Unbindable Kemmy Schmidt: ML-KEM is neither MAL-BIND-K-CT nor MAL-BIND-K-PK

Sophie Schmieg[1]

[1]Google, sschmieg@google.com

April 3, 2024

Abstract

In [CDM23] Cremers et al. introduced various binding models for KEMs. The authors show that ML-KEM is LEAK-BIND-K-CT and LEAK-BIND-K-PK, i.e. binding the ciphertext and the public key in the case of an adversary having access, but not being able to manipulate the key material. They further conjecture that ML-KEM also has MAL-BIND-K-PK, but not MAL-BIND-K-CT, the binding of public key or ciphertext to the shared secret in the case of an attacker with the ability to manipulate the key material.

This short paper demonstrates that ML-KEM does neither have MAL-BIND-K-CT nor MAL-BIND-K-PK, due to the attacker being able to produce mal-formed private keys, giving concrete examples for both. We also suggest mitigations, and sketch a proof for binding both ciphertext and public key when the attacker is not able to manipulate the private key as liberally.

1 Introduction

In the paper "Keeping Up With the KEMs" [CDM23], Cremers et al. defined the notion of a KEM binding shared secret, ciphertext, and public key to each other. These types of binding properties are of interest in protocols that implicitly rely on stronger security guarantees than IND-CCA security, with some examples listed in the paper. Similar problems have been discussed at some length in the context of key committing AEADs, called "invisible salamanders", for example in [DGRW19] and [ADG+20].

Of interest for this short note is the scenario of a malicious attacker, i.e. an attacker which can freely chose all key material, as well as encapsulation entropy and decapsulation ciphertexts. In order to be MAL-BIND-K-CT in this scenario, if two parties (each of which are either encapsulating or decapsulating), are able to obtain the same shared secret, but either produce or were given different ciphertexts, the ciphertext is not considered bound to the shared secret.

Similarly, if the parties are able to obtain the same shared secret while using different public keys, the public key is not considered bound to the shared secret.

There are some trivial binding attacks on the attacker model in Figure 6 of [CDM23], due to decapsulating parties not actually using the public key at all. In order to repair this trivial attack, we need to slightly change the APIs.

That is instead of the API

$$\begin{aligned} \text{KeyGen}\colon 0 &\xrightarrow{R} \mathcal{S} \times \mathcal{P} \\ \text{Encaps}\colon \mathcal{P} &\xrightarrow{R} \mathbb{B}^{32} \times \mathcal{C} \\ \text{Decaps}\colon \mathcal{S} \times \mathcal{C} &\to \mathbb{B}^{32} \end{aligned}$$

which we refer to as the NIST API, we need to use

$$\begin{aligned} \text{KeyGen}\colon 0 &\xrightarrow{R} \mathcal{S} \\ \text{PrivateToPublic}\colon \mathcal{S} &\to \mathcal{P} \\ \text{Encaps}\colon \mathcal{P} &\xrightarrow{R} \mathbb{B}^{32} \times \mathcal{C} \\ \text{Decaps}\colon \mathcal{S} \times \mathcal{C} &\to \mathbb{B}^{32} \end{aligned}$$

In the HONEST and LEAK attacker models, these APIs are trivially the same. In the case of a malicious attacker, we do need to explicitly specify how a public key is obtained from a (potentially malformed) private key. Note that PrivateToPublic is non-trivial in the case of some KEMs. For example, in the case of RSA-KEM, if the private key only consists of private exponent and modulus, being able to compute the public exponent can be computationally infeasible. In the case of ML-KEM, though, this is not a problem, as the private key contains a copy of the public key, and we can define PrivateToPublic as returning this copy.

With this API, we can formulate the attacker model of [CDM23] in the case of a malicious attacker as such: In other words, in the case of decapsulating parties, the attacker is only allowed to choose the private key to be used, and the public key is generated honestly from this private key. This fixes the trivial MAL-BIND-P-PK attack where an attacker choosing mode $g = 1$ or $g = 2$ otherwise can set the public key of the decapsulating party arbitrarily.

When discussing mitigations, we need to further refine this API to include

Algorithm 1 MAL-BIND-P-Q$^{KEM}_{\mathcal{A}}$

$g, st \leftarrow \mathcal{A}(st)$
if $g = 1$ **then**
  $sk_0, sk_1, ct_0, ct_1 \leftarrow \mathcal{A}(st)$
  $pk_0 \leftarrow \text{PrivateToPublic}(sk_0)$
  $pk_1 \leftarrow \text{PrivateToPublic}(sk_1)$
  $k_0 \leftarrow \text{Decaps}(sk_0, ct_0)$
  $k_1 \leftarrow \text{Decaps}(sk_1, ct_1)$
else if $g = 2$ **then**
  $pk_0, sk_1, r_0, ct_1 \leftarrow \mathcal{A}(st)$
  $k_0, ct_0 \leftarrow \text{Encaps}(pk_0; r_0)$
  $pk_1 \leftarrow \text{PrivateToPublic}(sk_1)$
  $k_1 \leftarrow \text{Decaps}(sk_1, ct_1)$
else if $g \notin 1, 2$ **then**
  $pk_0, pk_1, r_0, r_1 \leftarrow \mathcal{A}(st)$
  $k_0, ct_0 \leftarrow \text{Encaps}(pk_0; r_0)$
  $k_1, ct_1 \leftarrow \text{Encaps}(pk_1; r_1)$
end if
if $k_0 = \perp \vee k_1 = \perp$ **then**
  return 0
end if
// $\mathcal{A}$ wins if $\neg((\forall x \in P.x_0 = x_1) \Rightarrow (\forall y \in Q.y_0 = y_1))$
return $\forall x \in P.x_0 = x_1 \wedge \exists y \in Q.y_0 \neq y_1$

explicit serialization:

$$\begin{aligned}
\text{KeyGen}&: 0 \xrightarrow{R} \mathcal{S} \\
\text{PrivateToPublic}&: \mathcal{S} \to \mathcal{P} \\
\text{Encaps}&: \mathcal{P} \xrightarrow{R} \mathbb{B}^{32} \times \mathcal{C} \\
\text{Decaps}&: \mathcal{S} \times \mathcal{C} \to \mathbb{B}^{32} \\
\text{MarshalPrivate}&: \mathcal{S} \to \mathbb{B}^{len_s} \\
\text{ParsePrivate}&: \mathbb{B}^{len_s} \to \mathcal{S} \\
\text{MarshalPublic}&: \mathcal{P} \to \mathbb{B}^{len_p} \\
\text{ParsePublic}&: \mathbb{B}^{len_p} \to \mathcal{P}
\end{aligned}$$

In other words, in this API we assume that $\mathcal{S}$ and $\mathcal{P}$ are opaque types that a caller cannot serialize without the help of the corresponding functions. This kind of API can be seen in real world implementations of ML-KEM, as it allows for better caching of intermediate values in case of repeated Encaps or Decaps calls on the same key material.

2 Counterexamples

We can slightly simplify the attacker model in Algorithm 1, by observing that the attack happens in two distinct phases. First the attacker selects the relevant keys, ciphertexts, and encapsulation entropies for the corresponding scenario, and then the honest parties perform the necessary encapsulation and decapsulation operations. This means, in order to give a counterexample, it suffices to just give concrete instantiations of the necessary values, without the attacker having to perform any interactive operations.

2.1 Counterexamples for MAL-BIND-K-CT

We can construct a MAL-BIND-K-CT counterexample both in the case of one party encapsulating and the other party decapsulating, as well as the case of both parties decapsulating. Recall that an ML-KEM private key is the tuple (s, t, ρ, h, z), with the corresponding public key being the tuple (t, ρ). In a well-formed private key, $H(t\|\rho) = h$ holds. Our attack focuses on violating this property, and tricking the decapsulating party to use a different hash.

We will focus on the scenario of party 0 encapsulating and party 1 decapsulating (i.e. $g = 2$), with the construction for both parties decapsulating being analogous. If the decapsulation does not hit the FO rejection flow, we have $K_0, r_0 = G(m_0\|H(pk_0))$ and $K_1, r_1 = G(m_1\|h_1)$. Since we need $K_0 = K_1$, this implies that $m_0 = m_1$ and $H(pk_0) = h_1$. Party 1 obtains m_1 via the decryption of c_1, so we can obtain equality by setting $c_1 = \text{Encrypt}(pk_1, m_0, r_1)$. This ciphertext conveniently also will not trigger the FO rejection path, as we also have $r_0 = r_1$ as a side effect of the shared key equality.

In summary, the attacker selects:

Algorithm 2 Attacker for MAL-BIND-K-CT

$sk_0 \leftarrow (s_0, t_0, \rho_0, h_0, z_0) \leftarrow \text{KeyGen}()$
$pk_0 \leftarrow \text{PrivateToPublic}(sk_0)$
$(s_1, t_1, \rho_1, h_1, z_1) \leftarrow \text{KeyGen}()$
$sk_1 \leftarrow (s_1, t_1, \rho_1, h_0, z_1)$
$m_0 \leftarrow \text{Rng}(32)$
$r \leftarrow G(m_0 \| h_0)$
$c_1 \leftarrow \text{Encrypt}(pk_1, m_0; r)$
return (pk_0, sk_1, m_0, c_1)

On the encapsulating side, the Algorithm 1 now sets

$$\begin{aligned} K_0, r_0 &= G(m_0 \| H(pk_0)) \\ c_0 &= \text{Encrypt}(pk_0, m_0, r_0) \end{aligned}$$

On the decapsulating side, we get

$$\begin{aligned} m_0 &= \text{Decrypt}(s_1, c_1) \\ K_1, r_1 &= G(m_0 \| h_0) \\ c_1' &= \text{Encrypt}(pk_1, m_0; r_1) \end{aligned}$$

Note that by definition of c_1, we have $c_1' = c_1$.

The main trick behind the attack is in the line $K_1, r_1 = G(m_1 \| h_0)$. Due to the attackers preparation of the private key, we trick the honest decapsulating party into using the hash corresponding to the encapsulating public key.

2.2 Counterexamples for MAL-BIND-K-PK

Algorithm 2 already provides a counterexample for MAL-BIND-K-PK as well as $pk_1 = \text{PrivateToPublic}(sk_1) \neq pk_0$ in that scenario.

We can obtain a further MAL-BIND-K-PK counterexample in the case of both parties decapsulating ($g = 1$). This attack focuses on the FO rejection flow, and prepares mal-formed private keys that agree on their FO rejection secret, but differ otherwise. The attacker can then choose a random ciphertext c, equal for both parties, and abuse the fact that FO rejection does not include the public key. In other words, the attacker sets:

One interesting observation about this counterexample is that the private keys appear well-formed, since an auditor who is incapable of finding preimages of random number generators is not able to tell that a single private key was generated dishonestly.

3 Mitigations

There are several possible mitigations that can be used to address the lack of binding in ML-KEM.

Algorithm 3 Attacker for MAL-BIND-K-PK

$(s_0, t_0, \rho_0, h_0, z_0) \leftarrow \text{KeyGen}()$
$(s_1, t_1, \rho_1, h_1, z_1) \leftarrow \text{KeyGen}()$
$z \leftarrow \text{Rng}(32)$
$c \leftarrow \text{Rng}(len_c)$
$sk_0 \leftarrow (s_0, t_0, \rho_0, h_0, z)$
$sk_1 \leftarrow (s_1, t_1, \rho_1, h_1, z)$
return (sk_0, sk_1, c, c)

3.1 Omiting the hash in the private key serialization

As a simple mitigation of the lack of MAL-BIND-K-CT, the hash h can simply be omitted from the serialized private key format. In other words, changing the private key into a tuple (s, t, ρ, z). The main reason for implementations to use explicit serialization APIs is the ability to compute the expansion of ρ into the matrix A in ParsePrivate and ParsePublic with the computation of $H(pk)$ being another step in ParsePublic. Adding this step to ParsePrivate has relatively minor performance overhead.

In the most performance critical scenario of using an ephemeral key pair, both the matrix A and the public key hash $H(pk)$ have to computed exactly once for both encapsulating and generating/decapsulating side. The encapsulating side will usually compute this value in ParsePublic, while the generating/decapsulating side does so in KeyGen, never calling MarshalPrivate or ParsePrivate.

In the case of this mitigation, one can prove MAL-BIND-K-CT for ML-KEM.

Proposition 1. *ML-KEM without cached public key hashes is MAL-BIND-K-CT.*

Proof. We only give the proof for an attacker that cannot find collisions in hash functions, and leave the usual procdedure of changing hash functions to random oracles as an exercise to the reader. We also only give the proof for the scenario of both parties decapsulating, with the other scenarios being analogous to the subcases of this scenario.

We parse the two private keys sk_0 and sk_1 given by the attacker as the tuples (s_0, t_0, ρ_0, z_0) and (s_1, t_1, ρ_1, z_1).

Case 1. Both parties rejecting. In this case the shared secrets K_i are computed as $K_i = J(z_i \| c_i)$. As we assume our attacker cannot find collisions in hash functions, this implies that $z_0 = z_1$ and $c_0 = c_1$. The latter was the required property for MAL-BIND-K-CT, completing this case.

Case 2. Party 0 rejecting, Party 1 accepting. In this case we have $K_0 = J(z_0 \| c_0)$ and $K_1, r_1 = G(m_1 \| H(pk_1))$, where m_1 was obtained by decrypting c_1. This means that achieving $K_0 = K_1$ is only possible by finding a collision accross domain separated hash functions, which we assume our attacker is incapable off, leading to a contradiction.

Case 3. Both parties accepting In this case we have $K_i, r_i = G(m_i \| H(pk_i))$. Due to the attacker being unable to find collisions, $K_0 = K_1$ implies $m_0 = m_1$, $pk_0 = pk_1$, and further $r_0 = r_1$. Both parties then compute the reconstructed ciphertext c_i' as $\text{Encrypt}(pk_i, m_i; r_i)$. As Encrypt with defined entropy is a deterministic function, and all arguments agree between parties, this means $c_0' = c_1'$. Since we assume that both parties are accepting, we know that $c_i = c_i'$, and therefore $c_0 = c_1$ as required. □

Note that the MAL-BIND-K-PK attack of Algorithm 3 remains, so care must be taken that a given protocol is not relying on that property in the $g = 1$ case.

3.2 Validating private keys

The manipulation of the private key in Algorithm 2 is easy to detect, since it is internally inconsistent. Another mitigation option is to check for this inconsistency, by requiring that the implementor checks $H(pk) = h$ when deserializing the private key. This allows an implementation to conform to the ML-KEM standard, without being vulnerable to MAL-BIND-K-CT attacks.

Note that, as before, the MAL-BIND-K-PK attack of Algorithm 3 remains, so care must be taken that a given protocol is not relying on that property in the $g = 1$ case.

3.3 Storing private keys as seeds

The serialization format of the private key can in general be seen as a cache of the output of KeyGen under a given seed. If deserialization instead recomputed the whole private key via the initial seed, computing a malformed private key would be impossible. This option is also the only option to obtain MAL-BIND-K-PK, as long as z is not drawn independently of the rest of the seed. Unfortunately, this would require further modification of ML-KEM, computing σ, ρ, and z from a single seed being used in a KDF instead of, as is currently done, using two separate seeds.

In other words, this would modify KeyGen to be:

Algorithm 4 KeyGen for private keys serialized as seed

$g \leftarrow \text{Rng}(32)$
return g

and define ParsePrivate via

Algorithm 5 ParsePrivate when private keys are serialized as seed

$\sigma, \rho, z \leftarrow \text{KDF}(g)$
$sk = \text{KeyGen}_{NIST}(\sigma, \rho, z)$
return sk

Proposition 2. *ML-KEM with private keys being serialized as a single random seed is MAL-BIND-K-PK.*

Proof. We again only show a sketch of the proof without random oracles replacing the hash functions, and only prove the scenario of both parties decapsulating, with the other cases being analogous.

Case 1. Both parties rejecting. We have $K_i = J(z_i \| c_i)$, so $K_0 = K_1$ implies $z_0 = z_1$ and $c_0 = c_1$ due to the attacker not finding collisions. $\sigma_i, \rho_i, z_i = KDF(g_i)$ means that $z_0 = z_1$ implies $g_0 = g_1$. This in turn implies $\sigma_0 = \sigma_1$ and $\rho_0 = \rho_1$, leading to the public keys being equal as well.

Case 2. Party 0 accepts, Party 1 rejects. We have $K_0 = J(z_0 \| c_0)$ and $K_1, r_1 = G(m_1, H(pk_1))$. Like in the case of MAL-BIND-K-CT, $K_0 = K_1$ leads to a contradiction due to the domain separation bewtween the hash functions J and G.

Case 3. Both parties accepting. We have $K_i, r_i = G(m_i \| H(pk_i))$. Due to the attackers inability to find hash function collisions, $K_0 = K_1$ implies that $pk_0 = pk_1$, as required for the non-trivial notion of MAL-BIND-K-PK. □

3.4 Mitigations at the protocol layer

Binding properties were not required in the initial NIST competition and are not usually seen as a required property of a secure KEM. An argument can be made that binding properties already only matter in specific scenarios, and that they are relatively straightforward to mitigate at the protocol layer, by making sure that ciphertext and public key are always part of the key derivation of a key agreement. This is best practice for key agreement protocols, but unfortunately still a common oversight in protocol design.

Since all attacks on ML-KEM misbinding public key or ciphertext rely on manipulated private keys, only protocols in which the private key for one reason or another cannot be fully trusted are at risks. Protocols that receive the private key from a third party, or that require revealing of the private key under certain conditions, should use the seed of the private key as a representation of the key, and rerun KeyGen instead of relying on cached representations of the private key material.

4 Conclusions

The binding properties defined in [CDM23] depend very tightly with the formats used for the private key serialization. Outside of attacks on this aspect of ML-KEM, we obtain very strong binding properties for free. In particular, we do not have to hash the ciphertext or the public key into the shared secret more than the algorithms in ML-KEM already do, showing that the simplified key derivation NIST chose for ML-KEM over Kyber Round 3 was a correct choice, only improving performance without compromising security.

There are still misbinding problems left in ML-KEM, though, but they can be fixed with a relatively minor change in how private keys are serialized. Any

protocol in which revealing a private key or accepting private keys from a third party is part of the protocol flow, should use the seed used to generate the private key to accomplish these tasks instead of the serialized private key format of ML-KEM.

References

[ADG+20] Ange Albertini, Thai Duong, Shay Gueron, Stefan Kölbl, Atul Luykx, and Sophie Schmieg. How to abuse and fix authenticated encryption without key commitment. Cryptology ePrint Archive, Paper 2020/1456, 2020. https://eprint.iacr.org/2020/1456.

[CDM23] Cas Cremers, Alexander Dax, and Niklas Medinger. Keeping up with the kems: Stronger security notions for kems and automated analysis of kem-based protocols. Cryptology ePrint Archive, Paper 2023/1933, 2023. https://eprint.iacr.org/2023/1933.

[DGRW19] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. Cryptology ePrint Archive, Paper 2019/016, 2019. https://eprint.iacr.org/2019/016.