Advanced KEM Concepts

(Hybrid) Obfuscation and Verifiable Decapsulation

IBM **Research** Security

Felix Günther

IBM Research Europe – Zurich

based on work with

Lewis Glabush
EPFL

Britta Hale
NPS

Kathrin Hövelmanns
TU Eindhoven

Michael Rosenberg
Cloudflare

Douglas Stebila
U Waterloo

Shannon Veitch
ETH Zurich

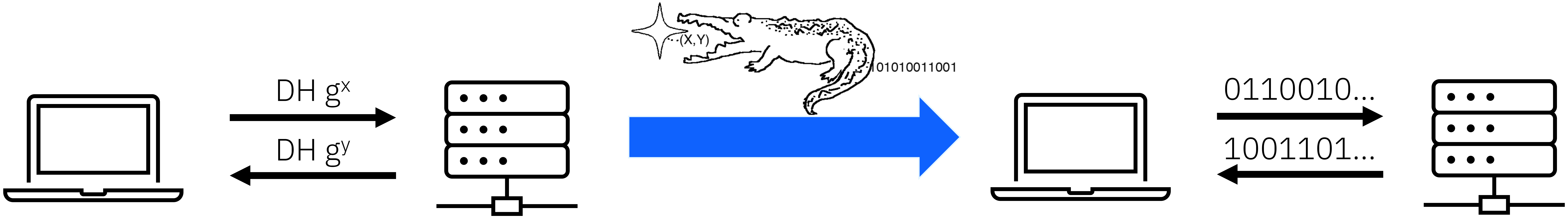
Elizabeth van Oorschot
McGill

Swiss Crypto Day 2025, Lausanne
October 31, 2025

Protocol Obfuscation

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

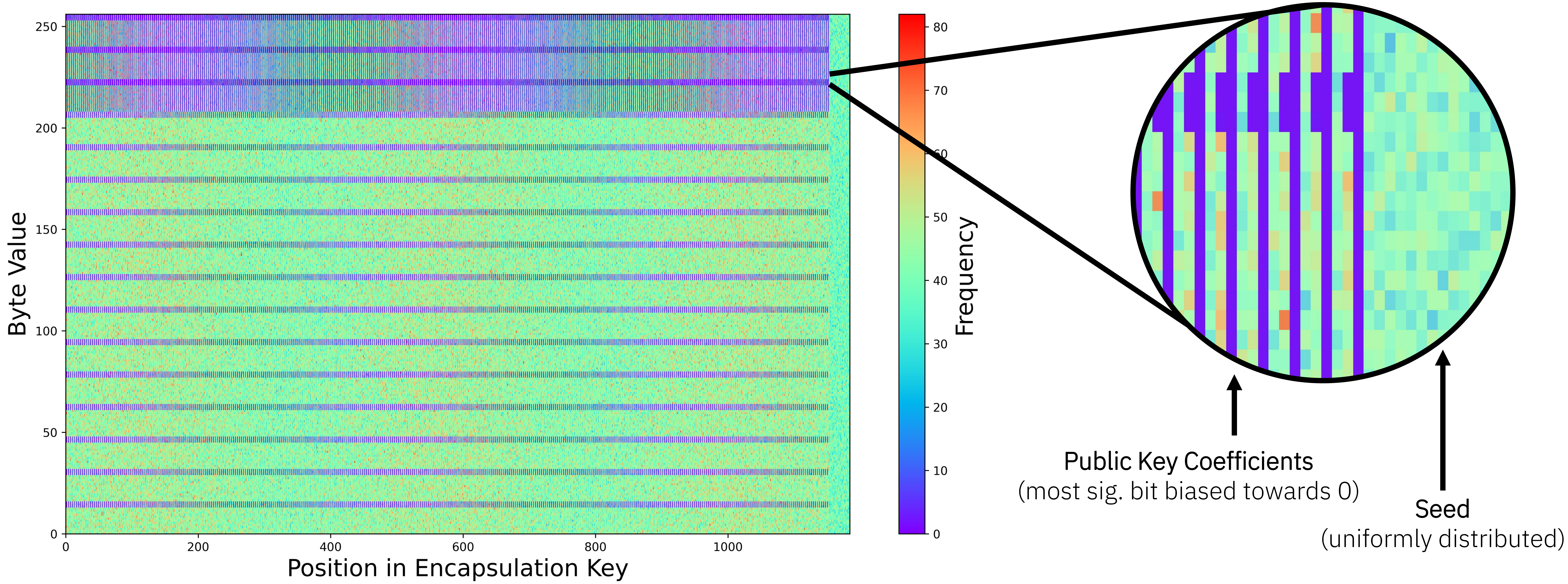


Quantum-safe transition?

ML-KEM public keys and ciphertexts **don’t look random!**

Byte Distribution of ML-KEM-768 Public Keys

ML-KEM public keys: $t = As + e$ (vectors of coefficients mod $q = 3329$), **ρ** (random seed generating **A**)



Kemeleon

ML-KEM public keys

- vector of coefficients mod q = 3329

$$[\ a_1\ ][\ a_2\ ][\ a_3\ ]\ldots[\ a_b\ ] \quad a_i \in \mathbb{Z}_q = \{0,\ldots,3328\} \text{ – each } a_i \text{ represented in 12 bits}$$

↑ ↑ ↑ ↑

most sig. bit of each value biased towards 0

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

- Encoding for public keys:
  1. accumulate into one big number
  2. rejection sampling: reject if msb is 1

ML-KEM-768 **likelihood of rejection is ~17%**

$$[\ A = a_1 + a_2 \cdot q + a_3 \cdot q^2 + \cdots + a_b \cdot q^{b-1}\ ]$$

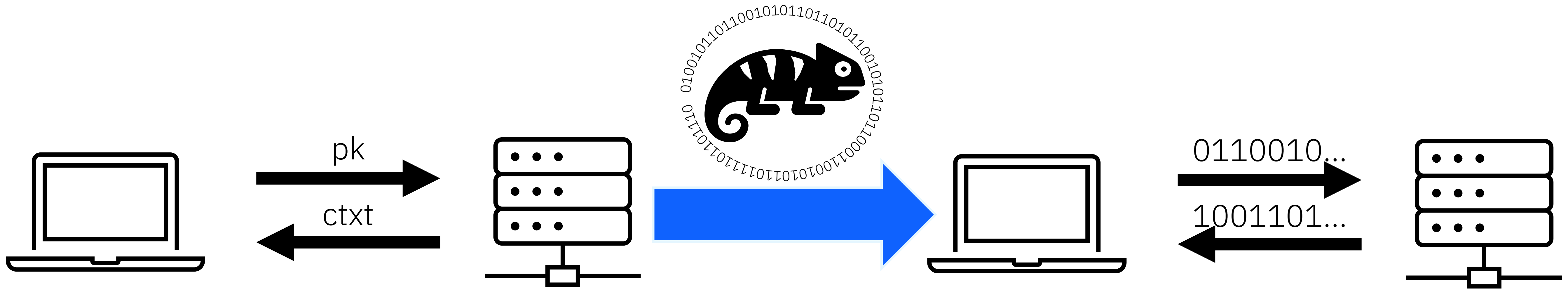
↑

most sig. bit still biased towards 0

ML-KEM ciphertexts

- vector of compressed coefficients – need to first “decompress”
- encoded ciphertexts larger than regular (6–15%)

Obfuscated KEMs



ML-KEM

\+ Kemeleon public key and ciphertext encoding

= Obfuscated KEM: **ML-Kemeleon**

- **IND-CCA:** indistinguishability of shared secrets
- **SPR-CCA:** ind. of secrets + ciphertexts simulatable (implies **anonymity**)
- **Ciphertext/Public-key Uniformity:** indistinguishable from random bit strings

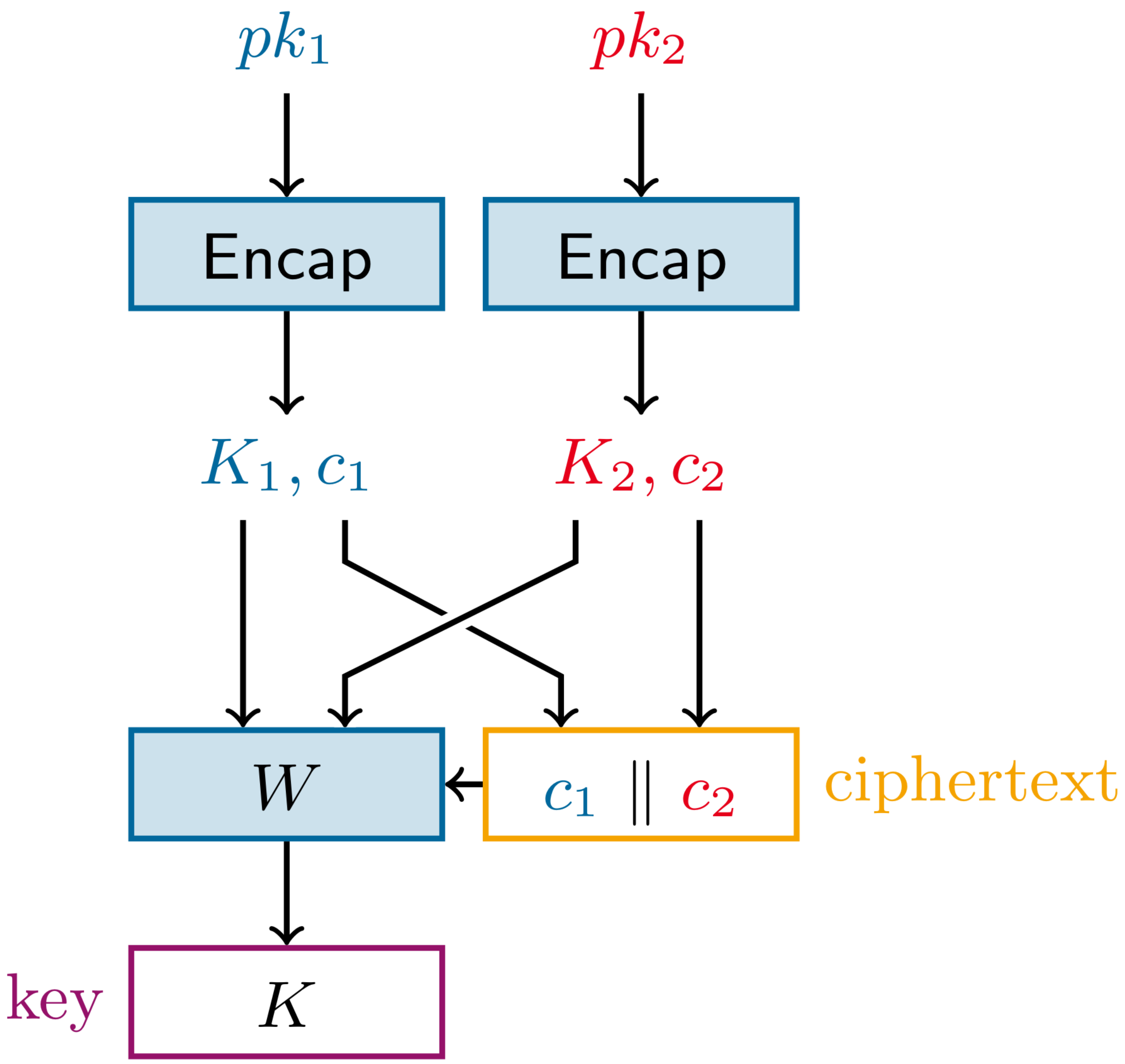
Kemeleon adopted by CFRG

`https://datatracker.ietf.org/doc/draft-irtf-cfrg-kemeleon/`

(more variants: no rejection, deterministic, ...)

Hybrid KEMs

Parallel Combiner

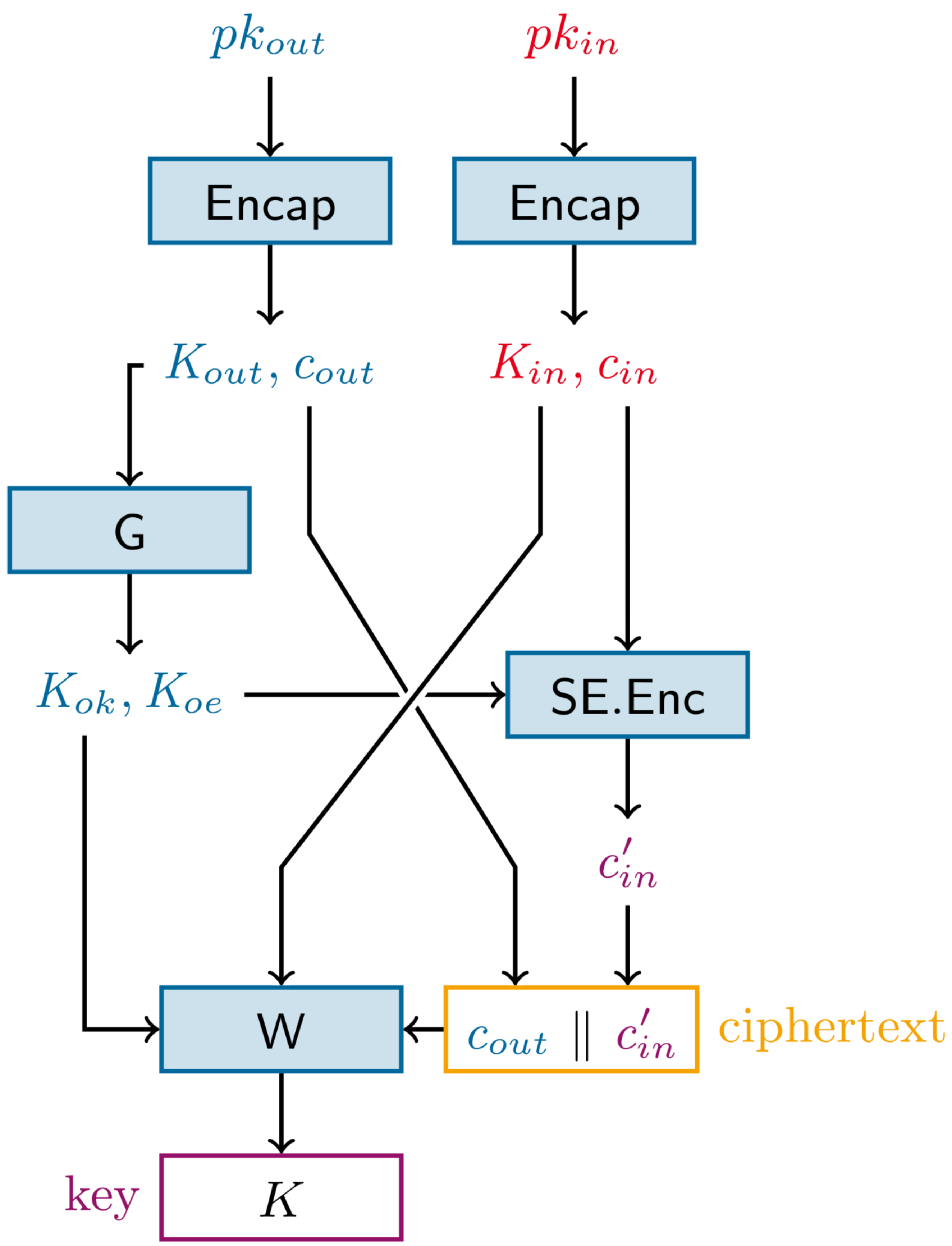


TLS 1.3 hybrid, HPKE Xyber, XWing, QSF, KitchenSink, Chempat, ...

- ☑ Hybrid IND-CCA security
- ☒ Hybrid Obfuscation

Hybrid Obfuscated KEMs

OEINC



Outer-encrypts-inner nested combiner

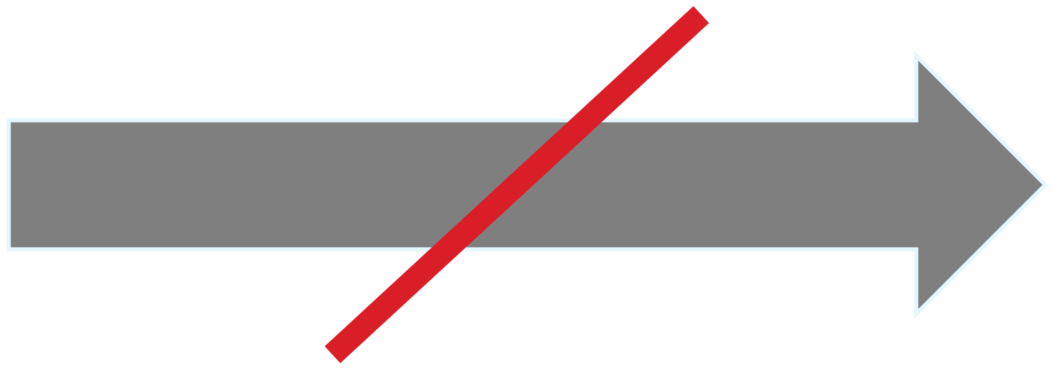
- ☑ Hybrid IND-CCA security
- ☑ Hybrid Obfuscation
- ☑ Low overhead: 1 PRG + 1 XOR

example:	outer =	DH-Elligator	(statistical)
	inner =	ML-Kemeleon	(computational)

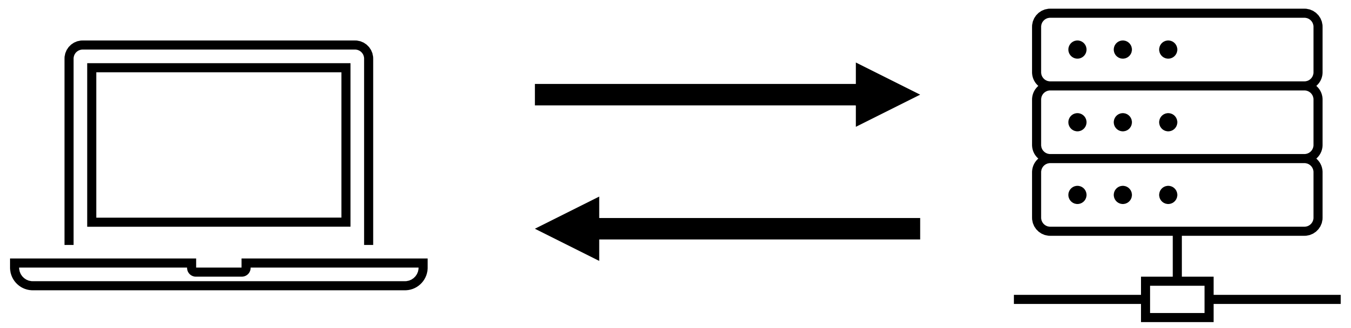
Use OEINC to build

- hybrid obfuscated key exchange
- hybrid PAKE (w/ adaptive corruptions)

Cryptography Is Brittle

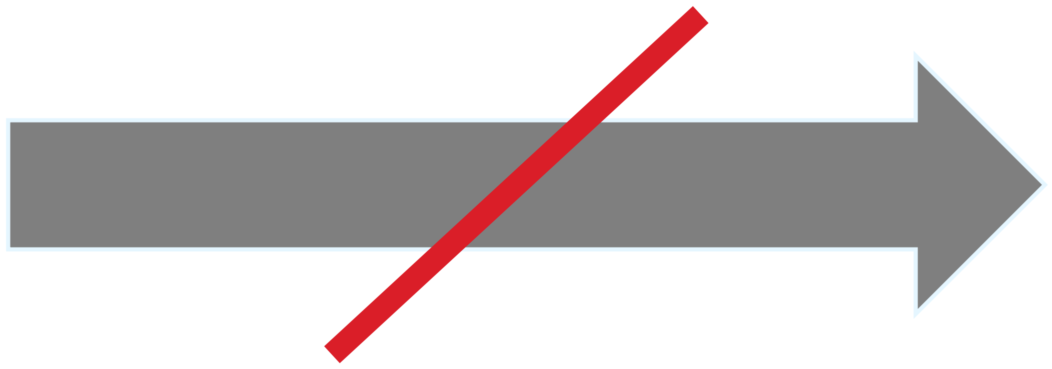
functionality  security

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ct
    SSLBuffer signedParams, uint8_t *signature,
{
    OSStatus        err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &se
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &si
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &has
        goto fail;
    ...

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```



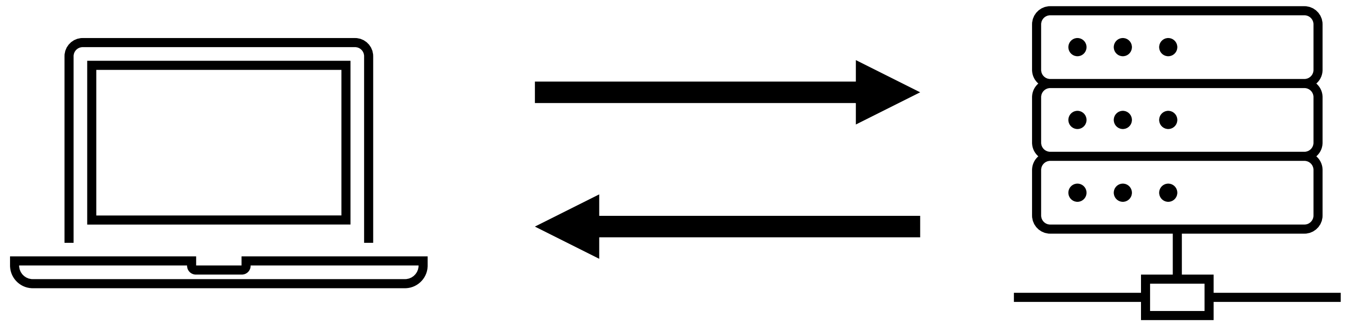
Cryptography Is Brittle

functionality



security

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ct
    SSLBuffer signedParams, uint8_t *signature,
{
    OSStatus        err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &se
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &si
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &has
        goto fail;
    ...

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```



Algorithm 18 ML-KEM.Decaps_internal(dk, c)

5: $m' \leftarrow$ K-PKE.Decrypt(dk$_{\text{PKE}}, c$)
6: $(K', r') \leftarrow \mathsf{G}(m' \| h)$
7: $\bar{K} \leftarrow \mathsf{J}(z \| c)$
8: $c' \leftarrow$ K-PKE.Encrypt(ek$_{\text{PKE}}, m', r'$) ?
9: **if** $c \neq c'$ **then**
10: $K' \leftarrow \bar{K}$
11: **end if**
12: **return** K'

FO transform

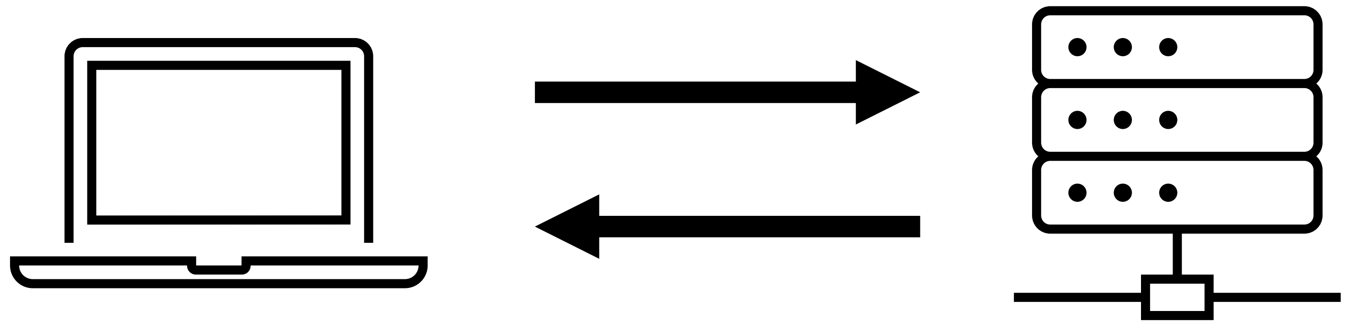
Cryptography Is Brittle

Can we tie **security** to **basic functionality**?

functionality

security

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ct
    SSLBuffer signedParams, uint8_t *signature,
{
    OSStatus        err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &se
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &si
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &has
        goto fail;
    ...

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```



```
int PQCLEAN_HQC128_CLEAN_crypto_kem_dec(uint8_t *ss, const uint8_t *ct, const uint8_t *sk) {

    uint8_t result;
    uint64_t u[VEC_N_SIZE_64] = {0};
    uint64_t v[VEC_N1N2_SIZE_64] = {0};
    const uint8_t *pk = sk + SEED_BYTES;
    uint8_t sigma[VEC_K_SIZE_BYTES] = {0};
    uint8_t theta[SHAKE256_512_BYTES] = {0};
    uint64_t u2[VEC_N_SIZE_64] = {0};
    uint64_t v2[VEC_N1N2_SIZE_64] = {0};
    uint8_t mc[VEC_K_SIZE_BYTES + VEC_N_SIZE_BYTES + VEC_N1N2_SIZE_BYTES] = {0};
    uint8_t tmp[VEC_K_SIZE_BYTES + PUBLIC_KEY_BYTES + SALT_SIZE_BYTES] = {0};
    uint8_t *m = tmp;
    uint8_t *salt = tmp + VEC_K_SIZE_BYTES + PUBLIC_KEY_BYTES;
    shake256incctx shake256state;

    // Retrieving u, v and d from ciphertext
    PQCLEAN_HQC128_CLEAN_hqc_ciphertext_from_string(u, v, salt, ct);

    // Decrypting
    result = PQCLEAN_HQC128_CLEAN_hqc_pke_decrypt(m, sigma, u, v, sk);

    // Computing theta
    memcpy(tmp + VEC_K_SIZE_BYTES, pk, PUBLIC_KEY_BYTES);
    PQCLEAN_HQC128_CLEAN_shake256_512_ds(&shake256state, theta, tmp, VEC_K_SIZE_BYTES + PUBLIC_

    // Encrypting m'
    PQCLEAN_HQC128_CLEAN_hqc_pke_encrypt(u2, v2, m, theta, pk);

    // Check if c != c'
    result |= PQCLEAN_HQC128_CLEAN_vect_compare((uint8_t *)u, (uint8_t *)u2, VEC_N_SIZE_BYTES);
    result |= PQCLEAN_HQC128_CLEAN_vect_compare((uint8_t *)v, (uint8_t *)v2, VEC_N1N2_SIZE_BYTE

    result = (uint8_t) (-((int16_t) result) >> 15);

    for (size_t i = 0; i < VEC_K_SIZE_BYTES; ++i) {
        mc[i] = (m[i] & result) ^ (sigma[i] & ~result);
    }
```

Verifiable Decapsulation

$\underline{\mathsf{Decaps}(\mathsf{sk}, c)}$

$m' \leftarrow \mathsf{Dec}(\mathsf{sk}, c)$

$(c' \qquad\qquad) \leftarrow \mathsf{Enc}(\mathsf{pk}, m')$

$\text{check } c' = c$

$K' \leftarrow \mathsf{KDF}(m', \mathsf{pk} \qquad\qquad)$

$\textbf{return } K'$

Verifiable Decapsulation

Enter: **Confirmation Codes**

building on ideas from Heninger, [Fischlin-G'23]

$$\underline{\mathsf{Decaps}(\mathsf{sk}, c)}$$

$$m' \leftarrow \mathsf{Dec}(\mathsf{sk}, c)$$

$$(c' \boxed{, \mathsf{cd}'}) \leftarrow \mathsf{Enc}(\mathsf{pk}, m')$$

$$\text{check } c' = c$$

$$K' \leftarrow \mathsf{KDF}(m', \mathsf{pk} \boxed{, \mathsf{cd}'})$$

$$\textbf{return } K'$$

Idea: faulty implementation of re-encryption → noticeable KEM correctness failure

ML-KEM with Confirmation Codes

ML-KEM ciphertext compression → lost entropy

leverage lost entropy for confirmation code

Using **12-20 bytes** of confirmation code

detect faulty re-encryption in ML-KM-512/768/1024

by single test w/ probability ~1/3

at ≤ 3.4% performance overhead

Algorithm 14 K-PKE.Encrypt$(\mathsf{ek}_{\mathsf{PKE}}, m, r)$

Uses the encryption key to encrypt a plaintext message using the randomness r.

Input: encryption key $\mathsf{ek}_{\mathsf{PKE}} \in \mathbb{B}^{384k+32}$.
Input: message $m \in \mathbb{B}^{32}$.
Input: randomness $r \in \mathbb{B}^{32}$.
Output: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.

$N \leftarrow 0$
$\hat{\mathbf{t}} \leftarrow \mathsf{ByteDecode}_{12}(\mathsf{ek}_{\mathsf{PKE}}[0:384k])$ ▷ run ByteDecode$_{12}$ k times to decode $\hat{\mathbf{t}} \in (\mathbb{Z}_q^{256})^k$
$\rho \leftarrow \mathsf{ek}_{\mathsf{PKE}}[384k : 384k+32]$ ▷ extract 32-byte seed from $\mathsf{ek}_{\mathsf{PKE}}$
for $(i \leftarrow 0; i < k; i$++) ▷ re-generate matrix $\hat{\mathbf{A}} \in (\mathbb{Z}_q^{256})^{k \times k}$ sampled in Alg. 13
  for $(j \leftarrow 0; j < k; j$++)
    $\hat{\mathbf{A}}[i,j] \leftarrow \mathsf{SampleNTT}(\rho \| j \| i)$ ▷ j and i are bytes 33 and 34 of the input
  end for
end for
for $(i \leftarrow 0; i < k; i$++) ▷ generate $\mathbf{y} \in (\mathbb{Z}_q^{256})^k$
  $\mathbf{y}[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_1}(\mathsf{PRF}_{\eta_1}(r, N))$ ▷ $\mathbf{y}[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
  $N \leftarrow N+1$
end for
for $(i \leftarrow 0; i < k; i$++) ▷ generate $\mathbf{e_1} \in (\mathbb{Z}_q^{256})^k$
  $\mathbf{e_1}[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_2}(\mathsf{PRF}_{\eta_2}(r, N))$ ▷ $\mathbf{e_1}[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
  $N \leftarrow N+1$
end for
$e_2 \leftarrow \mathsf{SamplePolyCBD}_{\eta_2}(\mathsf{PRF}_{\eta_2}(r, N))$ ▷ sample $e_2 \in \mathbb{Z}_q^{256}$ from CBD
$\hat{\mathbf{y}} \leftarrow \mathsf{NTT}(\mathbf{y})$ ▷ run NTT k times
$\mathbf{u} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^\top \circ \hat{\mathbf{y}}) + \mathbf{e_1}$ ▷ run NTT^{-1} k times
$\mu \leftarrow \mathsf{Decompress}_1(\mathsf{ByteDecode}_1(m))$
$v \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{t}}^\top \circ \hat{\mathbf{y}}) + e_2 + \mu$ ▷ encode plaintext m into polynomial v
$c_1 \leftarrow \mathsf{ByteEncode}_{d_u}(\mathsf{Compress}_{d_u}(\mathbf{u}))$ ▷ run ByteEncode$_{d_u}$ and Compress$_{d_u}$ k times
$c_2 \leftarrow \mathsf{ByteEncode}_{d_v}(\mathsf{Compress}_{d_v}(v))$
$\mathsf{cd} \leftarrow (\mathbf{u}[1][S], \ldots, \mathbf{u}[k][S], v[S])$
return $(c = c_1 \| c_2, \ \mathsf{cd})$

Verifiable Decapsulation: Confirmation-code Augmented FO

- We formalize **confirmation code unpredictability (cUP)** for PKE schemes:

limited access to F
intuition: won't accidentally compute

$$03 \quad (c, \mathsf{cd}) \leftarrow \mathsf{Enc}_{\mathsf{C}}^{\mathsf{F}}(\mathsf{pk}, m; r)$$
$$04 \quad \mathsf{cd}' \leftarrow \mathcal{A}^{\bar{\mathsf{F}}}(\mathsf{pk}, \mathsf{sk}, c, m, r)$$

- We introduce a **confirmation-code augmented FO transform** FOC = UC ∘ TC [following HHK'17]
  - TC transform: derandomize PKE with confirmation codes
  - UC transform: bind confirmation code into KEM key derivation

- We show: FOC transform of cUP PKE scheme → KEM with **noticeable incorrectness for faulty implementations**

Verifiable Decapsulation without Algorithm Modification

but confirmation codes themselves are

non-intrusive

$\underline{\mathsf{Decaps}(\mathsf{sk}, c)}$

$m' \leftarrow \mathsf{Dec}(\mathsf{sk}, c)$

$(c', \mathsf{cd}') \leftarrow \mathsf{Enc}(\mathsf{pk}, m')$

$\textbf{check } c' = c$

$K' \leftarrow \mathsf{KDF}(m', \mathsf{pk}, \mathsf{cd}')$

$\textbf{return } K'$

intrusive:

requires algorithm modification
= deviation from KEM standards

= problematic for certification

we can post-process confirmation codes via an external wrapper

- treat and formalize **read-only white-box access** to the original KEM implementation
- prove IND-CCA security of original KEM **under leakage of confirmation codes**
- interesting subtleties: how should an **implicitly rejecting confirmation code** look like?

Summary

(HYBRID) OBFUSCATION

Kemeleon: obfuscate ML-KEM pk/ctxt

– pk even 2.5% smaller

Obfuscated KEM

OEINC: hybrid KEM obfuscation

full versions @ IACR ePrint:

– Kemeleon: ia.cr/2024/1086

`https://datatracker.ietf.org/doc/draft-irtf-cfrg-kemeleon/`

– Hybrid OKEMs: ia.cr/2025/408

– Verifiable Decaps: ia.cr/2025/450

VERIFIABLE DECAPSULATION

functionality $\xrightarrow{\text{conf. code}}$ security

Confirmation-code augmented FO

ML-KEM: 12-20B → detect prob. ~1/3

HQC: 1B → basic tests catch bug

possible w/o algorithm modification

Thank You!

mail@felixguenther.info