Faster generic IND-CCA2 secure KEM using “encrypt-then-MAC”

Anonymous Submission

Abstract. The modular Fujisaki-Okamoto (FO) transformation takes public-key encryption with weaker security and constructs a key encapsulation mechanism (KEM) with indistinguishability under adaptive chosen ciphertext attacks. While the modular FO transform enjoys tight security bound and quantum resistance, it also suffers from computational inefficiency and vulnerabilities to side-channel attacks due to using de-randomization and re-encryption for providing ciphertext integrity. In this work, we propose an alternative KEM construction that achieves ciphertext integrity using a message authentication code (MAC) and instantiate a concrete instance using Kyber. Our experimental results showed that where the encryption routine incurs heavy computational cost, replacing re-encryption with MAC provides substantial performance improvements at comparable security level.

Keywords: Key encapsulation mechanism, post-quantum cryptography, lattice cryptography, Fujisaki-Okamoto transformation

1 Introduction

The Fujisaki-Okamoto transformation [FO99] is a generic construction that takes cryptographic primitives of lesser security and constructs a public-key encryption scheme with indistinguishability under adaptive chosen ciphertext attacks. Later works [HHK17] extended the original transformation to the construction of key encapsulation mechanism, which has been adopted by many post-quantum schemes such as Kyber [BDK$^+$18], FrodoKEM [BCD$^+$16], and SABER [DKSRV18].

The current state of the FO transformation enjoys proven tight security bound and quantum resistance [HHK17], but also leaves many deficiencies to be improved on. One such shortcoming is the use of re-encryption for providing ciphertext integrity [BP18], which requires the decapsulation routine to run the encryption routine as a subroutine. In many post-quantum schemes, such as Kyber, the encryption routine is substantially more expensive than the decryption routine, so running the encryption routine in the decapsulation routine inflates the computational cost of the decapsulator. In addition, running the encryption as a subroutine introduces risks of side-channel vulnerabilities that may expose the plaintext or the secret key [RRCB19][UXT$^+$22].

The problem of ciphertext integrity was solved in symmetric cryptography: given a semantically secure symmetric cipher and an existentially unforgeable message authentication code, combining them using “encrypt-then-mac” provides authenticated encryption [BN00]. We took inspiration from this strategy and applied a similar technique to transform an OW-PCA secure public-key encryption scheme into an IND-CCA2 secure key encapsulation mechanism. Using a message authentication code for ciphertext integrity replaces the re-encryption step in decryption with the computation of an authenticator, which offers significant performance improvements while maintaining comparable level of security.

The main challenge in applying “encrypt-then-mac” to public-key cryptography is the lack of a pre-shared symmetric key. We proposed to derive the symmetric key by hashing the plaintext message. In section 3, we prove that under the random oracle model, if the input public-key encryption scheme is one-way secure against plaintext-checking attack and the input message authentication code is one-time existentially unforgeable, then the transformed key encapsulation mechanism is IND-CCA2 secure.

In section 5, we instantiate concrete instances of our constructions by combining Kyber with GMAC and Poly1305. Our experimental results showed that replacing re-encryption with computing authenticator leads to significant performance improvements in the decapsulation routine while incurring only minimal runtime overhead in the encapsulation routine and a small increase in ciphertext size.

2 Preliminaries and previous results

2.1 Public-key encryption scheme

A public key encryption scheme PKE is a collection of three routines $(\texttt{KeyGen}, \texttt{Enc}, \texttt{Dec})$ defined over some message space $\mathcal{M}$ and some ciphertext space $\mathcal{C}$. Where the encryption routine is probabilistic, the source of randomness is denoted by the coin space $\mathcal{R}$.

The encryption routine $\texttt{Enc}(\texttt{pk}, m)$ takes a public key, a plaintext message, and outputs a ciphertext $c \in \mathcal{C}$. Where the encryption routine is probabilistic, specifying a pseudorandom seed $r \in \mathcal{R}$ will make the encryption routine behave deterministically. The decryption routine $\texttt{Dec}(\texttt{sk}, c)$ takes a secret key, a ciphertext, and outputs the decryption $\hat{m}$ if the ciphertext is valid. Some PKE will explicitly reject invalid ciphertext, in which case the decryption routine will output the rejection symbol $\perp$

We discuss the security of a PKE using the sequence of games described in [Sho04]. Specifically, we first define the OW-ATK as they pertain to a public key encryption scheme. In later section we will define the IND-CCA game as it pertains to a key encapsulation mechanism.

Algorithm 1 The OW-ATK game

```
1: $(\texttt{pk}, \texttt{sk}) \xleftarrow{\$} \texttt{KeyGen}(1^\lambda)$
2: $m^* \xleftarrow{\$} \mathcal{M}$
3: $c^* \xleftarrow{\$} \texttt{Enc}(\texttt{pk}, m^*)$
4: $\hat{m} \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_{\texttt{ATK}}}(1^\lambda, \texttt{pk}, c^*)$
5: return $[\![m^* = \hat{m}]\!]$
```

Figure 1: The OW-ATK game

Algorithm 2 $\texttt{PCO}(m \in \mathcal{M}, c \in \mathcal{C})$

```
1: return $[\![\texttt{Dec}(\texttt{sk}, c) = m]\!]$
```

Figure 2: Plaintext-checking oracle

In the OW-ATK game (see figure 1), an adversary's goal is to recover the decryption of a randomly generated ciphertext. A challenger randomly samples a keypair and a challenge plaintext m^*, encrypts the challenge plaintext $c^* \xleftarrow{\$} \texttt{Enc}(\texttt{pk}, m^*)$, then gives pk and c^* to the adversary A. The adversary A, with access to some oracle $\mathcal{O}_{\texttt{ATK}}$, outputs a guess decryption $\hat{m}$. A wins the game if its guess $\hat{m}$ is equal to the challenge plaintext m^*. The *advantage* $\texttt{Adv}_{\texttt{OW-ATK}}$ of an adversary in this game is the probability that it wins the game:

$$\texttt{Adv}_{\texttt{OW-ATK}}(A) = P\left[A(\texttt{pk}, c^*) = m^* | (\texttt{pk}, \texttt{sk}) \xleftarrow{\$} \texttt{KeyGen}(); m^* \xleftarrow{\$} \mathcal{M}; c^* \xleftarrow{\$} \texttt{Enc}(\texttt{pk}, m^*)\right]$$

The capabilities of the oracle $\mathcal{O}_{\texttt{ATK}}$ depends on the choice of security goal ATK. Particularly relevant to our result is security against plaintext-checking attack (PCA), for which the adversary has access to a plaintext-checking oracle (PCO) (see figure 2). A PCO takes as input a plaintext-ciphertext pair (m, c) and returns `True` if m is the decryption of c or `False` otherwise.

2.2 Key encapsulation mechanism (KEM)

A key encapsulation mechanism is a collection of three routines $(\texttt{KeyGen}, \texttt{Encap}, \texttt{Decap})$ defined over some ciphertext space $\mathcal{C}$ and some key space $\mathcal{K}$. The key generation routine takes the security parameter 1^λ and outputs a keypair $(\texttt{pk}, \texttt{sk}) \overset{\$}{\leftarrow} \texttt{KeyGen}(1^\lambda)$. $\texttt{Encap}(\texttt{pk})$ is a probabilistic routine that takes a public key `pk` and outputs a pair of values (c, K) where $c \in \mathcal{C}$ is the ciphertext (also called encapsulation) and $K \in \mathcal{K}$ is the shared secret (also called session key). $\texttt{Decap}(\texttt{sk}, c)$ is a deterministic routine that takes the secret key `sk` and the encapsulation c and returns the shared secret K if the ciphertext is valid. Some KEM constructions use explicit rejection, where if c is invalid then `Decap` will return a rejection symbol $\perp$; other KEM constructions use implicit rejection, where if c is invalid then `Decap` will return a fake session key that depends on the ciphertext and some other secret values.

The IND-CCA security of a KEM is defined by an adversarial game in which an adversary's goal is to distinguish pseudorandom shared secret (generated by running the `Encap` routine) and a truly random value.

Algorithm 3 `IND-CCA` game for `KEM`

```
1: $(\texttt{pk}, \texttt{sk}) \overset{\$}{\leftarrow} \texttt{KeyGen}(1^\lambda)$
2: $(c^*, K_0) \overset{\$}{\leftarrow} \texttt{Encap}(\texttt{pk})$
3: $K_1 \overset{\$}{\leftarrow} \mathcal{K}$
4: $b \overset{\$}{\leftarrow} \{0, 1\}$
5: $\hat{b} \overset{\$}{\leftarrow} A^{\mathcal{O}_{\texttt{Decap}}}(1^\lambda, \texttt{pk}, c^*, K_b)$
6: return $[\![\hat{b} = b]\!]$
```

Figure 3: The `KEM-IND-CCA2` game

Algorithm 4 $\mathcal{O}_{\texttt{Decap}}(c)$

```
1: return $\texttt{Decap}(\texttt{sk}, c)$
```

Figure 4: Decapsulation oracle

The decapsulation oracle $\mathcal{O}^{\texttt{Decap}}$ takes a ciphertext c and returns the output of the `Decap` routine using the secret key. The advantage $\epsilon_{\texttt{IND-CCA}}$ of an IND-CCA adversary $\mathcal{A}_{\texttt{IND-CCA}}$ is defined by

$$\texttt{Adv}_{\texttt{IND-CCA}}(A) = \left| P[A^{\mathcal{O}_{\texttt{Decap}}}(a^\lambda, \texttt{pk}, c^*, K_b) = b] - \frac{1}{2} \right|$$

2.3 Message authentication code (MAC)

A message authentication code `MAC` is a collection of routines $(\texttt{Sign}, \texttt{Verify})$ defined over some key space $\mathcal{K}$, some message space $\mathcal{M}$, and some tag space $\mathcal{T}$. The signing routine $\texttt{Sign}(k, m)$ takes the secret key $k \in \mathcal{K}$ and some message, and outputs a tag t. The verification routine $\texttt{Verify}(k, m, t)$ takes the triplet of secret key, message, and tag, and outputs `1` if the message-tag pair is valid under the secret key, or `0` otherwise. Many MAC constructions are deterministic. For these constructions it is simpler to denote the signing routine by $t \leftarrow \texttt{MAC}(k, m)$ and perform verification using a simple comparison.

The security of a MAC is defined in an adversarial game in which an adversary, with access to some signing oracle $\mathcal{O}_{\texttt{Sign}}(m)$, tries to forge a new valid message-tag pair that has never been queried before. The existential unforgeability under chosen message attack (EUF-CMA) game is shown below:

Algorithm 5 The EUF-CMA game

```
1: $k^* \xleftarrow{\$} \mathcal{K}$
2: $(\hat{m}, \hat{t}) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_{\texttt{Sign}}}()$
3: return $[\![\texttt{Verify}(k^*, \hat{m}, \hat{t}) \wedge (\hat{m}, \hat{t}) \notin \mathcal{O}_{\texttt{Sign}}]\!]$
```

Figure 5: The EUF-CMA game

The advantage $\texttt{Adv}_{\texttt{EUF-CMA}}$ of the existential forgery adversary is the probability that it wins the EUF-CMA game.

We are specifically interested in one-time MAC, whose security goal is identical to EUF-CMA described above, except for the constraint that each secret key can be used to sign exactly one distinct message. This translates to an attack model in which the signing oracle will only answer one signing query. Restricting to one-time usage allows for more efficient MAC constructions. One popular way to build one-time MAC is with universal hash functions (UHF), where each instance is parameterized by a finite field $\mathbb{F}$ and a maximal message length $L \geq 0$. The secret key is a pair of field elements $(k_1, k_2) \in \mathbb{F} \times \mathbb{F}$, and each message is a tuple of up to L field elements $m = (m_1, m_2, \ldots, m_l) \in \mathbb{F}^{\leq L}$. To compute the tag:

$$\texttt{MAC}((k_1, k_2), m) = H_{\text{xpoly}}(k_1, m) + k_2$$

Where the H_{xpoly} is a universal hash function:

$$H_{\text{xpoly}}(k_1, (m_1, m_2, \ldots, m_l)) = k_1^l \cdot m_1 + k_1^{l-1} \cdot m_2 + \ldots + k_1 \cdot m_l$$

Lemma 1. *For all adversaries (including unbounded ones) against the MAC described above, the probability of winning the one-time EUF-CMA game is at most:*

$$\texttt{Adv}_{OT\text{-}EUF\text{-}CMA}(A) \leq \frac{L+1}{|\mathbb{F}|}$$

Proof. See [BS20] lemma 7.11 □

2.4 Related works

The Fujisaki-Okamoto transformation [FO99][HHK17] is a family of generic transformations that takes as input a PKE with weaker security, such as OW-CPA, and outputs a PKE or KEM with IND-CCA2 security. The key ingredient in achieving ciphertext non-malleability is with *de-randomization* and *re-encryption*, which first transform a OW-CPA PKE into a *rigid* PKE, then transform the rigid PKE into a KEM. More specifically:

1. *de-randomization* means that a randomized encryption routine $c \xleftarrow{\$} \texttt{Enc}(\texttt{pk}, m)$ is made into a deterministic encryption routine by deriving randomization coin pseudorandomly: $c \leftarrow \texttt{Enc}(\texttt{pk}, m, r = H(m))$ for some hash function H

2. *re-encryption* means that the transformed decryption routine will run the transformed encryption routine to verify the integrity of the ciphertext. Because after *de-randomization*, each plaintext strictly corresponds exacty one ciphertext, tempering with a ciphertext means that even if the ciphertext decrypts back to the same plaintext, the re-encryption will detect that the ciphertext has been tempered with.

3. *rigidity* means that the decryption routine is a perfect inverse of the encryption routine: $c = \texttt{Enc}(\texttt{pk}, m) \Leftrightarrow m = \texttt{Dec}(\texttt{sk}, c)$. Converting a one-way secure rigid PKE (which is essentially a trapdoor function) into a IND-CCA2 KEM is well solved problem. We refer readers to [BS20] for details on such constructions.

let $\texttt{PKE} = (\texttt{KeyGen}_{\texttt{PKE}}, \texttt{Enc}, \texttt{Dec})$ be defined over message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$. Let $G : \mathcal{M} \to \mathcal{R}$ hash plaintexts into coincs, and let $H : \{0,1\}^* \to \{0,1\}^*$ hash byte stream into session keys. Depending on whether the constructed KEM uses implicit or explicit rejection, and the security property of the PKE, [HHK17] described four variations. They are summarized in table 1 and figure 6.

Table 1: Variants of modular FO transforms

name	rejection	PKE security
$U^{\perp}$	explicit	OW-PCVA
$U^{\not\perp}$	implicit	OW-PCA
$U^{\perp}_m$	explicit	OW-VA + rigid
$U^{\not\perp}_m$	implicit	OW-CPA + rigid

Algorithm 6 KeyGen()

```
1: $(\texttt{pk}, \texttt{sk}') \xleftarrow{\$} \texttt{KeyGen}_{\texttt{PKE}}()$
2: $z \xleftarrow{\$} \mathcal{M}$
3: $\texttt{sk} \leftarrow (\texttt{sk}', z)$                ▷ $U^{\not\perp}, U^{\not\perp}_m$
4: $\texttt{sk} \leftarrow \texttt{sk}'$                    ▷ $U^{\perp}, U^{\perp}_m$
5: return $(\texttt{pk}, \texttt{sk})$
```

Algorithm 7 Encap(pk)

```
1: $m \xleftarrow{\$} \mathcal{M}$
2: $r \leftarrow G(m)$
3: $c \leftarrow \texttt{Enc}(\texttt{pk}, m, r)$
4: $K \leftarrow H(m, c)$                    ▷ $U^{\perp}, U^{\not\perp}$
5: $K \leftarrow H(m)$                       ▷ $U^{\perp}_m, U^{\not\perp}_m$
6: return $(c, K)$
```

Algorithm 8 $\texttt{Decap}(\texttt{sk} = (\texttt{sk}', z), c)$

```
1: $\hat{m} \leftarrow \texttt{Dec}(\texttt{sk}', c)$
2: $\hat{r} \leftarrow G(\hat{m})$
3: $\hat{c} \leftarrow \texttt{Enc}(\texttt{pk}, \hat{m}, \hat{r})$
4: if $\hat{c} = c$ then
5:     $K \leftarrow H(\hat{m})$             ▷ $U^{\perp}_m, U^{\not\perp}_m$
6:     $K \leftarrow H(\hat{m}, c)$          ▷ $U^{\perp}, U^{\not\perp}$
7: else
8:     $K \leftarrow H(z, c)$                ▷ $U^{\not\perp}, U^{\not\perp}_m$
9:     $K \leftarrow \perp$                  ▷ $U^{\perp}, U^{\perp}_m$
10: end if
11: return $K$
```

Figure 6: Summary of the modular Fujisaki-Okamoto transformation variations

The modular FO transformations enjoy tight security bounds and proven quantum resistance. Variations have been deployed to many post-quantum KEMs submitted to NIST's post-quantum cryptography competition. Kyber, one of the round 3 finalists, uses the $U^{\not\perp}$ transformation. When it was later standardized into FIPS-203, it changed to use the $U^{\not\perp}_m$ transformation for computational efficiencies.

3 The "encrypt-then-MAC" transformation

Let $\mathcal{B}^*$ denote the set of finite bit strings. Let $\texttt{PKE}(\texttt{KeyGen}, \texttt{Enc}, \texttt{Dec})$ be a public-key encryption scheme defined over message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$. Let $\texttt{MAC} : \mathcal{K}_{\texttt{MAC}} \times \mathcal{B}^* \to \mathcal{T}$ be a deterministic message authentication code that takes a key $k \in \mathcal{K}_{\texttt{MAC}}$, some message $m \in \mathcal{B}^*$, and outputs a digest $t \in \mathcal{T}$. Let $G : \mathcal{M} \to \mathcal{K}_{\texttt{MAC}}$ be a hash function that maps from PKE's plaintext space to MAC's key space. Let $H : \mathcal{B}^* \to \mathcal{K}_{\texttt{KEM}}$ be a hash function that maps bit strings into the set of possible shared secrets. The "encrypt-then-MAC" transformation $\texttt{EtM}[\texttt{PKE}, \texttt{MAC}, G, H]$ constructs a key encapsulation mechanism $\texttt{KEM}_{\texttt{EtM}}(\texttt{KeyGen}_{\texttt{KEM}}, \texttt{Encap}, \texttt{Decap})$, whose routines are described in figure 7.

Algorithm 9 $\texttt{KeyGen}_{\texttt{EtM}}$

```
1: $(\texttt{pk}, \texttt{sk}') \xleftarrow{\$} \texttt{KeyGen}(1^\lambda)$
2: $z \xleftarrow{\$} \mathcal{M}$
3: $\texttt{sk} \leftarrow (\texttt{sk}', z)$
4: return $(\texttt{pk}, \texttt{sk})$
```

Algorithm 10 $\texttt{Encap}(\texttt{pk})$

```
1: $m \xleftarrow{\$} \mathcal{M}$
2: $k \leftarrow G(m)$
3: $c' \xleftarrow{\$} \texttt{Enc}(\texttt{pk}, m)$
4: $t \leftarrow \texttt{MAC}(k, c')$
5: $K \leftarrow H(m, c')$
6: $c \leftarrow (c', t)$
7: return $(c, K)$
```

Algorithm 11 $\texttt{Decap}(\texttt{sk}, c)$

```
1: $(c', t) \leftarrow c$
2: $(\texttt{sk}', z) \leftarrow \texttt{sk}$
3: $\hat{m} \leftarrow \texttt{Dec}(\texttt{sk}', c')$
4: $\hat{k} \leftarrow G(\hat{m})$
5: if $\texttt{MAC}(\hat{k}, c') \neq t$ then
6:     $K \leftarrow H(z, c')$
7: else
8:     $K \leftarrow H(\hat{m}, c')$
9: end if
10: return $K$
```

Figure 7: $\texttt{KEM}_{\texttt{EtM}}$ routines

The key generation routine of $\texttt{KEM}_{\texttt{EtM}}$ is largely identical to that of the PKE, only a secret value z is sampled as the implicit rejection symbol. In the encapsulation routine, a MAC key is derived from the randomly sampled plaintext $k \leftarrow G(m)$, then used to sign the unauthenticated ciphertext c'. Because the encryption routine might be randomized, the session key is derived from both the message and the ciphertext. Finally, the unauthenticated ciphertext c' and the tag t combine into the authenticated ciphertext c that would be transmitted to the peer. In the decapsulation routine, the decryption $\hat{m}$ of the unauthenticated ciphertext is used to re-derive the MAC key $\hat{k}$, which is then used to re-compute the tag $\hat{t}$. The ciphertext is considered valid if and only if the recomputed tag is identical to the input tag.

For an adversary A to produce a valid tag t for some unauthenticated ciphertext c' under the symmetric key $k \leftarrow G(\texttt{Dec}(\texttt{sk}', c'))$ implies that A must either know the symmetric key k or produce a forgery. Under the random oracle model, A also cannot know k without knowing its preimage $\texttt{Dec}(\texttt{sk}', c')$, so A must either have produced c' honestly, or have broken the one-way security of PKE. This means that the decapsulation oracle will not give out information on decryptions that the adversary does not already know.

Algorithm 12 $\texttt{PCO}(m, c)$
```
1: $k \leftarrow G(m)$
2: $t \leftarrow \texttt{MAC}(k, c)$
3: **return** $[\![\mathcal{O}^{\texttt{Decap}}((c, t)) = H(m, c)]\!]$
```

Figure 8: Every decapsulation oracle can be converted into a plaintext-checking oracle

However, a decapsulation oracle can still give out some information: for a known plaintext m, all possible encryptions $c' \overset{\$}{\leftarrow} \texttt{Enc}(\texttt{pk}, m)$ can be correctly signed, while ciphertexts that don't decrypt back to m cannot be correctly signed. This means that a decapsulation oracle can be converted into a plaintext-checking oracle (algorithm 12), so every chosen-ciphertext attack against the KEM can be converted into a plaintext-checking attack against the underlying PKE.

On the other hand, if the underlying PKE is one-way secure against plaintext-checking attack that makes q plaintext-checking queries, then "encrypt-then-MAC" KEM is semantically secure under chosen ciphertext attacks making the same number of decapsulation queries:

Theorem 1. *For every* IND-CCA2 *adversary A against* $\texttt{KEM}_{\texttt{EtM}}$ *that makes q decapsulation queries, there exists an* OW-PCA *adversary B who makes at least q plaintext-checking queries against the underlying* PKE*, and an one-time existential forgery adversary C against the underlying* MAC *such that*

$$\texttt{Adv}_{\texttt{IND-CCA2}}(A) \leq q \cdot \texttt{Adv}_{\texttt{OT-MAC}}(C) + 2 \cdot \texttt{Adv}_{\texttt{OW-PCA}}(B)$$

Theorem 1 naturally flows into an equivalence relationship between the security of the KEM and the security of the PKE:

Lemma 2. $\texttt{KEM}_{\texttt{EtM}}$ *is IND-CCA2 secure if and only if the input* PKE *is OW-PCA secure*

3.1 Proof of theorem 1

Proof. We will prove theorem 1 using a sequence of games.

Algorithm 13 IND-CCA2 game for KEM

```
1: $(\mathtt{pk}, \mathtt{sk}) \xleftarrow{\$} \mathtt{KeyGen}_{\mathtt{EtM}}()$
2: $m^* \xleftarrow{\$} \mathcal{M}$
3: $c' \xleftarrow{\$} \mathtt{Enc}(\mathtt{pk}, m^*)$
4: $k^* \leftarrow G(m^*)$
5: $t \leftarrow \mathtt{MAC}(k, c')$
6: $c^* \leftarrow (c', t)$
7: $K_0 \leftarrow H(m^*, c')$
8: $K_1 \xleftarrow{\$} \mathcal{K}_{\mathtt{KEM}}$
9: $b \xleftarrow{\$} \{0, 1\}$
10: $\hat{b} \leftarrow A^{\mathcal{O}^{\mathtt{Decap}}}(\mathtt{pk}, c^*, K_b)$
11: return $[\![\hat{b} = b]\!]$
```

Algorithm 14 $\mathcal{O}^{\mathtt{Decap}}(c)$

```
1: $(c', t) \leftarrow c$
2: $\hat{m} = \mathtt{Dec}(\mathtt{sk}', c')$
3: $\hat{k} \leftarrow G(\hat{m})$
4: if $\mathtt{MAC}(\hat{k}, c') = t$ then
5:     $K \leftarrow H(\hat{m}, c')$
6: else
7:     $K \leftarrow H(z, c')$
8: end if
9: return $K$
```

Algorithm 15 $\mathcal{O}^G(m)$

```
1: if $\exists(\tilde{m}, \tilde{k}) \in \mathcal{L}^G : \tilde{m} = m$ then
2:     return $\tilde{k}$
3: end if
4: $k \xleftarrow{\$} \mathcal{K}_{\mathtt{MAC}}$
5: $\mathcal{L}^G \leftarrow \mathcal{L}^G \cup \{(m, k)\}$
6: return $k$
```

Algorithm 16 $\mathcal{O}^H(m, c)$

```
1: if $\exists(\tilde{m}, \tilde{c}, \tilde{K}) \in \mathcal{L}^H : \tilde{m} = m \wedge \tilde{c} = c$
   then
2:     return $\tilde{K}$
3: end if
4: $K \xleftarrow{\$} \mathcal{K}_{\mathtt{KEM}}$
5: $\mathcal{L}^H \leftarrow \mathcal{L}^H \cup \{(m, c, K)\}$
6: return $K$
```

Game 0 is the standard IND-CCA2 game for KEMs. The decapsulation oracle $\mathcal{O}^{\mathtt{Decap}}$ executes the decapsulation routine using the challenge keypair and return the results faithfully. The queries made to the hash oracles $\mathcal{O}^G, \mathcal{O}^H$ are recorded to their respective tapes $\mathcal{L}^G, \mathcal{L}^H$.

Algorithm 17 $\mathcal{O}_1^{\mathtt{Decap}}(c)$

```
1: $(c', t) \leftarrow c$
2: if $\exists(\tilde{m}, \tilde{k}) \in \mathcal{L}^G : \tilde{m} = \mathtt{Dec}(\mathtt{sk}', c') \wedge \mathtt{MAC}(\tilde{k}, c') = t$ then
3:     $K \leftarrow H(\tilde{m}, c')$
4: else
5:     $K \leftarrow H(z, c')$
6: end if
7: return $K$
```

Figure 9: Simulated decapsulation oracle

Game 1 is identical to game 0 except that the true decapsulation oracle $\mathcal{O}^{\mathtt{Decap}}$ is replaced with a simulated oracle $\mathcal{O}_1^{\mathtt{Decap}}$. Instead of directly decrypting c' as in the decapsulation routine, the simulated oracle searches through the tape $\mathcal{L}^G$ to find a matching query $(\tilde{m}, \tilde{k})$ such that $\tilde{m}$ is the decryption of c'. The simulated oracle then uses $\tilde{k}$ to validate the tag t against c'.

If the simulated oracle accepts the queried ciphertext as valid, then there is a matching query that also validates the tag, which means that the queried ciphertext is honestly generated. Therefore, the true oracle must also accept the queried ciphertext. On the other hand, if the true oracle rejects the queried ciphertext (and output the implicit rejection $H(z, c'))$, then the tag is simply invalid under the MAC key $k = G(\texttt{Dec}(\texttt{sk}', c'))$. Therefore, there could not have been a matching query that also validates the tag, and the simulated oracle must also rejects the queried ciphertext.

This means that from the adversary A's perspective, game 1 and game 0 differ only when the true oracle accepts while the simulated oracle rejects, which means that t is a valid tag for c' under $k = G(\texttt{Dec}(\texttt{sk}', c'))$, but k has never been queried. Under the random oracle model, such k is a uniformly random sample of $\mathcal{K}_{\texttt{MAC}}$ that the adversary does not know, so for A to produce a valid tag is to produce a forgery against the MAC under an unknown and uniformly random key. Furthermore, the security game does not include a signing oracle, so this is a zero-time forgery. While zero-time forgery is not a standard security definition for a MAC, we can bound it by the advantage of a one-time forgery adversary C:

$$P\left[\mathcal{O}^{\texttt{Decap}}(c) \neq \mathcal{O}_1^{\texttt{Decap}}(c)\right] \leq \texttt{Adv}_{\texttt{OT-MAC}}(C)$$

Across all q decapsulation queries, the probability that at least one query is a forgery is thus at most $q \cdot P\left[\mathcal{O}^{\texttt{Decap}}(c) \neq \mathcal{O}_1^{\texttt{Decap}}(c)\right]$. By the difference lemma:

$$\texttt{Adv}_{G_0}(A) - \texttt{Adv}_{G_1}(A) \leq q \cdot \texttt{Adv}_{\texttt{OT-MAC}}(C)$$

Game 2 is identical to game 1, except that on line 4 of algorithm 13, the challenger samples a uniformly random MAC key $k^* \xleftarrow{\$} \mathcal{K}_{\texttt{MAC}}$ instead of deriving it from m. From A's perspective the two games are indistinguishable, unless A queries G with the value of m^*. Denote the probability that A queries G with m^* by $P[\texttt{QUERY G}]$, then:

$$\texttt{Adv}_{G_1}(A) - \texttt{Adv}_{G_2}(A) \leq P\left[\texttt{QUERY G}\right]$$

Game 3 is identical to game 2, except that on line 7 of algorithm 13, the challenger samples a uniformly random shared secret $K_0 \xleftarrow{\$} \mathcal{K}_{\texttt{KEM}}$ instead of deriving it from m^* and c'. From A's perspective the two games are indistinguishable, unless A queries H with $(m^*, \cdot)$. Denote the probability that A queries H with $(m^*, \cdot)$ by $P[\texttt{QUERY H}]$, then:

$$\texttt{Adv}_{G_2}(A) - \texttt{Adv}_{G_3}(A) \leq P\left[\texttt{QUERY H}\right]$$

Since in game 3, both K_0 and K_1 are uniformly random and independent of all other variables, no adversary can have any advantage: $\texttt{Adv}_{G_3}(A) = 0$.

Algorithm 18 $B(\mathtt{pk}, c'^*)$

```
1: $z \xleftarrow{\$} \mathcal{M}$
2: $k \xleftarrow{\$} \mathcal{K}_{\mathtt{MAC}}$
3: $t \leftarrow \mathtt{MAC}(k, c'^*)$
4: $c^* \leftarrow (c'^*, t)$
5: $K \xleftarrow{\$} \mathcal{K}_{\mathtt{KEM}}$
6: $\hat{b} \leftarrow A^{\mathcal{O}_B^{\mathtt{Decap}}, \mathcal{O}_B^G, \mathcal{O}_B^H}(\mathtt{pk}, c^*, K)$
7: if $\mathtt{ABORT}(m)$ then
8:     return $m$
9: end if
```

Algorithm 19 $\mathcal{O}_B^{\mathtt{Decap}}(c)$

```
1: $(c', t) \leftarrow c$
2: if $\exists(\tilde{m}, \tilde{k}) \in \mathcal{L}^G : \mathtt{PCO}(c', \tilde{m}) = 1 \land \mathtt{MAC}(\tilde{k}, c') = t$ then
3:     $K \leftarrow H(\tilde{m}, c')$
4: else
5:     $K \leftarrow H(z, c')$
6: end if
7: return $K$
```

Algorithm 20 $\mathcal{O}_B^G(m)$

```
1: if $\mathtt{PCO}(m, c'^*) = 1$ then
2:     $\mathtt{ABORT}(m)$
3: end if
4: if $\exists(\tilde{m}, \tilde{k}) \in \mathcal{L}^G : \tilde{m} = m$ then
5:     return $\tilde{k}$
6: end if
7: $k \xleftarrow{\$} \mathcal{K}_{\mathtt{MAC}}$
8: $\mathcal{L}^G \leftarrow \mathcal{L}^G \cup \{(m, k)\}$
9: return $k$
```

Algorithm 21 $\mathcal{O}_B^H(m, c)$

```
if $\mathtt{PCO}(m, c'^*) = 1$ then
    $\mathtt{ABORT}(m)$
end if
if $\exists(\tilde{m}, \tilde{c}, \tilde{K}) \in \mathcal{L}^H : \tilde{m} = m \land \tilde{c} = c$ then
    return $\tilde{K}$
end if
$K \xleftarrow{\$} \mathcal{K}_{\mathtt{KEM}}$
$\mathcal{L}^H \leftarrow \mathcal{L}^H \cup \{(m, c, K)\}$
return $K$
```

We will bound $P[\mathtt{QUERY\ G}]$ and $P[\mathtt{QUERY\ H}]$ by constructing a OW-PCA adversary B against the underlying PKE that uses A as a sub-routine. B's behaviors are described in algorithms 18, 19, 20, and 21.

B simulates game 3 for A: receiving the public key $\mathtt{pk}$ and challenge encryption c'^*, B samples random MAC key and session key to produce the challenge encapsulation, then feeds it to A. When simulating the decapsulation oracle, B uses the plaintext-checking oracle to look for matching queries in $\mathcal{L}^G$. When simulating the hash oracles, B uses the plaintext-checking oracle to detect when $m^* = \mathtt{Dec}(\mathtt{sk}', c'^\star)$ has been queried. When m^* is queried, B terminates A and returns m^* to win the OW-PCA game. In other words:

$$P[\mathtt{QUERY\ G}] \leq \mathtt{Adv}_{\mathtt{OW\text{-}PCA}}(B)$$
$$P[\mathtt{QUERY\ H}] \leq \mathtt{Adv}_{\mathtt{OW\text{-}PCA}}(B)$$

Combining all equations above produce the desired security bound. □

4 Implementation

Originally known as Kyber [BDK+18][ABD+19], ML-KEM is an `IND-CCA2` secure key encapsulation mechanism standardized in FIPS 203 by NIST. The `IND-CCA2` security of ML-KEM is achieved in two steps. First, ML-KEM constructs an `IND-CPA` secure public key encryption scheme $\mathtt{K\text{-}PKE}(\mathtt{KeyGen}, \mathtt{Enc}, \mathtt{Dec})$ whose security is based on the conjectured intractability of the module learning with error (MLWE) problems against both classical and quantum adversaries. Then, the $U_m^{\not\perp}$ variant of the Fujisaki-Okamoto transformation [HHK17] is used to construct the KEM $\mathtt{MLKEM}(\mathtt{KeyGen}, \mathtt{Encap}, \mathtt{Decap})$ by calling

`K-PKE.KeyGen`, `K-PKE.Enc`, `K-PKE.Dec` as sub-routines. Because `K-PKE.Enc` includes substantially more arithmetics than `K-PKE.Dec`, by using *re-encryption* and *de-randomization*, ML-KEM's decapsulation routine suffers from computational inefficiency.

We implemented the "encrypt-then-MAC" KEM construction using `K-PKE` as the input PKE and compared its performance against ML-KEM under a variety of scenarios. The experimental data showed that while the "encrypt-then-MAC" construction adds a small amount of computational overhead to the encapsulation routine and a small increase in ciphertext size when compared with ML-KEM, it boasts enormous runtime savings in the decapsulation routine, which makes it particularly suitable for deployment in constrained environment.

A detailed description of `K-PKE`'s routines can be found in FIPS 203 (TODO: citation). The "encrypt-then-MAC" routines are listed in algorithms 22, 23, and 24.

Algorithm 22 `ML-KEM-ETM.KeyGen()`

```
1: $z \xleftarrow{\$} \{0,1\}^{256}$
2: $(\texttt{pk}, \texttt{sk}') \xleftarrow{\$}$ K-PKE.KeyGen()
3: $h \leftarrow H(\texttt{pk})$                                ▷ $H$ is SHA3-256
4: $\texttt{sk} \leftarrow (sk' \| \texttt{pk} \| h \| z)$
5: return $(\texttt{pk}, \texttt{sk})$
```

Algorithm 23 `ML-KEM-ETM.Encap(pk)`

```
Require: Public key pk
1: $m \xleftarrow{\$} \{0,1\}^{256}$
2: $(\overline{K}, r, k) \leftarrow \texttt{XOF}(m \| H(\texttt{pk}))$          ▷ XOF is Shake256, outputting 768 bits
3: $c' \leftarrow \texttt{K-PKE.Enc}(\texttt{pk}, m, r)$
4: $t \leftarrow \texttt{MAC}(k, c')$
5: $K \leftarrow \texttt{KDF}(\overline{K} \| t)$                    ▷ KDF is Shake256, outputting 256 bits
6: $c \leftarrow (c', c)$
7: return $(c, K)$
```

Algorithm 24 `ML-KEM-ETM.Decap(sk, c)`

```
Require: Secret key $\texttt{sk} = (\texttt{sk}' \| \texttt{pk} \| h \| z)$
Require: Ciphertext $c = (c' \| t)$
 1: $(\texttt{sk}', \texttt{pk}, h, z) \leftarrow \texttt{sk}$
 2: $(c', t) \leftarrow c$
 3: $\hat{m} \leftarrow \texttt{K-PKE.Dec}(\texttt{sk}', c')$
 4: $(\overline{K}, \hat{r}, \hat{k}) \leftarrow \texttt{XOF}(\hat{m} \| h)$
 5: $\hat{t} \leftarrow \texttt{MAC}(\hat{k}, c')$
 6: if $\hat{t} = t$ then
 7:     $K \leftarrow \texttt{KDF}(\overline{K} \| t)$
 8: else
 9:     $K \leftarrow \texttt{KDF}(z \| t)$
10: end if
11: return $K$
```

Our implementation extended from the reference implementation by the PQCrystals team (https://github.com/pq-crystals/kyber). All C code is compiled with `GCC 11.4.1` and `OpenSSL 3.0.8`. All binaries are executed on an AWS c7a.medium instance with an AMD EPYC 9R14 CPU at 3.7 GHz and 1 GB of RAM.

4.1 Choosing a message authenticator

When instantiating `ML-KEM-ETM`, there are a variety of message authentication codes to choose from. We selected four instances covering a wide range of designs:

1. Poly1305, a Carter-Wegman style MAC operating on a prime field
2. GMAC (AES-256-GCM), a Carter-Wegman style MAC operating on a binary field
3. CMAC (AES-256-CBC), a CBC-MAC
4. KMAC-256, a keyed hash function based on Keccak

We tested each MAC's throughput by measuring the CPU cycles needed to compute a tag on the ciphertext returned by `K-PKE`. The length of the ciphertext varies depending on the security level. The measurements are summarized in table 2.

Table 2: MAC performance

ML-KEM-512							
Ciphertext size (bytes):				768			
Poly1305		GMAC		CMAC		KMAC	
Median	909	Median	3899	Median	6291	Median	6373
Average	2823	Average	4859	Average	6373	Average	7791
ML-KEM-768							
Ciphertext size (bytes):				1088			
Poly1305		GMAC		CMAC		KMAC	
Median	961	Median	3899	Median	7305	Median	9697
Average	2704	Average	4827	Average	7588	Average	9928
ML-KEM-1024							
Ciphertext size (bytes):				1568			
Poly1305		GMAC		CMAC		KMAC	
Median	1065	Median	4055	Median	8735	Median	11647
Average	1809	Average	5026	Average	8772	Average	12186

4.2 KEM performance

4.3 Key exchange performance

5 Application to Kyber

CRYSTALS-Kyber [BDK+18][ABD+19] is an IND-CCA2 secure key encapsulation mechanism whose security is based on the conjecture hardness of the decisional Module Learning with Error problem. To achieve the the IND-CCA2 security, Kyber first constructs an IND-CPA secure public key encryption scheme based on [LPR13], then apply a slightly modified variation of the Fujisaki-Okamoto transformation described in [HHK17]. The resulting decapsulation routine is especially inefficient because Kyber's IND-CPA encryption routine incurs significantly more computational cost than the decryption routine. This makes Kyber a prime target for demonstrating the performance improvements enjoyed by the "encrypt-then-MAC" KEM construction.

We took the IND-CPA PKE routines (algorithms 4, 5, 6 in [ABD+19]) and applied the "encrypt-then-MAC" transformation. The resulting KEM routines are described in algorithms 25, 26, and 27.

Algorithm 25 `Kyber.CCAKEM.KeyGen()`

```
1: $z \xleftarrow{\$} \{0,1\}^{256}$
2: $(\mathtt{pk}, \mathtt{sk}') \xleftarrow{\$}$ Kyber.CPAPKE.KeyGen()
3: $\mathtt{sk} = (\mathtt{sk}', \mathtt{pk}, H(\mathtt{pk}), z)$        ▷ H is instantiated with SHA3-256
4: return $(\mathtt{pk}, \mathtt{sk})$
```

Algorithm 26 `Kyber.CCAKEM.Encap`$^{+}$`(pk)`

```
1: $m \xleftarrow{\$} \{0,1\}^{256}$
2: $(\bar{K}, r, k) = G(m \| H(\mathtt{pk}))$        ▷ G is instantiated with SHA3-768
3: $c' \leftarrow$ Kyber.CPAPKE.Enc$(\mathtt{pk}, m, r)$
4: $t = \mathtt{MAC}(k, c')$
5: $K = \mathtt{KDF}(\bar{K} \| t)$        ▷ KDF is instantiated with Shake256
6: $c \leftarrow (c', t)$
7: return $(c, K)$
```

Algorithm 27 `Kyber.CCAKEM.Decap`$^{+}(\mathtt{sk}, c)$

```
Require: Secret key $\mathtt{sk} = (\mathtt{sk}', \mathtt{pk}, H(\mathtt{pk}), z)$
Require: Ciphertext $c = (c', t)$
 1: $(\mathtt{sk}', \mathtt{pk}, h, z) \leftarrow \mathtt{sk}$
 2: $(c', t) \leftarrow c$
 3: $\hat{m} =$ Kyber.CPAPKE.Dec$(\mathtt{sk}', c')$
 4: $(\overline{K}, k) = G(m' \| h)$
 5: $\hat{t} = \mathtt{MAC}(k, c)$
 6: if $\hat{t} = t$ then
 7:     $K = \mathtt{KDF}(\bar{K} \| t)$
 8: else
 9:     $K = \mathtt{KDF}(z \| t)$
10: end if
11: return $K$
```

Remark 1. We derive the MAC key from both the plaintext m and the public key `pk` for the same reason [ABD+19] derives the pseudorandom coin from both m and `pk`. One is to allow both parties to participate in the encryption process, the other is to prevent a quantum adversary from pre-computing a large table of MAC keys that can then be brute-forced.

Remark 2. We chose to derive the shared secret K from the ciphertext digest t instead of the ciphertext itself, which saves a few Keccak permutations since t is only 128 bits while the full ciphertext could span more than a thousand bytes. This should not impact the security of the scheme since finding collision for an unknown MAC key constitutes a forgery attack on the MAC.

Remark 3. We constructed the "encrypt-then-MAC" transformation to use implicit rejection so that we can directly use existing implementation of Kyber with minimal modification. In principle, a construction with explicit rejection should also be equally secure and efficient.

Remark 4. We chose to use MAC with 256-bit key size and 128-bit tag size. When modeling the security threats, we assumed that the adversary may have access to quantum computers, making it necessary to use the maximal key size of common MACs while maintaining the minimum 128-bit security. On the other hand, the tag size can be relative small because the decapsulator (aka the decapsulation oracle) is assumed to be a classical computer, so there is no quantum speedup on brute-forcing a valid tag.

When instantiating an instance of `KEM`(`KeyGen`, `Encap`, `Decap`), there are a variety of MAC’s to chose from. We experimented with Poly1305, AES-256-GCM (aka GMAC), AES-256-CBC (aka CMAC), and KMAC-256. We instantiated an intance of the “encrypt-then-MAC” KEM using each of the chosen MAC, then measured the median number of CPU cycles needed to perform the key generation, encapsulation, and decapsulation routines among 10000 runs. The Kyber implementation is taken from the reference implementation (https://github.com/pq-crystals/kyber). MAC implementations are taken from OpenSSL 3.3.1. The source code is compiled with GCC 11.4.1 on Amazon Linux 2. Performance measurements were taken from a c7a.medium AWS EC2 instance with a AMD EPYC 9R14 (1) @ 3.700GHz. The experimental results are listed in table 3.

Table 3: Performance measurements

Name	Security level	`KeyGen`	`Encap`	`Decap`
ML-KEM-512	128 bits	75945	91467	121185
ML-KEM-512 + Poly1305	128 bits	76907	93157	33733
ML-KEM-512 + GMAC	128 bits	76917	95419	37725
ML-KEM-512 + CMAC	128 bits	76907	99839	40117
ML-KEM-512 + KMAC-256	128 bits	76387	101009	40741
ML-KEM-768	192 bits	129895	146405	186445
ML-KEM-768 + Poly1305	192 bits	128205	146405	43315
ML-KEM-768 + GMAC	192 bits	127997	149525	46513
ML-KEM-768 + CMAC	192 bits	129167	151007	49841
ML-KEM-768 + KMAC-256	192 bits	128829	155219	52415
ML-KEM-1024	256 bits	194921	199185	246245
ML-KEM-1024 + Poly1305	256 bits	196013	205763	51375
ML-KEM-1024 + GMAC	256 bits	196039	208805	54573
ML-KEM-1024 + CMAC	256 bits	195389	213667	59175
ML-KEM-1024 + KMAC-256	256 bits	196117	216761	62269

Compared to Kyber using the FO transform, our proposed construction adds a small amount of runtime overhead (for computing a digest) in the encapsulation routine and a small increase in ciphertext size (for the 128-bit tag). In exchange, we see significant performance runtime savings in the decapsulation routines. This trade-off is especially meaningful in a key exchange protocol where one party has substantially more computationl resource than the other. For example, in many experimental implementation of TLS with post-quantum KEMs (such as CECPQ2), the client (might be IoT devices) runs `KeyGen` and `Decap` while the server (usually data centers) runs `Encap`.

5.1 Key exchange protocols

A common application of key encapsulation mechanism is key exchange protocols, where two parties establish a shared secret using a public channel. [BDK$^+$18] described three key exchange protocols: unauthenticated key exchange (KE), unilaterally authenticated key exchange (UAKE), and mutually authenticated key exchange (AKE). We instantiated an implementation for each of the three key exchange protocols using different variations of the "encrypt-then-MAC" KEM and compared round trip time with implementations instantiated using ML-KEM.

For clarity, we denote the party who sends the first message to be the client and the other party to be the server. Round trip time (RTT) is defined to be the time interval between the moment before the client starts generating ephemeral keypairs and the moment after the client derives the final session key. All experiements are run on a pair of AWS c7a.medium instances both located in the `us-west-2` region. For each experiment, a total of 10,000 rounds of key exchange are performed, with the median and average round trip time (measured in microsecond) recorded.

5.1.1 Unauthenticated key exchange (KE)

In unauthenticated key exchange, a single pair of ephemeral keypair $(\mathtt{pk}_e, \mathtt{sk}_e) \overset{\$}{\leftarrow} \mathtt{KeyGen}()$ is generated by the client. The client transmits the ephemeral public key $\mathtt{pk}_e$ to the server, who runs the encapsulation routine $(c_e, K_e) \overset{\$}{\leftarrow} \mathtt{Encap}(\mathtt{pk}_e)$ and transmits the ciphertext c_e back to the client. The client finally decapsulates the ciphertext to recover the shared secret $K_e \leftarrow \mathtt{Decap}(\mathtt{sk}_e, c_e)$. The specific steps are described in algorithms 28, 29.

Note that in our implementation, a key derivation function (KDF) is applied to the ephemeral shared secret to derive the final session key. This step is added to maintain consistency with other authenticated key exchange protocols, where the final session key is derived from multiple shared secrets. The key derivation function is instantiated using Shake256, and the final session key is 256 bits in length.

Algorithm 28 $\mathtt{KE}_\mathtt{C}$

```
1: $(\mathtt{pk}_e, \mathtt{sk}_e) \overset{\$}{\leftarrow} \mathtt{KeyGen}()$
2: $\mathtt{send}(\mathtt{pk}_e)$
3: $c_e \leftarrow \mathtt{read}()$
4: $K_e \leftarrow \mathtt{Decap}(\mathtt{sk}_e, c_e)$
5: $K \leftarrow \mathtt{KDF}(K)$
6: return $K$
```

Figure 10: KE Client

Algorithm 29 $\mathtt{KE}_\mathtt{S}$

```
1: $\mathtt{pk}_e \leftarrow \mathtt{read}()$
2: $(c_e, K_e) \overset{\$}{\leftarrow} \mathtt{Encap}(\mathtt{pk}_e)$
3: $\mathtt{send}(c_e)$
4: $K \leftarrow \mathtt{KDF}(K_e)$
5: return $K$
```

Figure 11: KE Server

The RTT comparison is summarized in table 4

Table 4: Unauthenticated key exchange RTT comparison

KEM	median RTT (μs)	average RTT (μs)
ML-KEM-512	92	97
ML-KEM-512 + Poly1305	70	72
ML-KEM-512 + GMAC	73	76
ML-KEM-512 + CMAC	75	79
ML-KEM-512 + KMAC	76	78
ML-KEM-768	135	140
ML-KEM-768 + Poly1305	99	104
ML-KEM-768 + GMAC	101	105
ML-KEM-768 + CMAC	103	109
ML-KEM-768 + KMAC	103	107
ML-KEM-1024	193	199
ML-KEM-1024 + Poly1305	138	141
ML-KEM-1024 + GMAC	140	145
ML-KEM-1024 + CMAC	143	148
ML-KEM-1024 + KMAC	144	149

5.1.2 Unilaterally authenticated key exchange (UAKE)

In unilaterally authenticated key exchange, the authenticating party proves its identity to the other party by demonstrating possession of a secret key that corresponds to a published long-term public key. In this implementation, the client possesses the long-term public key $\texttt{pk}_S$ of the server, and the server authenticates itself by demonstrating possession of the corresponding long-term secret key $\texttt{sk}_S$. Details are described in algorithms 30 and 31.

In addition to the long-term key, the client will also generate an ephemeral keypair as it does in an unauthenticated key exchange, and the session key is derived by applying the KDF to the concatenation of both the ephemeral shared secret and the shared secret encapsulated under server's long-term key. This helps the key exchange to achieve weak forward secrecy (citation needed).

Using KEM for authentication is especially interesting within the context of post-quantum cryptography: post-quantum KEM schemes usually enjoy better performance characteristics than post-quantum signature schemes with faster runtime, smaller memory footprint, and smaller communication sizes. KEMTLS was proposed in 2020 as an alternative to existing TLS handshake protocols, and many experimental implementations have demonstrated the performance advantage. (citation needed).

Algorithm 30 $\texttt{UAKE}_\texttt{C}(\texttt{pk}_S)$

```
Require: Server's long-term $\texttt{pk}_S$
1: $(\texttt{pk}_e, \texttt{sk}_e) \xleftarrow{\$} \texttt{KeyGen}()$
2: $(c_S, K_S) \xleftarrow{\$} \texttt{Encap}(\texttt{pk}_S)$
3: $\texttt{send}(\texttt{pk}_e, c_S)$
4: $c_e \leftarrow \texttt{read}()$
5: $K_e \leftarrow \texttt{Decap}(\texttt{sk}_e, c_e)$
6: $K \leftarrow \texttt{KDF}(K_e \| K_S)$
7: return $K$
```

Figure 12: UAKE Client

Algorithm 31 $\texttt{UAKE}_\texttt{S}(\texttt{sk}_S)$

```
Require: Server's long-term $\texttt{sk}_S$
1: $(\texttt{pk}_e, c_S) \leftarrow \texttt{read}()$
2: $K_S \leftarrow \texttt{Decap}(\texttt{sk}_S, c_S)$
3: $(c_e, K_e) \xleftarrow{\$} \texttt{Encap}(\texttt{pk}_e)$
4: $\texttt{send}(c_e)$
5: $K \leftarrow \texttt{KDF}(K_e \| K_S)$
6: return $K$
```

Figure 13: UAKE Server

Table 5: UAKE RTT comparison

KEM	median RTT (μs)	average RTT (μs)
ML-KEM-512	145	151
ML-KEM-512 + Poly1305	103	106
ML-KEM-512 + GMAC	106	110
ML-KEM-512 + CMAC	108	112
ML-KEM-512 + KMAC	109	113
ML-KEM-768	215	222
ML-KEM-768 + Poly1305	144	150
ML-KEM-768 + GMAC	149	156
ML-KEM-768 + CMAC	153	160
ML-KEM-768 + KMAC	154	159
ML-KEM-1024	310	318
ML-KEM-1024 + Poly1305	202	209
ML-KEM-1024 + GMAC	212	228
ML-KEM-1024 + CMAC	212	218
ML-KEM-1024 + KMAC	213	220

5.1.3 Mutually authenticated key exchange (AKE)

Mutually authenticated key exchange is largely identical to unilaterally authenticated key exchange, except for that client authentication is required. This means that client possesses server's long-term public key and its own long-term secret key, while the server possesses client's long-term public key and its own long-term secret key. The session key is derived by applying KDF onto the concatenation of shared secrets produced under the ephemeral keypair, server's long-term keypair, and client's long-term keypair, in this order.

Algorithm 32 $\texttt{AKE}_\texttt{C}(\texttt{pk}_S, \texttt{sk}_C)$

```
Require: Server's long-term $\texttt{pk}_S$
Require: Client's long-term $\texttt{sk}_C$
1: $(\texttt{pk}_e, \texttt{sk}_e) \xleftarrow{\$} \texttt{KeyGen}()$
2: $(c_S, K_S) \xleftarrow{\$} \texttt{Encap}(\texttt{pk}_S)$
3: $\texttt{send}(\texttt{pk}_e, c_S)$
4: $(c_e, c_C) \leftarrow \texttt{read}()$
5: $K_e \leftarrow \texttt{Decap}(\texttt{sk}_e, c_e)$
6: $K_C \leftarrow \texttt{Decap}(\texttt{sk}_e, c_C)$
7: $K \leftarrow \texttt{KDF}(K_e \| K_S \| K_C)$
8: return $K$
```

Figure 14: AKE Client

Algorithm 33 $\texttt{AKE}_\texttt{S}(\texttt{sk}_S, \texttt{pk}_C)$

```
Require: Server's long-term $\texttt{sk}_S$
Require: Client's long-term $\texttt{pk}_C$
1: $(\texttt{pk}_e, c_S) \leftarrow \texttt{read}()$
2: $K_S \leftarrow \texttt{Decap}(\texttt{sk}_S, c_S)$
3: $(c_e, K_e) \xleftarrow{\$} \texttt{Encap}(\texttt{pk}_e)$
4: $(c_C, K_C) \xleftarrow{\$} \texttt{Encap}(\texttt{pk}_C)$
5: $\texttt{send}(c_e, c_C)$
6: $K \leftarrow \texttt{KDF}(K_e \| K_S \| K_C)$
7: return $K$
```

Figure 15: AKE Server

Table 6: AKE RTT comparison

KEM	median RTT (μs)	average RTT (μs)
ML-KEM-512	200	213
ML-KEM-512 + Poly1305	133	138
ML-KEM-512 + GMAC	139	143
ML-KEM-512 + CMAC	143	148
ML-KEM-512 + KMAC	145	151
ML-KEM-768	294	301
ML-KEM-768 + Poly1305	190	196
ML-KEM-768 + GMAC	197	210
ML-KEM-768 + CMAC	202	208
ML-KEM-768 + KMAC	204	210
ML-KEM-1024	512	511
ML-KEM-1024 + Poly1305	266	273
ML-KEM-1024 + GMAC	273	282
ML-KEM-1024 + CMAC	280	287
ML-KEM-1024 + KMAC	282	288

6 Experimental Evaluation

We implement our scheme in C on top of CRYSTALS-Kyber's implementation available. We use the Poly1305 MAC from OpenSSL with a key of size 256 bits and a tag size of 128 bits. Our experiments were run on a desktop with a 2.3 GHz Intel Core i9 laptop (Coffee Lake) with 16 GB RAM. The codes were compiled using `clange 14.0.0`. Table 7 reports the cycle counts for three different instances of Keyber, namely . The cycle counts reported in Table 7 are the average of the cycle counts of 10000 executions of all algorithms.

Table 7: Cycle counts in AVX2 on a Coffee Lake laptop

Kyber512				Kyber512+Poly1305		Kyber512+GMAC	
Size in bytes		Clock cycles (AVX2)		Clock cycles (AVX2)		Clock cycles (AVX2)	
sk:	1632 (or 32)	**KeyGen:**		**KeyGen:**		**KeyGen:**	
pk:	800	**Encap:**		**Encap$^+$:**		**Encap$^+$:**	
ct:	768	**Decap:**		**Decap$^+$:**		**Decap$^+$:**	
Kyber768				Kyber768+Poly1305		Kyber768+GMAC	
Size in bytes		Clock cycles (AVX2)		Clock cycles (AVX2)		Clock cycles (AVX2)	
sk:	2400 (or 32)	**KeyGen:**		**KeyGen:**		**KeyGen:**	
pk:	1184	**Encap:**		**Encap$^+$:**		**Encap$^+$:**	
ct:	1088	**Decap:**		**Decap$^+$:**		**Decap$^+$:**	
Kyber1024				Kyber1024+Poly1305		Kyber1024+GMAC	
Size in bytes		Clock cycles (AVX2)		Clock cycles (AVX2)		Clock cycles (AVX2)	
sk:	3168 (or 32)	**KeyGen:**		**KeyGen:**		**KeyGen:**	
pk:	1568	**Encap:**		**Encap$^+$:**		**Encap$^+$:**	
ct:	1568	**Decap:**		**Decap$^+$:**		**Decap$^+$:**	

- Two tables, one-time MAC and many-time MAC, separate encap and decap in 2 tables
- Look at OpenSSL for GMAC with no AES-256.
- Also setup networking experiment on AWS

7 Conclusions and future works

Comparison with Fujisaki-Okamoto transformation: We applied the "encrypt-then-MAC" transformation to Kyber and saw meaningful performance improvements over using derandomization and re-encryption. Unfortunately the resulting KEM does not achieve the desired full IND-CCA2 security, because Kyber is known to be vulnerable to key-recovery plaintext-checking attack (KR-PCA) [RRCB19][UXT+22]. We speculate that while Kyber with "encrypt-then-MAC" could not achieve the full IND-CCA2 security, it can still be safe for use in ephemeral key exchange, where each secret key is used to decrypt at most one ciphertext (the KR-PCA requires a few hundred decryption queries to recover the secret key).

In section 3, we showed that if the input PKE is OW-PCA secure, then the resulting KEM is IND-CCA2 secure. One sufficient condition for OW-PCA security is one-way security plus rigidity. If the input PKE is rigid, then $m = \texttt{Dec}(\texttt{sk}, c)$ is equivalent to $c = \texttt{Enc}(\texttt{pk}, m)$, so a plaintext-checking oracle can be simulated without any secret information. However, the $U_m^{\not\perp}$ transformation in [HHK17] can already transform an OW-CPA secure and rigid PKE into an IND-CCA2 secure KEM with minimal overhead: the encapsulation and decapsulation routines each adds a hash of the plaintext to the encryption and decryption routine. In other words, where the input PKE is rigid, "encrypt-then-MAC" doesn't offer any performance advantage. It remains an open problem whether there exists a PKE that is OW-PCA secure but not rigid. If such a PKE exists, then "encrypt-then-MAC" would be a preferable strategy for constructing an IND-CCA2 KEM.

References

[ABD+19] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber algorithm specifications and supporting documentation. *NIST PQC Round*, 2(4):1–43, 2019.

[BCD+16] Joppe Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from lwe. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1006–1018, 2016.

[BDK+18] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber: a cca-secure module-lattice-based kem. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE, 2018.

[BN00] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 531–545. Springer, 2000.

[BP18] Daniel J Bernstein and Edoardo Persichetti. Towards kem unification. *Cryptology ePrint Archive*, 2018.

[BS20] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. *Draft 0.5*, 2020.

[DKSRV18] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-lwr based key exchange, cpa-secure encryption and cca-secure kem. In *Progress in Cryptology–AFRICACRYPT 2018: 10th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 7–9, 2018, Proceedings 10*, pages 282–305. Springer, 2018.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Annual international cryptology conference*, pages 537–554. Springer, 1999.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the fujisaki-okamoto transformation. In *Theory of Cryptography Conference*, pages 341–371. Springer, 2017.

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *J. ACM*, 60(6):43:1–43:35, 2013.

[RRCB19] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic side-channel attacks on cca-secure lattice-based pke and kem schemes. *Cryptology ePrint Archive*, 2019.

[Sho04] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. *cryptology eprint archive*, 2004.

[UXT$^+$22] Rei Ueno, Keita Xagawa, Yutaro Tanaka, Akira Ito, Junko Takahashi, and Naofumi Homma. Curse of re-encryption: a generic power/em analysis on post-quantum kems. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 296–322, 2022.