

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][HHM22] 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] (later standardized into ML-KEM [Nat24]), FrodoKEM [BCD⁺16], and SABER [DKRV18].

The current state of the FO transformation enjoys proven tight security bound and quantum resistance, but also leaves many open problems. One such problem is the substantial computational cost of using *de-randomization* and *re-encryption* [BP18] for providing ciphertext integrity. In many post-quantum schemes, including ML-KEM, the input encryption routine is substantially more expensive than the input decryption routine, so running the encryption as a subroutine substantially increases the runtime cost of the decapsulation routine. While not the focus of this project, *re-encryption* also introduces risks of side-channel vulnerabilities that may expose the plaintext or the secret key, as demonstrated in [RRCB19] and [UXT⁺22].

We are inspired by how ciphertext integrity is achieved in symmetric cryptography: given a semantically secure symmetric cipher and an existentially unforgeable message authentication code, combining them using in a pattern called “encrypt-then-MAC” provides proven authenticated encryption [BN00]. “encrypt-then-MAC” is now the most widely accepted method for doing symmetric encryption as AES-GCM [MV04] and ChaCha20-Poly1305 [NL15].

The main challenge in applying “encrypt-then-MAC” to public-key cryptography is the lack of a pre-shared symmetric key. Inspired by hybrid public-key encryption schemes such as [FO99] and [CS98], 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 4, we instantiate concrete instances of our constructions using ML-KEM. 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

Syntax. A public-key encryption scheme $\text{PKE}(\text{KeyGen}, \text{Enc}, \text{Dec})$ is a collection of three routines defined over some plaintext space \mathcal{M} and some ciphertext space \mathcal{C} . $(\text{pk}, \text{sk}) \xleftarrow{\$} \text{KeyGen}()$ is a randomized routine that returns a keypair. The encryption routine $\text{Enc} : (\text{pk}, m) \mapsto c$ encrypts the input plaintext under the input public key. The decryption routine $\text{Dec} : (\text{sk}, c) \mapsto m$ decrypts the input ciphertext under the input secret key. Where the encryption routine is randomized, we denote the randomness by $r \in \mathcal{R}$, where \mathcal{R} is called the coin space. The decryption routine is assumed to always be deterministic. Some decryption routines can detect malformed ciphertext and output the rejection symbol \perp accordingly.

Correctness. We usually demand that the decryption routine always correctly recovers the plaintext from an honest encryption, meaning that for all possible keypairs (pk, sk) and plaintext $m \in \mathcal{M}$.

$$P \left[\text{Dec}(\text{sk}, c) = m \mid c \xleftarrow{\$} \text{Enc}(\text{pk}, m) \right] = 1$$

However, many lattice cryptosystems do not achieve perfect correctness. [HHK17] defines δ -correctness to mean:

$$E \left[\max_{m \in \mathcal{M}} P \left[\text{Dec}(\text{sk}, c) \neq m \mid c \xleftarrow{\$} \text{Enc}(\text{pk}, m) \right] \right] \leq \delta$$

Where the expectation is taken over the probability distribution of possible keypairs. Decryption failures can leak information about the secret key and need to be delicately handled, though the concrete probability is small enough that we will ignore them in this paper (ML-KEM-512’s probability of decryption failure is estimated to be 2^{-80})

Security. 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.

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: $\hat{m} \xleftarrow{\$} \mathcal{A}^{\text{OW-ATK}}(1^\lambda, \text{pk}, c^*)$
- 5: **return** $\llbracket m^* = \hat{m} \rrbracket$

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

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

Figure 1: One-way security game of PKE (left) and plaintext-checking oracle (right)

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 1). 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 $(\text{pk}, \text{sk}) \xleftarrow{\$} \text{KeyGen}(1^\lambda)$. **Encap**(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). **Decap**(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.

KEM-IND-CCA2 game

1: $(\text{pk}, \text{sk}) \xleftarrow{\$} \text{KeyGen}(1^\lambda)$	
2: $(c^*, K_0) \xleftarrow{\$} \text{Encap}(\text{pk})$	
3: $K_1 \xleftarrow{\$} \mathcal{K}$	
4: $b \xleftarrow{\$} \{0, 1\}$	
5: $\hat{b} \xleftarrow{\$} A^{\mathcal{O}_{\text{Decap}}}(1^\lambda, \text{pk}, c^*, K_b)$	$\mathcal{O}_{\text{Decap}}(c)$
6: return $\llbracket \hat{b} = b \rrbracket$	1: return $\text{Decap}(\text{sk}, c)$

Figure 2: IND-CCA2 game for KEM (left) and decapsulation oracle (right)

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

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

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:

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 3: The existential forgery game

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

2.4 Related works

The modular Fujisaki-Okamoto transformation [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 exactly 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: $c = \text{Enc}(\text{pk}, m) \Leftrightarrow m = \text{Dec}(\text{sk}, c)$. Converting a one-way secure rigid PKE (which is essentially a trapdoor function) into a IND-CCA2 KEM is well solved problem.

let $\text{PKE} = (\text{KeyGen}_{\text{PKE}}, \text{Enc}, \text{Dec})$ be defined over message space \mathcal{M} and ciphertext space \mathcal{C} . Let $G : \mathcal{M} \rightarrow \mathcal{R}$ hash plaintexts into coins, and let $H : \{0, 1\}^* \rightarrow \{0, 1\}^*$ 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 4.

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

KEM.KeyGen()	KEM.Decap(sk, c)
1: $(pk, sk') \xleftarrow{\$} \text{PKE.KeyGen}()$ 2: $z \xleftarrow{\$} \mathcal{M}$ 3: $sk \leftarrow (sk', z)$ 4: $sk \leftarrow sk'$ 5: return (pk, sk)	Ensure: $sk = (sk', z)$ Ensure: sk' is valid PKE secret key Ensure: z is valid PKE plaintext 1: $\hat{m} \leftarrow \text{PKE.Dec}(sk', c)$ 2: $\hat{r} \leftarrow G(\hat{m})$ 3: $\hat{c} \leftarrow \text{PKE.Enc}(pk, \hat{m}, \hat{r})$ 4: if $\hat{c} = c$ then 5: $K \leftarrow H(\hat{m})$ 6: $K \leftarrow H(\hat{m}, c)$ 7: else 8: $K \leftarrow H(z, c)$ 9: $K \leftarrow \perp$ 10: end if 11: return K
KEM.Encap(pk)	
1: $m \xleftarrow{\$} \mathcal{M}$ 2: $r \leftarrow G(m)$ 3: $c \leftarrow \text{PKE.Enc}(pk, m, r)$ 4: $K \leftarrow H(m, c)$ 5: $K \leftarrow H(m)$ 6: return (c, K)	

Figure 4: Summary of the modular Fujisaki-Okamoto transformation variations

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

146 3 The “encrypt-then-MAC” transformation

147 Let \mathcal{B}^* denote the set of finite bit strings. Let $\text{PKE}(\text{KeyGen}, \text{Enc}, \text{Dec})$ be a public-key
148 encryption scheme defined over message space \mathcal{M} and ciphertext space \mathcal{C} . Let $\text{MAC} : \mathcal{K}_{\text{MAC}} \times \mathcal{B}^* \rightarrow \mathcal{T}$
149 be a deterministic message authentication code that takes a key $k \in \mathcal{K}_{\text{MAC}}$,
150 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
151 function that maps from PKE’s plaintext space to MAC’s key space. Let $H : \mathcal{B}^* \rightarrow \mathcal{K}_{\text{KEM}}$
152 be a hash function that maps bit strings into the set of possible shared secrets. The
153 “encrypt-then-MAC” transformation $\text{EtM}[\text{PKE}, \text{MAC}, G, H]$ constructs a key encapsulation
154 mechanism $\text{KEM}_{\text{EtM}}(\text{KeyGen}_{\text{KEM}}, \text{Encap}, \text{Decap})$, whose routines are described in figure 5.

$\text{KEM}_{\text{EtM}}.\text{KeyGen}()$	$\text{KEM}_{\text{EtM}}.\text{Decap}(\text{sk}, c)$
1: $(\text{pk}, \text{sk}') \xleftarrow{\$} \text{PKE}.\text{KeyGen}()$ 2: $z \xleftarrow{\$} \mathcal{M}$ 3: $\text{sk} \leftarrow \text{sk}' \ z$ 4: return (pk, sk)	Require: $c = c' \ t, \text{sk} = \text{sk}' \ z$ Ensure: c' is some PKE ciphertext Ensure: t is some MAC tag Ensure: sk' is some PKE secret key Ensure: z is some PKE plaintext 1: $(c', t) \leftarrow c$ 2: $(\text{sk}', z) \leftarrow \text{sk}$ 3: $\hat{m} \leftarrow \text{PKE}.\text{Dec}(\text{sk}', c')$ 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
$\text{KEM}_{\text{EtM}}.\text{Encap}(\text{pk})$	
Ensure: pk is some PKE public key 1: $m \xleftarrow{\$} \mathcal{M}$ 2: $k \leftarrow G(m)$ 3: $c' \xleftarrow{\$} \text{PKE}.\text{Enc}(\text{pk}, m)$ 4: $t \leftarrow \text{MAC}(k, c')$ 5: $K \leftarrow H(m, c')$ 6: $c \leftarrow c' \ t$ 7: return (c, K)	

Figure 5: KEM_{EtM} routines

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

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

$\text{PCO}(m, c)$
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 6: Every decapsulation oracle can be converted into a plaintext-checking oracle

172 However, a decapsulation oracle can still give out some information: for a known
173 plaintext m , all possible encryptions $c' \xleftarrow{\$} \text{Enc}(\text{pk}, m)$ can be correctly signed, while
174 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 (see figure 6), 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 1. *KEM_{EtM} is IND-CCA2 secure if and only if the input PKE is OW-PCA secure*

3.1 Proof of theorem 1

We will prove theorem 1 using a sequence of game. A summary of the the sequence of games can be found in figure 7 and 8. From a high level we made three incremental modifications to the IND-CCA2 game for KEM_{EtM} : replace true decapsulation with simulated decapsulation, replace the pseudorandom MAC key $k^* \leftarrow G(m^*)$ with a truly random MAC key $k^* \xleftarrow{\$} \mathcal{K}_{\text{MAC}}$, and finally replace pseudorandom shared secret $K_0 \leftarrow H(m^*, c')$ with a truly random shared secret $K_0 \xleftarrow{\$} \mathcal{K}_{\text{KEM}}$. A OW-PCA adversary can then simulate the modified IND-CCA2 game for the KEM adversary, and the advantage of the OW-PCA adversary is associated with the probability of certain behaviors of the KEM adversary.

Proof. *Game 0* is the standard IND-CCA2 game for KEMs. The decapsulation oracle $\mathcal{O}^{\text{Decap}}$ 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 $\mathcal{L}^G, \mathcal{L}^H$.

Game 1 is identical to game 0 except that the true decapsulation oracle $\mathcal{O}^{\text{Decap}}$ is replaced with a simulated oracle $\mathcal{O}_1^{\text{Decap}}$. 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 $k = G(\text{Dec}(\text{sk}', c'))$. Therefore, there could not have been a matching query that also validates the tag, and the simulated oracle must also 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 $k = G(\text{Dec}(\text{sk}', c'))$, 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

IND-CCA2 game for KEM_{EtM}	$\mathcal{O}^{\text{Decap}}(c)$
1: $(\text{pk}, \text{sk}) \xleftarrow{\$} \text{KEM}_{\text{EtM}}.\text{KeyGen}()$ 2: $m^* \xleftarrow{\$} \mathcal{M}$ 3: $c' \xleftarrow{\$} \text{PKE}.\text{Enc}(\text{pk}, m^*)$ 4: $k^* \leftarrow G(m^*)$ \triangleright Game 0-1 5: $k^* \xleftarrow{\$} \mathcal{K}_{\text{MAC}}$ \triangleright Game 2-3 6: $t \leftarrow \text{MAC}(k^*, c')$ 7: $c^* \leftarrow c' t$ 8: $K_0 \leftarrow H(m^*, c')$ \triangleright Game 0-2 9: $K_0 \xleftarrow{\$} \mathcal{K}_{\text{KEM}}$ \triangleright Game 3 10: $K_1 \xleftarrow{\$} \mathcal{K}_{\text{KEM}}$ 11: $b \xleftarrow{\$} \{0, 1\}$ 12: $\hat{b} \leftarrow A^{\mathcal{O}^{\text{Decap}}}(\text{pk}, c^*, K_b)$ \triangleright Game 0 13: $\hat{b} \leftarrow A^{\mathcal{O}_1^{\text{Decap}}}(\text{pk}, c^*, K_b)$ \triangleright Game 1-3 14: return $\llbracket \hat{b} = b \rrbracket$	1: $(c', t) \leftarrow c$ 2: $\hat{m} = \text{Dec}(\text{sk}', c')$ 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
$\mathcal{O}^G(m)$	$\mathcal{O}_1^{\text{Decap}}(c)$
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	1: $(c', t) \leftarrow c$ 2: if $\exists(\tilde{m}, \tilde{k}) \in \mathcal{L}^G : \tilde{m} = \text{Dec}(\text{sk}', c') \wedge \text{MAC}(\tilde{k}, c') = t$ then 3: $K \leftarrow H(\tilde{m}, c')$ 4: else 5: $K \leftarrow H(z, c')$ 6: end if 7: return K
$\mathcal{O}^H(m)$	$\mathcal{O}^H(m, c)$
	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

Figure 7: Sequence of games

221 security definition for a MAC, we can bound it by the advantage of a one-time forgery
222 adversary C :

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

223 Across all q decapsulation queries, the probability that at least one query is a forgery
224 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:

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

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

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

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

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

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

$B(\text{pk}, c'^*)$	$\mathcal{O}_B^{\text{Decap}}(c)$
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	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
$\mathcal{O}_B^H(m, c)$	$\mathcal{O}_B^G(m)$
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	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

Figure 8: OW-PCA adversary B simulates game 3 for IND-CCA2 adversary A

235 We will bound $P[\text{QUERY } G]$ and $P[\text{QUERY } H]$ by constructing a OW-PCA adversary B
 236 against the underlying PKE that uses A as a sub-routine. B 's behaviors are summarized
 237 in figure 8.

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

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

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

4 Implementation

ML-KEM is an IND-CCA2 secure key encapsulation mechanism standardized by NIST in FIPS 203. The IND-CCA2 security of ML-KEM is achieved in two steps. First, ML-KEM constructs an IND-CPA secure public key encryption scheme K-PKE(KeyGen, Enc, 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^L variant of the Fujisaki-Okamoto transformation [HHK17] is used to construct the KEM MLKEM(KeyGen, Encap, Decap) by calling K-PKE(KeyGen, Enc, 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 incurs significant computational cost.

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. See appendix 6.1 for comparison with Kyber’s third round submission to NIST’s PQC competition.

A detailed description of K-PKE’s routines can be found in FIPS 203 (TODO: citation). The “encrypt-then-MAC” routines are described in figure 9.

ML-KEM ⁺ .KeyGen()	ML-KEM ⁺ .Decap(sk, c)
<ol style="list-style-type: none"> 1: $z \xleftarrow{\\$} \{0, 1\}^{256}$ 2: $(pk, sk') \xleftarrow{\\$} \text{K-PKE.KeyGen}()$ 3: $h \leftarrow H(pk)$ 4: $sk \leftarrow (sk' pk h z)$ 5: return (pk, sk) 	<p>Require: Secret key $sk = (sk' pk h z)$</p> <p>Require: Ciphertext $c = (c' t)$</p> <ol style="list-style-type: none"> 1: $(sk', pk, h, z) \leftarrow sk$ 2: $(c', t) \leftarrow c$ 3: $\hat{m} \leftarrow \text{K-PKE.Dec}(sk', c')$ 4: $(\bar{K}, \hat{r}, \hat{k}) \leftarrow \text{XOF}(\hat{m} h)$ 5: $\hat{t} \leftarrow \text{MAC}(\hat{k}, c')$ 6: if $\hat{t} = t$ then 7: $K \leftarrow \text{KDF}(\bar{K} t)$ 8: else 9: $K \leftarrow \text{KDF}(z t)$ 10: end if 11: return K
ML-KEM ⁺ .Encap(pk)	
<p>Require: Public key pk</p> <ol style="list-style-type: none"> 1: $m \xleftarrow{\\$} \{0, 1\}^{256}$ 2: $(\bar{K}, r, k) \leftarrow \text{XOF}(m H(pk))$ 3: $c' \leftarrow \text{K-PKE.Enc}(pk, m, r)$ 4: $t \leftarrow \text{MAC}(k, c')$ 5: $K \leftarrow \text{KDF}(\bar{K} c')$ 6: $c \leftarrow (c', t)$ 7: return (c, K) 	

Figure 9: ML-KEM⁺ routines

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

For the ML-KEM⁺ implementation, we instantiated MAC with a selection that covered a wide range of MAC designs, including Poly1305 [Ber05], GMAC [MV04], CMAC [IK03][BR05], and KMAC [KCP16].

Poly1305 and GMAC are both Carter-Wegman style authenticators that compute the tag using finite field arithmetic. Generically speaking, Carter-Wegman MAC is parameterized by some finite field \mathbb{F} and the maximal message length $L > 0$. Each symmetric key $k = (k_1, k_2) \xleftarrow{\$} \mathbb{F}^2$ is a pair of uniformly random field elements, and the message is parsed into tuples of field elements up to length L : $m = (m_1, m_2, \dots, m_l) \in \mathbb{F}^{\leq L}$. The tag t is computed by evaluating a polynomial whose coefficients are the message blocks and whose indeterminate is the key:

$$\text{MAC}((k_1, k_2), m) = H_{\text{xpoly}}(k_1, m) + k_2 \quad (1)$$

Where H_{xpoly} is given by:

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

The authenticator formulated in equation 1 is a one-time MAC. To make the construction many-time secure, a non-repeating nonce r and a PRF is needed:

$$\text{MAC}((k_1, k_2), m, r) = H_{\text{xpoly}}(k_1, m) \oplus \text{PRF}(k_2, r)$$

Specifically, Poly1305 operates in the prime field \mathbb{F}_q where $q = 2^{130} - 5$ whereas GMAC operates in the binary field $\mathbb{F}_{2^{128}}$. In OpenSSL's implementation, standalone Poly1305 is a one-time secure MAC, whereas GMAC uses a nonce and AES as the PRF and is thus many-time secure (in OpenSSL GMAC is AES-256-GCM except all data is fed into the "associated data" section and thus not encrypted).

CMAC is based on the CBC-MAC with the block cipher instantiated from AES-256. To compute a CMAC tag, the message is first broke into 128-bit blocks with appropriate padding. Each block is first XOR'd with the previous block's output, then encrypted under AES using the symmetric key. The final output is XOR'd with a sub key derived from the symmetric key, before being encrypted for one last time. A summary of the computation can be found in figure 10

KMAC is defined in NIST SP 800-185 to be based on the family of sponge functions with Keccak permutation as the underlying function. We chose KMAC-256, which uses Shake256 as the underlying extendable output functions. KMAC allows variable-length key and tag, but we chose the 256 bits for key length and 128 bits for tag size for consistency with other authenticators.

To isolate the performance characteristics of each authenticator in our instantiation of ML-KEM⁺, we measured the CPU cycles needed for each authenticator to compute a tag on random inputs whose sizes correspond to the ciphertext sizes of ML-KEM. The measurements are summarized in table 2.

Table 2: CPU cycles needed to compute tag on various input sizes

Input size: 768 bytes			Input size: 1088 bytes			Input size: 1568 bytes		
MAC	Median	Average	MAC	Median	Average	MAC	Median	Average
Poly1305	909	2823	Poly1305	961	2704	Poly1305	1065	1809
GMAC	3899	4859	GMAC	3899	4827	GMAC	4055	5026
CMAC	6291	6373	CMAC	7305	7588	CMAC	8735	8772
KMAC	6373	7791	KMAC	9697	9928	KMAC	11647	12186

Sub-key derivation	CMAC(k, m)
Require: 256-bit key k Require: $\text{const_Rb} = 0x87$ 1: $l \leftarrow \text{AES-256}(k, 0^{128})$ 2: if $\text{MostSignificantBit}(l) = 0$ then 3: $k_1 \leftarrow l \ll 1$ 4: else 5: $k_1 \leftarrow l \ll 1 \oplus \text{const_Rb}$ 6: end if 7: if $\text{MostSignificantBit}(k_1) = 0$ then 8: $k_2 \leftarrow k_1 \ll 1$ 9: else 10: $k_2 \leftarrow k_1 \ll 1 \oplus \text{const_Rb}$ 11: end if 12: return k_1, k_2	Require: 256-bit symmetric key k 1: $(k_1, k_2) \leftarrow \text{deriveSubKey}(k)$ 2: $n \leftarrow \lceil \text{bytesLen}(m)/16 \rceil$ 3: if $n = 0$ then 4: $n \leftarrow 1$ 5: $m_{\text{last}} \leftarrow m_n \oplus k_2$ 6: else if $\text{bytesLen}(m) \bmod 16 = 0$ then 7: $m_{\text{last}} \leftarrow m_n \oplus k_1$ 8: else 9: $m_{\text{last}} \leftarrow m_n \oplus k_2$ 10: end if 11: $x = 0^{128}$ 12: for $i \in \{1, 2, \dots, n-1\}$ do 13: $y \leftarrow x \oplus m_i$ 14: $x \leftarrow \text{AES-256}(k, y)$ 15: end for 16: $y \leftarrow m_{\text{last}} \oplus x$ 17: $t \leftarrow \text{AES-256}(k, y)$ 18: return t

Figure 10: AES-256 CMAC

4.2 KEM performance

Compared to the $U_m^{\mathcal{K}}$ variant of Fujisaki-Okamoto transformed used in ML-KEM, the “encrypt-then-MAC” transformation the following trade-off when given the same input sub-routines:

- Both encapsulation and decapsulation add a small amount of overhead for needing to hash both the PKE plaintext and the PKE ciphertext when deriving the shared secret, where as the $U_m^{\mathcal{K}}$ transformation only needs to hash the PKE plaintext.
- The encapsulation routine adds a small amount of run-time overhead for computing the authenticator
- The decapsulation routine enjoys substantial runtime speedup because *re-encryption* is replaced with computing an authenticator
- Ciphertext size increases by the size of an authenticator

Since K-PKE.Enc carries significantly more computational complexity than K-PKE.Dec or any MAC we chose, the performance advantage of the “encrypt-then-MAC” transformation over the $U_m^{\mathcal{K}}$ transformation is dominated by the runtime saving gained from replacing *re-encryption* with MAC. A comparison between ML-KEM and variations of the ML-KEM-ETM can be found in table 3

4.3 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

Table 3: CPU cycles of each KEM routine

128-bit security		KEM variant	Encap cycles/tick		Decap cycles/tick	
size parameters (bytes)			Median	Average	Median	Average
pk size	800	ML-KEM-512	91467	92065	121185	121650
sk size	1632	Kyber512	97811	98090	119937	120299
ct size	768	ML-KEM-512 ⁺ w/ Poly1305	93157	93626	33733	33908
KeyGen cycles/tick		ML-KEM-512 ⁺ w/ GMAC	97369	97766	37725	37831
Median	75945	ML-KEM-512 ⁺ w/ CMAC	99739	99959	40117	39943
Average	76171	ML-KEM-512 ⁺ w/ KMAC	101009	101313	40741	40916

192-bit security		KEM variant	Encap cycles/tick		Decap cycles/tick	
size parameters (bytes)			Median	Average	Median	Average
pk size	1184	ML-KEM-768	136405	147400	186445	187529
sk size	2400	Kyber768	153061	153670	182129	182755
ct size	1088	ML-KEM-768 ⁺ w/ Poly1305	146405	146860	43315	43463
KeyGen cycles/tick		ML-KEM-768 ⁺ w/ GMAC	149525	150128	46513	46706
Median	129895	ML-KEM-768 ⁺ w/ CMAC	153139	153735	49841	50074
Average	130650	ML-KEM-768 ⁺ w/ KMAC	155219	155848	52415	52611

256-bit security		KEM variant	Encap cycles/tick		Decap cycles/tick	
size parameters (bytes)			Median	Average	Median	Average
pk size	1568	ML-KEM-1024	199185	199903	246245	247320
sk size	3168	Kyber1024	222351	223260	258231	259067
ct size	1568	ML-KEM-1024 ⁺ w/ Poly1305	205763	206499	51375	51562
KeyGen cycles/tick		ML-KEM-1024 ⁺ w/ GMAC	208805	209681	54573	54780
Median	194921	ML-KEM-1024 ⁺ w/ CMAC	213667	214483	59175	59408
Average	195465	ML-KEM-1024 ⁺ w/ KMAC	216761	217468	62269	62516

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.

4.3.1 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 key exchange routines are summarized in figure 11.

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.

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

Figure 11: Unauthenticated key exchange (KE) routines

348

The RTT comparison is summarized in table 4

Table 4: KE RTT comparison

KEM variant	Client TX bytes	Server TX bytes	RTT time (μs)	
			Median	Average
ML-KEM-512	800	768	92	97
ML-KEM-512 ⁺ w/ Poly1305	800	784	70	72
ML-KEM-512 ⁺ w/ GMAC	800	784	73	76
ML-KEM-512 ⁺ w/ CMAC	800	784	75	79
ML-KEM-512 ⁺ w/ KMAC	800	784	76	78

KEM variant	Client TX bytes	Server TX bytes	RTT time (μs)	
			Median	Average
ML-KEM-768	1184	1088	135	140
ML-KEM-768 ⁺ w/ Poly1305	1184	1104	99	104
ML-KEM-768 ⁺ w/ GMAC	1184	1104	101	105
ML-KEM-768 ⁺ w/ CMAC	1184	1104	103	109
ML-KEM-768 ⁺ w/ KMAC	1184	1104	103	107

KEM variant	Client TX bytes	Server TX bytes	RTT time (μs)	
			Median	Average
ML-KEM-1024	1568	1568	193	199
ML-KEM-1024 ⁺ w/ Poly1305	1568	1584	138	141
ML-KEM-1024 ⁺ w/ GMAC	1568	1584	140	145
ML-KEM-1024 ⁺ w/ CMAC	1568	1584	143	148
ML-KEM-1024 ⁺ w/ KMAC	1568	1584	144	149

349

4.3.2 Unilaterally authenticated key exchange (UAKE)

350

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 . UAKE routines are summarized in figure 12.

355

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).

360

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

362

363 footprint, and smaller communication sizes. KEMTLS was proposed in 2020 as an
 364 alternative to existing TLS handshake protocols, and many experimental implementations
 365 have demonstrated the performance advantage. (citation needed).

UAKE _c (pk _S)	UAKE _s (sk _S)
Require: Server's long-term pk _S	Require: Server's long-term sk _S
1: (pk _e , sk _e) $\xleftarrow{\$}$ KeyGen()	1: (pk _e , c _S) \leftarrow read()
2: (c _S , K _S) $\xleftarrow{\$}$ Encap(pk _S)	2: K _S \leftarrow Decap(sk _S , c _S)
3: send(pk _e , c _S)	3: (c _e , K _e) $\xleftarrow{\$}$ Encap(pk _e)
4: c _e \leftarrow read()	4: send(c _e)
5: K _e \leftarrow Decap(sk _e , c _e)	5: K \leftarrow KDF(K _e K _S)
6: K \leftarrow KDF(K _e K _S)	6: return K
7: return K	

Figure 12: Unilaterally authenticated key exchange (UAKE) routines

Table 5: UAKE RTT comparison

KEM variant	Client TX bytes	Server TX bytes	RTT time (μ s)	
			Median	Average
ML-KEM-512	1568	768	145	151
ML-KEM-512 ⁺ w/ Poly1305	1584	784	103	106
ML-KEM-512 ⁺ w/ GMAC	1584	784	106	110
ML-KEM-512 ⁺ w/ CMAC	1584	784	108	112
ML-KEM-512 ⁺ w/ KMAC	1584	784	109	113

KEM variant	Client TX bytes	Server TX bytes	RTT time (μ s)	
			Median	Average
ML-KEM-768	2272	1088	215	222
ML-KEM-768 ⁺ w/ Poly1305	2288	1104	144	150
ML-KEM-768 ⁺ w/ GMAC	2288	1104	149	156
ML-KEM-768 ⁺ w/ CMAC	2288	1104	153	160
ML-KEM-768 ⁺ w/ KMAC	2288	1104	154	159

KEM variant	Client TX bytes	Server TX bytes	RTT time (μ s)	
			Median	Average
ML-KEM-1024	3136	1568	310	318
ML-KEM-1024 ⁺ w/ Poly1305	3152	1584	202	209
ML-KEM-1024 ⁺ w/ GMAC	3152	1584	212	228
ML-KEM-1024 ⁺ w/ CMAC	3152	1584	212	218
ML-KEM-1024 ⁺ w/ KMAC	3152	1584	213	220

366 4.3.3 Mutually authenticated key exchange (AKE)

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

$\text{AKE}_C(\text{pk}_S, \text{sk}_C)$	$\text{AKE}_S(\text{sk}_S, \text{pk}_C)$
Require: Server’s long-term pk_S	Require: Server’s long-term sk_S
Require: Client’s long-term sk_C	Require: Client’s long-term pk_C
1: $(\text{pk}_e, \text{sk}_e) \xleftarrow{\$} \text{KeyGen}()$	1: $(\text{pk}_e, c_S) \leftarrow \text{read}()$
2: $(c_S, K_S) \xleftarrow{\$} \text{Encap}(\text{pk}_S)$	2: $K_S \leftarrow \text{Decap}(\text{sk}_S, c_S)$
3: send (pk_e, c_S)	3: $(c_e, K_e) \xleftarrow{\$} \text{Encap}(\text{pk}_e)$
4: $(c_e, c_C) \leftarrow \text{read}()$	4: $(c_C, K_C) \xleftarrow{\$} \text{Encap}(\text{pk}_C)$
5: $K_e \leftarrow \text{Decap}(\text{sk}_e, c_e)$	5: send (c_e, c_C)
6: $K_C \leftarrow \text{Decap}(\text{sk}_e, c_C)$	6: $K \leftarrow \text{KDF}(K_e \ K_S \ K_C)$
7: $K \leftarrow \text{KDF}(K_e \ K_S \ K_C)$	7: return K
8: return K	

Figure 13: Mutually authenticated key exchange (AKE) routines

Table 6: AKE RTT comparison

KEM variant	Client TX bytes	Server TX bytes	RTT time (μs)	
			Median	Average
ML-KEM-512	1568	1536	220	213
ML-KEM-512 ⁺ w/ Poly1305	1584	1568	133	138
ML-KEM-512 ⁺ w/ GMAC	1584	1568	139	143
ML-KEM-512 ⁺ w/ CMAC	1584	1568	143	148
ML-KEM-512 ⁺ w/ KMAC	1584	1568	145	151

KEM variant	Client TX bytes	Server TX bytes	RTT time (μs)	
			Median	Average
ML-KEM-768	2272	2176	294	301
ML-KEM-768 ⁺ w/ Poly1305	2288	2208	190	196
ML-KEM-768 ⁺ w/ GMAC	2288	2208	197	210
ML-KEM-768 ⁺ w/ CMAC	2288	2208	202	208
ML-KEM-768 ⁺ w/ KMAC	2288	2208	204	210

KEM variant	Client TX bytes	Server TX bytes	RTT time (μs)	
			Median	Average
ML-KEM-1024	3136	3136	512	511
ML-KEM-1024 ⁺ w/ Poly1305	3152	3168	266	273
ML-KEM-1024 ⁺ w/ GMAC	3152	3168	273	282
ML-KEM-1024 ⁺ w/ CMAC	3152	3168	280	287
ML-KEM-1024 ⁺ w/ KMAC	3152	3168	282	288

5 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

- [BCD⁺16] Joppe W. 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 Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1006–1018. ACM, 2016.
- [BDK⁺18] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrede 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 2018, London, United Kingdom, April 24-26, 2018*, pages 353–367. IEEE, 2018.
- [Ber05] Daniel J. Bernstein. The poly1305-aes message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, volume 3557 of *Lecture Notes in Computer Science*, pages 32–49. Springer, 2005.
- [BN00] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In Tatsuaki Okamoto, editor, *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545. Springer, 2000.
- [BP18] Daniel J. Bernstein and Edoardo Persichetti. Towards KEM unification. *IACR Cryptol. ePrint Arch.*, page 526, 2018.
- [BR05] John Black and Phillip Rogaway. CBC macs for arbitrary-length messages: The three-key constructions. *J. Cryptol.*, 18(2):111–131, 2005.
- [CS98] Ronald Cramer and Victor Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In Hugo Krawczyk, editor, *Advances in Cryptology - CRYPTO ’98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*, volume 1462 of *Lecture Notes in Computer Science*, pages 13–25. Springer, 1998.

- [DKRV18] 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 Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *Progress in Cryptology - AFRICACRYPT 2018 - 10th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 7-9, 2018, Proceedings*, volume 10831 of *Lecture Notes in Computer Science*, pages 282–305. Springer, 2018.
- [FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO ’99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 1999.
- [HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the fujisaki-okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part I*, volume 10677 of *Lecture Notes in Computer Science*, pages 341–371. Springer, 2017.
- [HHM22] Kathrin Hövelmanns, Andreas Hülsing, and Christian Majenz. Failing gracefully: Decryption failures and the fujisaki-okamoto transform. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part IV*, volume 13794 of *Lecture Notes in Computer Science*, pages 414–443. Springer, 2022.
- [IK03] Tetsu Iwata and Kaoru Kurosawa. OMAC: one-key CBC MAC. In Thomas Johansson, editor, *Fast Software Encryption, 10th International Workshop, FSE 2003, Lund, Sweden, February 24-26, 2003, Revised Papers*, volume 2887 of *Lecture Notes in Computer Science*, pages 129–153. Springer, 2003.
- [KCP16] John Kelsey, Shu-jen Chang, and Ray Perlner. Sha-3 derived functions: cshake, kmac, tuplehash, and parallelhash. *NIST special publication*, 800:185, 2016.
- [MV04] David A. McGrew and John Viega. The security and performance of the galois/counter mode (GCM) of operation. In Anne Canteaut and Kapalee Viswanathan, editors, *Progress in Cryptology - INDOCRYPT 2004, 5th International Conference on Cryptology in India, Chennai, India, December 20-22, 2004, Proceedings*, volume 3348 of *Lecture Notes in Computer Science*, pages 343–355. Springer, 2004.
- [Nat24] National Institute of Standards and Technology. Module-lattice-based key-encapsulation mechanism standard. Federal Information Processing Standards Publication (FIPS) NIST FIPS 203, Department of Commerce, Washington, D.C., 2024.
- [NL15] Yoav Nir and Adam Langley. Chacha20 and poly1305 for IETF protocols. *RFC*, 7539:1–45, 2015.
- [RRCB19] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic side-channel attacks on cca-secure lattice-based PKE and KEM schemes. *IACR Cryptol. ePrint Arch.*, page 948, 2019.
- [Sho04] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. *IACR Cryptol. ePrint Arch.*, page 332, 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 Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):296–322, 2022.

6 Appendix

6.1 Performance comparison between ML-KEM⁺ and Kyber

ML-KEM directly evolved from CRYSTALS-Kyber’s third round submission to NIST’s post quantum cryptography competition. While their IND-CPA subroutines (see figure 14) are identical, ML-KEM deviated from Kyber by choosing a different variant of the Fujisaki-Okamoto transformation.

K-PKE.KeyGen()	K-PKE.Enc(pk, m)	K-PKE.Dec(sk, c)
1: $A \xleftarrow{\$} R_q^{k \times k}$	Ensure: $\text{pk} = (A, \mathbf{t})$	Ensure: $c = (c_1, c_2)$
2: $\mathbf{s} \xleftarrow{\$} \mathcal{X}_{\eta_1}^k$	Ensure: $m \in R_2$	Ensure: $\text{sk} = \mathbf{s}$
3: $\mathbf{e} \xleftarrow{\$} \mathcal{X}_{\eta_1}^k$	1: $\mathbf{r} \xleftarrow{\$} \mathcal{X}_{\eta_1}^k$	1: $\hat{m} \leftarrow c_2 - \mathbf{c}_1^\top \cdot \mathbf{s}$
4: $\mathbf{t} \leftarrow A\mathbf{s} + \mathbf{e}$	2: $\mathbf{e}_1 \xleftarrow{\$} \mathcal{X}_{\eta_2}^k$	2: $\hat{m} \leftarrow \text{Round}(\hat{m})$
5: $\text{pk} \leftarrow (A, \mathbf{t})$	3: $e_2 \xleftarrow{\$} \mathcal{X}_{\eta_2}$	3: return \hat{m}
6: $\text{sk} \leftarrow \mathbf{s}$	4: $\mathbf{c}_1 \leftarrow A\mathbf{r} + \mathbf{e}_1$	
7: return (pk, sk)	5: $c_2 \leftarrow \mathbf{t}^\top \mathbf{r} + e_2 + m \cdot \lfloor \frac{q}{2} \rfloor$	
	6: return (c ₁ , c ₂)	

Figure 14: K-PKE routines are identical between Kyber and ML-KEM

CRYSTALS-Kyber uses the $U^\mathcal{L}$ variant, where the shared secret is derived from both the plaintext and the ciphertext. On the other hand, because by using *re-encryption* and *de-randomization*, the PKE is already made *rigid*, the CRYSTALS-Kyber team decided to use the $U_m^\mathcal{L}$ variant, where the shared secret is derived from the plaintext alone.

KEM.KeyGen()	KEM.Decap(sk, c)
1: $z \xleftarrow{\$} \{0, 1\}^{256}$ 2: $(pk, sk') \xleftarrow{\$} \text{PKE.KeyGen}()$ 3: $sk \leftarrow (sk' pk H(pk) z)$ 4: return (pk, sk)	Ensure: $sk = (sk' pk H(pk) z)$ 1: $\hat{m} \leftarrow \text{PKE.Dec}(sk', c)$ 2: $(\bar{K}, \hat{r}) \leftarrow G(\hat{m} H(pk))$ 3: if $\text{PKE.Enc}(pk, \hat{m}, \hat{r}) = c$ then 4: $K \leftarrow \text{KDF}(\bar{K}, H(c)) \quad \triangleright U^\mathcal{X}$ 5: $K \leftarrow \bar{K} \quad \triangleright U_m^\mathcal{X}$ 6: else 7: $K \leftarrow \text{KDF}(z H(c))$ 8: end if 9: return K
KEM.Encap(pk)	
1: $m \xleftarrow{\$} \{0, 1\}^{256}$ 2: $(\bar{K}, r) \leftarrow G(m H(pk))$ 3: $c \leftarrow \text{PKE.Enc}(pk, m, r)$ 4: $K \leftarrow \text{KDF}(\bar{K} H(c)) \quad \triangleright U^\mathcal{X}$ 5: $K \leftarrow \bar{K} \quad \triangleright U_m^\mathcal{X}$ 6: return (c, K)	

Figure 15: Kyber uses $U^\mathcal{X}$ variant. ML-KEM uses $U_m^\mathcal{X}$ variant.

489 The reason for ML-KEM to use a different variant of the Fujisaki-Okamoto trans-
490 formation is two-fold. The first reason is performance: using the $U_m^\mathcal{X}$ trans-
491 formation saves the need to hash the ciphertext, and since Kyber/ML-KEM’s performance is mainly
492 bottlenecked by the symmetric components, omitting the hash leads to significant runtime
493 savings (up to 17% in AVX-2 optimized implementations). The second reason is the
494 simplified security proof and tighter security bounds of the $U_m^\mathcal{X}$ variant compared to the
495 $U^\mathcal{X}$ variant. We will omit the details of the security proof and refer readers to [HHK17].

496 In section 4, we mainly compared ML-KEM⁺ with ML-KEM, but we would like to point
497 out that, because Kyber uses the $U^\mathcal{X}$ variant and needs to hash the ciphertext for deriving
498 the shared secret, the performance advantage of ML-KEM⁺ over Kyber will be even greater.