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

Key encapsulation mechanism (KEM) is a cryptographic primitive that allows two parties to establish a shared secret over an insecure channel. The combination of KEM and some data encapsulation mechanism (DEM), such as AES-GCM and ChaCha20-Poly1305, is the foundation of many of today's most popular secure communication protocols such as Transport Layer Security (TLS) and Secure Shell (SSH). The commonly accepted security standard of a KEM is *Indistinguishability under adaptive chosen-ciphertext attack (IND-CCA2)*: no efficient adversary can distinguish a shared secret obtained by running the encapsulation routine from random noise, even with access to a decapsulation oracle throughout the attack. KEM is related to another important cryptographic primitive called public-key encryption (PKE), and their security standards are similar. The desired security standard for PKE, also called IND-CCA2, requires that no efficient adversary can distinguish the encryption of two adversarially chosen messages even with access to decryption oracle.

It is difficult to build a provably IND-CCA2 secure KEM from scratch. Instead, secure KEMs are usually built on top of a PKE with weaker security property (e.g. being only OW-CPA or IND-CPA secure, not IND-CCA2). One such construction is the Fujisaki-Okamoto transformation, proposed in 1999 by Fujisaki Eiichiro and Okamoto Tetsuyaki in their seminal paper [FO99][FO13]. The first Fujisaki-Okamoto transformation combines an OW-CPA secure PKE with an IND-CPA secure symmetric cipher into a hybrid PKE (HPKE) with proven IND-CCA2 security under the random oracle model. Subsequent proposals such as [OP01], [CHJ+02], and [Den03] improved on the original proposal and adapted it to build KEM instead of HPKE. This line of work culminated in a landmark publication in 2017 by Hofhein, Hovelmann, and Kiltz [HHK17a][HHM22], where the authors provided a versatile variety of modular KEM constructions with tight security reduction in the random oracle model and non-tight security reduction in the quantum random oracle model.

The modular Fujisaki-Okamoto KEM transformation is remarkably successful. It was adopted by many submissions to NIST's post-quantum cryptography competition, including Kyber [BDK+18a], Saber [DKRV18], FrodoKEM [BCD+16], and classic McEliece [ABC+20] among others. When Kyber was standardized by NIST in FIPS 203 "Module-lattice key-encapsulation mechanism" (ML-KEM) [oST24], it kept the Fujisaki-Okamoto transformation in its KEM construction. However, the Fujisaki-Okamoto transformation is not perfect. Among its shortcomings is the use of *de-randomization* (transform a randomized encryption routine into a deterministic routine by using a pseudorandom coin derived from the input message) and *re-encryption* (the decapsulation routine of the output KEM runs the encryption routine of the input PKE to ensure ciphertext integrity), which causes two problems:

- **computational inefficiency:** where the PKE's encryption routine is substantially more expensive than the decryption routine, using re-encryption causes the decapsulation routine in the output KEM to become computationally expensive
- **side-channel vulnerability:** running the input PKE's encryption routine in the output KEM's decapsulation routine introduces risk of side-channel vulnerabilities not found in the input PKE's decryption routine alone. In fact, many practical attacks [UXT+22][RRCB19] exploit re-encryption to decrypt ciphertext or recover secret keys. Countermeasures such as masking have been proposed to address these side channels, but they inevitably carry substantial performance penalty.

1.1 Our contributions

Our main contribution is a novel generic construction that combines a PKE with weaker security property and a one-time secure message authentication code (MAC) into an IND-CCA2 secure KEM in the random oracle model. Specifically, we require the input PKE to be one-way secure against plaintext-checking attacks (OW-PCA): no efficient adversary can recover the decryption of a random encryption with access to a plaintext-checking oracle (PCO) that, when queried on a plaintext-ciphertext pair (m, c), answers m is the decryption of c. While OW-PCA is not a standard security definition, it has appeared useful in past IND-CCA2 KEM constructions [OP01][CHJ+02], and plays an important role in the modularity of the Fujisaki-Okamoto KEM transformation.

Our KEM construction is inspired by how symmetric cryptography achieves IND-CCA2 security (or equivalently, authenticated encryption [Kra01]) using the "encrypt-then-MAC" pattern. We applied "encrypt-then-MAC" to the construction of the KEM: at encapsulation, a symmetric key is derived from hashing a random PKE plaintext, then used to compute an authenticator against the PKE ciphertext; at decapsulation, the same symmetric key is derived from hashing the decryption of the PKE ciphertext, then used to verify the authenticator. Intuitively, for an adversary to be able to produce a valid authenticator, it must know the correct symmetric key and thus the corresponding PKE plaintext. In other words, the adversary either produced the ciphertext honestly, or have broken the one-wayness of the input PKE.

1.1.1 Performance improvements

The main advantage of our KEM construction over the Fujisaki-Okamoto transformation is the performance gains: our construction replaces re-encryption with computing a symmetric message authenticator, which is significantly faster, especially for one-time MAC. At the cost of computing an authenticator in encapsulation and increasing the ciphertext size by a message authenticator, our construction speeds up decapsulation by 10x.

When instantiated with the underlying PKE routines of ML-KEM and the Poly1305 message authenticator, our construction ML-KEM$^+$ achieves on average 72%-80% reduction of CPU cycles needed for decapsulation while only incurring 2%-7% increase of CPU cycles for encapsulation when compared to ML-KEM.

We also implemented and measured the round trip time of key exchange protocols with various modes of authentication. When compared to ML-KEM, ML-KEM$^+$ achieves 24%-28% reduction of round trip time in unauthenticated key exchange, 29%-35% reduction in unilaterally authenticated key exchange, and 35%-48% reduction in mutually authenticated key exchange.

1.2 Related works

A similar construction was proposed by Abdulla et al [ABR01], though our construction builds a KEM instead of a HPKE, and our construction is generic over all OW-PCA secure PKE.

2 Preliminaries and previous results

2.1 Public-key encryption scheme

Syntax. A public-key encryption scheme $\texttt{PKE}(\texttt{KeyGen}, \texttt{Enc}, \texttt{Dec})$ is a collection of three routines defined over some plaintext space $\mathcal{M}$ and some ciphertext space $\mathcal{C}$. $(\texttt{pk}, \texttt{sk}) \xleftarrow{\$} \texttt{KeyGen}()$ is a randomized routine that returns a keypair. The encryption routine $\texttt{Enc} : (\texttt{pk}, m) \mapsto c$ encrypts the input plaintext under the input public key. The decryption routine $\texttt{Dec} : (\texttt{sk}, c) \mapsto m$ 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. Following the definition in [DNR04] and [HHK17b], a $\texttt{PKE}$ is δ-correct if:

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

Where the expectation is taken with respect to the probability distribution of all possible keypairs $(\texttt{pk}, \texttt{sk}) \xleftarrow{\$} \texttt{PKE.KeyGen()}$. For many lattice-based cryptosystems, including ML-KEM, decryption failures could leak information about the secret key, although the probability of a decryption failure is low enough that classical adversaries cannot exploit decryption failure more than they can defeat the underlying lattice problem (see table 1).

Table 1: Estimated probability of decryption failure in ML-KEM

	ML-KEM-512	ML-KEM-768	ML-KEM-1024
δ	???	???	???

On the other hand, a quantum adversary may be able to exploit decryption failure in reasonable runtime by efficiently searching through all possible inputs using Grover's search algorithm. For that, ML-KEM made slight modifications in its KEM construction to prevent quantum adversary from precomputing large lookup table. We refer readers to [ABD$^+$19] and [BDK$^+$18b] for the details.

Security. 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

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

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

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 Decap routine using the secret key. The advantage $\epsilon_{\texttt{IND-CCA}}$ of an IND-CCA adversary $\mathcal{A}_{\texttt{IND-CCA}}$ is defined by

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

2.3 Message authentication code (MAC)

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

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

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

Figure 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.

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 $\texttt{PKE}$'s plaintext space to $\texttt{MAC}$'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 4.

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

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

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

Ensure: $\texttt{pk}$ is some $\texttt{PKE}$ 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 $\texttt{PKE}$ ciphertext
Ensure: t is some $\texttt{MAC}$ tag
Ensure: $\texttt{sk}'$ is some $\texttt{PKE}$ secret key
Ensure: z is some $\texttt{PKE}$ 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 4: $\texttt{KEM}_{\texttt{EtM}}$ routines

The key generation routine of $\texttt{KEM}_{\texttt{EtM}}$ is largely identical to that of the $\texttt{PKE}$, 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 $\texttt{PKE}$. This means that the decapsulation oracle will not give out information on decryptions that the adversary does not already know.

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

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

Figure 5: 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 5), 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* $\texttt{IND-CCA2}$ *adversary* A *against* $\texttt{KEM}_{\texttt{EtM}}$ *that makes* q *decapsulation queries, there exists an* $\texttt{OW-PCA}$ *adversary* B *who makes at least* q *plaintext-checking queries against the underlying* $\texttt{PKE}$*, and an one-time existential forgery adversary* C *against the underlying* $\texttt{MAC}$ *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* $\texttt{PKE}$ *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 6 and 7. 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

$\texttt{IND-CCA2}$ 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 6: Sequence of games

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 $\texttt{MAC}$ under an unknown and uniformly random key. Furthermore, the security game does not include a signing oracle, so this is a zero-time forgery. While zero-time forgery is not a standard security definition for a MAC, we can bound it by the advantage of a one-time forgery adversary C:

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

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

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

Game 2 is identical to game 1, except that 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:

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

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:

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

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

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

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

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

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

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

if $\texttt{PCO}(m, c'^*) = 1$ **then**
$\texttt{ABORT}(m)$
end if
if $\exists(\tilde{m}, \tilde{c}, \tilde{K}) \in \mathcal{L}^H : \tilde{m} = m \land \tilde{c} = c$ **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

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

1: **if** $\texttt{PCO}(m, c'^*) = 1$ **then**
2: $\texttt{ABORT}(m)$
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: $k \xleftarrow{\$} \mathcal{K}_{\texttt{MAC}}$
8: $\mathcal{L}^G \leftarrow \mathcal{L}^G \cup \{(m, k)\}$
9: **return** k

Figure 7: 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 7.

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

ML-KEM is an $\texttt{IND-CCA2}$ secure key encapsulation mechanism standardized by NIST in FIPS 203. The $\texttt{IND-CCA2}$ security of ML-KEM is achieved in two steps. First, ML-KEM constructs an $\texttt{IND-CPA}$ 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 [HHK17b] is used to construct the KEM $\texttt{MLKEM}(\texttt{KeyGen}, \texttt{Encap}, \texttt{Decap})$ by calling $\texttt{K-PKE}(\texttt{KeyGen}, \texttt{Enc}, \texttt{Dec})$ as sub-routines. Because $\texttt{K-PKE.Enc}$ includes substantially more arithmetics than $\texttt{K-PKE.Dec}$, 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 $\texttt{K-PKE}$ 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 $\texttt{K-PKE}$'s routines can be found in FIPS 203 (TODO: citation). The "encrypt-then-MAC" routines are described in figure 8.

`ML-KEM+.KeyGen()`

1: $z \xleftarrow{\$} \{0,1\}^{256}$
2: $(\mathtt{pk}, \mathtt{sk}') \xleftarrow{\$} \mathtt{K\text{-}PKE.KeyGen}()$
3: $h \leftarrow H(\mathtt{pk})$
4: $\mathtt{sk} \leftarrow (sk' \| \mathtt{pk} \| h \| z)$
5: **return** $(\mathtt{pk}, \mathtt{sk})$

`ML-KEM+.Encap(pk)`

Require: Public key pk
1: $m \xleftarrow{\$} \{0,1\}^{256}$
2: $(\overline{K}, r, k) \leftarrow \mathtt{XOF}(m \| H(\mathtt{pk}))$
3: $c' \leftarrow \mathtt{K\text{-}PKE.Enc}(\mathtt{pk}, m, r)$
4: $t \leftarrow \mathtt{MAC}(k, c')$
5: $K \leftarrow \mathtt{KDF}(\overline{K} \| c')$
6: $c \leftarrow (c', t)$
7: **return** (c, K)

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

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

Figure 8: `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 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:

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

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:

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

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).

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 9

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**
  $k_1 \leftarrow \texttt{l << 1}$
else
  $k_1 \leftarrow \texttt{l << 1} \oplus \texttt{const\_Rb}$
end if
if $\texttt{MostSignificantBit}(k_1) = 0$ **then**
  $k_2 \leftarrow k_1 \texttt{ << 1}$
else
  $k_2 \leftarrow k_1 \texttt{ << 1} \oplus \texttt{const\_Rb}$
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 9: AES-256 CMAC

KMAC is defined in NIST SP 800-185 to be based on the family of sponge functions with Keccak permutaiton 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 $\texttt{ML-KEM}^{+}$, 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

4.2 KEM performance

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

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 $U_m^{\not\bot}$ 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 $U_m^{\not\bot}$ 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

Table 3: CPU cycles of each KEM routine

128-bit security	
size parameters (bytes)	
`pk` size	800
`sk` size	1632
`ct` 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

192-bit security	
size parameters (bytes)	
`pk` size	1184
`sk` size	2400
`ct` 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

256-bit security	
size parameters (bytes)	
`pk` size	1568
`sk` size	3168
`ct` 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

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+18b] described three key exchange protocols: unauthenticated key exchange (KE), unilaterally authenticated key exchange (UAKE), and mutually authenticated key exchange (AKE). We instantiated an implementation for each of the three key exchange protocols using different variations of the "encrypt-then-MAC" KEM and compared round trip time with implementations instantiated using ML-KEM.

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

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 10.

Note that in our implementation, a key derivation function ($\mathtt{KDF}$) 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.

$\mathtt{KE}_\mathtt{C}()$

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

$\mathtt{KE}_\mathtt{S}()$

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

Figure 10: 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 $\texttt{pk}_S$ of the server, and the server authenticates itself by demonstrating possession of the corresponding long-term secret key $\texttt{sk}_S$. UAKE routines are summarized in figure 11.

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).

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

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

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

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

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

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

Figure 11: 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 $\mathtt{sk}_C$
$(\mathtt{pk}_e, \mathtt{sk}_e) \xleftarrow{\$} \mathtt{KeyGen}()$
$(c_S, K_S) \xleftarrow{\$} \mathtt{Encap}(\mathtt{pk}_S)$
$\mathtt{send}(\mathtt{pk}_e, c_S)$
$(c_e, c_C) \leftarrow \mathtt{read}()$
$K_e \leftarrow \mathtt{Decap}(\mathtt{sk}_e, c_e)$
$K_C \leftarrow \mathtt{Decap}(\mathtt{sk}_e, c_C)$
$K \leftarrow \mathtt{KDF}(K_e \| K_S \| K_C)$
return K

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

Require: Server's long-term $\mathtt{sk}_S$
Require: Client's long-term $\mathtt{pk}_C$
$(\mathtt{pk}_e, c_S) \leftarrow \mathtt{read}()$
$K_S \leftarrow \mathtt{Decap}(\mathtt{sk}_S, c_S)$
$(c_e, K_e) \xleftarrow{\$} \mathtt{Encap}(\mathtt{pk}_e)$
$(c_C, K_C) \xleftarrow{\$} \mathtt{Encap}(\mathtt{pk}_C)$
$\mathtt{send}(c_e, c_C)$
$K \leftarrow \mathtt{KDF}(K_e \| K_S \| K_C)$
return K

Figure 12: 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

The "encrypt-then-MAC" transformation is a generic KEM construction that achieves IND-CCA2 security under the random oracle model if the input PKE is OW-PCA secure. Compared to the Fujisaki-Okamoto transformation, our construction replaced *de-randomization* and *re-encryption* with a message authenticator. At the cost of some minimal increase in communication size and encapsulation runtime, our construction achieves significant efficiency gains in the decapsulation routine. In practical key exchange protocols, our construction saves between 35-45% in round trip time.

References

[ABC+20] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, RubenNiederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic mceliece. Technical report, National Institute of Standards and Technology, 2020.

[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.

[ABR01] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. The oracle diffie-hellman assumptions and an analysis of DHIES. In David Naccache, editor, *Topics in Cryptology - CT-RSA 2001, The Cryptographer's Track at RSA Conference 2001, San Francisco, CA, USA, April 8-12, 2001, Proceedings*, volume 2020 of *Lecture Notes in Computer Science*, pages 143–158. Springer, 2001.

[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+18a] Joppe W. 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 2018, London, United Kingdom, April 24-26, 2018*, pages 353–367. IEEE, 2018.

[BDK+18b] Joppe W. 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 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.

[BR05] John Black and Phillip Rogaway. CBC macs for arbitrary-length messages: The three-key constructions. *J. Cryptol.*, 18(2):111–131, 2005.

[CHJ+02] Jean-Sébastien Coron, Helena Handschuh, Marc Joye, Pascal Paillier, David Pointcheval, and Christophe Tymen. GEM: A generic chosen-ciphertext secure encryption method. In Bart Preneel, editor, *Topics in Cryptology - CT-RSA 2002, The Cryptographer's Track at the RSA Conference, 2002, San Jose, CA, USA, February 18-22, 2002, Proceedings*, volume 2271 of *Lecture Notes in Computer Science*, pages 263–276. Springer, 2002.

[Den03] Alexander W. Dent. A designer's guide to kems. In Kenneth G. Paterson, editor, *Cryptography and Coding, 9th IMA International Conference, Cirencester, UK, December 16-18, 2003, Proceedings*, volume 2898 of *Lecture Notes in Computer Science*, pages 133–151. Springer, 2003.

[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.

[DNR04] Cynthia Dwork, Moni Naor, and Omer Reingold. Immunizing encryption schemes from decryption errors. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, volume 3027 of *Lecture Notes in Computer Science*, pages 342–360. Springer, 2004.

[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.

[FO13] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *J. Cryptol.*, 26(1):80–101, 2013.

[HHK17a] 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.

[HHK17b] 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.

[Kra01] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is ssl?). In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 310–331. Springer, 2001.

[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.

[OP01] Tatsuaki Okamoto and David Pointcheval. REACT: rapid enhanced-security asymmetric cryptosystem transform. In David Naccache, editor, *Topics in Cryptology - CT-RSA 2001, The Cryptographer's Track at RSA Conference 2001, San Francisco, CA, USA, April 8-12, 2001, Proceedings*, volume 2020 of *Lecture Notes in Computer Science*, pages 159–175. Springer, 2001.

[oST24] National Institute of Standards and Technology. Module-lattice-based key-encapsulation mechanism standard. Technical Report Federal Information Processing Standards Publication (FIPS) NIST FIPS 203, U.S. Department of Commerce, Washington, D.C., 2024.

[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 $\texttt{ML-KEM}^{+}$ 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 13) are identical, ML-KEM deviated from Kyber by choosing a different variant of the Fujisaki-Okamoto transformation.

`K-PKE.KeyGen()`

$A \xleftarrow{\$} R_q^{k\times k}$
$\mathbf{s} \xleftarrow{\$} \mathcal{X}_{\eta_1}^k$
$\mathbf{e} \xleftarrow{\$} \mathcal{X}_{\eta_1}^k$
$\mathbf{t} \leftarrow A\mathbf{s} + \mathbf{e}$
$\mathtt{pk} \leftarrow (A, \mathbf{t})$
$\mathtt{sk} \leftarrow \mathbf{s}$
return $(\mathtt{pk}, \mathtt{sk})$

`K-PKE.Enc(pk, m)`

Ensure: $\mathtt{pk} = (A, \mathbf{t})$
Ensure: $m \in R_2$
$\mathbf{r} \xleftarrow{\$} \mathcal{X}_{\eta_1}^k$
$\mathbf{e}_1 \xleftarrow{\$} \mathcal{X}_{\eta_2}^k$
$e_2 \xleftarrow{\$} \mathcal{X}_{\eta_2}$
$\mathbf{c}_1 \leftarrow A\mathbf{r} + \mathbf{e}_1$
$c_2 \leftarrow \mathbf{t}^\intercal \mathbf{r} + e_2 + m \cdot \lfloor \frac{q}{2} \rceil$
return $(\mathbf{c}_1, c_2)$

`K-PKE.Dec(sk, c)`

Ensure: $c = (\mathbf{c}_1, c_2)$
Ensure: $\mathtt{sk} = \mathbf{s}$
$\hat{m} \leftarrow c_2 - \mathbf{c}_1^\intercal \cdot \mathbf{s}$
$\hat{m} \leftarrow \mathtt{Round}(\hat{m})$
return $\hat{m}$

Figure 13: `K-PKE` routines are identical between Kyber and ML-KEM

CRYSTALS-Kyber uses the $U^{\not\perp}$ 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 $U_m^{\not\perp}$ variant, where the shared secret is derived from the plaintext alone.

`KEM.KeyGen()`

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

`KEM.Encap(pk)`

$m \xleftarrow{\$} \{0,1\}^{256}$
$(\overline{K}, r) \leftarrow G(m \| H(\mathtt{pk}))$
$c \leftarrow \mathtt{PKE.Enc}(\mathtt{pk}, m, r)$
$K \leftarrow \mathtt{KDF}(\overline{K} \| H(c))$ ▷ $U^{\not\perp}$
$K \leftarrow \overline{K}$ ▷ $U_m^{\not\perp}$
return (c, K)

`KEM.Decap(sk, c)`

Ensure: $\mathtt{sk} = (\mathtt{sk}' \| \mathtt{pk} \| H(\mathtt{pk}) \| z)$
$\hat{m} \leftarrow \mathtt{PKE.Dec}(\mathtt{sk}', c)$
$(\overline{K}, \hat{r}) \leftarrow G(\hat{m} \| H(\mathtt{pk}))$
if $\mathtt{PKE.Enc}(\mathtt{pk}, \hat{m}, \hat{r}) = c$ **then**
  $K \leftarrow \mathtt{KDF}(\overline{K}, H(c))$ ▷ $U^{\not\perp}$
  $K \leftarrow \overline{K}$ ▷ $U_m^{\not\perp}$
else
  $K \leftarrow \mathtt{KDF}(z \| H(c))$
end if
return K

Figure 14: Kyber uses $U^{\not\perp}$ variant. ML-KEM uses $U_m^{\not\perp}$ variant.

The reason for ML-KEM to use a different variant of the Fujisaki-Okamoto transformation is two-fold. The first reason is performance: using the $U_m^{\not\perp}$ transformation saves the need to hash the ciphertext, and since Kyber/ML-KEM's performance is mainly bottlenecked by the symmetric components, omitting the hash leads to significant runtime savings (up to 17% in AVX-2 optimized implementations). The second reason is the simplified security proof and tighter security bounds of the $U_m^{\not\perp}$ variant compared to the $U^{\not\perp}$ variant. We will omit the details of the security proof and refer readers to [HHK17b].

In section 4, we mainly compared `ML-KEM`$^+$ with `ML-KEM`, but the we would like to point out that, because Kyber uses the $U^{\not\perp}$ variant and needs to hash the ciphertext for deriving the shared secret, the performance advantage of `ML-KEM`$^+$ over Kyber will be even greater.