

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

Figure 1: The OW-ATK game

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

```

1: return  $\llbracket \text{Dec}(\text{sk}, c) = m \rrbracket$ 
```

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 $c^* \xleftarrow{\$} \text{Enc}(\text{pk}, m^*)$, then gives pk and c^* to the adversary A . The adversary A , with access to some oracle \mathcal{O}_{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* $\text{Adv}_{\text{OW-ATK}}$ of an adversary in this game is the probability that it wins the game:

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

The capabilities of the oracle \mathcal{O}_{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 (**KeyGen**, **Encap**, **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 $(\mathbf{pk}, \mathbf{sk}) \xleftarrow{\$} \text{KeyGen}(1^\lambda)$. **Encap**(\mathbf{pk}) is a probabilistic routine that takes a public key \mathbf{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). **Decap**(\mathbf{sk}, c) is a deterministic routine that takes the secret key \mathbf{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:  $(\mathbf{pk}, \mathbf{sk}) \xleftarrow{\$} \text{KeyGen}(1^\lambda)$ 
2:  $(c^*, K_0) \xleftarrow{\$} \text{Encap}(\mathbf{pk})$ 
3:  $K_1 \xleftarrow{\$} \mathcal{K}$ 
4:  $b \xleftarrow{\$} \{0, 1\}$ 
5:  $\hat{b} \xleftarrow{\$} A^{\mathcal{O}_{\text{Decap}}}(1^\lambda, \mathbf{pk}, c^*, K_b)$ 
6: return  $\llbracket \hat{b} = b \rrbracket$ 
```

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

```

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

Figure 3: The KEM-IND-CCA2 game

Figure 4: Decapsulation oracle

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

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

2.3 Message authentication code (MAC)

A message authentication code MAC is a collection of routines (**Sign**, **Verify**) defined over some key space \mathcal{K} , some message space \mathcal{M} , and some tag space \mathcal{T} . The signing routine **Sign**(k, m) takes the secret key $k \in \mathcal{K}$ and some message, and outputs a tag t . The verification routine **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 $t \leftarrow \text{MAC}(k, m)$ 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 $\mathcal{O}_{\text{Sign}}(m)$, 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:  $(\hat{m}, \hat{t}) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_{\text{sign}}}()$ 
3: return  $\llbracket \text{Verify}(k^*, \hat{m}, \hat{t}) \wedge (\hat{m}, \hat{t}) \notin \mathcal{O}_{\text{sign}} \rrbracket$ 

```

Figure 5: The EUF-CMA game

The advantage $\text{Adv}_{\text{EUF-CMA}}$ 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 $m = (m_1, m_2, \dots, m_l) \in \mathbb{F}^{\leq L}$. To compute the tag:

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

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

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

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:*

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

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 $c \xleftarrow{\$} \text{Enc}(\text{pk}, m)$ is made into a deterministic encryption routine by deriving randomization coin pseudorandomly: $c \leftarrow \text{Enc}(\text{pk}, m, r = H(m))$ 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.

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

141 let $\text{PKE} = (\text{KeyGen}_{\text{PKE}}, \text{Enc}, \text{Dec})$ be defined over message space \mathcal{M} and ciphertext space
 142 \mathcal{C} . Let $G : \mathcal{M} \rightarrow \mathcal{R}$ hash plaintexts into coins, and let $H : \{0, 1\}^* \rightarrow \{0, 1\}^*$ hash byte
 143 stream into session keys. Depending on whether the constructed KEM uses implicit or
 144 explicit rejection, and the security property of the PKE, [HHK17] described four variations.
 145 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 |
| $U^\mathcal{L}$ | implicit | OW-PCA |
| U_m^\perp | explicit | OW-VA + rigid |
| $U_m^\mathcal{L}$ | implicit | OW-CPA + rigid |

Algorithm 6 $\text{KeyGen}()$

```

1:  $(\text{pk}, \text{sk}') \xleftarrow{\$} \text{KeyGen}_{\text{PKE}}()$ 
2:  $z \xleftarrow{\$} \mathcal{M}$ 
3:  $\text{sk} \leftarrow (\text{sk}', z)$   $\triangleright U^\mathcal{L}, U_m^\mathcal{L}$ 
4:  $\text{sk} \leftarrow \text{sk}'$   $\triangleright U^\perp, U_m^\perp$ 
5: return  $(\text{pk}, \text{sk})$ 
```

Algorithm 7 $\text{Encap}(\text{pk})$

```

1:  $m \xleftarrow{\$} \mathcal{M}$ 
2:  $r \leftarrow G(m)$ 
3:  $c \leftarrow \text{Enc}(\text{pk}, m, r)$ 
4:  $K \leftarrow H(m, c)$   $\triangleright U^\perp, U^\mathcal{L}$ 
5:  $K \leftarrow H(m)$   $\triangleright U_m^\perp, U_m^\mathcal{L}$ 
6: return  $(c, K)$ 
```

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

```

1:  $\hat{m} \leftarrow \text{Dec}(\text{sk}', c)$ 
2:  $\hat{r} \leftarrow G(\hat{m})$ 
3:  $\hat{c} \leftarrow \text{Enc}(\text{pk}, \hat{m}, \hat{r})$ 
4: if  $\hat{c} = c$  then
5:    $K \leftarrow H(\hat{m})$   $\triangleright U_m^\perp, U_m^\mathcal{L}$ 
6:    $K \leftarrow H(\hat{m}, c)$   $\triangleright U^\perp, U^\mathcal{L}$ 
7: else
8:    $K \leftarrow H(z, c)$   $\triangleright U^\mathcal{L}, U_m^\mathcal{L}$ 
9:    $K \leftarrow \perp$   $\triangleright U^\perp, U_m^\perp$ 
10: end if
11: return  $K$ 
```

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

146 The modular FO transformations enjoy tight security bounds and proven quantum
 147 resistance. Variations have been deployed to many post-quantum KEMs submitted to
 148 NIST's post-quantum cryptography competition. Kyber, one of the round 3 finalists, uses
 149 the $U^\mathcal{L}$ transformation. When it was later standardized into FIPS-203, it changed to use
 150 the $U_m^\mathcal{L}$ transformation for computational efficiencies.

3 The “encrypt-then-MAC” transformation

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

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

```

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

```

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

```

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

```

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

```

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

```

Figure 7: KEM_{EtM} routines

The key generation routine of KEM_{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 $k \leftarrow G(\text{Dec}(\text{sk}', c'))$ 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 $\text{Dec}(\text{sk}', c')$, 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 $\text{PCO}(m, c)$

```

1:  $k \leftarrow G(m)$ 
2:  $t \leftarrow \text{MAC}(k, c)$ 
3: return  $\llbracket \mathcal{O}^{\text{Decap}}((c, t)) = H(m, c) \rrbracket$ 

```

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 $c' \xleftarrow{\$} \text{Enc}(\text{pk}, m)$ 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 KEM_{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*

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

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

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

```

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:  $k \xleftarrow{\$} \mathcal{K}_{\text{MAC}}$ 
5:  $\mathcal{L}^G \leftarrow \mathcal{L}^G \cup \{(m, k)\}$ 
6: return  $k$ 

```

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

```

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

```

Algorithm 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:  $K \xleftarrow{\$} \mathcal{K}_{\text{KEM}}$ 
5:  $\mathcal{L}^H \leftarrow \mathcal{L}^H \cup \{(m, c, K)\}$ 
6: return  $K$ 

```

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

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

```

1:  $(c', t) \leftarrow c$ 
2: if  $\exists(\tilde{m}, \tilde{k}) \in \mathcal{L}^G : \tilde{m} = \text{Dec}(sk', c') \wedge \text{MAC}(\tilde{k}, c') = t$  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

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

205 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 $k = G(\text{Dec}(\text{sk}', c'))$. Therefore, there could not have been a matching query that also validates the tag, and the simulated oracle must also reject 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 $k = G(\text{Dec}(\text{sk}', c'))$, but k has never been queried. Under the random oracle model, such k is a uniformly random sample of \mathcal{K}_{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 :

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

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

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

Game 2 is identical to game 1, except that on line 4 of algorithm 13, the challenger samples a uniformly random MAC key $k^* \xleftarrow{\$} \mathcal{K}_{\text{MAC}}$ 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 $P[\text{QUERY } G]$, then:

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

Game 3 is identical to game 2, except that on line 7 of algorithm 13, the challenger samples a uniformly random shared secret $K_0 \xleftarrow{\$} \mathcal{K}_{\text{KEM}}$ 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 $P[\text{QUERY } H]$, then:

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

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: $\text{Adv}_{G_3}(A) = 0$.

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

```

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

```

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

```

1:  $(c', t) \leftarrow c$ 
2: if  $\exists(\tilde{m}, \tilde{k}) \in \mathcal{L}^G : \text{PCO}(c', \tilde{m}) = 1 \wedge$   

    $\text{MAC}(k, c') = t$  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  $\text{PCO}(m, c'^*) = 1$  then
2:    $\text{ABORT}(m)$ 
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:  $k \xleftarrow{\$} \mathcal{K}_{\text{MAC}}$ 
8:  $\mathcal{L}^G \leftarrow \mathcal{L}^G \cup \{(m, k)\}$ 
9: return  $k$ 

```

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

```

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

```

We will bound $P[\text{QUERY G}]$ and $P[\text{QUERY H}]$ 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 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 $m^* = \text{Dec}(\text{sk}', c'^*)$ has been queried. When m^* is queried, B terminates A and returns m^* to win the OW-PCA game. In other words:

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

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

Combining all equations above produce the desired security bound. \square

4 Implementation

ML-KEM (based on [BDK⁺18][ABD⁺19]) is an IND-CCA2 secure key encapsulation mechanism standardized by NIST in FIPS 203. ML-KEM achieves IND-CCA2 security first by constructing a public-key encryption scheme whose IND-CPA security is derived from the hardness of the module learning with error (MLWE) problems, then applying the Fujisaki-Okamoto transformation. We extended the reference C implementation of ML-KEM (<https://github.com/pq-crystals/kyber>) to implement our “encrypt-then-MAC” transformation and compared the performance characteristics of our instantiation with ML-KEM.

252 4.1 Choosing a message authenticator

253 4.2 KEM performance

254 4.3 Key exchange performance

255 5 Application to Kyber

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

265 We took the IND-CPA PKE routines (algorithms 4, 5, 6 in [ABD⁺19]) and applied
 266 the “encrypt-then-MAC” transformation. The resulting KEM routines are described in
 267 algorithms 22, 23, and 24.

Algorithm 22 Kyber.CCAKEM.KeyGen()

```

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

```

Algorithm 23 Kyber⁺.CCAEM.Encap⁺(pk)

```

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

```

Algorithm 24 $\text{Kyber.CCAKEM.Decap}^+(\text{sk}, c)$ **Require:** Secret key $\text{sk} = (\text{sk}', \text{pk}, H(\text{pk}), z)$ **Require:** Ciphertext $c = (c', t)$

```

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

```

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

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

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

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

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

Table 2: 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 computational 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 experiments 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 $(\mathbf{pk}_e, \mathbf{sk}_e) \xleftarrow{\$} \text{KeyGen}()$ is generated by the client. The client transmits the ephemeral public key \mathbf{pk}_e to the server, who runs the encapsulation routine $(c_e, K_e) \xleftarrow{\$} \text{Encap}(\mathbf{pk}_e)$ and transmits the ciphertext c_e back to the client. The client finally decapsulates the ciphertext to recover the shared secret $K_e \leftarrow \text{Decap}(\mathbf{sk}_e, c_e)$. The specific steps are described in algorithms 25, 26.

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 25 KE_c

```

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

```

Figure 10: KE Client

Algorithm 26 KE_s

```

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

```

Figure 11: KE Server

The RTT comparison is summarized in table 3

Table 3: 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 pk_S of the server, and the server authenticates itself by demonstrating possession of the corresponding long-term secret key sk_S . Details are described in algorithms 27 and 28.

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 27 $\text{UAKE}_c(\text{pk}_S)$

Require: Server's long-term pk_S

- 1: $(\text{pk}_e, \text{sk}_e) \xleftarrow{\$} \text{KeyGen}()$
 - 2: $(c_S, K_S) \xleftarrow{\$} \text{Encap}(\text{pk}_S)$
 - 3: $\text{send}(\text{pk}_e, c_S)$
 - 4: $c_e \leftarrow \text{read}()$
 - 5: $K_e \leftarrow \text{Decap}(\text{sk}_e, c_e)$
 - 6: $K \leftarrow \text{KDF}(K_e \| K_S)$
 - 7: **return** K
-

Algorithm 28 $\text{UAKE}_s(\text{sk}_S)$

Require: Server's long-term sk_S

- 1: $(\text{pk}_e, c_S) \leftarrow \text{read}()$
 - 2: $K_S \leftarrow \text{Decap}(\text{sk}_S, c_S)$
 - 3: $(c_e, K_e) \xleftarrow{\$} \text{Encap}(\text{pk}_e)$
 - 4: $\text{send}(c_e)$
 - 5: $K \leftarrow \text{KDF}(K_e \| K_S)$
 - 6: **return** K
-

Figure 12: UAKE Client

Figure 13: UAKE Server

Table 4: 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 29 $\text{AKE}_C(\text{pk}_S, \text{sk}_C)$ **Require:** Server’s long-term pk_S **Require:** Client’s long-term sk_C

- 1: $(\text{pk}_e, \text{sk}_e) \xleftarrow{\$} \text{KeyGen}()$
- 2: $(c_S, K_S) \xleftarrow{\$} \text{Encap}(\text{pk}_S)$
- 3: **send** (pk_e, c_S)
- 4: $(c_e, c_C) \leftarrow \text{read}()$
- 5: $K_e \leftarrow \text{Decap}(\text{sk}_e, c_e)$
- 6: $K_C \leftarrow \text{Decap}(\text{sk}_e, c_C)$
- 7: $K \leftarrow \text{KDF}(K_e \| K_S \| K_C)$
- 8: **return** K

Algorithm 30 $\text{AKE}_S(\text{sk}_S, \text{pk}_C)$ **Require:** Server’s long-term sk_S **Require:** Client’s long-term pk_C

- 1: $(\text{pk}_e, c_S) \leftarrow \text{read}()$
- 2: $K_S \leftarrow \text{Decap}(\text{sk}_S, c_S)$
- 3: $(c_e, K_e) \xleftarrow{\$} \text{Encap}(\text{pk}_e)$
- 4: $(c_C, K_C) \xleftarrow{\$} \text{Encap}(\text{pk}_C)$
- 5: **send** (c_e, c_C)
- 6: $K \leftarrow \text{KDF}(K_e \| K_S \| K_C)$
- 7: **return** K

Figure 14: AKE Client**Figure 15:** AKE Server**Table 5:** 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 clang 14.0.0. Table 6 reports the cycle counts for three different instances of Keyber, namely . The cycle counts reported in Table 6 are the average of the cycle counts of 10000 executions of all algorithms.

- 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

Table 6: 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⁺: | |

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 de-randomization 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 $m = \text{Dec}(\text{sk}, c)$ is equivalent to $c = \text{Enc}(\text{pk}, m)$, so a plaintext-checking oracle can be simulated without any secret information. However, the U_m^\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, Tancrede 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*

- 396 *2016 ACM SIGSAC conference on computer and communications security*,
397 pages 1006–1018, 2016.
- 398 [BDK⁺18] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky,
399 John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-
400 kyber: a cca-secure module-lattice-based kem. In *2018 IEEE European*
401 *Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE, 2018.
- 402 [BN00] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Rela-
403 tions among notions and analysis of the generic composition paradigm. In
404 *International Conference on the Theory and Application of Cryptology and*
405 *Information Security*, pages 531–545. Springer, 2000.
- 406 [BP18] Daniel J Bernstein and Edoardo Persichetti. Towards kem unification. *Cryp-*
407 *tology ePrint Archive*, 2018.
- 408 [BS20] Dan Boneh and Victor Shoup. A graduate course in applied cryptography.
409 *Draft 0.5*, 2020.
- 410 [DKSRV18] Jan-Pieter D’Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik
411 Vercauteren. Saber: Module-lwr based key exchange, cpa-secure encryption
412 and cca-secure kem. In *Progress in Cryptology–AFRICACRYPT 2018: 10th*
413 *International Conference on Cryptology in Africa, Marrakesh, Morocco, May*
414 *7–9, 2018, Proceedings 10*, pages 282–305. Springer, 2018.
- 415 [FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and
416 symmetric encryption schemes. In *Annual international cryptology conference*,
417 pages 537–554. Springer, 1999.
- 418 [HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of
419 the fujisaki-okamoto transformation. In *Theory of Cryptography Conference*,
420 pages 341–371. Springer, 2017.
- 421 [LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and
422 learning with errors over rings. *J. ACM*, 60(6):43:1–43:35, 2013.
- 423 [RRCB19] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin.
424 Generic side-channel attacks on cca-secure lattice-based pke and kem schemes.
425 *Cryptology ePrint Archive*, 2019.
- 426 [Sho04] Victor Shoup. Sequences of games: a tool for taming complexity in security
427 proofs. *cryptology eprint archive*, 2004.
- 428 [UXT⁺22] Rei Ueno, Keita Xagawa, Yutaro Tanaka, Akira Ito, Junko Takahashi, and
429 Naofumi Homma. Curse of re-encryption: a generic power/em analysis on
430 post-quantum kems. *IACR Transactions on Cryptographic Hardware and*
431 *Embedded Systems*, pages 296–322, 2022.