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 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 dominates 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 [NL18].

The main challenge in applying "encrypt-then-MAC" to public-key cryptography is the lack of a pre-shared symmetric key. We took inspiration from hybrid public-key encryption schemes (HPKE) such as .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 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 $\texttt{PKE}$ is a collection of three routines $(\texttt{KeyGen}, \texttt{Enc}, \texttt{Dec})$ defined over some message space $\mathcal{M}$ and some ciphertext space $\mathcal{C}$. Where the encryption routine is probabilistic, the source of randomness is denoted by the coin space $\mathcal{R}$.

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

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

The $\texttt{OW-ATK}$ game

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

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

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

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

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

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

The capabilities of the oracle $\mathcal{O}_{\texttt{ATK}}$ depends on the choice of security goal $\texttt{ATK}$. 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 $\texttt{True}$ if m is the decryption of c or $\texttt{False}$ otherwise.

2.2 Key encapsulation mechanism (KEM)

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

KEM-IND-CCA2 game

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

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

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

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

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

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

2.3 Message authentication code (MAC)

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

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

`EUF-CMA` game

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

Figure 3: The existential forgery game

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

2.4 Related works

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

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

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

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

let $\texttt{PKE} = (\texttt{KeyGen}_{\texttt{PKE}}, \texttt{Enc}, \texttt{Dec})$ be defined over message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$. Let $G : \mathcal{M} \rightarrow \mathcal{R}$ hash plaintexts into coincs, 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
$U^{\not\perp}$	implicit	OW-PCA
$U_m^{\perp}$	explicit	OW-VA + rigid
$U_m^{\not\perp}$	implicit	OW-CPA + rigid

$\texttt{KEM.KeyGen}()$

- $(\texttt{pk}, \texttt{sk}') \xleftarrow{\$} \texttt{PKE.KeyGen}()$
- $z \xleftarrow{\$} \mathcal{M}$
- $\texttt{sk} \leftarrow (\texttt{sk}', z)$ $\quad \triangleright\ U^{\not\perp}, U_m^{\not\perp}$
- $\texttt{sk} \leftarrow \texttt{sk}'$ $\quad \triangleright\ U^{\perp}, U_m^{\perp}$
- **return** $(\texttt{pk}, \texttt{sk})$

$\texttt{KEM.Encap}(\texttt{pk})$

- $m \xleftarrow{\$} \mathcal{M}$
- $r \leftarrow G(m)$
- $c \leftarrow \texttt{PKE.Enc}(\texttt{pk}, m, r)$
- $K \leftarrow H(m, c)$ $\quad \triangleright\ U^{\perp}, U^{\not\perp}$
- $K \leftarrow H(m)$ $\quad \triangleright\ U_m^{\perp}, U_m^{\not\perp}$
- **return** (c, K)

$\texttt{KEM.Decap}(\texttt{sk}, c)$

Ensure: $\texttt{sk} = (\texttt{sk}', z)$
Ensure: $\texttt{sk}'$ is valid PKE secret key
Ensure: z is valid PKE plaintext

- $\hat{m} \leftarrow \texttt{PKE.Dec}(\texttt{sk}', c)$
- $\hat{r} \leftarrow G(\hat{m})$
- $\hat{c} \leftarrow \texttt{PKE.Enc}(\texttt{pk}, \hat{m}, \hat{r})$
- **if** $\hat{c} = c$ **then**
  - $K \leftarrow H(\hat{m})$ $\quad \triangleright\ U_m^{\perp}, U_m^{\not\perp}$
  - $K \leftarrow H(\hat{m}, c)$ $\quad \triangleright\ U^{\perp}, U^{\not\perp}$
- **else**
  - $K \leftarrow H(z, c)$ $\quad \triangleright\ U^{\not\perp}, U_m^{\not\perp}$
  - $K \leftarrow \perp$ $\quad \triangleright\ U^{\perp}, U_m^{\perp}$
- **end if**
- **return** K

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

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

3 The "encrypt-then-MAC" transformation

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

$\texttt{KEM}_{\texttt{EtM}}\texttt{.KeyGen}()$

$(\texttt{pk}, \texttt{sk}') \xleftarrow{\$} \texttt{PKE.KeyGen}()$
$z \xleftarrow{\$} \mathcal{M}$
$\texttt{sk} \leftarrow \texttt{sk}' \| z$
return $(\texttt{pk}, \texttt{sk})$

$\texttt{KEM}_{\texttt{EtM}}\texttt{.Encap}(\texttt{pk})$

Ensure: pk is some PKE public key
$m \xleftarrow{\$} \mathcal{M}$
$k \leftarrow G(m)$
$c' \xleftarrow{\$} \texttt{PKE.Enc}(\texttt{pk}, m)$
$t \leftarrow \texttt{MAC}(k, c')$
$K \leftarrow H(m, c')$
$c \leftarrow c' \| t$
return (c, K)

$\texttt{KEM}_{\texttt{EtM}}\texttt{.Decap}(\texttt{sk}, c)$

Require: $c = c' \| t, \texttt{sk} = sk' \| z$
Ensure: c' is some PKE ciphertext
Ensure: t is some MAC tag
Ensure: $\texttt{sk}'$ is some PKE secret key
Ensure: z is some PKE plaintext
$(c', t) \leftarrow c$
$(\texttt{sk}', z) \leftarrow \texttt{sk}$
$\hat{m} \leftarrow \texttt{PKE.Dec}(\texttt{sk}', c')$
$\hat{k} \leftarrow G(\hat{m})$
if $\texttt{MAC}(\hat{k}, c') \neq t$ **then**
    $K \leftarrow H(z, c')$
else
    $K \leftarrow H(\hat{m}, c')$
end if
return K

Figure 5: $\texttt{KEM}_{\texttt{EtM}}$ routines

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

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

$\texttt{PCO}(m, c)$

$k \leftarrow G(m)$
$t \leftarrow \texttt{MAC}(k, c)$
return $[\![\mathcal{O}^{\texttt{Decap}}((c, t)) = H(m, c)]\!]$

Figure 6: 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 $c' \xleftarrow{\$} \texttt{Enc}(\texttt{pk}, m)$ can be correctly signed, while ciphertexts that don't decrypt back to m cannot be correctly signed. This means that a decapsulation oracle can be converted into a plaintext-checking oracle (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* $\texttt{KEM}_{\texttt{EtM}}$ *that makes q decapsulation queries, there exists an* `OW-PCA` *adversary B who makes at least q plaintext-checking queries against the underlying* `PKE`*, and an one-time existential forgery adversary C against the underlying* `MAC` *such that*

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

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

Lemma 1. $\texttt{KEM}_{\texttt{EtM}}$ *is IND-CCA2 secure if and only if the input* `PKE` *is OW-PCA secure*

3.1 Proof of theorem 1

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 $\texttt{KEM}_{\texttt{EtM}}$: replace true decapsulation with simulated decapsulation, replace the pseudorandom MAC key $k^* \leftarrow G(m^*)$ with a truly random MAC key $k^* \xleftarrow{\$} \mathcal{K}_{\texttt{MAC}}$, and finally replace pseudorandom shared secret $K_0 \leftarrow H(m^*, c')$ with a truly random shared secret $K_0 \xleftarrow{\$} \mathcal{K}_{\texttt{KEM}}$. 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 $\mathcal{O}^{\texttt{Decap}}$ 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 $\mathcal{O}^{\texttt{Decap}}$ is replaced with a simulated oracle $\mathcal{O}_1^{\texttt{Decap}}$. Instead of directly decrypting c' as in the decapsulation routine, the simulated oracle searches through the tape $\mathcal{L}^G$ to find a matching query $(\tilde{m}, \tilde{k})$ such that $\tilde{m}$ is the decryption of c'. The simulated oracle then uses $\tilde{k}$ to validate the tag t against c'.

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

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

$\texttt{IND-CCA2}$ game for $\texttt{KEM}_{\texttt{EtM}}$

$(\texttt{pk}, \texttt{sk}) \xleftarrow{\$} \texttt{KEM}_{\texttt{EtM}}.\texttt{KeyGen}()$
$m^* \xleftarrow{\$} \mathcal{M}$
$c' \xleftarrow{\$} \texttt{PKE.Enc}(\texttt{pk}, m^*)$
$k^* \leftarrow G(m^*)$ ▷ Game 0-1
$k^* \xleftarrow{\$} \mathcal{K}_{\texttt{MAC}}$ ▷ Game 2-3
$t \leftarrow \texttt{MAC}(k^*, c')$
$c^* \leftarrow c' \| t$
$K_0 \leftarrow H(m^*, c')$ ▷ Game 0-2
$K_0 \xleftarrow{\$} \mathcal{K}_{\texttt{KEM}}$ ▷ Game 3
$K_1 \xleftarrow{\$} \mathcal{K}_{\texttt{KEM}}$
$b \xleftarrow{\$} \{0, 1\}$
$\hat{b} \leftarrow A^{\mathcal{O}^{\texttt{Decap}}}(\texttt{pk}, c^*, K_b)$ ▷ Game 0
$\hat{b} \leftarrow A^{\mathcal{O}_1^{\texttt{Decap}}}(\texttt{pk}, c^*, K_b)$ ▷ Game 1-3
return $[\![\hat{b} = b]\!]$

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

if $\exists(\tilde{m}, \tilde{k}) \in \mathcal{L}^G : \tilde{m} = m$ **then**
  return $\tilde{k}$
end if
$k \xleftarrow{\$} \mathcal{K}_{\texttt{MAC}}$
$\mathcal{L}^G \leftarrow \mathcal{L}^G \cup \{(m, k)\}$
return k

$\mathcal{O}^{\texttt{Decap}}(c)$

$(c', t) \leftarrow c$
$\hat{m} = \texttt{Dec}(\texttt{sk}', c')$
$\hat{k} \leftarrow G(\hat{m})$
if $\texttt{MAC}(\hat{k}, c') = t$ **then**
  $K \leftarrow H(\hat{m}, c')$
else
  $K \leftarrow H(z, c')$
end if
return K

$\mathcal{O}_1^{\texttt{Decap}}(c)$

$(c', t) \leftarrow c$
if $\exists(\tilde{m}, \tilde{k}) \in \mathcal{L}^G : \tilde{m} = \texttt{Dec}(\texttt{sk}', c') \wedge \texttt{MAC}(\tilde{k}, c') = t$ **then**
  $K \leftarrow H(\tilde{m}, c')$
else
  $K \leftarrow H(z, c')$
end if
return K

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

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
$K \xleftarrow{\$} \mathcal{K}_{\texttt{KEM}}$
$\mathcal{L}^H \leftarrow \mathcal{L}^H \cup \{(m, c, K)\}$
return K

Figure 7: Sequence of games

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

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

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

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

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

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

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

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

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

$B(\texttt{pk}, c'^*)$

- $z \xleftarrow{\$} \mathcal{M}$
- $k \xleftarrow{\$} \mathcal{K}_{\texttt{MAC}}$
- $t \leftarrow \texttt{MAC}(k, c'^*)$
- $c^* \leftarrow (c'^*, t)$
- $K \xleftarrow{\$} \mathcal{K}_{\texttt{KEM}}$
- $\hat{b} \leftarrow A^{\mathcal{O}_B^{\texttt{Decap}}, \mathcal{O}_B^G, \mathcal{O}_B^H}(\texttt{pk}, c^*, K)$
- **if** $\texttt{ABORT}(m)$ **then**
  - **return** m
- **end if**

$\mathcal{O}_B^{\texttt{Decap}}(c)$

- $(c', t) \leftarrow c$
- **if** $\exists(\tilde{m}, \tilde{k}) \in \mathcal{L}^G : \texttt{PCO}(c', \tilde{m}) = 1 \wedge \texttt{MAC}(\tilde{k}, c') = t$ **then**
  - $K \leftarrow H(\tilde{m}, c')$
- **else**
  - $K \leftarrow H(z, c')$
- **end if**
- **return** K

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

- **if** $\texttt{PCO}(m, c'^*) = 1$ **then**
  - $\texttt{ABORT}(m)$
- **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**
- $K \xleftarrow{\$} \mathcal{K}_{\texttt{KEM}}$
- $\mathcal{L}^H \leftarrow \mathcal{L}^H \cup \{(m, c, K)\}$
- **return** K

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

- **if** $\texttt{PCO}(m, c'^*) = 1$ **then**
  - $\texttt{ABORT}(m)$
- **end if**
- **if** $\exists(\tilde{m}, \tilde{k}) \in \mathcal{L}^G : \tilde{m} = m$ **then**
  - **return** $\tilde{k}$
- **end if**
- $k \xleftarrow{\$} \mathcal{K}_{\texttt{MAC}}$
- $\mathcal{L}^G \leftarrow \mathcal{L}^G \cup \{(m, k)\}$
- **return** k

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

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

B simulates game 3 for A: receiving the public key $\texttt{pk}$ 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 $m^* = \texttt{Dec}(\texttt{sk}', c'^{\star})$ has been queried. When m^* is queried, B terminates A and returns m^* to win the OW-PCA game. In other words:

$$P\left[\texttt{QUERY G}\right] \leq \texttt{Adv}_{\texttt{OW-PCA}}(B)$$
$$P\left[\texttt{QUERY H}\right] \leq \texttt{Adv}_{\texttt{OW-PCA}}(B)$$

Combining all equations above produce the desired security bound. □

4 Implementation

Originally known as Kyber [BDK+18][ABD+19], ML-KEM is an `IND-CCA2` secure key encapsulation mechanism standardized in FIPS 203 by NIST. The `IND-CCA2` security of ML-KEM is achieved in two steps. First, ML-KEM constructs an `IND-CPA` secure public key encryption scheme $\texttt{K-PKE}(\texttt{KeyGen}, \texttt{Enc}, \texttt{Dec})$ whose security is based on the conjectured intractability of the module learning with error (MLWE) problems against both classical and quantum adversaries. Then, the $U_m^{\not\perp}$ variant of the Fujisaki-Okamoto transformation [HHK17] is used to construct the KEM $\texttt{MLKEM}(\texttt{KeyGen}, \texttt{Encap}, \texttt{Decap})$ by calling $\texttt{K-PKE.KeyGen}, \texttt{K-PKE.Enc}, \texttt{K-PKE.Dec}$ as sub-routines. Because `K-PKE.Enc` includes substantially more arithmetics than `K-PKE.Dec`, by using *re-encryption* and *de-randomization*, ML-KEM's decapsulation routine suffers from computational inefficiency.

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

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

`ML-KEM+.KeyGen()`

$z \xleftarrow{\$} \{0,1\}^{256}$
$(\texttt{pk}, \texttt{sk}') \xleftarrow{\$} \texttt{K-PKE.KeyGen}()$
$h \leftarrow H(\texttt{pk})$
$\texttt{sk} \leftarrow (sk' \| \texttt{pk} \| h \| z)$
return $(\texttt{pk}, \texttt{sk})$

`ML-KEM+.Encap(pk)`

Require: Public key `pk`
$m \xleftarrow{\$} \{0,1\}^{256}$
$(\overline{K}, r, k) \leftarrow \texttt{XOF}(m \| H(\texttt{pk}))$
$c' \leftarrow \texttt{K-PKE.Enc}(\texttt{pk}, m, r)$
$t \leftarrow \texttt{MAC}(k, c')$
$K \leftarrow \texttt{KDF}(\overline{K} \| t)$
$c \leftarrow (c', c)$
return (c, K)

`ML-KEM+.Decap(sk, c)`

Require: Secret key $\texttt{sk} = (\texttt{sk}' \| \texttt{pk} \| h \| z)$
Require: Ciphertext $c = (c' \| t)$
$(\texttt{sk}', \texttt{pk}, h, z) \leftarrow \texttt{sk}$
$(c', t) \leftarrow c$
$\hat{m} \leftarrow \texttt{K-PKE.Dec}(\texttt{sk}', c')$
$(\overline{K}, \hat{r}, \hat{k}) \leftarrow \texttt{XOF}(\hat{m} \| h)$
$\hat{t} \leftarrow \texttt{MAC}(\hat{k}, c')$
if $\hat{t} = t$ **then**
    $K \leftarrow \texttt{KDF}(\overline{K} \| t)$
else
    $K \leftarrow \texttt{KDF}(z \| t)$
end if
return 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][BR00], 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 ranodm field elements, and the message is parsed into tuples of field elements up to length L: $m = (m_1, m_2, \ldots, m_l) \in \mathbb{F}^{\leq L}$. The tag t is computed by evaluating a polynomial whose coefficients the message blocks and whose indeterminate is the key:

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

Where H_{xpoly} is given by:

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

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:

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

Specifically, Poly1035 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).

The CMAC is based on the CBC-MAC, where the block chosen is chosen to be 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

Table 2: MAC performance

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

4.2 KEM performance

Compared to the $U_m^{\not\perp}$ 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:

Sub-key derivation

Require: 256-bit key k
Require: `const_Rb = 0x87`
$l \leftarrow \texttt{AES-256}(k, 0^{128})$
if $\texttt{MostSignificantBit}(l) = 0$ **then**
  $k_1 \leftarrow$ `l << 1`
else
  $k_1 \leftarrow$ `l << 1` $\oplus$ `const_Rb`
end if
if $\texttt{MostSignificantBit}(k_1) = 0$ **then**
  $k_2 \leftarrow k_1$ `<< 1`
else
  $k_2 \leftarrow k_1$ `<< 1` $\oplus$ `const_Rb`
end if
return k_1, k_2

CMAC(k, m)

Require: 256-bit symmetric key k
$(k_1, k_2) \leftarrow \texttt{deriveSubKey}(k)$
$n \leftarrow \lceil \texttt{bytesLen}(m)/16 \rceil$
if $n = 0$ **then**
  $n \leftarrow 1$
  $m_{\texttt{last}} \leftarrow m_n \oplus k_2$
else if $\texttt{bytesLen(m)} \mod 16 = 0$ **then**
  $m_{\texttt{last}} \leftarrow m_n \oplus k_1$
else
  $m_{\texttt{last}} \leftarrow m_n \oplus k_2$
end if
$x = 0^{128}$
for $i \in \{1, 2, \ldots, n-1\}$ **do**
  $y \leftarrow x \oplus m_i$
  $x \leftarrow \texttt{AES-256}(k, y)$
end for
$y \leftarrow m_{\texttt{last}} \oplus x$
$t \leftarrow \texttt{AES-256}(k, y)$
return t

Figure 10: AES-256 CMAC

1. 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 $U_m^{\not\perp}$ transformation only needs to hash the PKE plaintext.

2. The encapsulation routine adds a small amount of run-time overhead for computing the authenticator

3. The decapsulation routine enjoys substantial runtime speedup because *re-encryption* is replaced with computing an authenticator

4. 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 $U_m^{\not\perp}$ 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 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

Table 3: CPU cycles of each KEM routine

ML-KEM-512	
size parameters (bytes)	
pubkey size	800
seckey size	1632
ciphertext size	768
`KeyGen` cycles/tick	
Median	75945
Average	76171

KEM variant	`Encap` cycles/tick		`Decap` cycles/tick	
	Median	Average	Median	Average
`ML-KEM-512`	91467	92065	121185	121650
`Kyber512`	97811	98090	119937	120299
`ML-KEM-512`$^+$ w/ Poly1305	93157	93626	33733	33908
`ML-KEM-512`$^+$ w/ GMAC	97369	97766	37725	37831
`ML-KEM-512`$^+$ w/ CMAC	99739	99959	40117	39943
`ML-KEM-512`$^+$ w/ KMAC	101009	101313	40741	40916

ML-KEM-768	
size parameters (bytes)	
pubkey size	1184
seckey size	2400
ciphertext size	1088
`KeyGen` cycles/tick	
Median	129895
Average	130650

KEM variant	`Encap` cycles/tick		`Decap` cycles/tick	
	Median	Average	Median	Average
`ML-KEM-768`	136405	147400	186445	187529
`Kyber768`	153061	153670	182129	182755
`ML-KEM-768`$^+$ w/ Poly1305	146405	146860	43315	43463
`ML-KEM-768`$^+$ w/ GMAC	149525	150128	46513	46706
`ML-KEM-768`$^+$ w/ CMAC	153139	153735	49841	50074
`ML-KEM-768`$^+$ w/ KMAC	155219	155848	52415	52611

ML-KEM-1024	
size parameters (bytes)	
pubkey size	1568
seckey size	3168
ciphertext size	1568
`KeyGen` cycles/tick	
Median	194921
Average	195465

KEM variant	`Encap` cycles/tick		`Decap` cycles/tick	
	Median	Average	Median	Average
`ML-KEM-1024`	199185	199903	246245	247320
`Kyber1024`	222351	223260	258231	259067
`ML-KEM-1024`$^+$ w/ Poly1305	205763	206499	51375	51562
`ML-KEM-1024`$^+$ w/ GMAC	208805	209681	54573	54780
`ML-KEM-1024`$^+$ w/ CMAC	213667	214483	59175	59408
`ML-KEM-1024`$^+$ w/ KMAC	216761	217468	62269	62516

between the moment before the client starts generating ephemeral keypairs and the moment after the client derives the final session key. All experiements are run on a pair of AWS c7a.medium instances both located in the `us-west-2` region. For each experiment, a total of 10,000 rounds of key exchange are performed, with the median and average round trip time (measured in microsecond) recorded.

4.3.1 Unauthenticated key exchange (KE)

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

$\mathtt{KE_C}()$

$(\mathtt{pk}_e, \mathtt{sk}_e) \xleftarrow{\$} \mathtt{KeyGen}()$
$\mathtt{send}(\mathtt{pk}_e)$
$c_e \leftarrow \mathtt{read}()$
$K_e \leftarrow \mathtt{Decap}(\mathtt{sk}_e, c_e)$
$K \leftarrow \mathtt{KDF}(K)$
return K

$\mathtt{KE_S}()$

$\mathtt{pk}_e \leftarrow \mathtt{read}()$
$(c_e, K_e) \xleftarrow{\$} \mathtt{Encap}(\mathtt{pk}_e)$
$\mathtt{send}(c_e)$
$K \leftarrow \mathtt{KDF}(K_e)$
return K

Figure 11: Unauthenticated key exchange (KE) routines

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

4.3.2 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 $\mathtt{pk}_S$ of the server, and the server authenticates itself by demonstrating possession of the corresponding long-term secret key $\mathtt{sk}_S$. UAKE routines are summarized in figure 12.

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).

$\mathtt{UAKE_C}(\mathtt{pk}_S)$

Require: Server's long-term $\mathtt{pk}_S$

$(\mathtt{pk}_e, \mathtt{sk}_e) \xleftarrow{\$} \mathtt{KeyGen}()$
$(c_S, K_S) \xleftarrow{\$} \mathtt{Encap}(\mathtt{pk}_S)$
$\mathtt{send}(\mathtt{pk}_e, c_S)$
$c_e \leftarrow \mathtt{read}()$
$K_e \leftarrow \mathtt{Decap}(\mathtt{sk}_e, c_e)$
$K \leftarrow \mathtt{KDF}(K_e \| K_S)$
return K

$\mathtt{UAKE_S}(\mathtt{sk}_S)$

Require: Server's long-term $\mathtt{sk}_S$

$(\mathtt{pk}_e, c_S) \leftarrow \mathtt{read}()$
$K_S \leftarrow \mathtt{Decap}(\mathtt{sk}_S, c_S)$
$(c_e, K_e) \xleftarrow{\$} \mathtt{Encap}(\mathtt{pk}_e)$
$\mathtt{send}(c_e)$
$K \leftarrow \mathtt{KDF}(K_e \| K_S)$
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 (μs)	
			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 (μs)	
			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 (μs)	
			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

4.3.3 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.

$\mathtt{AKE_C}(\mathtt{pk}_S, \mathtt{sk}_C)$

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

$\mathtt{AKE_S}(\mathtt{sk}_S, \mathtt{pk}_C)$

Require: Server's long-term $\mathtt{sk}_S$
Require: Client's long-term $\mathtt{pk}_C$
1: $(\mathtt{pk}_e, c_S) \leftarrow \mathtt{read}()$
2: $K_S \leftarrow \mathtt{Decap}(\mathtt{sk}_S, c_S)$
3: $(c_e, K_e) \xleftarrow{\$} \mathtt{Encap}(\mathtt{pk}_e)$
4: $(c_C, K_C) \xleftarrow{\$} \mathtt{Encap}(\mathtt{pk}_C)$
5: $\mathtt{send}(c_e, c_C)$
6: $K \leftarrow \mathtt{KDF}(K_e \| K_S \| K_C)$
7: **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 $m = \texttt{Dec}(\texttt{sk}, c)$ is equivalent to $c = \texttt{Enc}(\texttt{pk}, m)$, so a plaintext-checking oracle can be simulated without any secret information. However, the $U_m^{\not\perp}$ transformation in [HHK17] can already transform an OW-CPA secure and rigid PKE into an IND-CCA2 secure KEM with minimal overhead: the encapsulation and decapsulation routines each adds a hash of the plaintext to the encryption and decryption routine. In other words, where the input PKE is rigid, "encrypt-then-MAC" doesn't offer any performance advantage. It remains an open problem whether there exists a PKE that is OW-PCA secure but not rigid. If such a PKE exists, then "encrypt-then-MAC" would be a preferable strategy for constructing an IND-CCA2 KEM.

References

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

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

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

[Ber05] Daniel J Bernstein. The poly1305-aes message-authentication code. In *International workshop on fast software encryption*, pages 32–49. Springer, 2005.

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

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

[BR00] John Black and Phillip Rogaway. Cbc macs for arbitrary-length messages: The three-key constructions. In *Annual International Cryptology Conference*, pages 197–215. Springer, 2000.

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

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

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

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

[IK03] Tetsu Iwata and Kaoru Kurosawa. Omac: One-key cbc mac. In *Fast Software Encryption: 10th International Workshop, FSE 2003, Lund, Sweden, February 24-26, 2003. Revised Papers 10*, 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 McGrew and John Viega. The galois/counter mode of operation (gcm). *submission to NIST Modes of Operation Process*, 20:0278–0070, 2004.

[NL18] Yoav Nir and Adam Langley. Chacha20 and poly1305 for ietf protocols. Technical report, 2018.

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

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

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