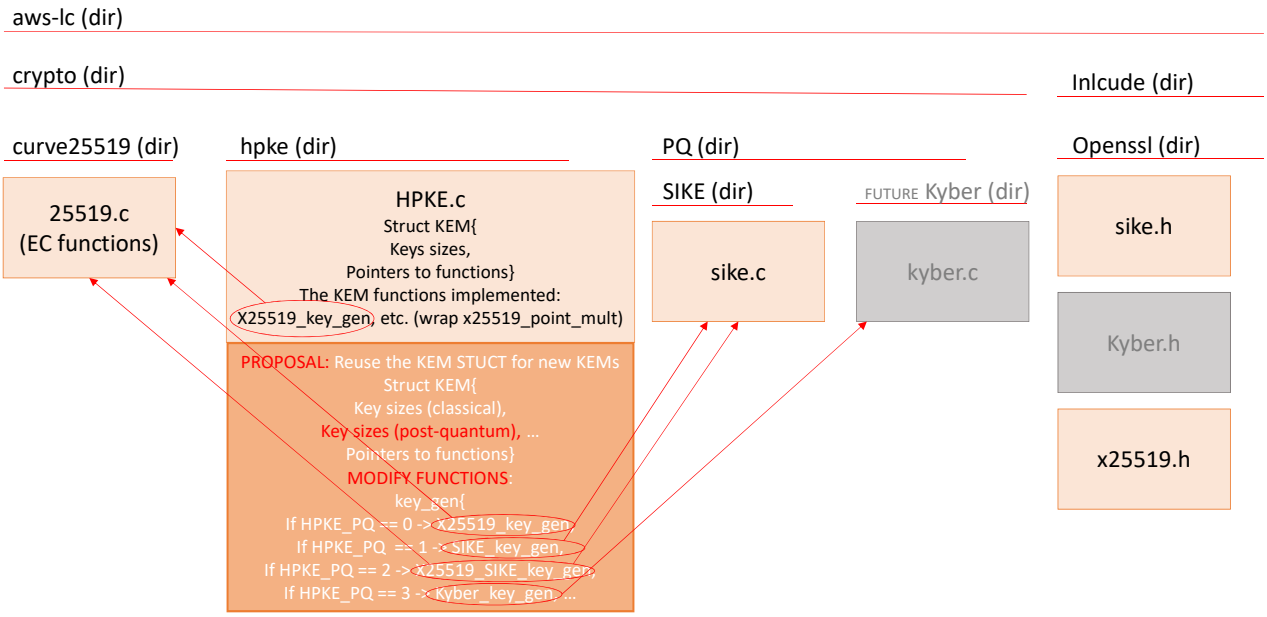


Figure 10: Implementation design using the KEM defined functions from the SIKE submission [7].

The decision to consider the design in Figure 10 is based on the use of the original files of the post-quantum primitives to consume them. It is important to note that the classical crypto primitives, used into HPKE are also consumed directly from the original source file, omitting the use of high level EVP APIs. By implementing this design we follow the logic of the HPKE implementation, where the `hpke` file constitutes the high level wrapper of the crypto primitives. The arrows show that there are no specific `pq_kem` structures involved in the use of the underlying crypto primitives: the structure used to wrap the parameters of the crypto schemes are designed inside the HPKE files, avoiding creation and invocation of EVP PQ specific APIs. Therefore, as described in more details in the following section and the section 7, the HPKE_KEM and HPKE_KEY structs require some modifications. In particular, several fields are added to the HPKE_KEM struct specifying the lengths of the PQ parameters. Moreover, for supporting any crypto primitives (classical, post-quantum or hybrid) we change the KEM struct to contain pointers to the values of the private and public keys, where the length of the keys is going to be dynamically specified with the user's choice of HPKE underlying primitive(s). This is particularly useful for the easy integration of new crypto schemes into HPKE since the key sizes vary depending on the specified length in the HPKE_KEM structure. Additionally, by omitting the use of EVP APIs, we avoid the concatenation of the keys from the fields of their structures to another array since in the HPKE file we store them in continuous memory locations.

For the design, with the progress of the work, we have notice another advantage of the design for the easy integration of new cryptographic primitives. The different HPKE key generation functions, serving as a high level wrappers for the KEMs, used to initialize the keys for the various KEMs, can be combined into a single HPKE key generation function. The logic about the KEM is implemented inside the given function and requires simply a new case integration inside a switch branch statement (this logic is later modified as explain in the following sections by creating arrays of function pointers but we skip the details in this part since they are not directly related to the design choice). It should be noted that this is possible only after applying changes to the HPKE_KEM and HPKE_KEY structures, which allows no need of change of the keys, shared secrets and ciphertexts lengths when new primitive is added.

5.3 Design #2

The second design that we have considered is represented on Figure 11, where we use the same notation to describe the repository hierarchy and the source files used for creating and executing the HPKE protocol. This design follows the PQ EVP APIs implemented as a higher level wrapper of the PQ KEMs. Furthermore, to follow the same logic for the classical and PQ primitives, it should expand the EVP API usage to the classical underlying crypto (in the scope of AWS-LC HPKE it is only X25519 for now).

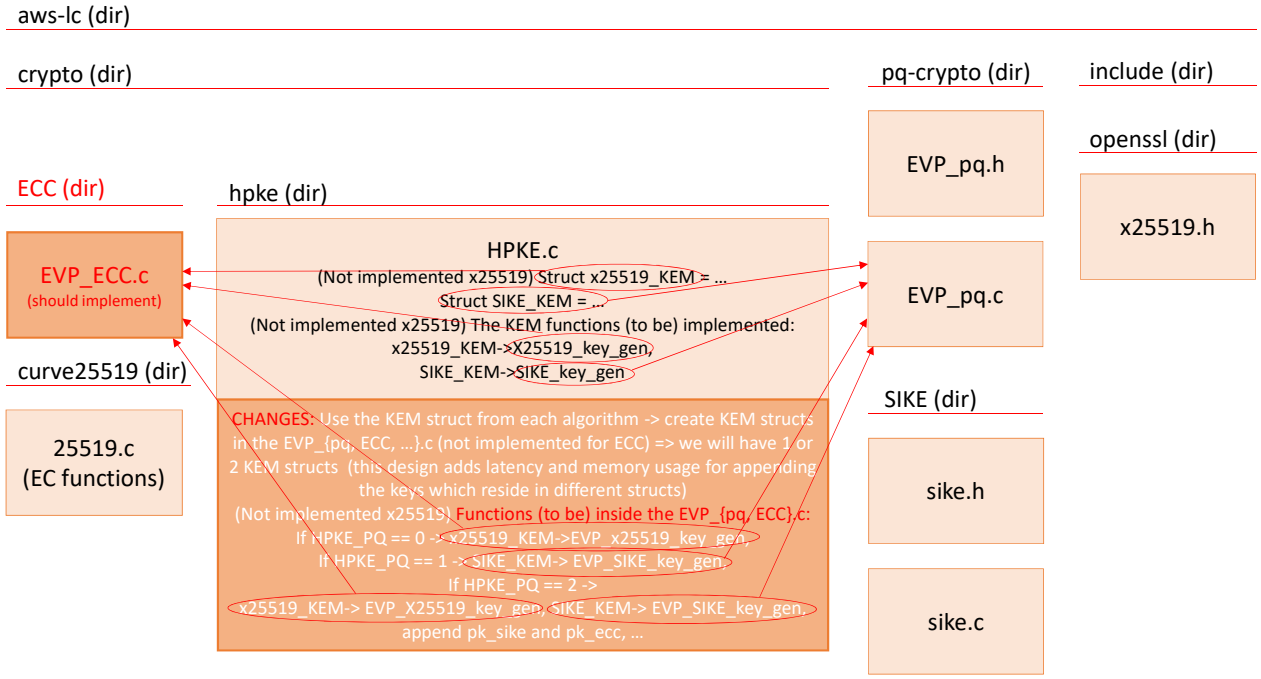


Figure 11: Implementation design using the EVP defined functions from the pq-crypto folder

The API based design imposes the use of high level structures for the creation of a classical or PQ instance. Therefore, the HPKE file will have to use APIs to instantiate a structure, containing the classical primitive values (parameters lengths and key values) and another structure, containing the PQ primitive values.

We have evaluated the given design and considered several disadvantages of it, listed below.

First, the high level EVP API file for the ECC is not developed, therefore, the given design will require its implementation. However, the EVP API file for the PQ KEMs is not definite and multiple changes may occur in future, which will also impose changes in the HPKE files and a corresponding redesign of the ECC API so that they follow the same structure and are properly used in the HPKE instances.

Second, the before-mentioned design requires two different KEM structures for ECC and for PQ to be instantiated. However, the value of the public keys cannot be easily appended. To do so, we need to copy the values of the ECC key into a separate buffer and then copy the value of the PQ key in the same buffer, concatenating both values. This adds a performance overhead (which may not be too high but it can be avoided by applying our first design).

Third, the use of separate structs for the classical and the PQ KEM values requires the invocation of different keygen, encaps, decaps funcitons (one for ECC and one for PQ). However, when using a single struct for both primitives (as in Design 1) we can easily combine all these functions under a global `HPKE_{keygen, encaps, decaps}` function. Later, this will allow to invoke only the global function (independent of the KEMs used) where inside it the logic for the crypto primitives is "hidden". Therefore, the need of separate functions `EVP_{X25519, SIKE, Kyber}_keygen` can be avoided.

The comparison between both designs and the advantages of each one is summarized in Table 1.

Feature	KEM (Design 1)	EVP (Design 2)
Allow to re-use the same KEM struct independently of the scheme.	✓	
Allow to use the PQ crypto directory without changes.		✓
Optimal memory usage when appending the key (ciphertext) values.	✓	
Optimal performance when appending the key (ciphertext) values.	✓	
Follow the structure of the AWS-LC HPKE directory.	✓	
Allows a global function containing the logic for the HPKE primitives.	✓	
Allows no changes in the HPKE KEM and KEY structures.		✓
Allows no implementation of ECC API file.	✓	

Table 1: Implementation Design evaluation criteria. The KEM represents the direct invocation of the KEM functions (Design 1), whereas the EVP represents the API usage (Design 2).

6 HPKE Structures and Functions

6.1 HPKE KEM structure

The HPKE KEM structure Figure 12 contains information about the cryptographic primitives parameters lengths. Moreover, it serves as a wrapper to a specific KEM instance, therefore, it contains pointers to the key generation, encapsulation and decapsulation functions.

NOTE: The current integration of ECDHKEM(curve25519, SHA256) does not require to add additional field containing the length of the classical cryptography scheme shared secret since the values of the shared secret, the public and the secret keys are of the same size - 32B. However, the integration of other ECC schemes will require the integration of another field containing the length of the share secret of the ECC underlying algorithm (changed are marked in Figure 14 in blue).

The fields `id`, `public_key_len`, `private_key_len`, `seed_len`, `* init_key`,`* generate_key` and `* encap_with_seed` are initialized using a specific function which returns a HPKE KEM structure with the corresponding field values to the scheme used. The original implementation of the HPKE integrates 7 different fields in the HPKE KEM where they contain the lengths of the crypto primitive parameters and the corresponding (pointers to) functions needed for the initialization and the generation of the key pair, the encapsulation and the decapsulation of the shared secret value. There is an additional field reserved for the id of the HPKE KEM used, where, as described later, will be used to define the combination of crypto primitives used for the execution of the HPKE protocol.

To integrate a Post-Quantum KEM into the HPKE KEM structure we add new fields to store the parameter lengths of the PQ primitive. We define the new fields as PQ_public_key_len, PQ_private_key_len, PQ_ciphertext_len, PQ_shared_secret_len. Depending on the schemes used (ECC only, PQ only or hy-

```
HPKE_KEM
id
public_key_len
private_key_len
seed_len
int (*init_key)(EVP_HPKE_KEY *key, const uint8_t *priv_key, size_t priv_key_len);
int (*generate_key)(EVP_HPKE_KEY *key);
int (*encap_with_seed)(const EVP_HPKE_KEM *kem, uint8_t *out_shared_secret,
            size_t *out_shared_secret_len, uint8_t *out_enc, size_t *out_enc_len,
            size_t max_enc, const uint8_t *peer_public_key, size_t peer_public_key_len,
            const uint8_t *seed, size_t seed_len);
int (*decap)(const EVP_HPKE_KEY *key, uint8_t *out_shared_secret, size_t *out_shared_secret_len,
            const uint8_t *enc, size_t enc_len);
```

Figure 12: HPKE KEM structure before modification

```
EVP_HPKE_KEM EVP_hpke_x25519_hkdf_sha256()
static const EVP_HPKE_KEM kKEM = {
  /*id=*/ EVP_HPKE_DHKEM_X25519_HKDF_SHA256,
  /*public_key_len=*/ X25519_PUBLIC_VALUE_LEN,
  /*private_key_len=*/ X25519_PRIVATE_KEY_LEN,
  /*seed_len=*/ X25519_PRIVATE_KEY_LEN,
  x25519_init_key,
  x25519_generate_key,
  x25519_encap_with_seed,
  x25519_decap};
return &kKEM;
```

Figure 13: HPKE KEM structure after modification

brid) the values of these fields are initialized to the corresponding values, i.e. when using X25519 only, we will initialize the classical parameter length fields with the corresponding values and the post-quantum values all to zero lengths. In the case of only post-quantum algorithm the values of the ECC crypto lengths will be initialized to zero and the PQ fields will take the corresponding values. In case hybrid HPKE is instantiated, the values of all the HPKE KEM structure are set to the values, corresponding to the given underlying cryptography primitives.

The modifications applied to the HPKE KEM structure, including extra fields for the lengths of the PQ parameters is a design decision that allows the implementation of the HPKE (classical only, PQ only or hybrid) using the addition of these values as the total length of the parameters used. Therefore, if using ECC only, the value of the public key will be the addition of `kem->ECC_public_key_len` and `kem->PQ_public_key_len` which will equal to the value of `kem->ECC_public_key_len` since `kem->PQ_public_key_len` is initialized to zero. Similarly, in the case of PQ only, the length of the public key will be only the length of the `kem->PQ_public_key_len` since the value of `kem->ECC_public_key_len` is

```
HPKE_KEM
id
ECC_public_key_len
ECC_private_key_len
ECC_shared_secret_len
ECC_seed_len
PQ_public_key_len
PQ_private_key_len
PQ_ciphertext_len
PQ_shared_secret_len
int (*init_key)(EVP_HPKE_KEY *key, const uint8_t *priv_key, size_t priv_key_len);
int (*generate_key)(EVP_HPKE_KEY *key);
int (*encap_with_seed)(const EVP_HPKE_KEM *kem, uint8_t *out_shared_secret,
            size_t *out_shared_secret_len, uint8_t *out_enc, size_t *out_enc_len,
            size_t max_enc, const uint8_t *peer_public_key, size_t peer_public_key_len,
            const uint8_t *seed, size_t seed_len);
int (*decap)(const EVP_HPKE_KEY *key, uint8_t *out_shared_secret, size_t *out_shared_secret_len,
            const uint8_t *enc, size_t enc_len);
```

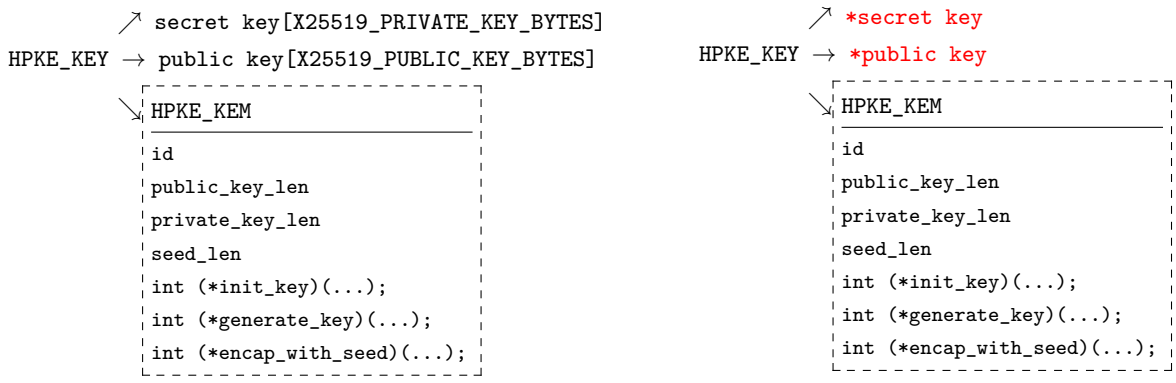
Figure 14: HPKE KEM structure after modification

zero. In the case of hybrid HPKE instantiated, the actual value of the public key will be the addition of both underlying primitives public key lengths. Therefore, the instantiation of the protocol using any combination of schemes will not impose a change in the code and the implementation of the `HPKE_setup_sender` and `HPKE_setup_recipient` funcitons. The design solution allows also the integration of different ECC and PQ schemes adding simply a KEM instantiation function specifying the values of the parameters' lengths as shown in Figure 13.

6.2 HPKE KEY structure

Another structure used in the scope of the HPKE standard is the HPKE KEY which contains a pointer to a HPKE KEM structure (subsection 6.1) and the actual content of the public and private keys illustrated in Figure 15a. The structure contains two arrays of fixed-length representing the public and the secret keys. The original design of the HPKE protocol does not allow to initialize the arrays using variable sizes, i.e. if we use curve25519 we should use the defined constants for the lengths of the arrays. Therefore, for the declaration of the arrays with different size, there should be several functions initializing the key structure, or there should be a global variable specifying which algorithm is used (we used the define preprocessor directive to set up a constant with the id of the underlying primitive used and then if else endif branching to initialize the arrays with the correct lengths). However, the given solution results in constant manual setup of the directive before compiling, thus, prevents further automation of the testing functions to perform test on all the HPKE variants.

The design decision taken to eliminate this issue consist of replacing the arrays by vectors as shown in Figure 15b. Therefore, the structure itself will be formed by three pointers - two to the keys and another one to the new HPKE KEM structure. The new design, however, requires the dynamic memory allocation for



(a) HPKE KEY before modifications

(b) HPKE KEY after modifications

Figure 15: HPKE KEY structure

the public and private keys with size depending on the algorithm used. Furthermore, it should be considered that the size of the HPKE KEY structure will be equal to three pointers, not to the sizes of the keys plus a pointer to the HPKE KEM structure. Therefore, all occurrences of `sizeof(HPKE_KEY)` inside the code should be modified as well. We summarize the code changes performed in Figure 16 where we denote the dynamic memory allocation for the keys and the memory free after we do not need the keys anymore. The functions `secretkeybytes(int alg)` and `publickeybytes(int alg)`, auxiliary functions for the purpose of automated testing of the HPKE with different primitives, take the algorithm id as an input and return the length of the secret/public keys per algorithms, respectively.

```
ScopedEVP_HPKE_KEY key;
key->private_key = (uint8_t *)(malloc(sizeof(uint8_t) * (secretkeybytes(algorithm))));
key->public_key = (uint8_t *)(malloc(sizeof(uint8_t) * (publickeybytes(algorithm))));
...
free(key->private_key);
free(key->public_key);
```

Figure 16: HPKE KEY dynamic initialize

6.3 PQ HPKE modifications

The first strategy for integration of post-quantum primitives into the HPKE protocol followed strictly the HPKE files structure. The functions `x25519_init_key`, `x25519_generate_key`, `x25519_encapsulate`, `x25519_decapsulate` were implemented specifically for the ECC based on curve25519 implementation. The multiple uses of constants, particular for the given curve, e.g. `X25519_PRIVATE_KEY_LEN`, did not allow the easy integration of other ECC and PQ schemes. The initial startegy was the creation of similar functions for each combination of classical and post-quantum HPKE implementation. Therefore, we created similar function for SIKE only and x25519||SIKE, e.g. `SIKE_init_key`, `x25519_SIKE_init_key`, etc. However, with the integration of Kyber, we noticed that there is unnecessary increase in the number of functions, which complicates the HPKE API and requires additional work in the case of newly integrated schemes (classical or PQ). Therefore, we merged all the functions is a single one, e.g. `HPKE_init_key`, `HPKE_generate_key`, `HPKE_encapsulate`, `HPKE_decapsulate`. We had to implement large logical blocks using if-else or switch

statements which were also increasing the code size and were imposing the developer to make changes on the given block with the integration of new schemes (we refer the details about the logical block and the design strategy for its elimination in the following paragraph). Due to the continuation of the NIST PQ standardization process and the novelty, therefore, unstudied aspects of some of the PQ algorithms, the urge for fast integration/elimination into/from standards and protocols, such as TLS, HPKE, etc, was objective of these PQ HPKE implementation project. Therefore, our design integrates the most efficient solution for future expansion of the work. We describe the development decisions related to the functional design in the following sections, along with discussion about the changes in the functions, where we first describe the auxiliary structures, arrays or functions implemented which allow the actual development of an optimal design.

6.3.1 PQ HPKE auxiliary design

The decision between the multiple key generation/encapsulation/decapsulation functions to be invoked requires the implementation of a logic block inside the function based on large `if-else` branching statements, depending on the combination of primitives chosen. These branches are based on the `key->kem->id` field described in subsection 6.1. To make the design a bit more compact, we have used switch statement where each case defines if a pq algorithm is used. Therefore, in case we use X25519_SIKE combination we will execute the X25519 key generation/encapsulation/decpasulation and then will perform the SIKE key generation/encapsulation/decpasulation execution on of the `switch` cases. Similarly, another `switch` case will contain the functions corresponding to Kyber. Following the given implementation design, the integration of a new scheme (ECC or PQ) will require to additionally modify the logic block integrating new cases or `if-else` branches.

```
//Define the function pointer types (with arguments and return value)
typedef int (*pq_crypto_keygen)(unsigned char *, unsigned char *);
typedef int (*pq_crypto_enc)(unsigned char *, unsigned char *, const unsigned char *);
typedef int (*pq_crypto_dec)(unsigned char *, const unsigned char *, const unsigned char *);

//Create vectors of pointers to functions
pq_crypto_keygen pq_keygen[] = {&crypto_kem_keypair_SIKEp434, &crypto_kem_keypair_kyber,
    NULL};
pq_crypto_enc pq_enc[] = {&crypto_kem_enc_SIKEp434, &crypto_kem_enc_kyber, NULL};
pq_crypto_dec pq_dec[] = {&crypto_kem_dec_SIKEp434, &crypto_kem_dec_kyber, NULL};
```

Listing 1: HPKE function pointer types used to initialize arrays containing the possible PQ functions

After considering these modifications when new scheme is integrated to HPKE, we have considered a new, more optimal, implementation design. We have created a new type which defines pointers to the functions to be called (Listing 1). These type defines the return type of the function, a pointer to the function itself and the arguments that it takes. We defined type for all the three function types - key generation, encapsulation and decapsulation. This definition of the function pointers is possible due to equivalent structures of the PQ KEM algorithms. In the case of ECC it will be also possible if an auxiliary function is added to standardize the arguments and return value of the ECC functions. After the definition of the function pointers we create arrays of these pointers. Each position of the array specifies a function belonging to a cryptographic algorithm. When invoking a specific function (position of the array) we use the `key->kem->id` as index. Therefore, we avoid the use of the large logic blocks, allowing easier integration of new schemes. The only change required will be the correct management of the `id` values.

```
//Use pq_keygen array to invoke given PQ algorithm key generation function
pq_keygen[key->kem->id % SUPPORTED_PQ_ALGORITHMS](
        (unsigned char *)key->public_key + key->kem->public_key_len,
        (unsigned char *)key->private_key + key->kem->private_key_len);

```

```
//Use pq_enc array to invoke given PQ algorithm encapsulation function
enc_res = pq_enc[kem->id % SUPPORTED_PQ_ALGORITHMS](
        out_enc + kem->public_key_len, hybrid_ss + kem->public_key_len,
        peer_public_key + kem->public_key_len);

//Use pq_dec array to invoke given PQ algorithm decapsulation function
enc_res = pq_dec[key->kem->id % SUPPORTED_PQ_ALGORITHMS](
        hybrid_ss + key->kem->public_key_len, enc + key->kem->public_key_len,
        key->private_key + key->kem->private_key_len);
```

Listing 2: HPKE function pointer types used to initialize arrays containing the possible PQ functions

The invocation of the PQ functions using the same set of arguments, requires a unified way of passing the arguments, independently of the size of the public/private key values. The new HPKE KEM structure (subsection 6.1) including the values of both KEMs (ECC and PQ) allows to refer to the keys independently of their sizes, e.i. since the public key consists of ECC public key length and PQ public key length, to access the public key of the PQ primitive requires simply to move the pointer to the position of the PQ key by adding the value of the ECC public key length to the original pointer (Listing 2).

6.3.2 HPKE initialize key function

The key initialization function takes as an input the value of the private key and generates the value of the public key given the input information. In the case of ECC underlying primitives the function performs one point multiplication resulting in the value of the public key. In the case of the post-quantum algorithms, the low level APIs of the KEMs provide a function called `crypto_kem_keygen` which, however, does not take the private key value as an input but rather generates the private key as a pseudo-random value and the public key based on the newly generated private key. The original implementation of the function is shown in Listing 3. The length of the private key passed is checked and the public key is generated.

In Listing 4 the performed modifications are shown. First, we eliminate the use of constants replacing them with the values from the HPKE KEM structure. Comparing the length of the private key with the HPKE KEM values allows the integration of any new cryptographical primitive (classical or post-quantum) without need of code change. Second, we have integrated the PQ KEMs where we invoke the key generation function. Due to the similar PQ KEM APIs and the creation of the array of function pointers, we managed to eliminate the use of logic blocks, previously required. Therefore, in case of PQ/hybris HPKE, the PQ KEM API for key generation will be invoked. The index of the array is determined by the `kem->id` as previously discussed.

```
static int x25519_init_key(EVP_HPKE_KEY *key, const uint8_t *priv_key, size_t priv_key_len){
  if (priv_key_len != X25519_PRIVATE_KEY_LEN) {
    OPENSSL_PUT_ERROR(EVP, EVP_R_DECODE_ERROR);
    return 0;
  }
  OPENSSL_memcpy(key->private_key, priv_key, priv_key_len);
  X25519_public_from_private(key->public_key, priv_key);
  return 1;
}
```

Listing 3: X25519 initialize key original implementation

```
static int HPKE_init_key(EVP_HPKE_KEY *key, const uint8_t *priv_key, size_t priv_key_len){
  if (priv_key_len !=
      key->kem->private_key_len + key->kem->PQ_private_key_len) {
```

```
    OPENSSL_PUT_ERROR(EVP, EVP_R_DECODE_ERROR);
    return 0;
  }
  // Check if the HPKE involves x25519 KEM
  if (key->kem->id == EVP_HPKE_DHKEM_X25519_HKDF_SHA256 ||
      key->kem->id == EVP_HPKE_HKEM_X25519_SIKE_HKDF_SHA256 ||
      key->kem->id == EVP_HPKE_HKEM_X25519_KYBER_HKDF_SHA256) {
    OPENSSL_memcpy(key->private_key, priv_key, priv_key_len);
    X25519_public_from_private(key->public_key, priv_key);
  }
  // Check if the HPKE involves PQ KEM
  if (key->kem->id != EVP_HPKE_DHKEM_X25519_HKDF_SHA256) {
    pq_keygen[key->kem->id % SUPPORTED_PQ_ALGORITHMS]((unsigned char *)key->public_key + key
    ->kem->public_key_len, (unsigned char *)key->private_key + key->kem->private_key_len);
  }
  return 1;
}
```

Listing 4: HPKE initialize key PQ implementation

6.3.3 HPKE generate key function

```
static int x25519_generate_key(EVP_HPKE_KEY *key) {
  X25519_keypair(key->public_key, key->private_key);
  return 1;
}
```

Listing 5: X25519 generate key original implementation

The key generation function should take as an input an HPKE and should generate the values of the private and public keys of the recipient. In the original implementation (Listing 5) of the HPKE protocol the `X25519_keypair` is invoked to generate ECC key pair.

```
static int HPKE_generate_key(EVP_HPKE_KEY *key) {
    X25519_keypair(key->public_key, key->private_key);

    pq_keygen[(key->kem->id) % SUPPORTED_PQ_ALGORITHMS]((unsigned char *)key->public_key +
    key->kem->public_key_len,(unsigned char *)key->private_key + key->kem->private_key_len);

  return 1;
}
```

Listing 6: HPKE generate key PQ implementation

The implementation of the key generation function and the changes that have been implemented (Listing 6) are very similar to the developed key initialization function described in subsubsection 6.3.2. Summarizing them briefly, we have replaced the constants and have invoked the PQ KEM APIs using a vector of pointers, allowing easy integration of other crypto primitives.

6.3.4 HPKE encapsulate function

```
static int x25519_encap_with_seed(
    const EVP_HPKE_KEM *kem, uint8_t *out_shared_secret,
    size_t *out_shared_secret_len, uint8_t *out_enc, size_t *out_enc_len,
    size_t max_enc, const uint8_t *peer_public_key, size_t peer_public_key_len,
```

```
    const uint8_t *seed, size_t seed_len) {
  ...
  X25519_public_from_private(out_enc, seed);
  uint8_t dh[X25519_SHARED_KEY_LEN];
  X25519(dh, seed, peer_public_key));

  uint8_t kem_context[2 * X25519_PUBLIC_VALUE_LEN];
  OPENSSL_memcpy(kem_context, out_enc, X25519_PUBLIC_VALUE_LEN);
  OPENSSL_memcpy(kem_context + X25519_PUBLIC_VALUE_LEN, peer_public_key,
    X25519_PUBLIC_VALUE_LEN);

  dhkem_extract_and_expand(kem->id, EVP_sha256(), out_shared_secret, SHA256_DIGEST_LENGTH,
    dh, sizeof(dh), kem_context, sizeof(kem_context));

  *out_enc_len = X25519_PUBLIC_VALUE_LEN;
  *out_shared_secret_len = SHA256_DIGEST_LENGTH;
  return 1;
}
```

Listing 7: X25519 encapsulate original implementation

The KEM encapsulation function generates the ephemeral key pair of the sender (Alice) and computes the shared secret based on the public key of the recipient and the ephemeral private key. The ECDH KEM algorithm does not provide a particular APIs for the encapsulation and the decapsulation functions, therefore, they are implemented inside the HPKE functions. We have kept the original implementation design (Listing 7), where, again, we replace the constants with the value inside the HPKE KEM structure.

```
static int HPKE_encap_with_seed(const EVP_HPKE_KEM *kem, uint8_t *out_shared_secret, size_t
    *out_shared_secret_len, uint8_t *out_enc, size_t *out_enc_len, size_t max_enc, const
    uint8_t *peer_public_key, size_t peer_public_key_len, const uint8_t *seed, size_t
    seed_len) {
...
  // create the public key for ECC
    X25519_public_from_private(out_enc, seed);

  // create the shared secret
  uint8_t *hybrid_ss = malloc(sizeof(uint8_t) * (kem->public_key_len + kem->
    PQ_shared_secret_len));
 ...
    X25519(hybrid_ss, seed, peer_public_key));

  // add space for the SIKE pk's to the kem_context
  uint8_t *kem_context = malloc(sizeof(uint8_t) * (2 * kem->public_key_len + kem->
    PQ_public_key_len + kem->PQ_ciphertext_len));
  OPENSSL_memcpy(kem_context, out_enc, kem->public_key_len);
  OPENSSL_memcpy(kem_context + kem->public_key_len, peer_public_key, kem->public_key_len);
  ...
    enc_res = pq_enc[kem->id % SUPPORTED_PQ_ALGORITHMS](out_enc + kem->public_key_len,
    hybrid_ss + kem->public_key_len, peer_public_key + kem->public_key_len);

  OPENSSL_memcpy(kem_context + 2 * kem->public_key_len, out_enc + kem->public_key_len, kem->
    PQ_ciphertext_len);
  OPENSSL_memcpy(kem_context + 2 * kem->public_key_len + kem->PQ_ciphertext_len,
    peer_public_key + kem->public_key_len, kem->PQ_public_key_len);

  dhkem_extract_and_expand(kem->id, EVP_sha256(), out_shared_secret, SHA256_DIGEST_LENGTH,
    hybrid_ss, (kem->public_key_len + kem->PQ_shared_secret_len), kem_context, (2 * kem->
    public_key_len + kem->PQ_public_key_len + kem->PQ_ciphertext_len)));

  *out_enc_len = kem->public_key_len + kem->PQ_ciphertext_len;
  *out_shared_secret_len = SHA256_DIGEST_LENGTH;
  free(hybrid_ss);
```

```
  free(kem_context);
  return 1;
}
```

Listing 8: HPKE encapsulate PQ implementation

We have increased the size of the shared secret and the `kem_context` (Listing 8) which contains the values of the public keys (and ciphertexts in the case of PQ primitives), where by dynamically allocating memory, we define their sizes again based on the HPKE KEM field values. Therefore, independently of the algorithms used, the length of the shared_secret variable will always equal to the length of the classical scheme shared secret added to the length of the PQ shared secret. The case of non-hybrid HPKE, the value of a given length filed will equal to zero, therefore, no need of change in the addition is required. This design decision again facilitates the integration of other crypto primitives into the HPKE protocol.

At the end of the encapsulation mechanism we input both, the ECC shared secret and the PQ shared secret to the KDF, particular for the ECDH KEM, where we obtain a fixed-length key.

6.3.5 HPKE decapsulate function

```
static int x25519_decap(const EVP_HPKE_KEY *key, uint8_t *out_shared_secret, size_t *
    out_shared_secret_len, const uint8_t *enc, size_t enc_len) {

  uint8_t dh[X25519_SHARED_KEY_LEN];
  X25519(dh, key->private_key, enc);

  uint8_t kem_context[2 * X25519_PUBLIC_VALUE_LEN];
  OPENSSL_memcpy(kem_context, enc, X25519_PUBLIC_VALUE_LEN);
  OPENSSL_memcpy(kem_context + X25519_PUBLIC_VALUE_LEN, key->public_key,
    X25519_PUBLIC_VALUE_LEN);

  dhkem_extract_and_expand(key->kem->id, EVP_sha256(), out_shared_secret,
    SHA256_DIGEST_LENGTH, dh, sizeof(dh), kem_context, sizeof(kem_context));

  *out_shared_secret_len = SHA256_DIGEST_LENGTH;
  return 1;
}
```

Listing 9: X25519 decapsulation original implementation

The decapsulation function takes as an input the value of the recipient private key and the public key (or ciphertext in the case of PQ crypto scheme) of the sender. Similar changes are applied to the original decapsulation function (Listing 9). We show the PQ HPKE decapsulate funciton in Listing 10.

```

static int HPKE_decap(const EVP_HPKE_KEY *key, uint8_t *out_shared_secret,  size_t *
    out_shared_secret_len, const uint8_t *enc,  size_t enc_len) {
  uint8_t *hybrid_ss = malloc(sizeof(uint8_t) * (key->kem->public_key_len + key->kem->
    PQ_shared_secret_len));

  !X25519(hybrid_ss, key->private_key, enc);

  uint8_t *kem_context = malloc(sizeof(uint8_t) * (2 * key->kem->public_key_len + key->kem->
    PQ_public_key_len + key->kem->PQ_ciphertext_len));
  OPENSSL_memcpy(kem_context, enc, key->kem->public_key_len);
  OPENSSL_memcpy(kem_context + key->kem->public_key_len, key->public_key, key->kem->
    public_key_len);

```

```
    enc_res = pq_dec[key->kem->id % SUPPORTED_PQ_ALGORITHMS](hybrid_ss + key->kem->
    public_key_len, enc + key->kem->public_key_len, key->private_key + key->kem->
    private_key_len);

  OPENSSL_memcpy(kem_context + 2 * key->kem->public_key_len, enc + key->kem->public_key_len,
     key->kem->PQ_ciphertext_len);
  OPENSSL_memcpy(kem_context + 2 * key->kem->public_key_len + key->kem->PQ_ciphertext_len,
    key->public_key + key->kem->public_key_len, key->kem->PQ_public_key_len);

  dhkem_extract_and_expand(key->kem->id, EVP_sha256(), out_shared_secret,
    SHA256_DIGEST_LENGTH, hybrid_ss, key->kem->public_key_len + key->kem->
    PQ_shared_secret_len, kem_context, (2 * key->kem->public_key_len + key->kem->
    PQ_public_key_len +  key->kem->PQ_ciphertext_len));

  *out_shared_secret_len = SHA256_DIGEST_LENGTH;
  free(hybrid_ss);
  free(kem_context);
  return 1;
}
```

Listing 10: HPKE decapsulation PQ implementation

6.4 PSK operation mode changes

The implementation of the Pre-Shared Secret Mode allows the authentication of the sender of the data. Therefore, based on a key, established in a previous session between the sender and the recipient, the symmetric communication among the parties continues only in case of correct authentication.
As described in section 3, the PSK value is used as a Input Key Material (ikm) for the extract function (part of the KDF) inside the key schedule subroutine. The base mode of operation will use NULL as ikm value and 0 as the length of the ikm. The implementation of the PSK mode requires to pass the value and its length to the KDF as a ikm in order to obtain a pseudo-random key (prk) with fixed length, used for the derivation of the common shared secret later.

```
int EVP_HPKE_CTX_setup_sender_with_seed_for_testing(
    EVP_HPKE_CTX *ctx, uint8_t *out_enc, size_t *out_enc_len, size_t max_enc,
    const EVP_HPKE_KEM *kem, const EVP_HPKE_KDF *kdf, const EVP_HPKE_AEAD *aead,
    const uint8_t *peer_public_key, size_t peer_public_key_len,
    const uint8_t *info, size_t info_len, const uint8_t *seed,
    size_t seed_len);

int EVP_HPKE_CTX_setup_recipient(EVP_HPKE_CTX *ctx, const EVP_HPKE_KEY *key,
                                 const EVP_HPKE_KDF *kdf,
                                 const EVP_HPKE_AEAD *aead, const uint8_t *enc,
                                 size_t enc_len, const uint8_t *info,
                                 size_t info_len);
```

Listing 11: HPKE BASE mode function APIs

```
int EVP_HPKE_CTX_setup_sender_with_seed_for_testing(
    EVP_HPKE_CTX *ctx, uint8_t *out_enc, size_t *out_enc_len, size_t max_enc,
    const EVP_HPKE_KEM *kem, const EVP_HPKE_KDF *kdf, const EVP_HPKE_AEAD *aead,
    const uint8_t *peer_public_key, size_t peer_public_key_len,
    const uint8_t *info, size_t info_len, const uint8_t *seed, size_t seed_len, const
    uint8_t * psk, size_t psk_len, const uint8_t * psk_id, size_t psk_id_len);

int EVP_HPKE_CTX_setup_recipient(
    EVP_HPKE_CTX *ctx, const EVP_HPKE_KEY *key, const EVP_HPKE_KDF *kdf,
    const EVP_HPKE_AEAD *aead, const uint8_t *enc, size_t enc_len,
```

```
10      const uint8_t *info, size_t info_len, const uint8_t * psk, size_t psk_len, const uint8_t
         * psk_id, size_t psk_id_len);
```

Listing 12: HPKE PSK mode function APIs

For the implementation of these operation mode, we had to modify the API of several functions in order to pass the value of the psk and the value of the psk_id along with their lengths form the user application to the low level HPKE functions.
The original interface used for the BASE operation mode is shown in Listing 11 whereas the Listing 12 shows the modifications applied consisting of adding the value of the pre-shared key, the length, the value of the pre-shared key id and the length. We had to change the interface of the `hpke_key_schedule` function similarly.
Inside the `hpke_key_schedule` function we modified the function calls to the extract functions, passing the value of the psk and psk length or psk is and psk id length as ikm when required by the HPKE standard. The original calls of the extract functions are shown on Listing 13 and the changes on Listing 14.

```
1  hpke_labeled_extract(hkdf_md, psk_id_hash, &psk_id_hash_len, NULL, 0,
2                               suite_id, sizeof(suite_id), "psk_id_hash", NULL,
3                               0);
4
5  hpke_labeled_extract(hkdf_md, secret, &secret_len, shared_secret,
6                               shared_secret_len, suite_id, sizeof(suite_id),
7                               "secret", NULL, 0)
```

Listing 13: HPKE BASE mode function APIs

```
1  hpke_labeled_extract(hkdf_md, psk_id_hash, &psk_id_hash_len, NULL, 0,
2                               suite_id, sizeof(suite_id), "psk_id_hash", psk_id,
3                               psk_id_len);
4
5  hpke_labeled_extract(hkdf_md, secret, &secret_len, shared_secret,
6                               shared_secret_len, suite_id, sizeof(suite_id),
7                               "secret", psk, psk_len);
```

Listing 14: HPKE PSK mode function APIs

The HPKE PSK operation mode requires the implantation of and additional (Listing 16) function to verify the consistency of the psk, `psk_id` and their lengths. The function is specified in the HPKE standard draft [6] as VerifyPSKInputs. It can be declared as a static function, therefore, there is no need of previous prototype declaration. Additionally, we have defined three new error codes (Listing 15) inside the `/aws-lc/include/openssl/evp_errors.h`.

```
1  #define EVP_R_INCONSISTENT_PSK_INPUTS 138
2  #define EVP_R_PSK_PROVIDED_WHEN_NOT_NEEDED 139
3  #define EVP_R_PSK_MISSING 140
```

Listing 15: HPKE PSK new error codes

```
1  static int VerifyPSKInputs(uint8_t mode, const uint8_t *psk, size_t psk_len, const uint8_t *
       psk_id, size_t psk_id_len){
2
3    if ((!(psk == NULL) && (psk_id == NULL)) ||
```

```
      (psk == NULL && (!(psk_id == NULL)))) {
    OPENSSL_PUT_ERROR(EVP, EVP_R_INCONSISTENT_PSK_INPUTS);
    return 0;
  }

  if (mode == HPKE_MODE_BASE && psk != NULL) {
    OPENSSL_PUT_ERROR(EVP, EVP_R_PSK_PROVIDED_WHEN_NOT_NEEDED);
    return 0;
  }

  if (mode == HPKE_MODE_PSK && psk == NULL) {
   OPENSSL_PUT_ERROR(EVP, EVP_R_PSK_MISSING);
    return 0;
  }

return 1;
}
```

Listing 16: HPKE PSK correct input verification function

6.5 ASM integration

For better results of the integrated post quantum schemes, we have added the assembly implementation of both SIKE and Kyber KEMs. We have made changes to the `CMakeList.txt` files corresponding to both protocols, where we have changes the list of compiled files depending on the platform, on which the code is being compiled. SIKE offers ARM64 and x64 assembly optimizations and Kyber offers avx2 optimizations. Our measurements are based on the assembly code of the PQ KEMs.

6.6 Tests and benchmark changes

Several modifications were required for the testing of the functionality of PQ HPKE and for the benchmarking of the protocol. The changes and modifications to the code are specified below.

6.6.1 Tests functions

For the testing of the implementation in BASE and PSK mode we have mainly focused on the functions HPKEVerifyTestVectors and HPKERoundTrip. The first one is using the test vectors to obtain an encryption ciphertext and compare it with the expected value specified in the standard draft. The HPKE standard, however, integrates only ECC KEMs, therefore, we could not use the testvectors for verifying the results after integrating the PQ algorithms. For testing the correctness of the PQ HPKE implementation we used the HPKERoundTRip test, where an entire HPKE execution is performed: the recipient has his static key pair issued, the sender is setup, the recipient is setup, the sender encrypts a message using his/her own symmetric key, the recipient decrypts the ciphertext from the sender using his/her symmetric key and the original and decrypted messages are compared.

We have implemented a separate test function for each one of the HPKE combinations of underlying primitives, e.g. SIKE, x25519_SIKE, Kyber, x25519_Kyber, which test a specific version of the HPKE variants. Each one of the tests initialized the KEM with the corresponding parameter lengths and executed the entire round of the HPKE. The completion of the test indicates the success of the algorithm. Additionally, each file performs benchmarking of the key generation setup sender setup recipient, seal and open functions.

For easier usage of the HPKE protocol and automated testing, we have modified the original HPKERoundTrip test function, implanting a loop iterating through the entire set of HPKE primitives. Since this function also measures the performance of the protocol, we changed the name to HPKERoundTripBenchmark.

The integration of the PSK mode into the HPKE implementation required also to develop some check statements when passing the mode (since it may be 0 in BASE mode and 1 in PSK mode). To ensure the implementation of the PSK mode is correct, we had to update the test vector file, including the test vectors in PSK mode. The file `hpke_test_vecotrs.txt` is created using the python file `translate_test_vectors.py` and the json file `test-vectors.json`. For the creation of the text file we have modified the filtering inside the python file in order to include the PSK mode tests as well.

The have modified the test function verifyTestVector. We added variables for the `psk and psk_id` values, read them from the file (only in PSK mode) and pass them to the setup sender and setup recipient functions. In BASE mode NULL values are passed to the functions with corresponding zero lengths. After running the HPKEVerifyTestVectors function, all the tests for BASE and PSK modes are executed and the values of the encrypted plaintext is compared with the expected value, specified in the standard draft.

The automation of the HPKERoundTrip test allows the execution of all the HPKE different combination of underlying primitives. To do so we implemented a loop iterating through the PQ HPKE versions. To declare variables of different lengths for the keys we decided to implement functions that return the parameter length per algorithm (Listing 17). We have considered also the implementation of macros. However, macros in C/C++ only consume constants and do not allow variables as arguments. Therefore, to avoid the large logic blocks inside the testing function, we have created a set of auxiliary functions inside the aux_functions.c file. Their implementation also avoids having the user to input the desired mode as a constant before compilation (which made the automated testing possible). Apart from the straightforward functions returning the lengths of the secret key, public key and ciphertext, we have included a function algorithm_kdf which creates an HPKE KEM and return a structure, initialized with the correct parameter set lengths. The rest of the functions simply facilitate the printing of the information (through the standard output or a file output stream).

```
int algorithm_secretkeybytes(int alg);
int algorithm_publickeybytes(int alg);
int algorithm_ciphertextbytes(int alg);
const EVP_HPKE_KEM *algorithm_kdf(int alg);
void print_info(int aead, int kdf, int alg);
void print_info_file(int aead, int kdf, int alg, std::ofstream &MyFile);
void init_plaintext(uint8_t *plaintext, int size);
void print_text(std::vector<uint8_t> cleartext, int cleartext_len);
```

Listing 17: HPKE auxiliary functions for automated testing of all HPKE classical, post-quantum and hybrid variants

6.6.2 Benchmark functions

For measuring the performance of the HPKE protocol based on different underlying crypto primitives, we have used the dedicated testing functions: x25519, SIKE, x25519_SIKE, Kyber, x25519_Kyber and the automated test function HPKERoundTripBenchmark. The clock cycles required per function are interpreted and are printed into a file stored into `aws-lc/results/HPKE_XXX_results.txt`, with XXX being the name of the underlying primitives concatenated with an underscore between them (if more than one).

The platform specifications and the surrounding conditions (such as power supply, number of running programs, etc.) can impact the number of clock cycles obtained. Therefore, we integrate additional functions to obtain the statistical information about the results and analyze them better. We have created a separate file aux_functions.cc with header aux_functions.h (Listing 18), where we implement functions for sorting of the array values, computing the mean, median, standard deviation and quartiles values. For the analysis of the performance we collect the clock cycles per function for the number of tests specified by the user. Furthermore, for each test we input different info and ad values, which emulate different scenarios for the key scheduling subroutine. Therefore, the number of samples we obtain is the number of tests $\times$ the number of different combinations of ad and info values sets. For analysing these samples reliably, we first sort the array and then eliminate the lowest and the highest 25% of the set of samples. Therefore, we eliminate some extreme scenarios where the multitasking slows or benefits the performance.

```
float mean(unsigned long long array[], int n);
float median(unsigned long long array[], int n);
void sort_array(unsigned long long arr[], int n);
double standarddeviation(unsigned long long array[], const int n);
void calculate_quartiles(unsigned long long arr[], int n, float quartiles[4],
                         int quartiles_positions[4]);
float analyze_statistics(uint8_t mode, unsigned long long arr_cycles[], int n,
                         std::ofstream &MyFile);
void analyze_percentage(unsigned long long cycles_set_up_sender_total,
                        unsigned long long cycles_set_up_recipient_total,
                        unsigned long long cycles_seal_total,
                        unsigned long long cycles_open_total,
                        unsigned long long clean_protocol,
                        std::ofstream &MyFile);
void analyze_protocol(uint8_t mode, unsigned long long *arr_cycles_setup_sender,
                      unsigned long long *arr_cycles_setup_recipient,
                      unsigned long long *arr_cycles_seal,
                      unsigned long long *arr_cycles_open, int n,
                      std::ofstream &MyFile);
```

Listing 18: HPKE auxiliary functions for statistical analysis of the collected performance samples

For the better user experience, we defined a constant called ANALYZE_RESULTS_MODE into the hpke_test.cc file. The user can modify the value to add more details to the statistical analysis of the data. In particular, when the variable is set to zero (the default value) the analysis omits to print the details to the output files. The information printed consists of analysis of the middle 50% of collected data and outputs the average number of clock cycles required per function. We report this information about the `HPKE_generate_key`, `HPKE_setup_sender`, `HPKE_setup_recipient`, `HPKE_seal` and `HPKE_open` functions. We also report the total protocol timing and the percentage that each function takes. In case the ANALYZE_RESULTS_MODE variable is set to one, the output files will also contain information about the median and the standard deviation for each one of the functions (again, considering only the middle 50% of collected data).

Similar measurement was performed targeting the RSA-encrypt since it is used as an encryption scheme. For the measurement of the RSA-encrypt, since it limits the plaintext by the modulus value, we have measured the encryption and decryption performance for the maximum plaintext length depending on the different key sizes and the padding modes. We have measured RSA-2048, RSA-3072 and RSA-4096 using the `RSA_PKCS1_PADDING`, `RSA_PKCS1_OAEP_PADDING` and `RSA_NO_PADDING` modes. The padding mode require 11, 42 and 0 bytes reserved, respectively, therefore, additionally shrink the plaintext size. To provide more comparable results to the HPKE protocol (since it does not impose limitation in the plaintext size) we have performed RSA-encrypt measurements for longer plaintext sizes (up to 1MB) where we split the input information into different chunks of the maximum plaintext size per benchmarked algorithm mode and report the encryption and decryption of all the separate pieces of information- using the same key. This experiment

does not ensure security due the the reuse of the keys and is only performed for the sake of HPKE comparison.

7 Performance Evaluation

Our measurements are performed on an `ec2` instance running **Linux 20.04 OS** running on an **8-core Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz** with **8GB RAM**. We may consider benchmarking the code after the completion of the project to reflect better the performance. However, the main focus of the project is the implementation of the post-quantum HPKE. Furthermore, the additional functions for removing the outliers of the collected samples provides more precise evaluation of the performance data.

For the evaluation of the data we show exhaustive table with the information about RSA-encrypt (currently used in may Amazon applications) and the data collected using HPKE in the different classical, post-quantum and hybrid variants. In particular, we measure the performance of RSA-encrypt using key sizes of 2048, 3072 and 4096-bit lengths. We report the data collected using the `RSA_PKCS1_OAEP_PADDING` mode in Table 7. Due to the amount of different modes, plaintext sizes and key lengths we represent only one of the modes in the body of the documentation. The rest of the collected and analyzed data, including all the RSA-encrypt padding modes can be found in section 7. In Table 7 we show the performance of (PQ) HPKE including all modes: x25519, SIKE, x25519 and SIKE, Kyber, x25519 and Kyber. We analyze the data collected for the setup of the sender and the recipient. These functions (with and overhead due to the protocol execution) show mainly the execution time of the key encapsulation and decapsulation functions, respectively. We report the performance of the seal and open functions as well, where they correspond to the symmetric encryption and decryption functions. We report the collected measurements for the corresponding maximum plaintext sizes for RSA-encrypt 2048, 3072 and 4096 for comparison purposes. Again, for simplicity and clarity purposes, we show the results of only one AEAD primitive, particularly, AES_256_GCM. The benchmark of the other two AEAD underlying primitives are reported in section 7. For precise evaluation of the performance we use 1,000 executions with 9 different (combinations of) info and ad value sets (subsubsection 6.6.2), therefore, we run 9,000 executions for the collection of the data.

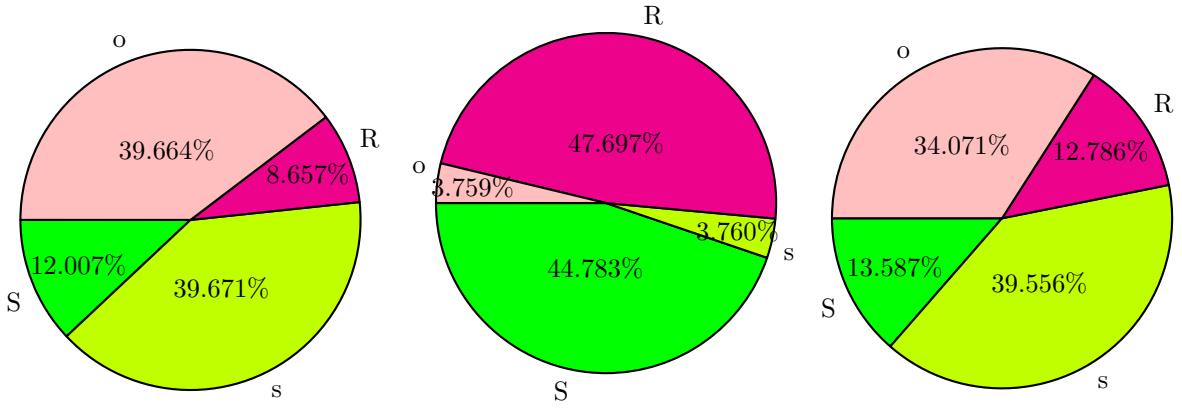
Function	Pt	RSA-encrypt			HPKE with AES_256_GCM														
		2048	3072	4092	x25519			Kyber			x25519_Kyber			SIKE			x25519_SIKE		
	[B]		[CC×10^3]		[CC×10^3]	[%]	Total	[CC×10^3]	[%]	Total	[CC×10^3]	[%]	Total	[CC×10^3]	[%]	Total	[CC×10^3]	[%]	Total
KG	214	152,192	-	-	50.79	-	372	40.05	-	194	99.170	-	564	5,225.65	-	17,687	5,278.63	-	18,007
S		-	-	-	211.16	56.746		103.58	53.313		318.24	56.369		8,523.45	48.19		8,709.30	48.365	
R		-	-	-	159.61	42.893		89.12	45.874		244.71	43.345		9,162.03	51.8		9,296.56	51.626	
s		53	-	-	0.69	0.187		0.79	0.407		0.83	0.146		0.89	0.005		0.88	0.005	
o		1,655	-	-	0.65	0.175		0.79	0.406		0.79	0.140		0.8	0.005		0.80	0.004	
KG	342	-	868,299	-	50.79	-	372	40.05	-	194	99.170	-	564	5,225.65	-	17,686	5,278.63	-	18,007
S		-	-	-	211.18	56.709		103.49	53.228		318.13	56.340		8,523.67	48.188		8,708.89	48.364	
R		-	-	-	159.61	42.861		89.10	45.828		244.64	43.326		9,161.64	51.801		9,296.22	51.625	
s		-	104	-	0.85	0.227		0.91	0.468		0.96	0.170		1.04	0.006		1.04	0.006	
o		-	5044	-	0.76	0.203		0.93	0.477		0.93	0.164		0.93	0.005		0.89	0.005	
KG	470	-	-	2,071,684	50.79	-	372	40.05	-	194	99.170	-	565	5,225.65	-	17,684	5,278.63	-	18,004
S		-	-	-	211.11	56.667		103.46	53.127		318.17	56.311		8,523.34	48.186		8,706.96	48.361	
R		-	-	-	159.59	42.839		89.12	45.764		244.67	43.304		9,160.69	51.801		9,295.05	51.627	
s		-	-	175	0.94	0.253		1.07	0.522		1.10	0.195		1.17	0.007		1.18	0.007	
o		-	-	11,343	0.9	0.241		1.09	0.557		1.08	0.190		1.04	0.006		1.04	0.006	
KG	1K	187,311	1,873,198	5,609,350	50.79	-	373	40.05	-	195	99.170	-	565	5,225.65	-	17,687	5,278.63	-	18,007
S		-	-	-	211.16	56.547		103.52	52.882		318.15	56.224		8,522.93	48.186		8,709.22	48.364	
R		-	-	-	159.63	42.748		89.18	45.559		244.67	43.238		9,161.50	51.797		9,295.60	51.620	
s		263	313	3,005	1.34	0.359		1.53	0.784		1.52	0.269		1.54	0.009		1.54	0.009	
o		8,294	15,127	179,232	1.29	0.345		1.52	0.775		1.52	0.269		1.42	0.008		1.42	0.008	
KG	10K	424,543	937,286	1,993,637	50.79	-	386	40.05	-	210	99.170	-	581	5,225.65	-	17,701	5,278.63	-	18,021
S		-	-	-	211.19	54.653		103.58	49.110		318.27	54.763		8,523.68	48.152		8,709.47	48.327	
R		-	-	-	159.62	41.308		89.10	42.260		244.64	42.094		9,161.92	51.758		9,296.46	51.584	
s		2,478	3,132	31,524	7.84	2.028		9.13	4.327		9.19	1.581		8.07	0.046		8.09	0.045	
o		79,459	153,195	1,878,696	7.77	2.012		9.08	4.304		9.08	1.563		7.87	0.044		7.88	0.044	
KG	100K	80,726	727,418	3,191,746	50.79	-	518	40.05	-	364	99.170	-	734	5,225.65	-	17,830	5,278.63	-	18,151
S		-	-	-	211.29	40.752		103.68	28.467		318.49	43.361		8,521.47	47.792		8,708.16	47.974	
R		-	-	-	159.63	30.789		89.18	24.486		244.62	33.303		9,161.07	51.379		9,295.62	51.211	
s		24,596	30,483	37,211	73.80	14.234		85.68	23.526		85.72	11.670		73.98	0.415		74.08	0.408	
o		791,299	1,494,822	2,436,978	73.76	14.225		85.67	23.521		85.69	11.666		73.79	0.414		73.83	0.407	
KG		253,760	688,264	1,587,592	50.79	-	1,847	40.05	-	1,837	99.170	-	2,151	5,225.65	-	19,174	5,278.63	-	19,495
S		-	-	-	221.87	12.007		135.24	53.228		292.31	13.587		8,543.03	44.554		8,730.86	44.783	
R		-	-	-	159.97	8.657		117.68	45.828		275.06	12.786		9,165.89	47.802		9,298.92	47.697	
s		245,681	304,194	370,868	531.54	39.671		851.29	0.468		850.98	39.556		732.81	3.822		733.10	3.760	
o		7,899,892	14,916,783	24,338,208	532.03	39.664		733.18	0.477		732.98	34.071		733.02	3.823		732.93	3.759	

Table 2: Measure HPKE AES_256_GCM clock cycles (ASM) (avg of 10,000 executions) on an ec2 instance 8-core Intel(R) Core(TM) i7-10610U CPU 8GB RAM @1.80GHz.

The notation in Table 7 shows the function measured in the leftmost column, where **KG**, **S** and **R** denote the **HPKE key generation**, **HPKE setup Sender** and **HPKE setup Recipient**, including mainly the performance of the KEM key generation, encapsulation and decapsulation functions, respectively. The **s** and **o** symbols denote the **HPKE seal** and **HPKE open** functions (in Table 7 the performance with underlying AES_256_GCM). The second column shows the length of the encrypted plaintext in Bytes. We should denote that due to RSA-encrypt `RSA_PKCS1_OAEP_PADDING` the size of the plaintext is 41 bytes less than the actual encrypted data. The following three columns report the collected and analyzed data for RSA-2048, RSA-3072 and RSA-4096, respectively. For RSA-encrypt we report the encryption and decryption steps under the **s** and **o** symbols, respectively. . The rest of Table 7 shows the performance evaluation of HPKE based on the different KEM underlying primitives. We report the total as the sum of the HPKE setup sender, HPKE setup recipient, HPKE seal and HPKE open functions, where we exclude the clock cycles required for the generation of the key pair for the recipient since it is static and is executed only once in the very beginning of the protocol. All clock cycles are reported in **thousands**. Additionally, we present information for the HPKE setup sender, HPKE setup recipient, HPKE seal and HPKE open functions as a percentage of the entire protocol, showing that with increasing the plaintext length, the weight of the KEM protocol decreases.

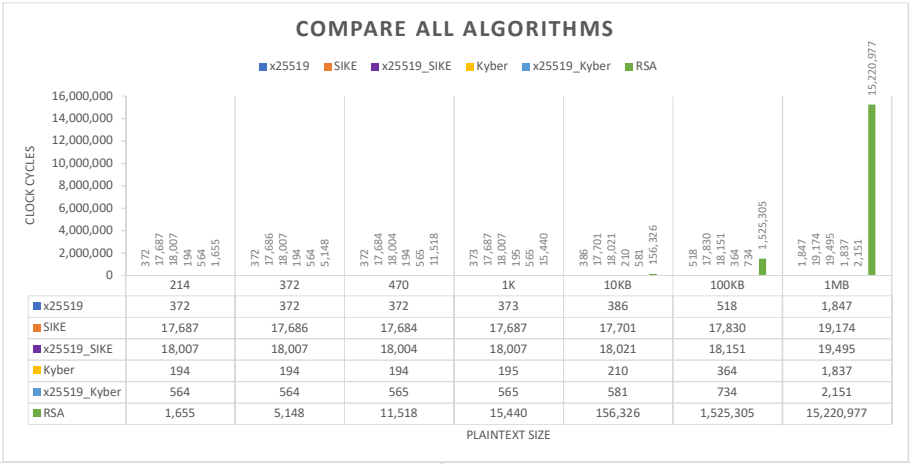
We show the performance of all the evaluated protocols in
The primitives used for the evaluation of the HPKE protocol are X25519 (C code implementation), SIKE (assembly x64 optimizations) and Kyber (avx2 assembly optimizations). It is important to note that the assembly code allows the execution of Kyber in less clock cycles that the used ECC primitive. When more efficient and better studied ECC primitive is added, the classical and hybrid modes will become even more efficient. The assembly optimizations for SIKE target also ARM64 and are included in the code, however, our target platform is Intel based.

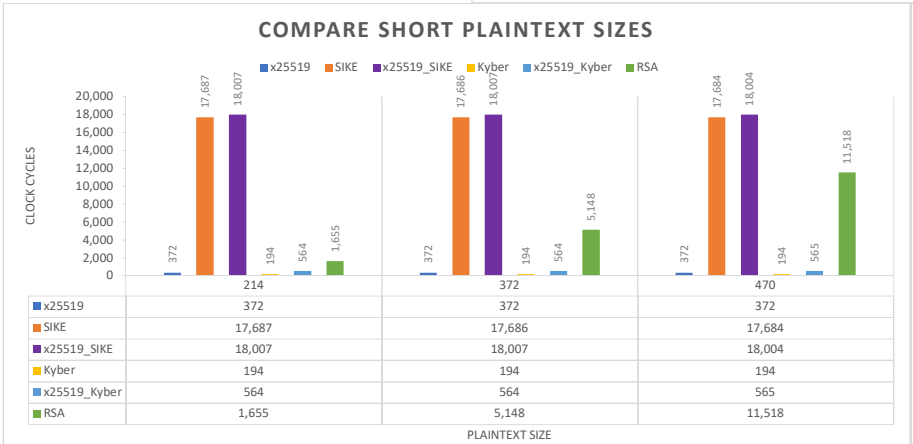
For better visualizing the performance on the client (Sender) and the Server (Recipient) sides, we illustrate the values using pie charts, where we show the Server in pink color and the client in green color. Additionally, we represent the in light pink/green the overall percentage of the KEM dedicated function (client - encapsulate, server - decapsulate) and in dark pink/green the overall percentage of the symmetric primitive (client seal, server - open). We report the perfromance based on 10MB en/decrypted data, using AES_256_GCM AEAD primitive.

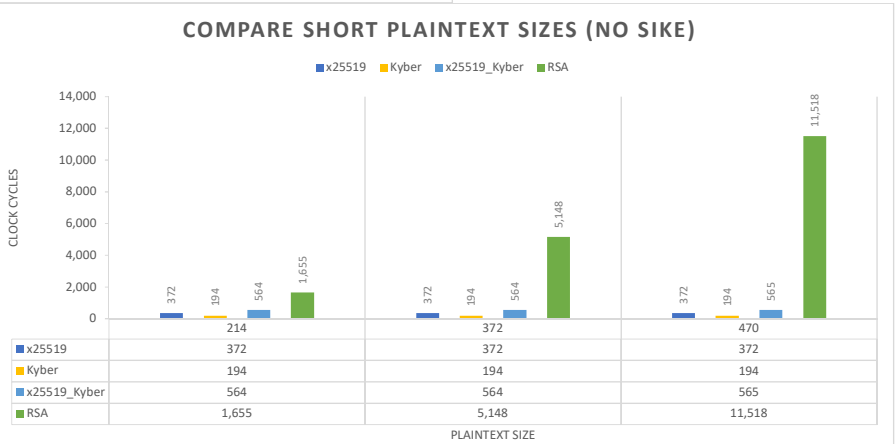


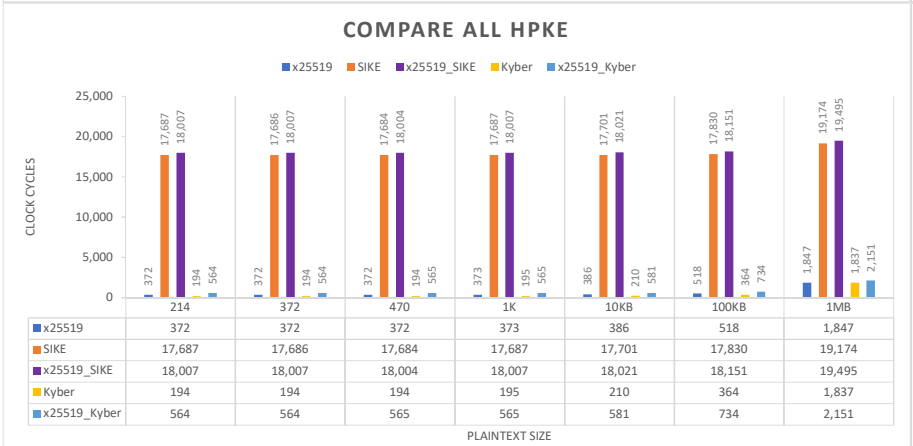
(a) x25519. (b) x25519 and SIKE. (c) x25519 and Kyber.

Figure 17: HPKE using AES_256_GCM function performance for encryption of 1MB plaintext.









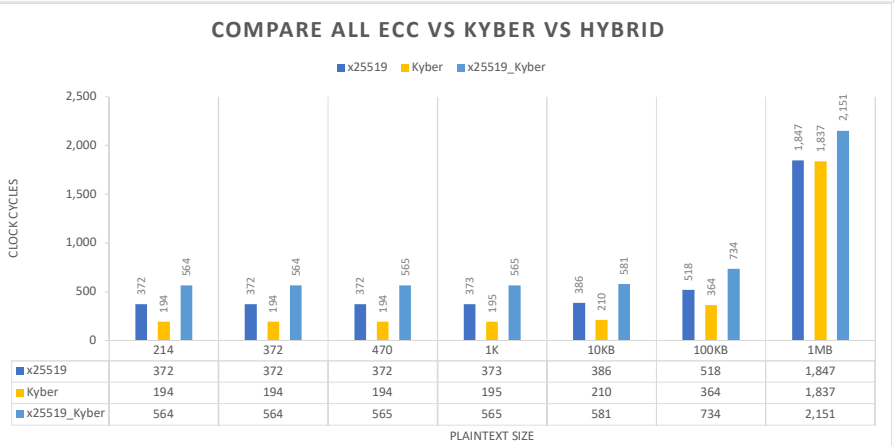


Figure 18: Total timing of protocols [CC $\times 10^3$]

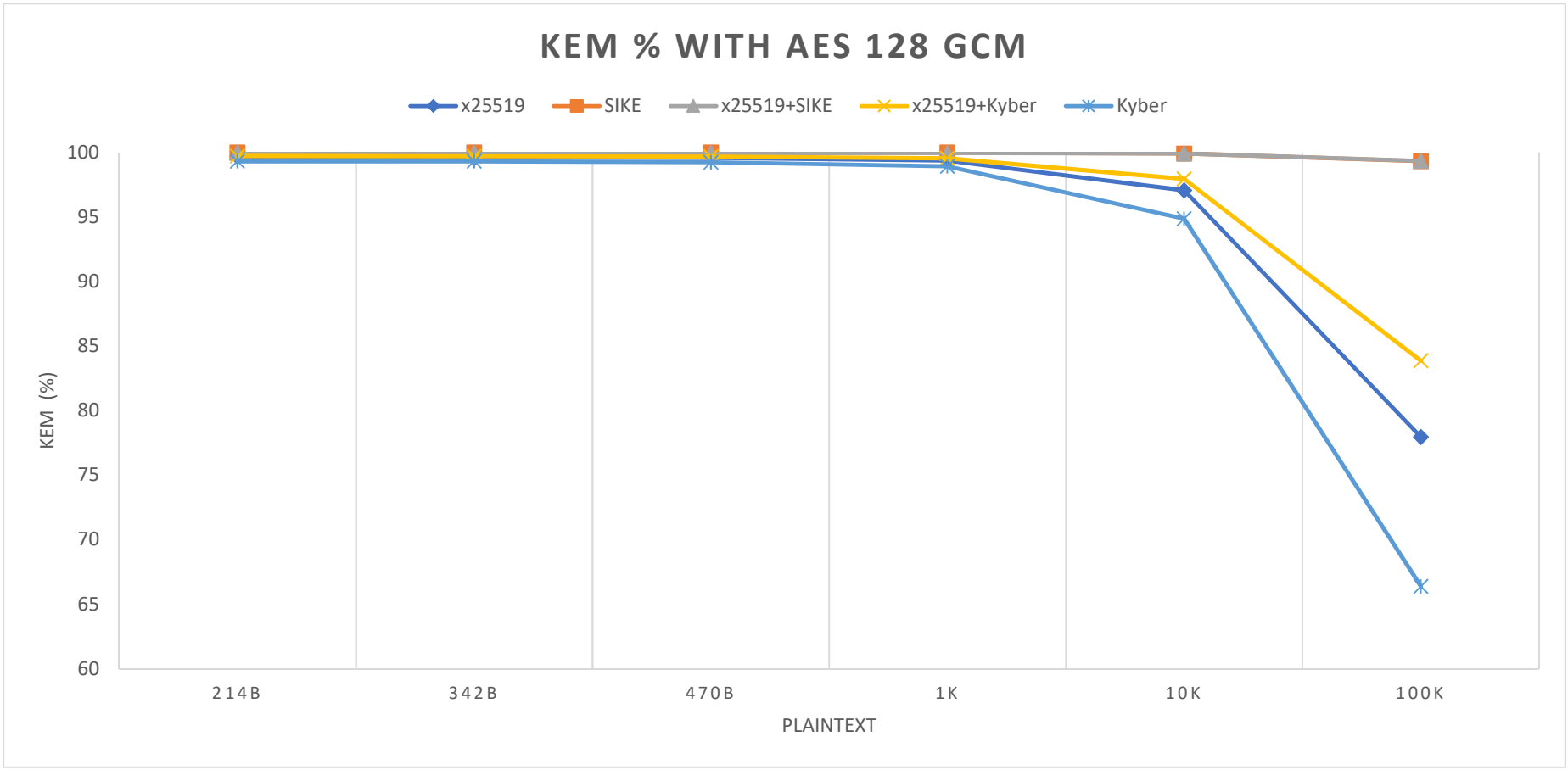


Figure 19: Percentage time use of KEMs

References

[1] Joël Alwen, Bruno Blanchet, Eduard Hauck, Eike Kiltz, Benjamin Lipp, and Doreen Riepel. Analysing the hpke standard. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 87–116. Springer, 2021.

[2] Mila Anastasova, Reza Azarderakhsh, and Mehran Mozaffari Kermani. Fast strategies for the implementation of sike round 3 on arm cortex-m4. *IACR Cryptol. ePrint Arch.*, 2021:115, 2021.

[3] X9 ANSI. 63: Elliptic curve key agreement and key transport protocols. *Working Draft, Oct*, 1998.

[4] Reza Azarderakhsh, David Jao, Kassem Kalach, Brian Koziel, and Christopher Leonardi. Key compression for isogeny-based cryptosystems. In *Proceedings of the 3rd ACM International Workshop on ASIA Public-Key Cryptography*, pages 1–10, 2016.

[5] R Barnes, K Bhargavan, B Lipp, and C Wood. Hybrid public key encryption (2021).

[6] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. Hybrid Public Key Encryption. Internet-Draft draft-irtf-cfrg-hpke-12, Internet Engineering Task Force, September 2021. Work in Progress.

[7] Matthew Campagna, Craig Costello, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, David Urbanik, et al. Supersingular isogeny key encapsulation. 2019.

[8] Rami Elkhatib, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. Efficient and fast hardware architectures for sike round 2 on fpga. *Cryptology ePrint Archive*, 2020.

[9] Rami Elkhatib, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. Highly optimized montgomery multiplier for sike primes on fpga. In *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*, pages 64–71. IEEE, 2020.

[10] Eiichiro Fujisaki and Tatsuaki Okamoto. How to enhance the security of public-key encryption at minimum cost. In *International Workshop on Public Key Cryptography*, pages 53–68. Springer, 1999.

[11] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *International Workshop on Post-Quantum Cryptography*, pages 19–34. Springer, 2011.

[12] Burton S Kaliski. Ieee p1363: A standard for rsa, diffie-hellman, and elliptic-curve cryptography. In *International Workshop on Security Protocols*, pages 117–118. Springer, 1996.

[13] Brian Koziel, A-Bon Ackie, Rami El Khatib, Reza Azarderakhsh, and Mehran Mozaffari Kermani. Sike'd up: Fast hardware architectures for supersingular isogeny key encapsulation. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(12):4842–4854, 2020.

[14] Benjamin Lipp. *An analysis of hybrid public key encryption.* PhD thesis, IACR Cryptology ePrint Archive, 2020.

[15] Microsoft. Pqcrypto-sidh.

[16] National Institute of Standards and Technology. Security requirements for cryptographic modules. Technical Report Federal Information Processing Standards Publications (FIPS PUBS) 140-2, Change Notice 2 December 03, 2002, U.S. Department of Commerce, Washington, D.C., 2001.

[17] Eric Rescorla et al. Diffie-hellman key agreement method. 1999.

[18] Elliptic Curve Cryptography SEC. Standards for efficient cryptography group, 2000.

[19] Hwajeong Seo, Mila Anastasova, Amir Jalali, and Reza Azarderakhsh. Supersingular isogeny key encapsulation (sike) round 2 on arm cortex-m4. *IEEE Transactions on Computers*, 2020.

[20] Hwajeong Seo, Pakize Sanal, Amir Jalali, and Reza Azarderakhsh. Optimized implementation of sike round 2 on 64-bit arm cortex-a processors. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(8):2659–2671, 2020.

[21] Seog Chung Seo. Sike on gpu: Accelerating supersingular isogeny-based key encapsulation mechanism on graphic processing units. *IEEE Access*, 9:116731–116744, 2021.

[22] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[23] Victor Shoup. A proposal for an iso standard for public key encryption (version 2.1). *IACR e-Print Archive*, 112, 2001.

[24] Victor Shoup. Information technology-security techniques-encryption algorithms-part 2: Asymmetric ciphers. *ISO/IEC 18033-2*, 2004.

Appendix

x25519 Key Generation (x25519 represents the original HPKE implementation.)

1. HPKE_KEY key;

HPKE_KEM →

```
Info about KEM
-> public parameters length,
-> keygen, encap, decp functions
id
public_key_len
private_key_len
seed_len
int (*init_key)(EVP_HPKE_KEY *key, const uint8_t *priv_key, size_t priv_key_len);
int (*generate_key)(EVP_HPKE_KEY *key);
int (*encap_with_seed)(const EVP_HPKE_KEM *kem, uint8_t *out_ss, size_t *out_ss_len,
             uint8_t *out_enc, size_t *out_enc_len, size_t max_enc,
             const uint8_t *peer_public_key, size_t peer_public_key_len,
             const uint8_t *seed, size_t seed_len);
int (*decap)(const EVP_HPKE_KEY *key, uint8_t *out_ss, size_t *out_ss_len,
              const uint8_t *enc, size_t enc_len);
```

↗

HPKE_KEY → public key[X25519_PUBLIC_KEY_LEN]

↘ secret key[X25519_PRIVATE_KEY_LEN]

2. EVP_HPKE_KEY_generate(key.get(), EVP_hpke_x25519_hkdf_sha256());

EVP_HPKE_KEM EVP_hpke_x25519_hkdf_sha256() →

```
Assign the KEM var info about the params
static const EVP_HPKE_KEM kKEM = {
  /*id=*/ EVP_HPKE_DHKEM_X25519_HKDF_SHA256,
  /*public_key_len=*/ X25519_PUBLIC_KEY_LEN,
  /*private_key_len=*/ X25519_PRIVATE_KEY_LEN,
  /*seed_len=*/ X25519_PRIVATE_KEY_LEN,
  x25519_init_key,
  x25519_generate_key,
  x25519_encap_with_seed,
  x25519_decap};
return &kKEM;
```

2.1 key->kem=kem;

2.2 kem->generate_key(key);

2.2.1 X25519_keypair(key->public_key, key->private_key);

2.2.1.1 private_key ← random value;

2.2.1.2 X25519_public_from_private(public_key, private_key);

2.2.1.2.1 x25519_ge_scalarmult_base(public_key, private_key);

Just point multiplication. $pk_R = [sk_R] \cdot P$

3. EVP_HPKE_KEY_public_key(key.get(), public_key_r, &public_key_r_len, sizeof(public_key_r));

Copy the pk into public_key_r & update the public_key_r_len value.

x25519 Set Up Sender (x25519 represents the original HPKE implementation.)

1. HPKE_CTX sender_ctx;
uint8_t enc[X25519_PUBLIC_KEY_LEN];
size_t enc_len;

HPKE_CTX →

```
Info about HPKE
 -> the Key Derivation Function (KDF),
-> the Authenticated Encryption with Additional Data (AEAD),
-> the nonce, ...
const EVP_HPKE_AEAD *aead;
const EVP_HPKE_KDF *kdf;
EVP_AEAD_CTX aead_ctx;
uint8_t base_nonce[EVP_AEAD_MAX_NONCE_LENGTH];
uint8_t exporter_secret[EVP_MAX_MD_SIZE];
uint64_t seq;
int is_sender;
```

2. EVP_HPKE_CTX_setup_sender(sender_ctx.get(), enc, &enc_len, sizeof(enc), EVP_hpke_x25519_hkdf_sha256(), kdf(), aead(), public_key_r, public_key_r_len, info.data(), info.size());

EVP_HPKE_KEM EVP_hpke_x25519_hkdf_sha256() →

```
Assign the same KEM var info about the params
static const EVP_HPKE_KEM kKEM = {
  /*id=*/ EVP_HPKE_DHKEM_X25519_HKDF_SHA256,
  /*public_key_len=*/ X25519_PUBLIC_KEY_LEN,
  /*private_key_len=*/ X25519_PRIVATE_KEY_LEN,
  /*seed_len=*/ X25519_PRIVATE_KEY_LEN,
  x25519_init_key,
  x25519_generate_key,
  x25519_encap_with_seed,
  x25519_decap};
return &kKEM;
```

2.1 int8_t seed[MAX_SEED_LEN]; → seed represents the pk of Bob(Sender) -> sk_S
RAND_bytes(seed, kem->seed_len); → MAX_SEED_LEN = X25519_PRIVATE_KEY_LEN

2.2 EVP_HPKE_CTX_setup_sender_with_seed_for_testing(ctx, out_enc, out_enc_len, max_enc, kem, kdf, aead, peer_public_key, peer_public_key_len, info, info_len, seed, kem->seed_len);
Pass the same args as in setup_sender + the seed value & length -> sk & length

2.3. uint8_t ss[MAX_SHARED_SECRET_LEN]; → MAX_SHARED_SECRET_LEN = SHA256_DIGEST_LENGTH
size_t ss_len;

2.4. kem->encap_with_seed(kem, ss, &ss_len, out_enc, out_enc_len, max_enc, peer_public_key, peer_public_key_len, seed, seed_len);

→

```
Create pk of Bob(Sender) & find the shared secret
X25519_public_from_private(out_enc, seed); → Again just point multiplication
                                ↓      ↓
                                pk_B   sk_B
X25519(dh, seed, peer_public_key); → Point multiplication ->
                                     dh = [seed] · peer_public_key
                                     dh = [sk_B] · pk_A = [sk_B] · [sk_A] · P
dhkem_extract_and_expand(kem->id, EVP_sha256(),
  ss, SHA256_DIGEST_LENGTH, dh, sizeof(dh), kem_context, sizeof(kem_context));
  ↓              ↓          ↓
out_key      out_key_len  shared secret
  *out_enc_len = X25519_PUBLIC_VALUE_LEN;
  *out_shared_secret_len = SHA256_DIGEST_LENGTH;
```

2.5. hpke_key_schedule(ctx, ss, sizeof(ss), info, info_len);

x25519 Set Up Recipient(x25519 represents the original HPKE implementation.)

1. EVP_HPKE_CTX_setup_recipient(recipient_ctx.get(), key.get(), kdf(), aead(), enc, enc_len,
   info.data(), info.size()); ↓ sk_A ↓ pk_B

   1.2. key->kem->decap(key, ss, &ss_len, enc, enc_len);

   → Find the shared secret

   X25519(dh, key->private_key, enc); → ss = $[sk_A] \cdot$ enc =
   $= [sk_A] \cdot pk_B =$
   $= [sk_A] \cdot [sk_B] \cdot P$
   x25519_scalar_mult(dh, private_key, peer_public_value);
   Just the multiplication
   dhkem_extract_and_expand(key->kem->id, EVP_sha256(),
   ss, SHA256_DIGEST_LENGTH, dh,
   ↓ out_key ↓ out_key_len ↓ shared secret
   sizeof(dh), kem_context, sizeof(kem_context));

   1.2. hpke_key_schedule(ctx, ss, sizeof(ss), info, info_len);

SIKE Key Generation (Recipient)(SIKE represents the PQ only HPKE implementation.)

1. HPKE_KEY key;

HPKE_KEM →

Info about KEM

```
-> public parameters length,
-> keygen, encap, decp functions
id
public_key_len
private_key_len
seed_len
int (*init_key)(EVP_HPKE_KEY *key,
                const uint8_t *priv_key, size_t priv_key_len);
int (*generate_key)(EVP_HPKE_KEY *key);
int (*encap_with_seed)(const EVP_HPKE_KEM *kem,
                uint8_t *out_shared_secret, size_t *out_shared_secret_len,
                uint8_t *out_enc, size_t *out_enc_len, size_t max_enc,
                const uint8_t *peer_public_key, size_t peer_public_key_len,
                const uint8_t *seed, size_t seed_len);
int (*decap)(const EVP_HPKE_KEY *key, uint8_t *out_ss, size_t *out_ss_len,
                const uint8_t *enc, size_t enc_len);
```

HPKE_KEY → public key
HPKE_KEY → secret key

2. EVP_HPKE_KEY_generate(key.get(), EVP_HPKE_KEM EVP_hpke_SIKE_hkdf_sha256());

EVP_HPKE_KEM EVP_hpke_SIKE_hkdf_sha256(); →

Assign the KEM var info about the params

```
static const EVP_HPKE_KEM kKEM = {
  /*id=*/ EVP_HPKE_DHKEM_SIKE_HKDF_SHA256,
  /*public_key_len=*/ SIKE_PUBLIC_KEY_LEN,
  /*private_key_len=*/ SIKE_PRIVATE_KEY_LEN,
  /*seed_len=*/ SIKE_PRIVATE_KEY_LEN,
  SIKE_init_key,
  SIKE_generate_key,
  SIKE_encap_with_seed,
  SIKE_decap};
return &kKEM;
```

2.1 key->kem=kem;

2.2 kem->generate_key(key); → Pointer to the new function SIKE_generate_key

2.2.1 SIKE_keypair(key->public_key, key->private_key);

~~2.2.1.1 private_key ← random value;~~

2.2.1.1 SIKE_keygen(public_key, private_key); →

SIKE key generation

sk_A ← random value
ϕ_A : $E_0 \rightarrow E_A$
Ker(ϕ_A) = $\langle P_A + [sk_A] \cdot Q_A \rangle$
$pk_A = (E_A, \phi_A(P_B), \phi_A(Q_B))$

3. EVP_HPKE_KEY_public_key(key.get(), public_key_r, &public_key_r_len, sizeof(public_key_r));

Copy the pk into public_key_r & update the public_key_r_len value.

SIKE Set Up Sender(SIKE represents the PQ only HPKE implementation.)

1. HPKE_CTX sender_ctx;
uint8_t enc[SIKE_CIPHERTEXT_LEN];
size_t enc_len;

HPKE_CTX →

```
Info about HPKE
 -> the Key Derivation Function (KDF),
-> the Authenticated Encryption with Additional Data (AEAD),
-> the nonce, ...
const EVP_HPKE_AEAD *aead;
const EVP_HPKE_KDF *kdf;
EVP_AEAD_CTX aead_ctx;
uint8_t base_nonce[EVP_AEAD_MAX_NONCE_LENGTH];
uint8_t exporter_secret[EVP_MAX_MD_SIZE];
uint64_t seq;
int is_sender;
```

2. EVP_HPKE_CTX_setup_sender(sender_ctx.get(), enc, &enc_len, sizeof(enc), EVP_hpke_SIKE_hkdf_sha256(), kdf(), aead(), public_key_r, public_key_r_len, info.data(), info.size());

EVP_HPKE_KEM EVP_hpke_SIKE_hkdf_sha256() →

```
Assign the same KEM var info about the params
static const EVP_HPKE_KEM kKEM = {
  /*id=*/ EVP_HPKE_PQKEM_SIKE_HKDF_SHA256,
  /*public_key_len=*/ SIKE_PUBLIC_KEY_LEN,
  /*private_key_len=*/ SIKE_PRIVATE_KEY_LEN,
  /*seed_len=*/ SIKE_PRIVATE_KEY_LEN,
  SIKE_init_key,
  SIKE_generate_key,
  SIKE_encap_with_seed,
  SIKE_decap};
return &kKEM;
```

~~2.1. int8_t seed[MAX_SEED_LEN]; → seed represents the pk of Bob(Sender) -> sk_S~~
~~RAND_bytes(seed, kem->seed_len); → MAX_SEED_LEN = PRIVATE_KEY_LEN_BYTES~~

2.1. EVP_HPKE_CTX_setup_sender_with_seed_for_testing(ctx, enc, enc_len, max_enc, kem, kdf, aead, peer_public_key, peer_public_key_len, info, info_len, seed, kem->seed_len);

2.2. uint8_t ss[SHA256_DIGEST_LENGTH];
size_t ss_len;

2.3. key->kem->encap(kem, ss, &ss_len, out_enc, out_enc_len, max_enc, peer_public_key, peer_public_key_len, seed, seed_len);
→ Pointer to the new function SIKE_encap_with_seed
→ (We use the name _with_seed but seed is not used due to SIKE KEM struct)

→

```
Create ct of Bob(Sender) & find the shared secret
SIKE_encap(out_enc, SIKE_ss, peer_public_key); → ct = (c0, c1)
             ↓              ↓        c0 = pkB = (EB, φB(PA), φB(QA))
             ct             pkA      c1 = j(EAB) ⊕ m
                                     SIKE_ss = H(m||c)
dhkem_extract_and_expand(kem->id, EVP_sha256(),
  ss, SHA256_DIGEST_LENGTH, SIKE_ss, sizeof(SIKE_ss), kem_context, sizeof(kem_context));
  ↓          ↓              ↓
out_key   out_key_len       ss
  *out_enc_len = SIKE_CIPHERTEXT_LEN;
  *out_ss_len = SHA256_DIGEST_LENGTH;
```

2.5. hpke_key_schedule(ctx, ss, sizeof(ss), info, info_len);

SIKE Set Up (SIKE represents the PQ only HPKE implementation.)

1. `EVP_HPKE_CTX_setup_recipient(recipient_ctx.get(), key.get(), kdf(), aead(), enc, enc_len,`
   `info.data(), info.size());` ↓ sk_A ↓ ct

   1.2. `key->kem->decap(key, ss, &ss_len, enc, enc_len);` → Pointer to the new function SIKE_decap

   → Find the shared secret

```
SIKE_decap(SIKE_ss, enc, key->private_key);
          x25519_scalar_mult(out_shared_key, private_key, peer_public_value);
                    Just the multiplication
dhkem_extract_and_expand(key->kem->id, EVP_sha256(),
                         ss, SHA256_DIGEST_LENGTH, SIKE_ss,
                         ↓          ↓              ↓
                      out_key   out_key_len   shared secret
                         sizeof(SIKE_ss), kem_context, sizeof(kem_context));
```

   1.2. `hpke_key_schedule(ctx, ss, sizeof(ss), info, info_len);`

Hybrid x25519 SIKE Key Generation (Recipient)(x25519 SIKE represents the x25519 SIKE HPKE implementation.)

1. HPKE_KEY key;

HPKE_KEM →

Info about KEM

```
-> public parameters length,
-> keygen, encap, decp functions
id
public_key_len
private_key_len
seed_len
int (*init_key)(EVP_HPKE_KEY *key,
                const uint8_t *priv_key, size_t priv_key_len);
int (*generate_key)(EVP_HPKE_KEY *key);
int (*encap_with_seed)(const EVP_HPKE_KEM *kem,
                uint8_t *out_shared_secret, size_t *out_shared_secret_len,
                uint8_t *out_enc, size_t *out_enc_len, size_t max_enc,
                const uint8_t *peer_public_key, size_t peer_public_key_len,
                const uint8_t *seed, size_t seed_len);
                const uint8_t *seed, size_t seed_len);
int (*decap)(const EVP_HPKE_KEY *key, uint8_t *out_ss, size_t *out_ss_len,
                const uint8_t *enc, size_t enc_len);
```

↗

HPKE_KEY → public key[x25519_PUBLIC_KEY_LEN + SIKE_PUBLIC_KEY_LEN]

↘ secret key[x25519_SECRET_KEY_LEN + SIKE_SECRET_KEY_LEN]

2. EVP_HPKE_KEY_generate(key.get(), EVP_HPKE_KEM EVP_hpke_x25519_SIKE_hkdf_sha256());

EVP_HPKE_KEM EVP_hpke_x25519_SIKE_hkdf_sha256(); →

Assign the KEM var info about the params

```
static const EVP_HPKE_KEM kKEM = {
  /*id=*/ EVP_HPKE_DHKEM_x25519_SIKE_HKDF_SHA256,
  /*private_key_len=*/ x25519_PRIVATE_KEY_LEN + SIKE_PRIVATE_KEY_LEN,
  /*public_key_len=*/ x25519_PUBLIC_KEY_LEN + SIKE_PUBLIC_KEY_LEN,
  /*seed_len=*/ x25519_PRIVATE_KEY_LEN,
  Hybrid_init_key,
  Hybrid_generate_key,
  Hybrid_encap_with_seed,
  Hybrid_decap};
return &kKEM;
```

2.1 key->kem=kem;

2.2 kem->generate_key(key); → Pointer to the new function Hybrid_generate_key

2.2.1 private_key → randomly generate x25519 secret key

2.2.2 X25519_public_from_privatekey(key->public_key, key->private_key);

2.2.2.1 x25519_ge_scalarmult_base(public_key, private_key);

2.2.3 SIKE_keygen(public_key + x25519_PUBLIC_KEY_LEN, private_key + x25519_PRIVATE_KEY_LEN);

→

SIKE key generation

$sk_A \leftarrow$ random value

$\phi_A : E_0 \rightarrow E_A$

$Ker(\phi_A) = \langle P_A + [sk_A] \cdot Q_A \rangle$

$pk_A = (E_A, \phi_A(P_B), \phi_A(Q_B))$

3. EVP_HPKE_KEY_public_key(key.get(), public_key_r, &public_key_r_len, sizeof(public_key_r));

Copy the pk into public_key_r & update the public_key_r_len value.

Hybrid x25519 SIKE Set Up Sender(x25519 SIKE represents the x25519 SIKE HPKE implementation.)

1. HPKE_CTX sender_ctx;
   uint8_t enc[x25519_PUBLIC_VALUE_LEN + SIKE_CIPHERTEXT_VALUE_LEN];
   size_t enc_len;
2. EVP_HPKE_CTX_setup_sender(sender_ctx.get(), enc, &enc_len, sizeof(enc), EVP_hpke_x25519_SIKE_hkdf_sha256(), kdf(), aead(), public_key_r, public_key_r_len, info.data(), info.size());

EVP_HPKE_KEM EVP_hpke_x25519_SIKE_hkdf_sha256() →

Assign the same KEM var info about the params

```
static const EVP_HPKE_KEM kKEM = {
  /*id=*/ EVP_HPKE_DHKEM_x25519_SIKE_HKDF_SHA256,
  /*private_key_len=*/ x25519_PRIVATE_KEY_LEN + SIKE_PRIVATE_KEY_LEN,
  /*public_key_len=*/ x25519_PUBLIC_KEY_LEN + SIKE_PUBLIC_KEY_LEN,
  /*PQ_public_key_len=*/ SIKE_PUBLIC_VALUE_LEN,
  Hybrid_init_key,
  Hybrid_generate_key,
  Hybrid_encap_with_seed,
  Hybrid_decap};
return &kKEM;
```

2.1. int8_t seed[MAX_SEED_LEN]; → seed represents the x25519 pk of Bob(Sender) -> sk_S
RAND_bytes(seed, kem->seed_len); → MAX_SEED_LEN = x25519_PRIVATE_KEY_LEN

2.1. EVP_HPKE_CTX_setup_sender_with_seed_for_testing(ctx, out_enc, out_enc_len, max_enc, kem, kdf, aead, peer_public_key, peer_public_key_len, info, info_len, seed, kem->seed_len);

2.2. uint8_t ss[SHA256_DIGEST_LEN];
size_t ss_len;

2.3. kem->encap(kem, ss, &ss_len, out_enc, out_enc_len, max_enc, peer_public_key, peer_public_key_len, seed, seed_len);
→ Pointer to the new function Hybrid_generate_key

→ Create x25519_pk || ct of Bob(Sender) & find the shared secret = x25519_ss || SIKE_ss

```
X25519_public_from_private(out_enc, seed);
                            ↓      ↓
                       x25519_pk_B   x25519_sk_B
X25519(hybrid_ss, seed, peer_public_key); → hybrid_ss = [seed] · peer_public_key = [sk_B] · pk_A = [sk_B] · [sk_A] · P
SIKE_encap(out_enc + x25519_PUBLIC_KEY_LEN, hybrid_ss + x25519_SHARED_SECRET_LEN,
                            ↓                    ct = (c0, c1) = (E_B, φ_B(P_A), φ_B(Q_A)), j(E_AB) ⊕ m)
                           ct                    SIKE_ss = H(m||c)
            peer_public_key + x25519_PUBLIC_KEY_LEN);
dhkem_extract_and_expand(kem->id, EVP_sha256(), ss,
  SHA256_DIGEST_LEN, hybrid_ss, sizeof(hybrid_ss), kem_context, sizeof(kem_context));
          ↓            ↓
     out_key_len   hybrid_ss = (x25519_ss || SIKE_ss)
  *out_enc_len = SHA256_DIGEST_LEN;
  *out_shared_secret_len = X25519_SHARED_KEY_LEN + SIKE_SHARED_SECRET_BYTES
```

3. hpke_key_schedule(ctx, ss, sizeof(ss), info, info_len);

Hybrid x25519 SIKE Set Up Recipient(x25519 SIKE represents the x25519 SIKE HPKE implementation.)

1. EVP_HPKE_CTX_setup_recipient(recipient_ctx.get(), key.get(), kdf(), aead(), enc, enc_len,
   info.data(), info.size()); ↓ x25519_sk$_A$ || SIKE_sk$_A$ ↓ x25519_pk$B$ || SIKE_ct

   1.2. key->kem->decap(key, ss, &ss_len, enc, enc_len); → Pointer to the new function Hybrid_decaps

→ Find the shared secret

```
X25519(hybrid_ss, key->private_key, enc); → hybrid_ss = [sk_A] · enc =
                                          = [sk_A] · pk_B =
                                          = [sk_A] · [sk_B] · P
         x25519_scalar_mult(hybrid_ss, private_key, peer_public_value);
SIKE_decap(hybrid_ss + X25519_SHARED_SECRET_LEN, enc + x25519_PUBLIC_KEY_LEN,
         key->private_key + x25519_KEY_VALUE_LEN);
dhkem_extract_and_expand(key->kem->id, EVP_sha256(), ss,
     SHA256_DIGEST_LEN, hybrid_ss,
        ↓               ↓
     out_key_len  x25519_ss || SIKE_ss
                          sizeof(ss), kem_context, sizeof(kem_context));
```

2. hpke_key_schedule(ctx, ss, sizeof(ss), info, info_len);

HPKE Key Generation (Recipient)(HPKE represents the integrating any PQ HPKE implementation.)

1. HPKE_KEY key;

HPKE_KEM →

```
Info about KEM
id
ECC_private_key_len
ECC_public_key_len
ECC_shared_secret_len
ECC_seed_len
PQ_private_key_len
PQ_public_key_len
PQ_shared_secret_len
PQ_ciphertext_len
int (*init_key)(EVP_HPKE_KEY *key, const uint8_t *priv_key, size_t priv_key_len);
int (*generate_key)(EVP_HPKE_KEY *key);
int (*encap_with_seed)(const EVP_HPKE_KEM *kem,uint8_t *out_shared_secret,
            size_t *out_shared_secret_len, uint8_t *out_enc, size_t *out_enc_len,
            size_t max_enc, const uint8_t *peer_public_key, size_t peer_public_key_len,
            const uint8_t *seed, size_t seed_len);
```

HPKE_KEY → public key[key->kem->ECC_public_key_len + key->kem->PQ_public_key_len]

HPKE_KEY → secret key[key->kem->ECC_private_key_len + key->kem->PQ_private_key_len]

2. EVP_HPKE_KEY_generate(key.get(), algorithm_kdf(alg));

algorithm_kdf(alg); →

```
Assign the KEM var info about the params
static const EVP_HPKE_KEM kKEM = {
  /*id=*/ EVP_HPKE_(alg)_HKDF_SHA256,
  /*ECC_private_key_len=*/ ECC_(alg)_PRIVATE_KEY_LEN,
  /*ECC_public_key_len=*/ ECC_(alg)_PUBLIC_VALUE_LEN,
  /*ECC_shared_secret_len=*/ ECC_(alg)_SHARED_SECRET_LEN,
  /*ECC_seed_len=*/ ECC_(alg)_PRIVATE_KEY_LEN,
  /*PQ_private_key_len=*/ PQ_(alg)_PRIVATE_KEY_LEN,
  /*PQ_public_key_len=*/ PQ_(alg)_PUBLIC_VALUE_LEN,
  /*PQ_shared_secret_len=*/ PQ_(alg)_SHARED_SECRET_LEN,
  /*PQ_ciphertext_len=*/ PQ_(alg)_CIPHERTEXT_LEN,
  HPKE_init_key,
  HPKE_generate_key,
  HPKE_encap_with_seed,
  HPKE_decap};
return &kKEM;
```

2.1 key->kem=kem;

2.2 kem->generate_key(key); → Pointer to HPKE_generate_key, one function for any ECC and/or PQ HPKE

```
IF (key->kem->id == EVP_HPKE_DHKEM_x25519_HKDF_SHA256 ||
    key->kem->id == EVP_HPKE_HKEM_x25519_SIKE_HKDF_SHA256 ||
    key->kem->id == EVP_HPKE_HKEM_x25519_Kyber_HKDF_SHA256)
        X25519_keypair(key->public_key, key->private_key);
IF (key->kem->id != EVP_HPKE_DHKEM_X25519_HKDF_SHA256)
    pq_keygen[(key->kem->id) % SUPPORTED_PQ_ALGORITHMS](
    key->public_key + key->kem->ECC_public_key_len,
    key->private_key + key->kem->ECC_private_key_len);
```

3. EVP_HPKE_KEY_public_key(key.get(), public_key_r, &public_key_r_len, publickeybytes(alg));

Copy the pk into public_key_r & update the public_key_r_len value.

HPKE Set Up Sender(HPKE represents the integrating any PQ HPKE implementation.)

1. HPKE_CTX sender_ctx;
   uint8_t enc[ciphertextbytes(alg)];
   size_t enc_len;
2. EVP_HPKE_CTX_setup_sender(sender_ctx.get(), enc, &enc_len, ciphertextbytes(alg), kdf(alg),
   kdf(), aead(), public_key_r, public_key_r_len, info.data(), info.size());

algorithm_kdf(alg); →

Assign the KEM var info about the params

```
static const EVP_HPKE_KEM kKEM = {
  /*id=*/ EVP_HPKE_(alg)_HKDF_SHA256,
  /*ECC_private_key_len=*/ ECC_(alg)_PRIVATE_KEY_LEN,
  /*ECC_public_key_len=*/ ECC_(alg)_PUBLIC_VALUE_LEN,
  /*ECC_shared_secret_len=*/ ECC_(alg)_SHARED_SECRET_LEN,
  /*ECC_seed_len=*/ ECC_(alg)_PRIVATE_KEY_LEN,
  /*PQ_private_key_len=*/ PQ_(alg)_PRIVATE_KEY_LEN,
  /*PQ_public_key_len=*/ PQ_(alg)_PUBLIC_VALUE_LEN,
  /*PQ_shared_secret_len=*/ PQ_(alg)_SHARED_SECRET_LEN,
  /*PQ_ciphertext_len=*/ PQ_(alg)_CIPHERTEXT_LEN,
  HPKE_init_key,
  HPKE_generate_key,
  HPKE_encap_with_seed,
  HPKE_decap};
return &kKEM;
```

2.1. int8_t seed[kem->ECC_seed_len];
RAND_bytes(seed, kem->ECC_seed_len);

2.1. EVP_HPKE_CTX_setup_sender_with_seed_for_testing(ctx, out_enc, out_enc_len, max_enc, kem, kdf,
aead, peer_public_key, peer_public_key_len, info, info_len, seed, kem->seed_len,
*psk, psk_len, *psk_id, psk_id_len);

2.2. uint8_t ss[SHA256_DIGEST_LENGTH];
size_t ss_len;

2.3. kem->encap_with_seed(kem, ss, &ss_len, out_enc, out_enc_len,
max_enc, peer_public_key, peer_public_key_len, seed, seed_len);
→ Pointer to HPKE_encap_with_seed, one function for any ECC and/or PQ HPKE

→

Create ECC_pk || PQ_ct of Bob(Sender) & ECC_ss || PQ_ss

```
uint8_t hybrid_ss[kem->ECC_shared_secret_len + kem->PQ_shared_secret_len];
IF (key->kem->id == EVP_HPKE_DHKEM_x25519_HKDF_SHA256 ||
    key->kem->id == EVP_HPKE_HKEM_x25519_SIKE_HKDF_SHA256 ||
    key->kem->id == EVP_HPKE_HKEM_x25519_Kyber_HKDF_SHA256)
        X25519_public_from_private(out_enc, seed);
        X25519(hybrid_ss, seed, peer_public_key);
IF (key->kem->id != EVP_HPKE_DHKEM_X25519_HKDF_SHA256)
              pq_enc[(key->kem->id) % SUPPORTED_PQ_ALGORITHMS](out_enc + kem->ECC_public_key_len,
              hybrid_ss + kem->ECC_public_key_len, peer_public_key + kem->ECC_public_key_len);
dhkem_extract_and_expand(kem->id, EVP_sha256(), ss, SHA256_DIGEST_LENGTH, hybrid_ss,
        kem->ECC_shared_secret_len + kem->PQ_shared_secret_len, kem_context,
        2 * kem->ECC_public_key_len + kem->PQ_public_key_len + kem->PQ_ciphertext_len);
*out_enc_len = kem->ECC_public_key_len + kem->PQ_ciphertext_len;
*out_shared_secret_len = SHA256_DIGEST_LENGTH
```

3. hpke_key_schedule(mode, ctx, ss, sizeof(ss), info, info_len, psk, psk_len, psk_id, psk_id_len);

HPKE Set Up Recipient(HPKE represents the integrating any PQ HPKE implementation.)

```
1.  EVP_HPKE_CTX_setup_recipient(recipient_ctx.get(), key.get(), kdf(), aead(), enc, enc_len,
                info.data(), info.size(), NULL, 0, NULL, 0));
    1.1.  uint8_t ss[SHA256_DIGEST_LENGTH];
          size_t ss_len;
    1.2.  key->kem->decap(key, ss, &ss_len, enc, enc_len, psk, psk_len, psk_id, psk_id_len);
```

→ Find the shared secret

```
  uint8_t hybrid_ss[kem->ECC_shared_secret_len + kem->PQ_shared_secret_len];
  IF (key->kem->id == EVP_HPKE_DHKEM_x25519_HKDF_SHA256 ||
      key->kem->id == EVP_HPKE_HKEM_x25519_SIKE_HKDF_SHA256 ||
      key->kem->id == EVP_HPKE_HKEM_x25519_Kyber_HKDF_SHA256)
          X25519(hybrid_ss, key->private_key, enc);
  IF (key->kem->id != EVP_HPKE_DHKEM_X25519_HKDF_SHA256)
          pq_dec[(key->kem->id) % SUPPORTED_PQ_ALGORITHMS](hybrid_ss + key->kem->ECC_shared_secret_len,
                  enc + key->kem->ECC_public_key_len, key->private_key + key->kem->ECC_private_key_len);
  dhkem_extract_and_expand(key->kem->id, EVP_sha256(), ss, SHA256_DIGEST_LENGTH, hybrid_ss,
                    key->kem->ECC_shared_secret_len + key->kem->PQ_shared_secret_len, kem_context,
                    2 * kem->ECC_public_key_len + kem->PQ_public_key_len + kem->PQ_ciphertext_len);
  *out_shared_secret_len = SHA256_DIGEST_LENGTH;
```

```
2.  hpke_key_schedule(mode, ctx, ss, sizeof(ss), info, info_len, psk, psk_len, psk_id, psk_id_len);
```

Modifications to the AWS-LC

Modifications based on aws-s2n SIKE code

Tree structure of the library and files (main branch) modified marked in red:

```
- aws-lc
      - CMakeList.txt
      - include
            - openssl
                  - hpke.h
                  - curve25519.h
                  - sike_internal.h
                  - ...
      - crypto
            - hpke
            - curve25519
            - sike
            - ...
```

aws-lc/crypto/sike

Add the files:

```
sikep434r3.c
sikep434r3.h
sikep434r3_api.h (header for sidh)
sikep434r3_fips202.c
sikep434r3_fips202.h
sikep434r3_fp.c
sikep434r3_fp.h
sikep434r3_fpx.c
sikep434r3_fpx.h
sikep434r3_ec_isogeny.c
sikep434r3_ec_isogeny.h
sikep434r3_sidh.c
sikep434r3_sidh.h
sikep434r3_kem.c
```

All the files are taken from the aws-s2n library and the previous intern project of Jyoti Lama.

Modify the `*.c` files by including the header file:

```
#include <openssl/sike_internal.h>
```

aws-lc/crypto/CMakeList.txt

Add the `*.c` files in the `CMakeFiles.txt` in the `add_library` section :

```
sike/sikep434r3.c
sike/sikep434r3_fips202.c
```

```
sike/sikep434r3_fp.c
sike/sikep434r3_fpx.c
sike/sikep434r3_ec_isogeny.c
sike/sikep434r3_sidh.c
sike/sikep434r3_kem.c
```

aws-lc/inslude/openssl

Add the file:

`sike_internal.h (header for sike_kem.c)`
Modify the `hpke.h` file by including the header file:

```
#include <openssl/sike_internal.h>
```

Modifications based on AWS-LC SIKE and Kyber code (imported from the SIKE submission package [15])

Tree structure of the library and files (integrate-pq branch) modified marked in red:

```
- aws-lc
      - CMakeList.txt
      - include
            - openssl
                  - hpke.h
                  - curve25519.h
                  - P434_api.h
                  - kyber_kem.h
                  - ...
      - crypto
            - hpke
            - curve25519
            - ...
      - pq-crypto
            - pq_kem.h
            - pq_kem.c
            - pq_kem_test.cc
            - sike_r3
            - kyber_r3
                  - ...
            - ...
```

`NOTE`: We move the content of the `pq-crypto` folder under the `crypto` folder since `pq-crypto` represents a subset of the cryptographical primitives and `crypto` is the folder containing the crypto primitives.

aws-lc/crypto/pq-crypto

Tree structure of the files:

```
- CMAkeList.txt
```

- pq_kem.c
- pq_kem.h
- pq_kem_params_size.h
- pq_kem_test.cc
- **sike_r3**
    - CMAkeList.txt
    - config.h
    - ec_isogeny.c
    - fpx.c
    - internal.c
    - sidh.c
    - sike.c
    - **P434**
        - P434.c
        - P434_internal.h
        - **generic**
            - fp_generic.c
    - **random**
        - random.c
        - random.h
    - **sha3**
        - fips202.c
        - fips202.h
- **kyber_r3**
    - CMAkeList.txt
    - aes256ctr.c
    - aes256ctr.h
    - cbd.c
    - cbd.h
    - fips202.c
    - fips202.h
    - indcpa.c
    - indcpa.h
    - kem.c
    - ntt.c
    - ntt.h
    - params.h
    - poly.c
    - poly.h
    - polyvec.c
    - polyvec.h
    - randombytes.c
    - randombytes.h
    - reduce.c
    - reduce.h
    - sha256.c
    - sha2.h
    - sha512.c
    - symmetric.h
    - symmetric-aes.c
    - symmetric-shake.c

```
- verify.c
- verify.h
- avx
      - aes256ctr.c
      - basemul.S
      - aes256ctr.c
      - aes256ctr.h
      - align.h
      - api.h
      - basemul.S
      - cbd.c
      - cbd.h
      - consts.c
      - consts.h
      - fips202.c
      - fips202.h
      - fips202x4.c
      - fips202x4.h
      - fq.S
      - fq.inc
      - indcpa.c
      - indcpa.h
      - ntt.S
      - ntt.h
      - invntt.S
      - kem.c
      - poly.c
      - poly.h
      - params.h
      - keccak4x
            - KeccakP-1600-times4-SIMD256.c
            - KeccakP-1600-times4-SnP.h
            - KeccakP-1600-unrolling.macros
            - KeccakP-align.h
            - KeccakP-brg_endian.h
            - KeccakP-SIMD256-config.h
```

Modify the `*.c` files that include `#include "P434_api.h"` and `#include "kyber_kem.h"` by including the header file:

`#include <openssl/P434_api.h>` for SIKE and `#include <openssl/kyber_kem.h>` for SIKE and Kyber, respectively.

aws-lc/CMakeList.txt

Remove the line:

```
add_subdirectory(pq-crypto)
```

aws-lc/crypto/CMakeList.txt

Add the lines:

```
add_subdirectory(pq-crypto)
target_link_libraries(crypto pq_crypto)
```

aws-lc/crypto/pq-crypto/CMakeList.txt

Change the line `include_directories(../include)` to the line:

```
include_directories(../../include)
```

Add the lines:

```
install(TARGETS pq_crypto
                 EXPORT crypto-targets
                 ARCHIVE DESTINATION $CMAKE_INSTALL_LIBDIR
                 LIBRARY DESTINATION $CMAKE_INSTALL_LIBDIR)
```

7.0.1 aws-lc/include/openssl

Add the file:

```
P434_api.h (header for sike.c and sidh.c)

kyber_kem.h
```

Modify the `hpke.h` file by including the header file:

`#include <openssl/P434_api.h>` and `#include <openssl/kyber_kem.h>`

aws-lc/crypto/pq-crypto/{sike_r3, kyber_r3}/CMakeList.txt

For SIKE specify the optimized assembly files for AMD64 and ARM64:

```
if($CMAKE_SYSTEM_PROCESSOR MATCHES "x86_64|amd64|AMD64")
# If ARCH is originally detected as 64-bit, perform an additional check
# to determine whether to build as 32-bit or 64-bit.  This happens in some
# cases such as when building in Docker, where the host-level architecture is 64-bit
# but the Docker image should result in building for a 32-bit architecture.
if(CMAKE_SIZEOF_VOID_P EQUAL 8)
    set($ARCHITECTURE "_AMD64_")
    set(USE_OPT_LEVEL "_FAST_")
    set(SIKE_R3_SRCS $SIKE_R3_SRCS P434/AMD64/fp_x64.c P434/AMD64/fp_x64_asm.S)
else()
    set($ARCHITECTURE "_X86_")
    set(USE_OPT_LEVEL "_GENERIC_")
    set(SIKE_R3_SRCS $SIKE_R3_SRCS P434/generic/fp_generic.c)
endif()
elseif($CMAKE_SYSTEM_PROCESSOR MATCHES "x86|i386|i686")
```

```
    set($ARCHITECTURE "_X86_")
    set(USE_OPT_LEVEL "_GENERIC_")
    set(SIKE_R3_SRCS $SIKE_R3_SRCS P434/generic/fp_generic.c)
elseif($CMAKE_SYSTEM_PROCESSOR MATCHES "arm64.*|ARM64|aarch64")
    set($ARCHITECTURE "_ARM64_")
    set(USE_OPT_LEVEL "_FAST_")
    set(SIKE_R3_SRCS $SIKE_R3_SRCS P434/ARM64/fp_arm64.c P434/AMD64/fp_arm64_asm.S)
elseif($CMAKE_SYSTEM_PROCESSOR MATCHES "arm*")
    set($ARCHITECTURE "_ARM_")
    set(USE_OPT_LEVEL "_GENERIC_")
    set(SIKE_R3_SRCS $SIKE_R3_SRCS P434/generic/fp_generic.c)
   endif()
```

For Kyber specify the optimized assembly files for avx2 (in two separate CMakeList.txt since all avx2 files are under different folder) and set flags:

```
if($CMAKE_SYSTEM_PROCESSOR MATCHES "x86_64|amd64|AMD64")

 if(CMAKE_SIZEOF_VOID_P EQUAL 8)
    target_compile_options(kyber_r3 PRIVATE $CFLAGS -D_AMD64_)
else()
    target_compile_options(kyber_r3 PRIVATE $CFLAGS -D_X86_)
endif()
elseif($CMAKE_SYSTEM_PROCESSOR MATCHES "x86|i386|i686")
    target_compile_options(kyber_r3 PRIVATE -D_X86_)
elseif($CMAKE_SYSTEM_PROCESSOR MATCHES "arm64.*|ARM64|aarch64")
    target_compile_options(kyber_r3 PRIVATE -D_ARM64_)
elseif($CMAKE_SYSTEM_PROCESSOR MATCHES "arm*")
    target_compile_options(kyber_r3 PRIVATE -D_ARM_)
endif()

if(UNIX)
    target_compile_options(kyber_r3 PRIVATE $CFLAGS -D__NIX__)
elseif(WIN32)
    target_compile_options(kyber_r3 PRIVATE $CFLAGS -D__WINDOWS__)
endif()
```