Kemeleon:

Elligator-like Obfuscation for Post-Quantum Cryptography

Felix Günther
IBM Research – Zurich

Michael Rosenberg
Cloudflare

Douglas Stebila
University of Waterloo

Shannon Veitch
ETH Zurich

Uniform Representations

Internet protocols **hide metadata** to protect user privacy, dissuade protocol fingerprinting, and prevent network ossification

Uniform Representations

Internet protocols **hide metadata** to protect user privacy, dissuade protocol fingerprinting, and prevent network ossification

- TLS 1.3 Encrypted Client Hello, QUIC, obfs4, Shadowsocks, ...
- "Fully encrypted" protocols, with **obfuscated key exchange**

Uniform Representations

Internet protocols **hide metadata** to protect user privacy, dissuade protocol fingerprinting, and prevent network ossification

- TLS 1.3 Encrypted Client Hello, QUIC, obfs4, Shadowsocks, ...
- "Fully encrypted" protocols, with **obfuscated key exchange**

Some PAKEs need to operate on random bytestrings

Uniform Representations

Internet protocols **hide metadata** to protect user privacy, dissuade protocol fingerprinting, and prevent network ossification

- TLS 1.3 Encrypted Client Hello, QUIC, obfs4, Shadowsocks, ...
- "Fully encrypted" protocols, with **obfuscated key exchange**

Some PAKEs need to operate on random bytestrings

Previously: Elligator maps elliptic curve public keys to random bytestrings

Uniform Representations

Internet protocols **hide metadata** to protect user privacy, dissuade protocol fingerprinting, and prevent network ossification

- TLS 1.3 Encrypted Client Hello, QUIC, obfs4, Shadowsocks, ...
- "Fully encrypted" protocols, with **obfuscated key exchange**

Some PAKEs need to operate on random bytestrings

Previously: Elligator maps elliptic curve public keys to random bytestrings

Uniform Representations

Internet protocols **hide metadata** to protect user privacy, dissuade protocol fingerprinting, and prevent network ossification

- TLS 1.3 Encrypted Client Hello, QUIC, obfs4, Shadowsocks, ...
- "Fully encrypted" protocols, with **obfuscated key exchange**

Some PAKEs need to operate on random bytestrings

Previously: Elligator maps elliptic curve public keys to random bytestrings

What about post-quantum key exchanges? Can use Saber or FrodoKEM.

Uniform Representations

Internet protocols **hide metadata** to protect user privacy, dissuade protocol fingerprinting, and prevent network ossification

- TLS 1.3 Encrypted Client Hello, QUIC, obfs4, Shadowsocks, ...
- "Fully encrypted" protocols, with **obfuscated key exchange**

Some PAKEs need to operate on random bytestrings

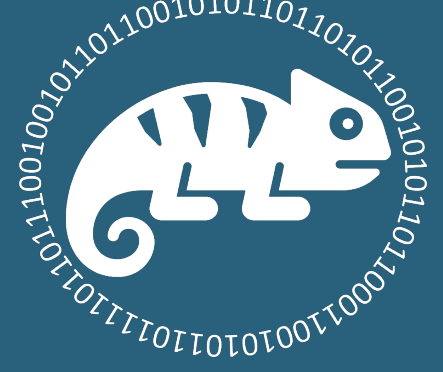
Previously: Elligator maps elliptic curve public keys to random bytestrings

What about post-quantum key exchanges? Can use Saber or FrodoKEM.

What about standardized post-quantum key exchanges — ML-KEM?

Overview

- **Kemeleon Encoding**
- ML-Kemeleon: Obfuscated PQ KEM
- OEINC: Obfuscated KEM Combiner
- Hybrid Applications:
  - Obfuscated Key Exchange
  - Password Authenticated Key Exchange (with adaptive security)



Byte Distribution of ML-KEM-768 Public Keys

ML-KEM public keys: Polynomials with coefficients mod q=3329, and a random seed

Image inspired by: Jack Wampler

Byte Distribution of ML-KEM-768 Public Keys

ML-KEM public keys: Polynomials with coefficients mod q=3329, and a random seed

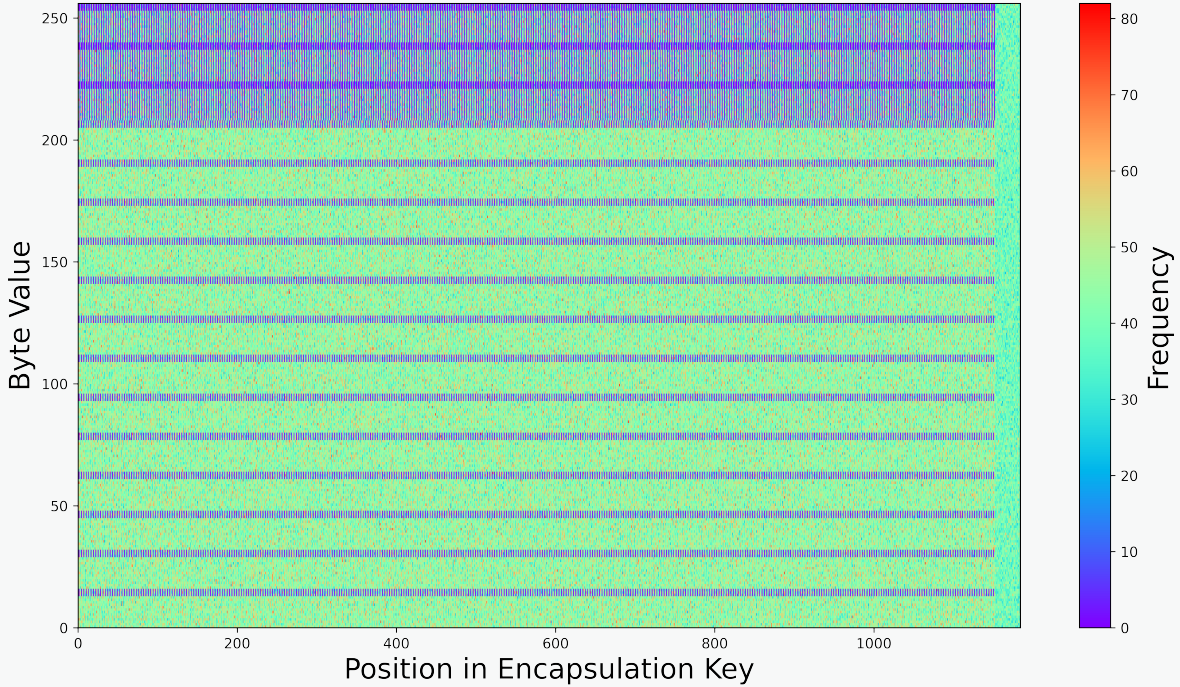


Image inspired by: Jack Wampler

Byte Distribution of ML-KEM-768 Public Keys

ML-KEM public keys: Polynomials with coefficients mod q=3329, and a random seed

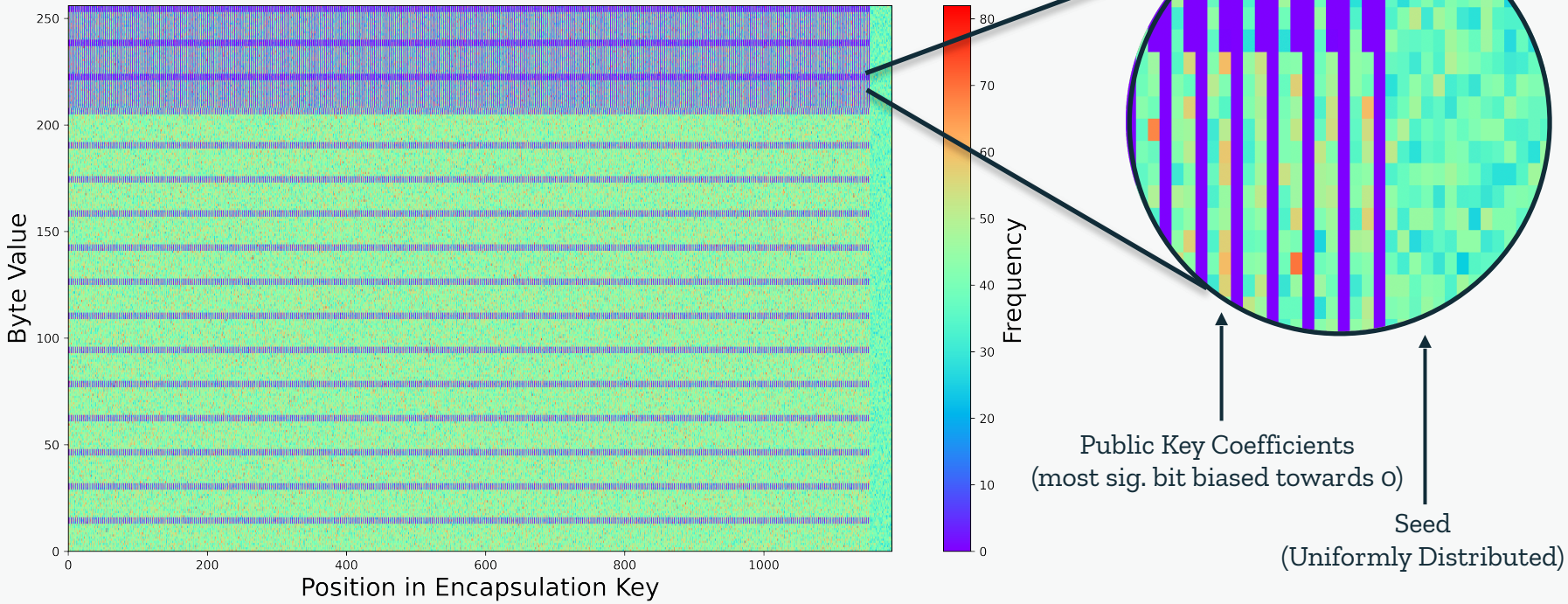


Image inspired by: Jack Wampler

Kemeleon: Rejection-Sampling Public Keys

ML-KEM public keys

$[\ a_1\ ][\ a_2\ ][\ a_3\ ]\ \ldots\ [\ a_b\ ]$ (a_i is a number mod q=3329)

Kemeleon: Rejection-Sampling Public Keys

ML-KEM public keys

$[\ a_1\ ][\ a_2\ ][\ a_3\ ]\ \ldots\ [\ a_b\ ]$ (a_i is a number mod q=3329)

1. Accumulate into one **big integer**

$[\ A = a_1 + a_2 \times q + a_3 \times q^3 + \ldots + a_b \times q^{b-1}\ ]$ (A is a number mod q^{b-1})

Kemeleon: Rejection-Sampling Public Keys

ML-KEM public keys

$[\ a_1\ ][\ a_2\ ][\ a_3\ ]\ \ldots\ [\ a_b\ ]$ (a_i is a number mod q=3329)

1. Accumulate into one **big integer**
2. Rejection sampling: **reject if msb is 1**

$[\ A = a_1 + a_2 \times q + a_3 \times q^3 + \ldots + a_b \times q^{b-1}\ ]$ (A is a number mod q^{b-1})

Most sig. bit still biased towards 0

Kemeleon: Rejection-Sampling Public Keys

ML-KEM public keys

$[\ a_1\ ][\ a_2\ ][\ a_3\ ]\ \ldots\ [\ a_b\ ]$ (a_i is a number mod q=3329)

Encoded public keys **~2.5% smaller** than regular (19/28/38 bytes for ML-KEM-512/768/1024)

1. Accumulate into one **big integer**
2. Rejection sampling: **reject if msb is 1**

$[\ A = a_1 + a_2 \times q + a_3 \times q^3 + \ldots + a_b \times q^{b-1}\ ]$ (A is a number mod q^{b-1})

Most sig. bit still biased towards 0

Kemeleon: Rejection-Sampling Public Keys

ML-KEM public keys

$[\ a_1\ ][\ a_2\ ][\ a_3\ ]\ \ldots\ [\ a_b\ ]$ (a_i is a number mod q=3329)

1. Accumulate into one **big integer**
2. Rejection sampling: **reject if msb is 1**

Encoded public keys **~2.5% smaller** than regular (19/28/38 bytes for ML-KEM-512/768/1024)

MLKEM-768 **likelihood of rejection is 17%** (Elligator likelihood of rejection is ~50%)

$[\ A = a_1 + a_2 \times q + a_3 \times q^3 + \ldots + a_b \times q^{b-1}\ ]$ (A is a number mod q^{b-1})

Most sig. bit still biased towards 0

Distribution of ML-KEM-768 Ciphertext Coefficients

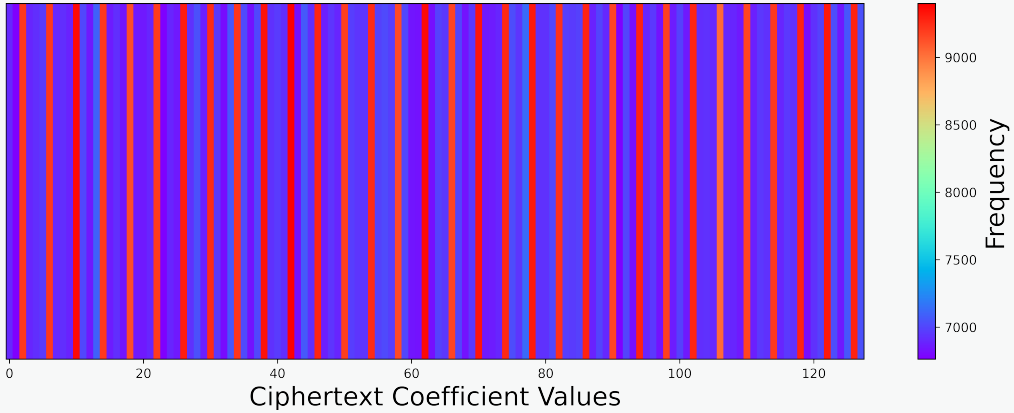
ML-KEM ciphertexts

Vector of polynomials, *compressed* before returned by Encap

Distribution of ML-KEM-768 Ciphertext Coefficients

ML-KEM ciphertexts

Vector of polynomials, *compressed* before returned by Encap

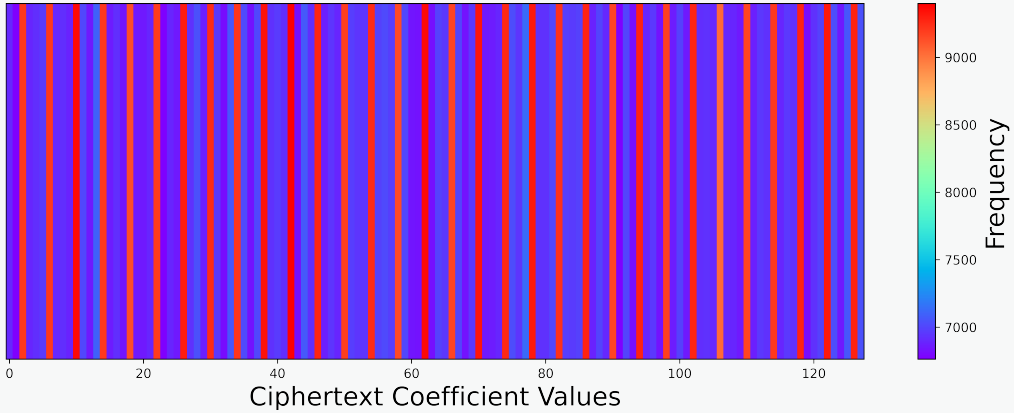


Compression step in Encap performs rounding which results in a non-uniform ciphertext distribution.

Distribution of ML-KEM-768 Ciphertext Coefficients

ML-KEM ciphertexts

Vector of polynomials, *compressed* before returned by Encap



Compression step in Encap performs rounding which results in a non-uniform ciphertext distribution.

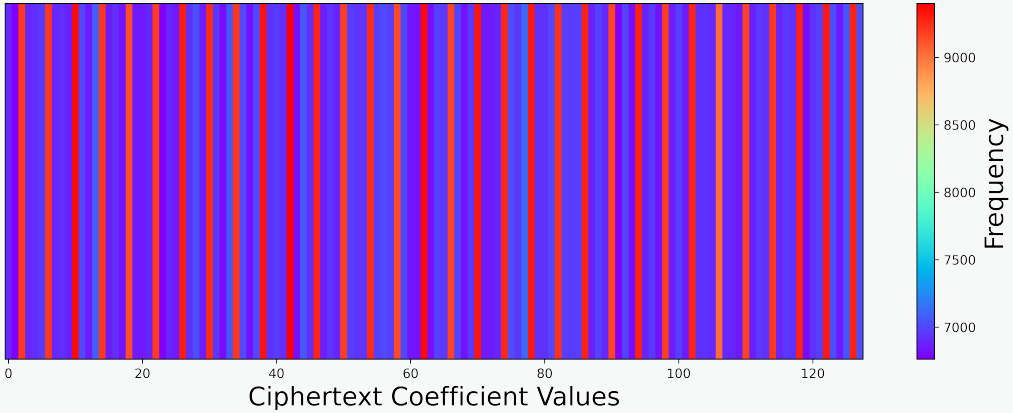
Kemeleon encoding for ciphertexts:

1. Decompress and "recover" randomness from ciphertexts
2. Use same rejection-sampling method as was used for public keys

Distribution of ML-KEM-768 Ciphertext Coefficients

ML-KEM ciphertexts

Vector of polynomials, *compressed* before returned by Encap



Compression step in Encap performs rounding which results in a non-uniform ciphertext distribution.

Kemeleon encoding for ciphertexts:

1. Decompress and "recover" randomness from ciphertexts
2. Use same rejection-sampling method as was used for public keys

Encoded ciphertexts are **6-15% larger** than regular (109/164/90 bytes for ML-KEM-512/768/1024)

Kemeleon without Rejection

Applying techniques from Tibouchi 2014 (Elligator²):

1. Take the accumulated integer, $\mathbf{A}$ (mod $\mathbf{q^{b-1}}$), from the original encoding with byte length < n

$[\ A = a_1 + a_2 \times q + a_3 \times q^3 + \ldots + a_b \times q^{b-1}\ ]$ (A is a number mod q^{b-1}, msb still biased towards 0)

Kemeleon without Rejection

Applying techniques from Tibouchi 2014 (Elligator²):

1. Take the accumulated integer, $\mathbf{A}$ (mod $\mathbf{q^{b-1}}$), from the original encoding with byte length < n
2. Add random value $\mathbf{r}$ such that result, $\mathbf{R = A + r}$, has byte length n + 32 (or alternative value, depending on security parameters)

$\mathbf{[\ A = a_1 + a_2 \times q + a_3 \times q^3 + ... + a_b \times q^{b-1}\ ]}$ (A is a number mod q^{b-1}, msb still biased towards 0)

$\mathbf{[\ R = A + r\ ]}$ (r random s.t. R is a number modulo some power of 2)

Kemeleon without Rejection

Applying techniques from Tibouchi 2014 (Elligator²):

1. Take the accumulated integer, **A** (mod $\mathbf{q^{b-1}}$), from the original encoding with byte length < n
2. Add random value **r** such that result, **R = A + r**, has byte length n + 32 (or alternative value, depending on security parameters)

Decoding: Take the result R modulo q^{b-1}, to recover A, and decode as usual

[$A = a_1 + a_2 \times q + a_3 \times q^3 + ... + a_b \times q^{b-1}$] (A is a number mod q^{b-1}, msb still biased towards 0)

[R = A + r] (r random s.t. R is a number modulo some power of 2)

Kemeleon without Rejection

Applying techniques from Tibouchi 2014 (Elligator²):

1. Take the accumulated integer, **A** (mod $\mathbf{q^{b-1}}$), from the original encoding with byte length < n
2. Add random value **r** such that result, **R = A + r**, has byte length n + 32 (or alternative value, depending on security parameters)

Encoded public keys are **~ same size** as in standard ML-KEM.

Likelihood of rejection is **0%**

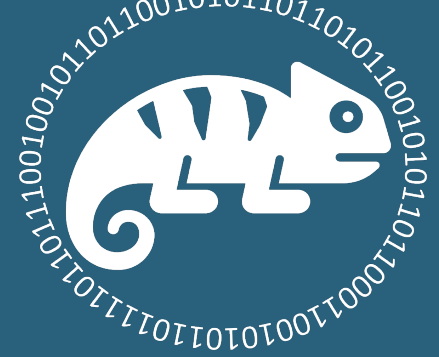
Decoding: Take the result R modulo q^{b-1}, to recover A, and decode as usual

[A = a_1 + a_2×q + a_3×q^3 + ... + a_b×q^{b-1}] (A is a number mod q^{b-1}, msb still biased towards 0)

[R = A + r] (r random s.t. R is a number modulo some power of 2)

Overview

- Kemeleon Encoding
- **ML-Kemeleon: Obfuscated PQ KEM**
- OEINC: Obfuscated KEM Combiner
- Hybrid Applications:
  - Obfuscated Key Exchange
  - Password Authenticated Key Exchange (with adaptive security)



Using Kemeleon with ML-KEM: an OKEM

ML-KEM can naturally be combined with Kemeleon (by encoding public keys and ciphertexts) to obtain an *Obfuscated KEM* (OKEM)

ML-KEM + Kemeleon = ML-Kemeleon

Using Kemeleon with ML-KEM: an OKEM

ML-KEM can naturally be combined with Kemeleon (by encoding public keys and ciphertexts) to obtain an *Obfuscated KEM* (OKEM)

ML-KEM + Kemeleon = ML-Kemeleon

OKEM Security Properties:

- **IND-CCA**: indistinguishability of shared secrets

Using Kemeleon with ML-KEM: an OKEM

ML-KEM can naturally be combined with Kemeleon (by encoding public keys and ciphertexts) to obtain an *Obfuscated KEM* (OKEM)

ML-KEM + Kemeleon = ML-Kemeleon

OKEM Security Properties:

- **IND-CCA**: indistinguishability of shared secrets
- **SPR-CCA (strong pseudorandomness)**: indistinguishability of shared secrets, random/ simulatable ciphertexts
    - implies anonymity (ANO-CCA)

Using Kemeleon with ML-KEM: an OKEM

ML-KEM can naturally be combined with Kemeleon (by encoding public keys and ciphertexts) to obtain an *Obfuscated KEM* (OKEM)

ML-KEM + Kemeleon = ML-Kemeleon

OKEM Security Properties:

- **IND-CCA**: indistinguishability of shared secrets
- **SPR-CCA (strong pseudorandomness)**: indistinguishability of shared secrets, random/ simulatable ciphertexts
    - implies anonymity (ANO-CCA)
- **Ciphertext and Public Key Uniformity**: indistinguishable from random bit strings

Using Kemeleon with ML-KEM: an OKEM

ML-KEM can naturally be combined with Kemeleon (by encoding public keys and ciphertexts) to obtain an *Obfuscated KEM* (OKEM)

ML-KEM + Kemeleon = ML-Kemeleon

OKEM Security Properties:

- **IND-CCA**: indistinguishability of shared secrets
- **SPR-CCA (strong pseudorandomness)**: indistinguishability of shared secrets, random/ simulatable ciphertexts
    - implies anonymity (ANO-CCA)
- **Ciphertext and Public Key Uniformity**: indistinguishable from random bit strings

ML-Kemeleon Properties:

- **IND-CCA**: IND-CCA of ML-KEM
- **SPR-CCA**: SPR-CCA of ML-KEM and ciphertext uniformity
- **Ciphertext Uniformity**: SPR-CCA of ML-KEM
- **Public Key Uniformity**: reduces to MLWE

\+ small loss from rejection rates in each case

Using Kemeleon with ML-KEM: an OKEM

ML-KEM can naturally be combined with Kemeleon (by encoding public keys and ciphertexts) to obtain an *Obfuscated KEM* (OKEM)

ML-KEM + Kemeleon = ML-Kemeleon

OKEM Security Properties:

- **IND-CCA**: indistinguishability of shared secrets
- **SPR-CCA (strong pseudorandomness)**: indistinguishability of shared secrets, random/ simulatable ciphertexts
    - implies anonymity (ANO-CCA)
- **Ciphertext and Public Key Uniformity**: indistinguishable from random bit strings

ML-Kemeleon Properties:

- **IND-CCA**: IND-CCA of ML-KEM
- **SPR-CCA**: SPR-CCA of ML-KEM and ciphertext uniformity
- **Ciphertext Uniformity**: SPR-CCA of ML-KEM
- **Public Key Uniformity**: reduces to MLWE

\+ small loss from rejection rates in each case

Note: while Elligator is statistically uniform, Kemeleon relies on MLWE assumption.

Using Kemeleon

Dos!

- Consider a constant-time implementation for big integer arithmetic, if this is in your threat model (also, consider timing side channels due to rejection sampling)

Using Kemeleon

Dos!

- Consider a constant-time implementation for big integer arithmetic, if this is in your threat model (also, consider timing side channels due to rejection sampling)

Don’ts!

- Use randomness derived from the KEM shared secret to seed the encoding algorithm (i.e., careful with key separation)
- Reveal randomness used for the encoding algorithm (i.e., randomness must be kept secret)

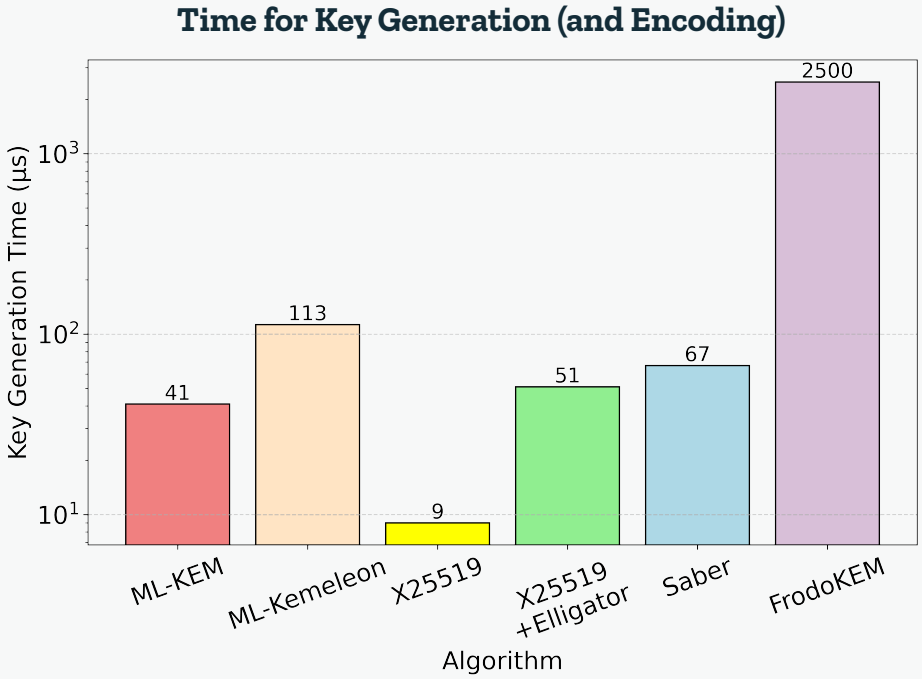
Using Kemeleon

Dos!

- Consider a constant-time implementation for big integer arithmetic, if this is in your threat model (also, consider timing side channels due to rejection sampling)

Don'ts!

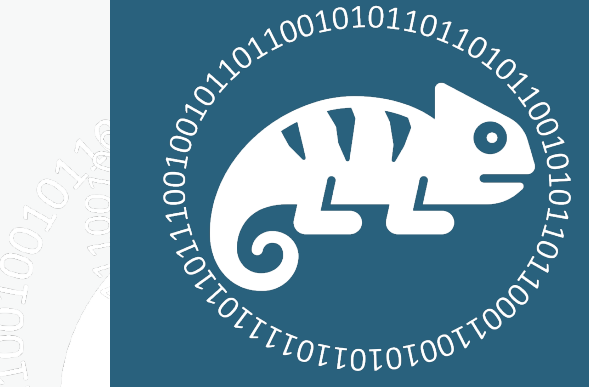
- Use randomness derived from the KEM shared secret to seed the encoding algorithm (i.e., careful with key separation)
- Reveal randomness used for the encoding algorithm (i.e., randomness must be kept secret)



Constant-time Kemeleon implementation, non-rejection variant, NIST security level 3

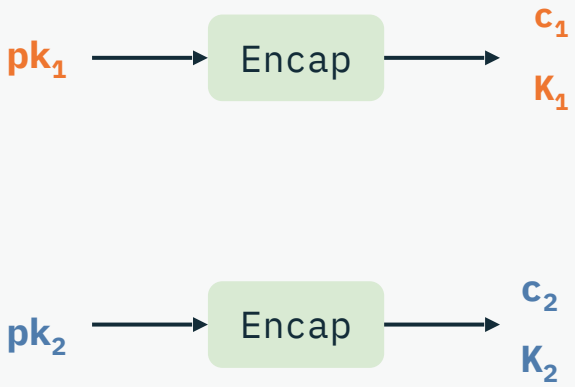
Overview

- Kemeleon Encoding
- ML-Kemeleon: Obfuscated PQ KEM
- **OEINC: Obfuscated KEM Combiner**
- Hybrid Applications:
  - Obfuscated Key Exchange
  - Password Authenticated Key Exchange (with adaptive security)

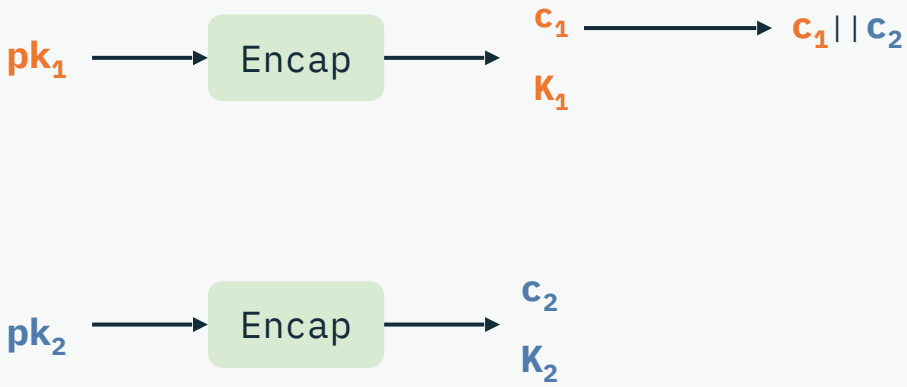


Hybrid KEMs: The Parallel Approach

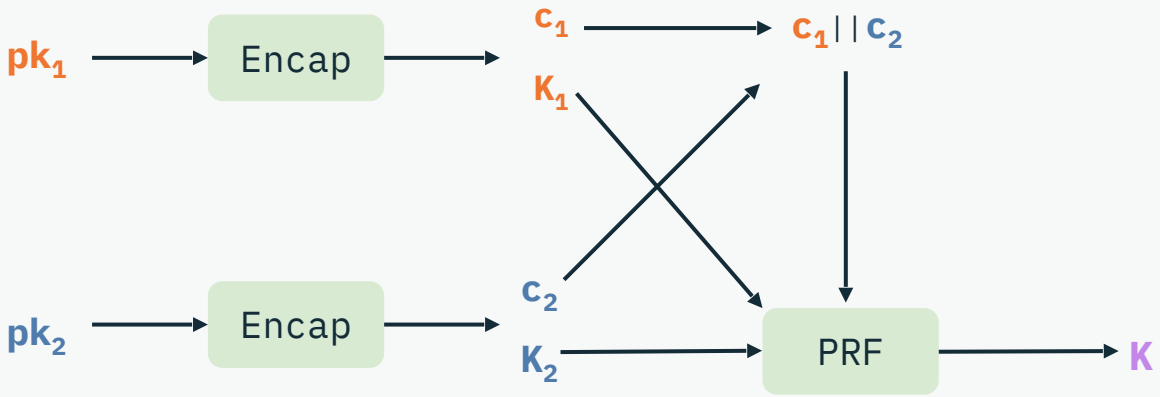
Hybrid KEMs: The Parallel Approach



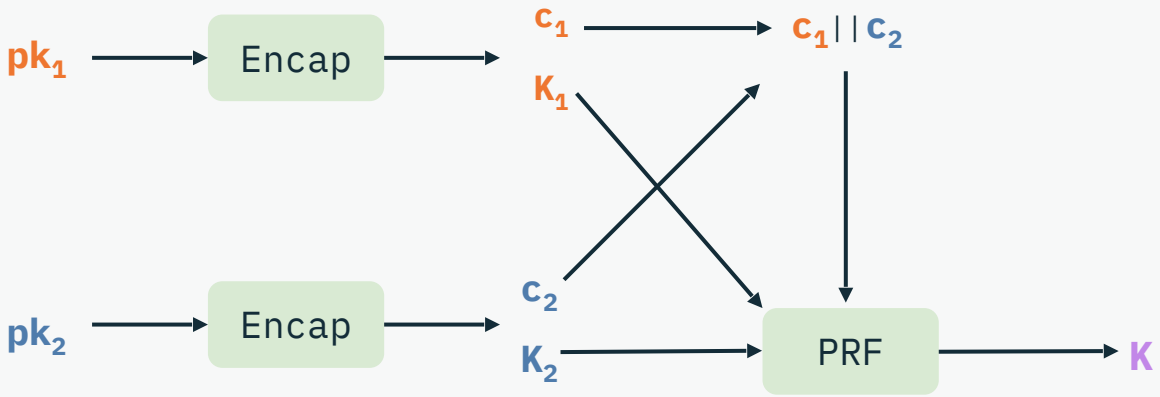
Hybrid KEMs: The Parallel Approach



Hybrid KEMs: The Parallel Approach

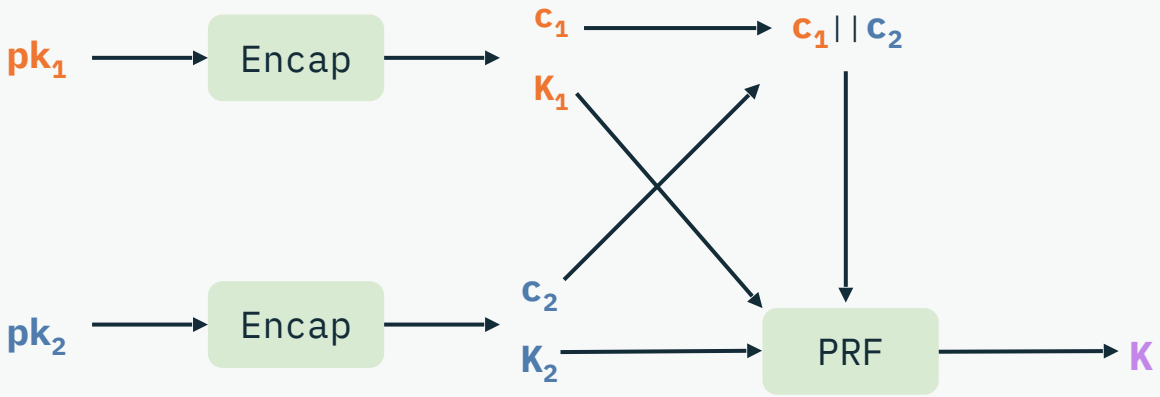


Hybrid KEMs: The Parallel Approach



Approach used in hybrid TLS 1.3, Xyber, X-Wing, ...

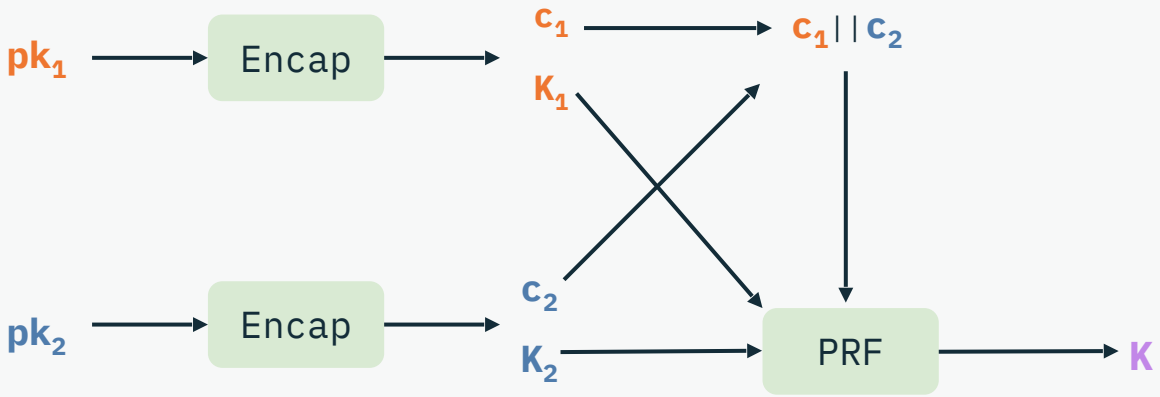
Hybrid KEMs: The Parallel Approach



Approach used in hybrid TLS 1.3, Xyber, X-Wing, ...

- ☑ **Hybrid IND-CCA**

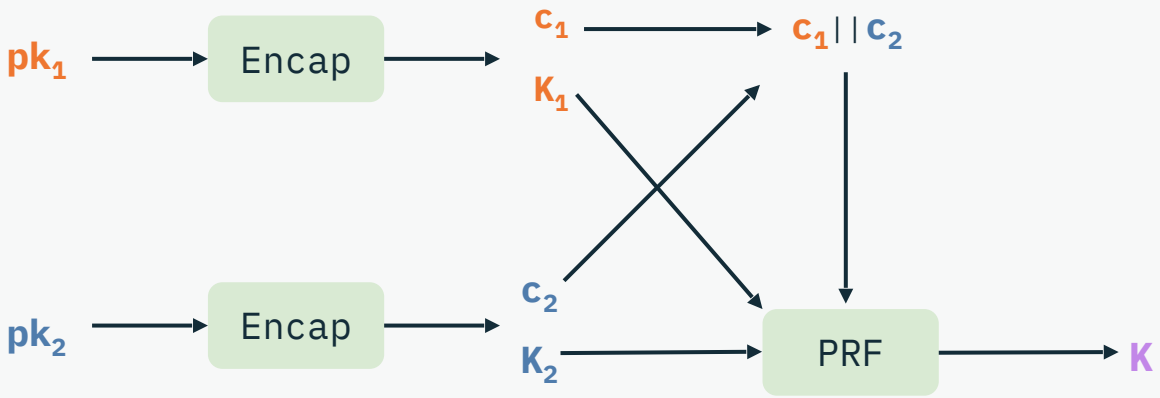
Hybrid KEMs: The Parallel Approach



Approach used in hybrid TLS 1.3, Xyber, X-Wing, …

- ☑ **Hybrid IND-CCA**
- ☒ **Hybrid Obfuscation (also, SPR-CCA, which implies anonymity)**

Hybrid KEMs: The Parallel Approach

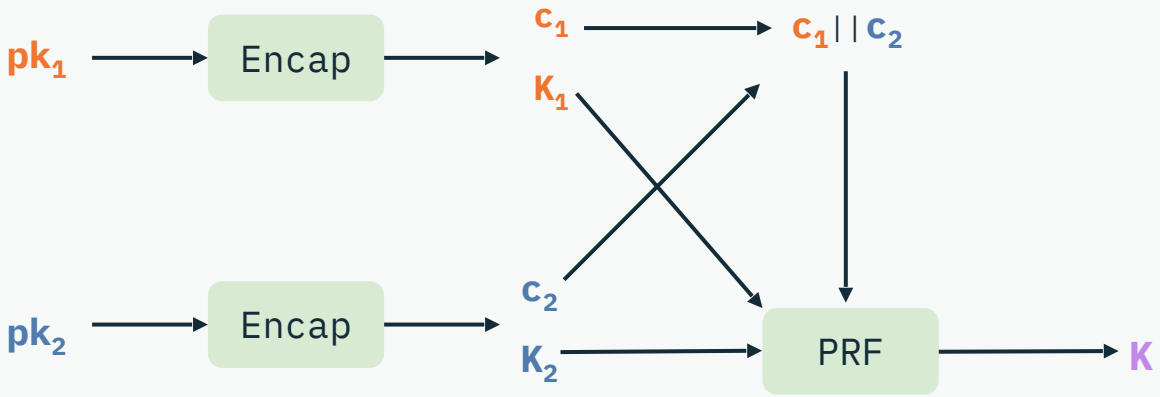


Suffices to distinguish **either** ciphertext

Approach used in hybrid TLS 1.3, Xyber, X-Wing, …

- ☑ **Hybrid IND-CCA**
- ☒ **Hybrid Obfuscation (also, SPR-CCA, which implies anonymity)**

Hybrid KEMs: The Parallel Approach



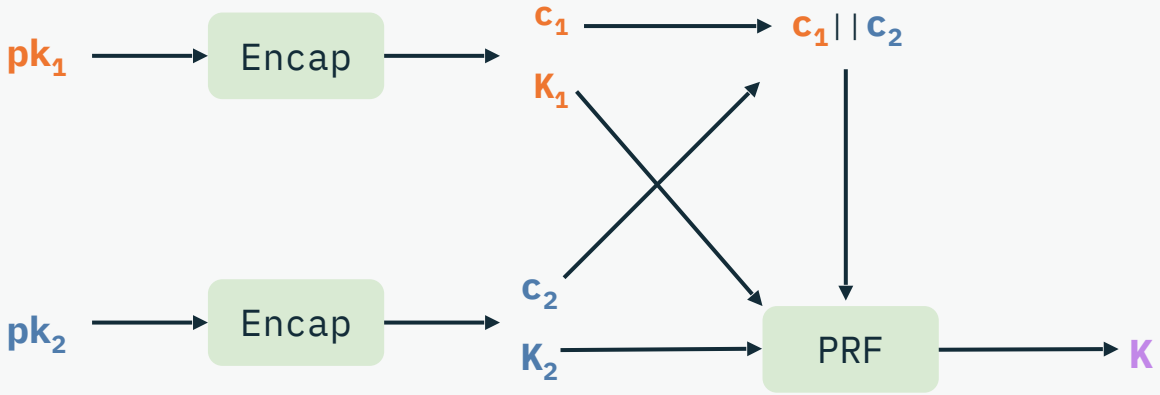
Suffices to distinguish **either** ciphertext

Sometimes c_1 is **unconditionally uniform**

Approach used in hybrid TLS 1.3, Xyber, X-Wing, …

- ☑ **Hybrid IND-CCA**
- ☒ **Hybrid Obfuscation (also, SPR-CCA, which implies anonymity)**

Hybrid KEMs: The Parallel Approach



Suffices to distinguish **either** ciphertext

Sometimes c_1 is **unconditionally uniform**

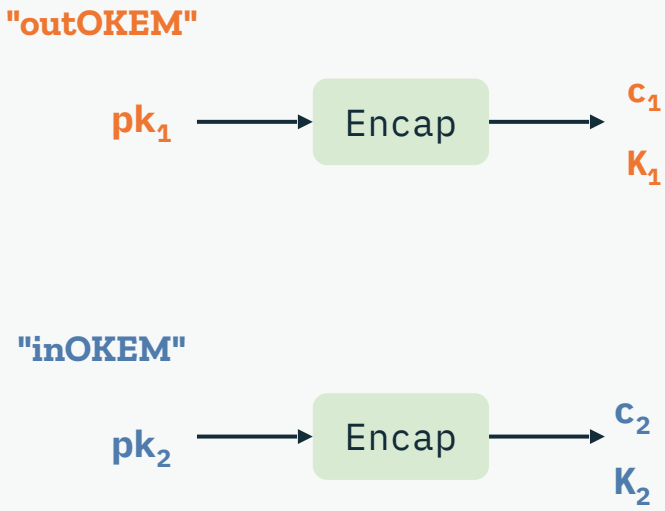
Can we use c_1 to encrypt c_2?

Approach used in hybrid TLS 1.3, Xyber, X-Wing, …

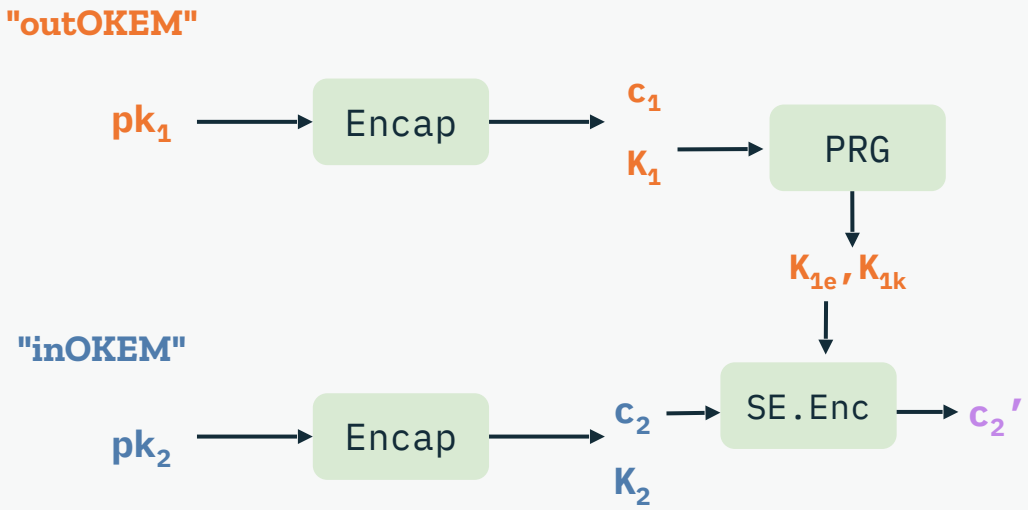
- ☑ **Hybrid IND-CCA**
- ☒ **Hybrid Obfuscation (also, SPR-CCA, which implies anonymity)**

Outer-Encrypts-Inner Nested Combiner (OEINC)

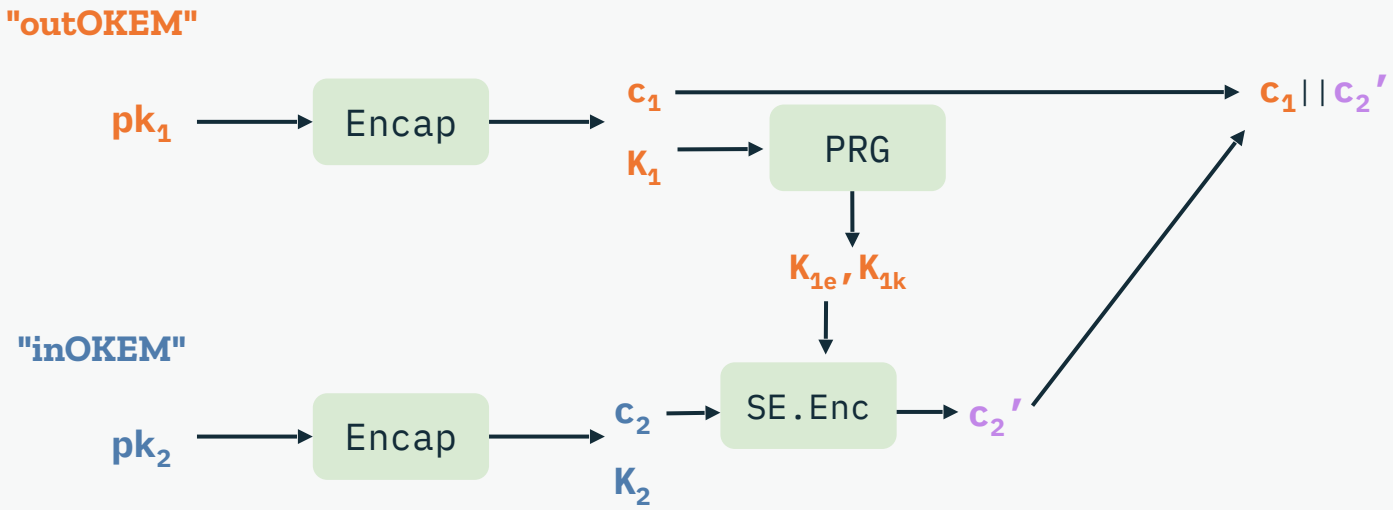
Outer-Encrypts-Inner Nested Combiner (OEINC)



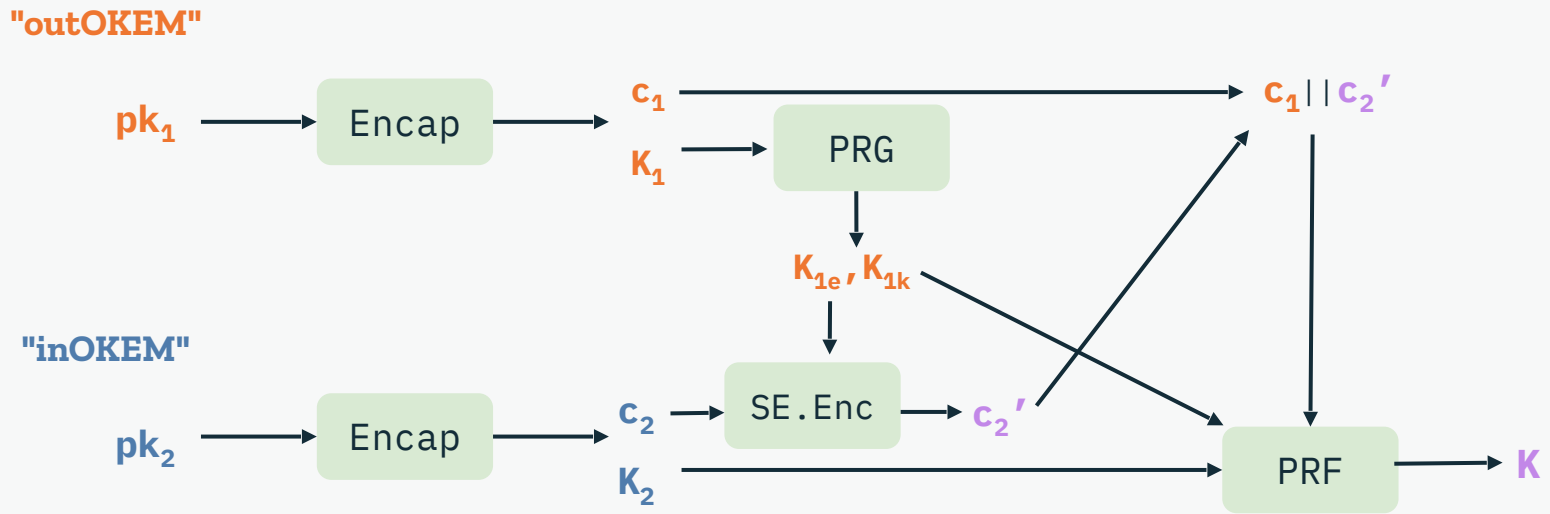
Outer-Encrypts-Inner Nested Combiner (OEINC)



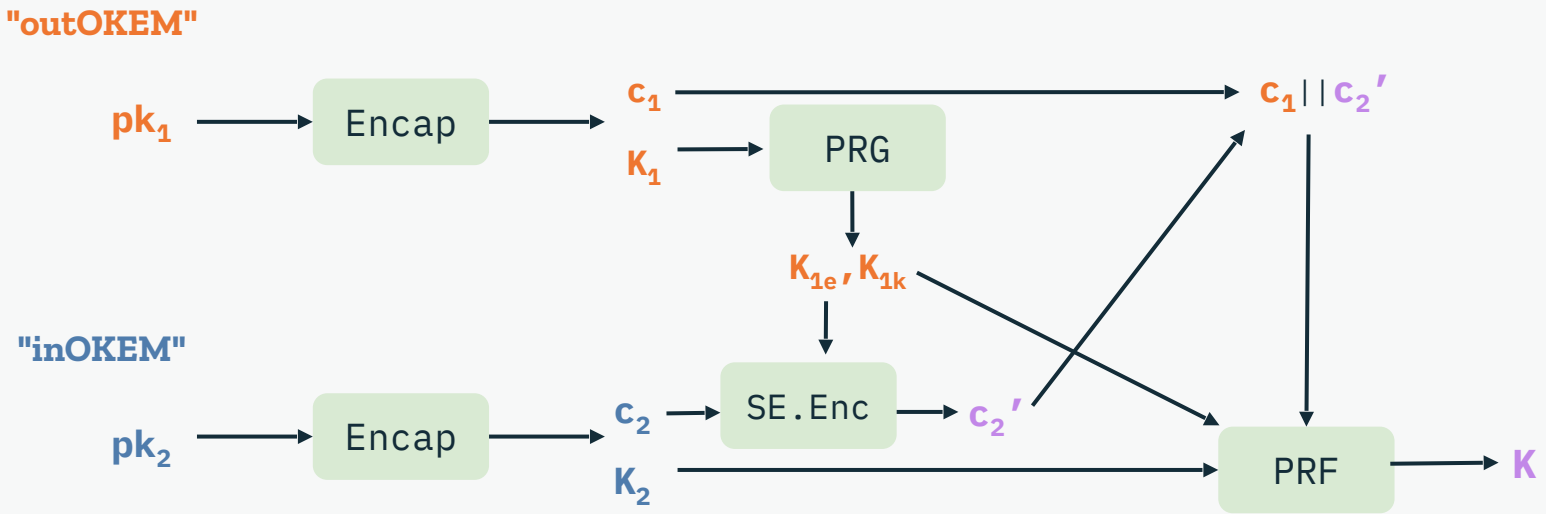
Outer-Encrypts-Inner Nested Combiner (OEINC)



Outer-Encrypts-Inner Nested Combiner (OEINC)

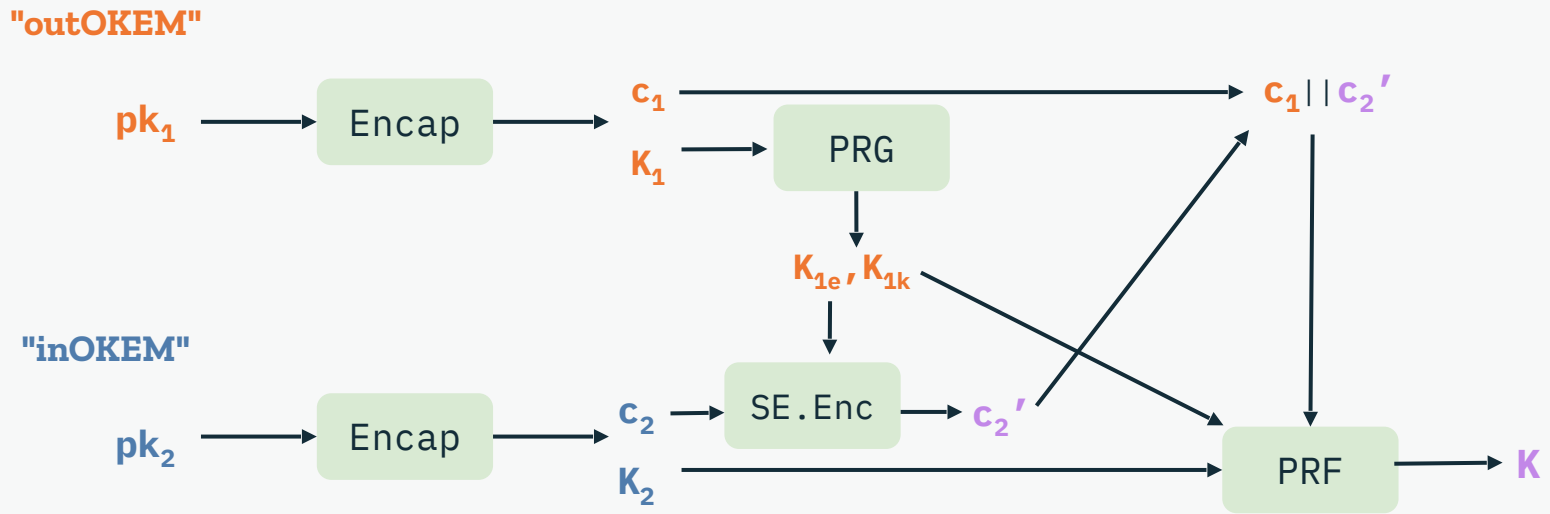


Outer-Encrypts-Inner Nested Combiner (OEINC)



☑ **Hybrid IND-CCA**

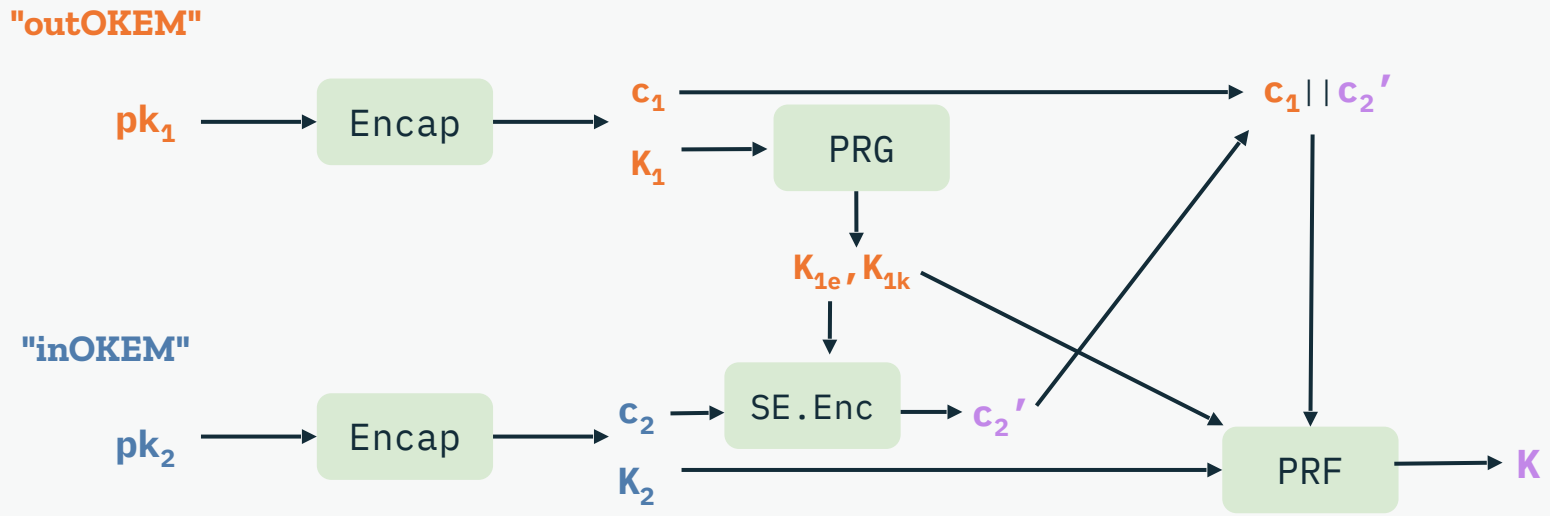
Outer-Encrypts-Inner Nested Combiner (OEINC)



☑ **Hybrid IND-CCA**

☑ **Low overhead: 1 PRG + 1 XOR**

Outer-Encrypts-Inner Nested Combiner (OEINC)



- ☑ **Hybrid IND-CCA**
- ☑ **Low overhead: 1 PRG + 1 XOR**
- ☑ **Hybrid Obfuscation (also, SPR-CCA, which implies anonymity)**

Instantiating OEINC

Instantiating OEINC

Security Properties

Instantiating OEINC

Security Properties

Requires:

Instantiating OEINC

Security Properties

Requires:

- outOKEM must have **statistical strong ciphertext uniformity** (ciphertexts must look uniform, even if you know sk, pk)

Instantiating OEINC

Security Properties

Requires:

- outOKEM must have **statistical strong ciphertext uniformity** (ciphertexts must look uniform, even if you know sk, pk)

outOKEM can be DHKEM

$sk = x \qquad pk = xG$

$ct = \text{Elligator2}(r \cdot pk)$

Instantiating OEINC

Security Properties

Requires:

- outOKEM must have **statistical strong ciphertext uniformity** (ciphertexts must look uniform, even if you know sk, pk)

Achieves IND-CCA/SPR-CCA, and:

outOKEM can be DHKEM

sk = x pk = xG

ct = Elligator2(r · pk)

Instantiating OEINC

Security Properties

Requires:

- outOKEM must have **statistical strong ciphertext uniformity** (ciphertexts must look uniform, even if you know sk, pk)

Achieves IND-CCA/SPR-CCA, and:

- **Ciphertext uniformity** outOKEM is IND-CCA or inOKEM is ct-unif

outOKEM can be DHKEM

sk = x pk = xG

ct = Elligator2(r · pk)

Instantiating OEINC

Security Properties

Requires:

- outOKEM must have **statistical strong ciphertext uniformity** (ciphertexts must look uniform, even if you know sk, pk)

Achieves IND-CCA/SPR-CCA, and:

- **Ciphertext uniformity** outOKEM is IND-CCA or inOKEM is ct-unif
- **Public key uniformity** outOKEM is pk-unif and inOKEM is pk-unif

outOKEM can be DHKEM

sk = x pk = xG

ct = Elligator2(r · pk)

Instantiating OEINC

Security Properties

Requires:

- outOKEM must have **statistical strong ciphertext uniformity** (ciphertexts must look uniform, even if you know sk, pk)

Achieves IND-CCA/SPR-CCA, and:

- **Ciphertext uniformity** outOKEM is IND-CCA or inOKEM is ct-unif
- **Public key uniformity** outOKEM is pk-unif and inOKEM is pk-unif

We don't get hybrid public key uniformity! (Likely impossible)

outOKEM can be DHKEM

sk = x pk = xG

ct = Elligator2(r · pk)

Instantiating OEINC

Security Properties

Requires:

- outOKEM must have **statistical strong ciphertext uniformity** (ciphertexts must look uniform, even if you know sk, pk)

outOKEM can be DHKEM

sk = x pk = xG

ct = Elligator2(r · pk)

Achieves IND-CCA/SPR-CCA, and:

- **Ciphertext uniformity** outOKEM is IND-CCA or inOKEM is ct-unif
- **Public key uniformity** outOKEM is pk-unif and inOKEM is pk-unif

We don't get hybrid public key uniformity! (Likely impossible)

We also **don't always need hybrid pk-unif**

Instantiating OEINC

Security Properties

Requires:

- outOKEM must have **statistical strong ciphertext uniformity** (ciphertexts must look uniform, even if you know sk, pk)

Achieves IND-CCA/SPR-CCA, and:

- **Ciphertext uniformity** outOKEM is IND-CCA or inOKEM is ct-unif
- **Public key uniformity** outOKEM is pk-unif and inOKEM is pk-unif

We don't get hybrid public key uniformity! (Likely impossible)

We also **don't always need hybrid pk-unif**

outOKEM can be DHKEM

sk = x pk = xG

ct = Elligator2(r · pk)

inOKEM can basically be any ct-unif KEM

(and pk-unif if you want it)

Instantiating OEINC

Security Properties

Requires:

- outOKEM must have **statistical strong ciphertext uniformity** (ciphertexts must look uniform, even if you know sk, pk)

Achieves IND-CCA/SPR-CCA, and:

- **Ciphertext uniformity** outOKEM is IND-CCA or inOKEM is ct-unif
- **Public key uniformity** outOKEM is pk-unif and inOKEM is pk-unif

We don't get hybrid public key uniformity! (Likely impossible)

We also **don't always need hybrid pk-unif**

outOKEM can be DHKEM

sk = x pk = xG

ct = Elligator2(r · pk)

inOKEM can basically be any ct-unif KEM

(and pk-unif if you want it)

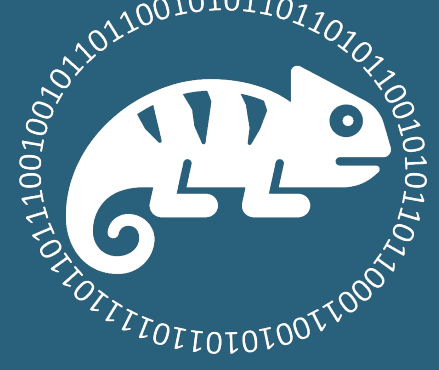
Concrete Instantiation

outOKEM = DHKEM[Ristretto]+Elligator2

inOKEM = ML-Kemeleon/Saber/Frodo

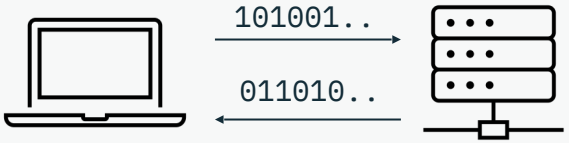
Overview

- Kemeleon Encoding
- ML-Kemeleon: Obfuscated PQ KEM
- OEINC: Obfuscated KEM Combiner
- **Hybrid Applications:**
  - **Obfuscated Key Exchange**
  - Password Authenticated Key Exchange (with adaptive security)



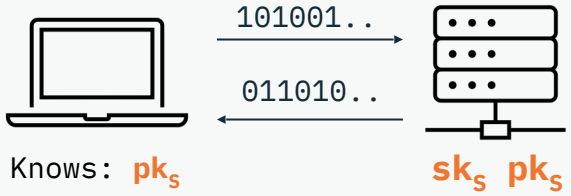
Hybrid Obfuscated Key Exchange

- All **handshake traffic** should appear **indistinguishable from random** (or simulatable).



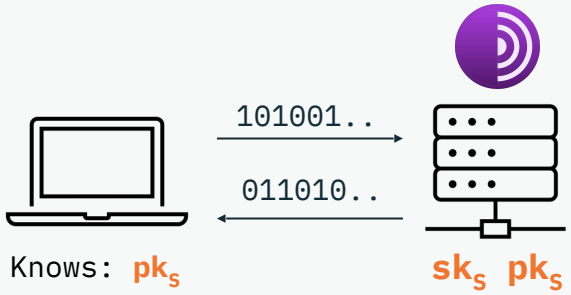
Hybrid Obfuscated Key Exchange

- All **handshake traffic** should appear **indistinguishable from random** (or simulatable).
- **Honest clients** assumed to know **server public key**.



Hybrid Obfuscated Key Exchange

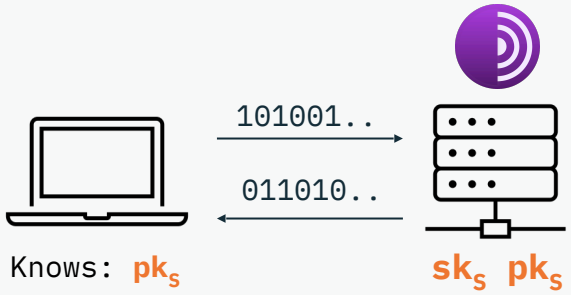
- All **handshake traffic** should appear **indistinguishable from random** (or simulatable).
- **Honest clients** assumed to know **server public key**.
- **Applications in censorship circumvention:**
  - obfs4 as a Tor pluggable transport
  - Traffic does not match any blocklists



Hybrid Obfuscated Key Exchange

- All **handshake traffic** should appear **indistinguishable from random** (or simulatable).
- **Honest clients** assumed to know **server public key**.
- **Applications in censorship circumvention:**
  - obfs4 as a Tor pluggable transport
  - Traffic does not match any blocklists

Existing KEM-based protocol requires **hybrid public key uniformity**



Hybrid Obfuscated Key Exchange

Drivel: A Hybrid Obfuscated Key Exchange Protocol

(O)KEM-based AKE

Client **Server** sk_S pk_S

Hybrid Obfuscated Key Exchange

Drivel: A Hybrid Obfuscated Key Exchange Protocol

(O)KEM-based AKE

Ephemeral and static hybrid (O)KEM encapsulations

Client

Server sk_s pk_s

$(sk_e, pk_e) := KEM.Keygen()$

pk_e →

$(c_2, K_2) := KEM.Encap(pk_e)$

← c_2

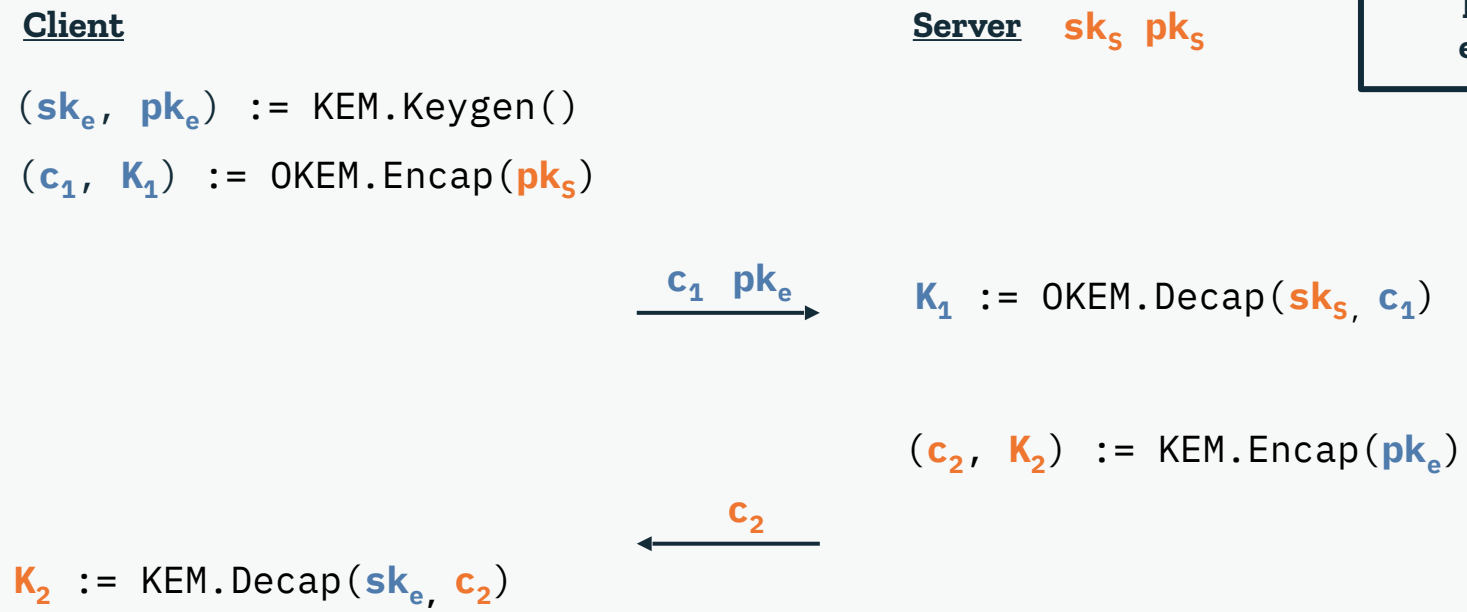
$K_2 := KEM.Decap(sk_e, c_2)$

Hybrid Obfuscated Key Exchange

Drivel: A Hybrid Obfuscated Key Exchange Protocol

(O)KEM-based AKE

Ephemeral and static hybrid (O)KEM encapsulations

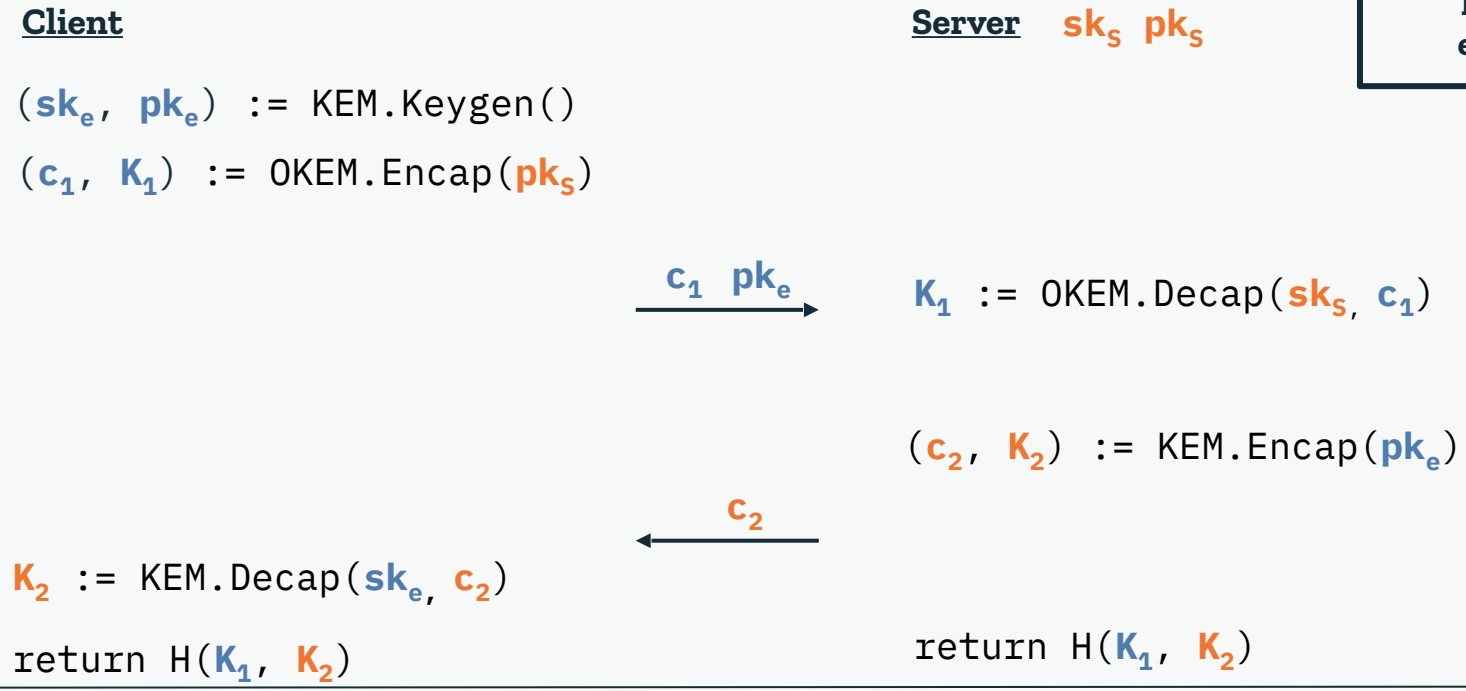


Hybrid Obfuscated Key Exchange

Drivel: A Hybrid Obfuscated Key Exchange Protocol

(O)KEM-based AKE

Ephemeral and static hybrid (O)KEM encapsulations



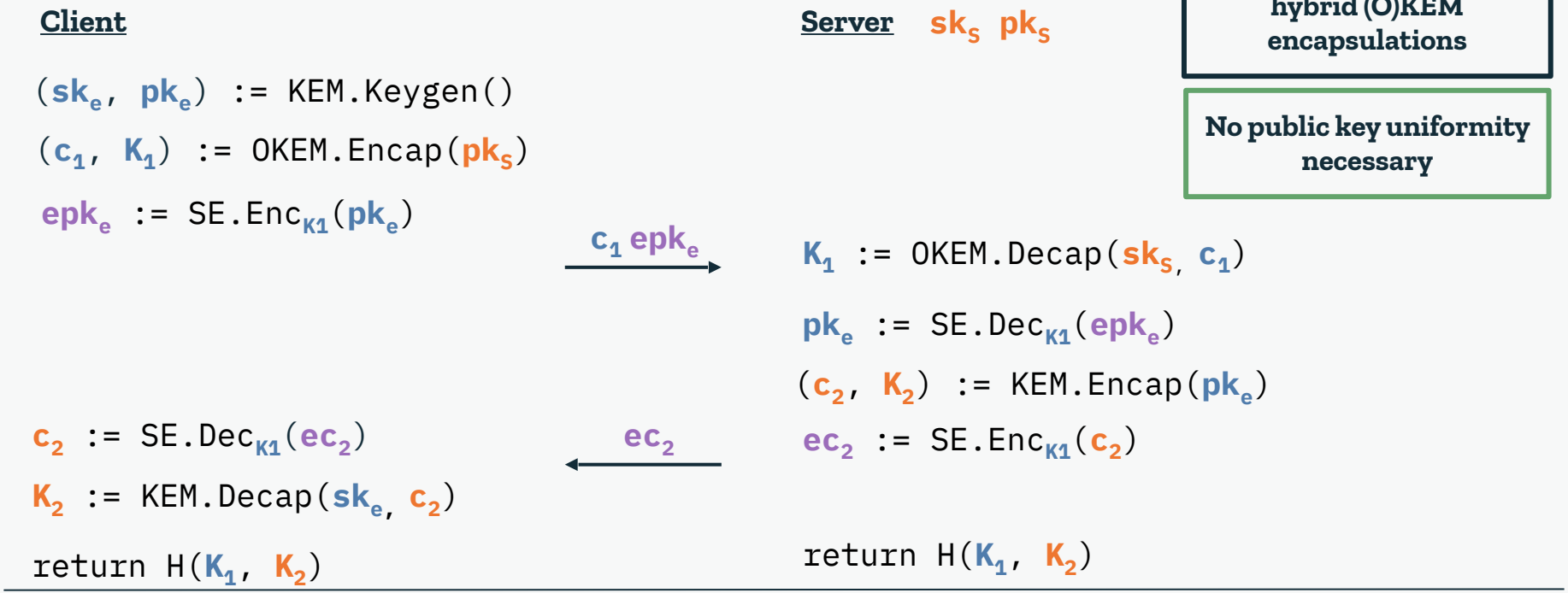
Hybrid Obfuscated Key Exchange

Drivel: A Hybrid Obfuscated Key Exchange Protocol

(O)KEM-based AKE

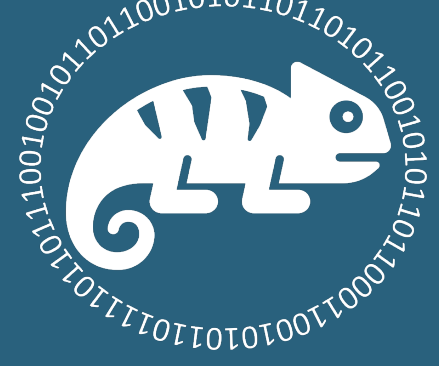
Ephemeral and static hybrid (O)KEM encapsulations

No public key uniformity necessary



Overview

- Kemeleon Encoding
- ML-Kemeleon: Obfuscated PQ KEM
- OEINC: Obfuscated KEM Combiner
- **Hybrid Applications:**
  - Obfuscated Key Exchange
  - **Password Authenticated Key Exchange (with adaptive security)**



Applications of OEINC: Hybrid PAKE

Applications of OEINC: Hybrid PAKE

Password authenticated key exchange (PAKE)

Applications of OEINC: Hybrid PAKE

Password authenticated key exchange (PAKE)

- Parties w/ **low-entropy password** want to establish a **high-entropy shared secret**:
  - **Active adversary has 1 pw guess** per protocol execution
  - **Passive adversary has no advantage at all**

Applications of OEINC: Hybrid PAKE

Password authenticated key exchange (PAKE)

- Parties w/ **low-entropy password** want to establish a **high-entropy shared secret**:
  - **Active adversary has 1 pw guess** per protocol execution
  - **Passive adversary has no advantage at all**

KEM-based PAKEs (NoIC, CHIC, EKE-PRF, CAKE, OCAKE, …)

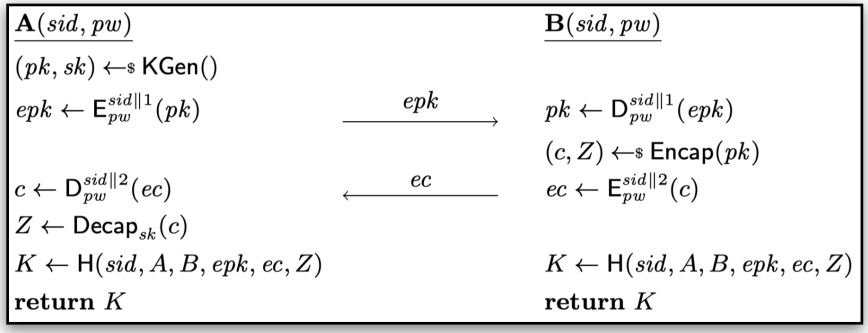
Applications of OEINC: Hybrid PAKE

Password authenticated key exchange (PAKE)

- Parties w/ **low-entropy password** want to establish a **high-entropy shared secret**:
  - **Active adversary has 1 pw guess** per protocol execution
  - **Passive adversary has no advantage at all**

KEM-based PAKEs (NoIC, CHIC, EKE-PRF, CAKE, OCAKE, …)

- CAKE proven secure in the UC model with **adaptive corruptions** (adversaries can corrupt any user at any time)



CAKE

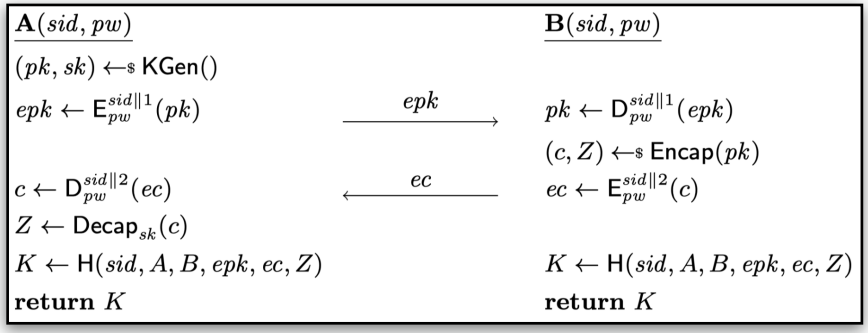
Applications of OEINC: Hybrid PAKE

Password authenticated key exchange (PAKE)

- Parties w/ **low-entropy password** want to establish a **high-entropy shared secret**:
  - **Active adversary has 1 pw guess** per protocol execution
  - **Passive adversary has no advantage at all**

KEM-based PAKEs (NoIC, CHIC, EKE-PRF, CAKE, OCAKE, …)

- CAKE proven secure in the UC model with **adaptive corruptions** (adversaries can corrupt any user at any time)
- Needs ciphertext and **public key uniformity**



CAKE

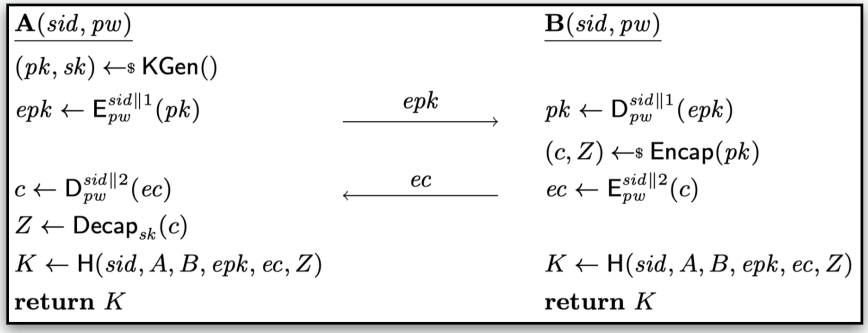
Applications of OEINC: Hybrid PAKE

Password authenticated key exchange (PAKE)

- Parties w/ **low-entropy password** want to establish a **high-entropy shared secret**:
  - **Active adversary has 1 pw guess** per protocol execution
  - **Passive adversary has no advantage at all**

KEM-based PAKEs (NoIC, CHIC, EKE-PRF, CAKE, OCAKE, …)

- CAKE proven secure in the UC model with **adaptive corruptions** (adversaries can corrupt any user at any time)
- Needs ciphertext and **public key uniformity**
  - LWE schemes only fail this bc of an optimization



CAKE

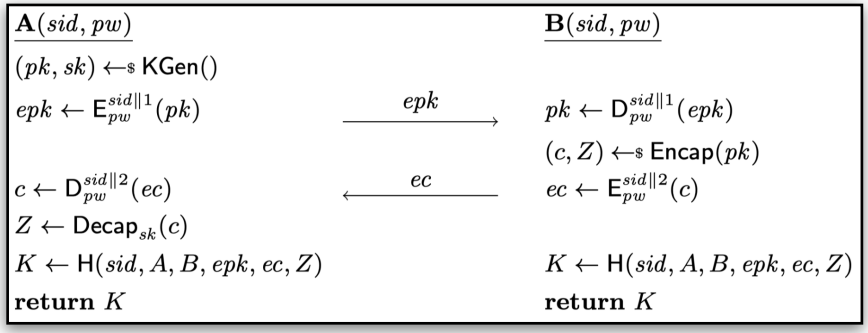
Applications of OEINC: Hybrid PAKE

Password authenticated key exchange (PAKE)

- Parties w/ **low-entropy password** want to establish a **high-entropy shared secret**:
  - **Active adversary has 1 pw guess** per protocol execution
  - **Passive adversary has no advantage at all**

KEM-based PAKEs (NoIC, CHIC, EKE-PRF, CAKE, OCAKE, …)

- CAKE proven secure in the UC model with **adaptive corruptions** (adversaries can corrupt any user at any time)
- Needs ciphertext and **public key uniformity**
  - LWE schemes only fail this bc of an optimization



CAKE

We can instantiate CAKE with **OEINC[DHKEM+Elligator, StatFrodoKEM]**

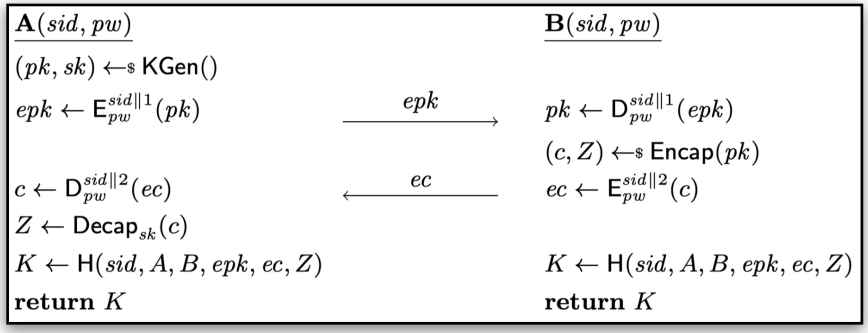
Applications of OEINC: Hybrid PAKE

Password authenticated key exchange (PAKE)

- Parties w/ **low-entropy password** want to establish a **high-entropy shared secret**:
  - **Active adversary has 1 pw guess** per protocol execution
  - **Passive adversary has no advantage at all**

KEM-based PAKEs (NoIC, CHIC, EKE-PRF, CAKE, OCAKE, ...)

- CAKE proven secure in the UC model with **adaptive corruptions** (adversaries can corrupt any user at any time)
- Needs ciphertext and **public key uniformity**
  - LWE schemes only fail this bc of an optimization
- First hybrid PAKE with security against adaptive corruptions



CAKE

We can instantiate CAKE with **OEINC[DHKEM+Elligator, StatFrodoKEM]**

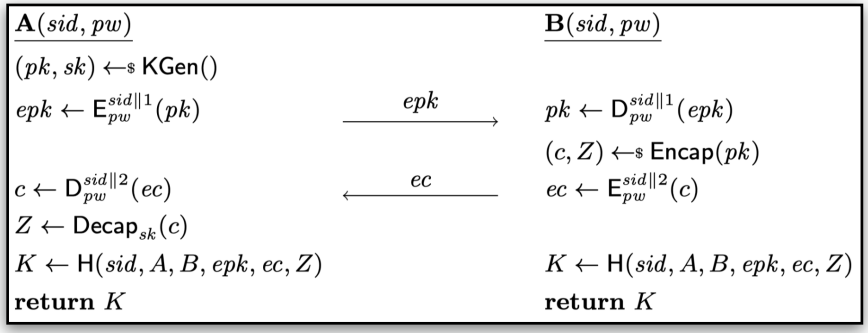
Applications of OEINC: Hybrid PAKE

Password authenticated key exchange (PAKE)

- Parties w/ **low-entropy password** want to establish a **high-entropy shared secret**:
  - **Active adversary has 1 pw guess** per protocol execution
  - **Passive adversary has no advantage at all**

KEM-based PAKEs (NoIC, CHIC, EKE-PRF, CAKE, OCAKE, …)

- CAKE proven secure in the UC model with **adaptive corruptions** (adversaries can corrupt any user at any time)
- Needs ciphertext and **public key uniformity**
  - LWE schemes only fail this bc of an optimization
- First hybrid PAKE with security against adaptive corruptions



CAKE

We can instantiate CAKE with **OEINC[DHKEM+Elligator, StatFrodoKEM]**

This is **2 rounds**. Other PAKEs are 3 rounds or inefficient (350x slowdown).

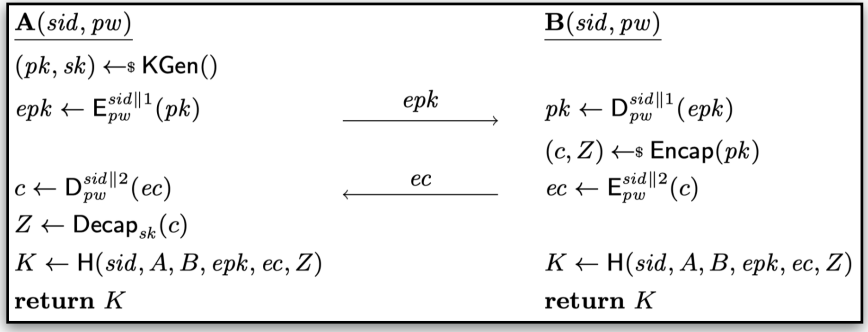
Applications of OEINC: Hybrid PAKE

Password authenticated key exchange (PAKE)

- Parties w/ **low-entropy password** want to establish a **high-entropy shared secret**:
  - **Active adversary has 1 pw guess** per protocol execution
  - **Passive adversary has no advantage at all**

KEM-based PAKEs (NoIC, CHIC, EKE-PRF, CAKE, OCAKE, ...)

- CAKE proven secure in the UC model with **adaptive corruptions** (adversaries can corrupt any user at any time)
- Needs ciphertext and **public key uniformity**
  - LWE schemes only fail this bc of an optimization
- First hybrid PAKE with security against adaptive corruptions



CAKE

We can instantiate CAKE with **OEINC[DHKEM+Elligator, StatFrodoKEM]**

This is **2 rounds**. Other PAKEs are 3 rounds or inefficient (350x slowdown).

7.5x comms overhead compared to 3-round PAKEs

Thanks! Questions?

Kemeleon: Elligator-like Obfuscation for Post-Quantum Cryptography

We made:

- an encoding for ML-KEM
- an OKEM from ML-KEM
- an OKEM combiner

We got:

- Hybrid obfuscated key exchange
- Hybrid PAKE

References:

- Günther, Stebila, Veitch. Obfuscated Key Exchange. CCS 2024. ia.cr/2024/1086
- Günther, Stebila, Veitch. Kemeleon Encodings. Internet-Draft. https://datatracker.ietf.org/doc/draft-veitch-kemeleon/ **<— send us your feedback!**
- Günther, Rosenberg, Stebila, Veitch. Hybrid Obfuscated KEMs and Key Exchange. ia.cr/2025/408