□ Projectdocument – Quantum Proof Encryptieplatform

Deel 1 – Fundament & Architectuur

(Hoofdstukken 1 t/m 3)

Inhoud

---

1. Inleiding

1.1 Doel van het project

Dit document beschrijft de technische architectuur en implementatie van een **Quantum-Proof Encryptieplatform**.
Het platform is ontwikkeld om gevoelige data te beschermen tegen huidige en toekomstige bedreigingen, inclusief **post-quantum-computing**.
De implementatie is volledig gebaseerd op **end-to-end-encryptie (E2EE)**, waarbij uitsluitend de gebruiker toegang heeft tot zijn of haar sleutelmateriaal.

Het platform bestaat uit een **Laravel-backend** (PHP 8.3 +) en een **Next.js-frontend**.
Alle cryptografische bewerkingen vinden plaats aan de **client-zijde**, zodat de server slechts fungeert als transportlaag en auditlog.

1.2 Probleemstelling

Huidige cryptografische standaarden, zoals RSA en ECC (elliptische-curve-cryptografie), worden als **kwetsbaar** beschouwd voor aanvallen door toekomstige kwantumcomputers.
Een grootschalige kwantumcomputer kan met het **Shor-algoritme** binnen minuten de

private sleutel berekenen uit een publieke sleutel, waarmee de vertrouwelijkheid van data volledig verloren gaat.

Het doel van dit project is daarom om over te stappen op **post-quantum-cryptografie (PQC)**, met behoud van praktische bruikbaarheid binnen een web- en mobiele context.

1.3 Terminologie en afkortingen

Afkorting	Betekenis	Uitleg
PQC	Post-Quantum Cryptography	Cryptografie die bestand is tegen aanvallen van kwantumcomputers.
KEM	Key Encapsulation Mechanism	Algoritme voor het veilig uitwisselen van sleutelmateriaal zonder directe sleuteltransmissie.
ML-KEM	Module Lattice-based Key Encapsulation Mechanism	Een KEM-variant gebaseerd op roosters (lattices); veilig tegen kwantumaanvallen.
AEAD	Authenticated Encryption with Associated Data	Versleuteling die tegelijkertijd vertrouwelijkheid én integriteit biedt.
AES-256-GCM	Advanced Encryption Standard – Galois Counter Mode	Moderne AEAD-methode voor data-encryptie en -authenticatie.
HKDF	HMAC-based Key Derivation Function	Algoritme om afgeleide sleutels te genereren uit één bron-sleutel.
FFI	Foreign Function Interface	PHP-mechanisme om functies uit C-bibliotheken (zoals liboqs) aan te roepen.
E2EE	End-to-End Encryption	Sleutels bestaan uitsluitend bij de gebruikers; de server heeft geen toegang.

2. Doelen en functionele eisen

2.1 Hoofddoelen

1. **Volledige end-to-end-encryptie:**
   Alleen de gebruiker bezit de *Master Private Key (MPK)*.

2. **Quantum-bestendige communicatie:**
   Gebruik van **ML-KEM-1024** voor sleuteluitwisseling en **AES-256-GCM** voor data-encryptie.

3. **Abstracte integratie:**
   Het systeem moet als zelfstandige module kunnen functioneren binnen elk PHP-framework en JavaScript-frontend.

4. **Herhaalbare sleutel-afleiding:**
   Alle sleutels worden deterministisch afgeleid via **HKDF-SHA-512** met unieke salts.

5. **Forward Secrecy & Post-Compromise Security:**
   Elke nieuwe sessie of bestandsactie gebruikt nieuwe sleutels; oude sleutels worden vernietigd.

6. **Sleutelherstel via Capsule-systeem:**
   Gebruikers kunnen optioneel een “Recovery Capsule” genereren die door derden kan worden bewaard maar niet ontsleuteld.

2.2 Niet-functionele eisen

Categorie	Eisen
Prestatie	Encryptie- en decryptie-acties moeten < 200 ms duren op een moderne client.
Schaalbaarheid	Het systeem moet horizontaal uitbreidbaar zijn (stateless API).
Veiligheid	Sleutelmateriaal verlaat nooit de client.
Herstelbaarheid	Sleutels kunnen worden teruggezet via het Capsule-mechanisme, mits de gebruiker de *Recovery Secret Key* bezit.
Compatibiliteit	PHP 8.3+, Node 18+, LibOQS (FFI) op AlmaLinux 9.5.

2.3 Projectscope

Het project richt zich uitsluitend op:

- bestand- en berichtversleuteling (*data-at-rest* en *data-in-transit*),

- veilige sleuteluitwisseling en -afleiding,
- herstel- en sleutelrotatie.

Authenticatie, autorisatie en logging vallen buiten de primaire scope, maar de architectuur is voorbereid op integratie met bestaande identity-providers (Azure AD, OAuth 2.0, enz.).

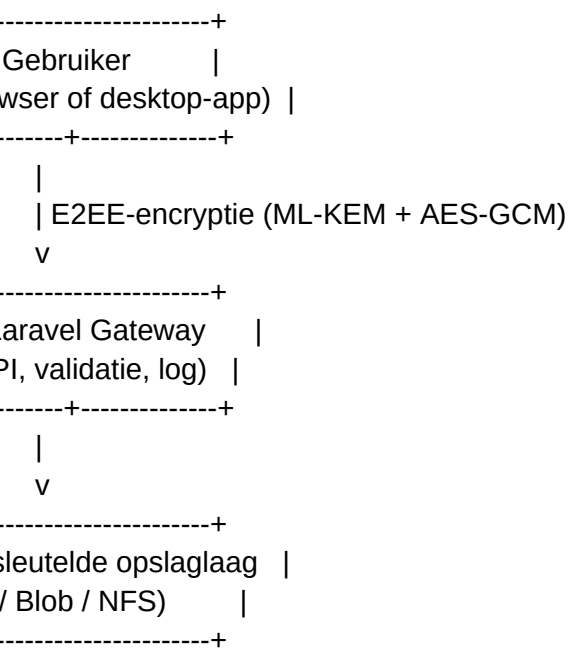
---

3. Systeemarchitectuur

3.1 Overzicht

De architectuur bestaat uit drie hoofdcomponenten:

1. **Client-side cryptografie (Next.js)**
   – Sleutelgeneratie, encryptie/decryptie, capsule-aanmaak.
2. **Backend-gateway (Laravel)**
   – Validatie, opslag en logging; geen toegang tot sleutels.
3. **Beveiligde opslaglaag (S3, Azure Blob, of on-prem NFS)**
   – Bewaart uitsluitend versleutelde data en capsules.



3.2 Componenten

Client (Next.js)

- Genereert **ML-KEM-1024-sleutelpaar** bij eerste gebruik.
- Beheert **Ratchet-keten** voor sleutelrotatie.
- Gebruikt de **WebCrypto-API** voor AES-GCM-operaties.
- Ondersteunt **IndexedDB** voor lokale opslag van versleutelde sleutels.
- Kan sleutels exporteren/importeren naar **1Password** via SDK-integratie.

Server (Laravel API)

- Verzorgt enkel **validatie en opslag** van ciphertext, metadata en logs.
- Controleert dat `capsule_hash` of `file_hash` geldig is.
- Registreert events voor audit: `upload`, `download`, `capsule_created`, `capsule_returned`.
- Heeft **geen decryptie-logica** en geen toegang tot private keys.

Opslaglaag

- Bestanden worden uitsluitend opgeslagen als **versleutelde blobs**.
- Metadata (zoals bestandsnaam, timestamp, gebruikers-ID) is afzonderlijk versleuteld.
- Sleutels voor deze encryptie (DEK's – *Data Encryption Keys*) worden client-side aangemaakt en nooit gedeeld met de server.

3.3 Communicatie- en datastromen

3.3.1 Uploadproces

1. Gebruiker kiest bestand → client genereert **random DEK**.
2. Bestand wordt versleuteld met `AES-256-GCM(DEK)`.
3. DEK wordt *afgeleid of verpakt* via **ML-KEM-1024** en **HKDF-SHA-512**.
4. De resulterende blob + metadata wordt via HTTPS naar de Laravel-API gestuurd.
5. Server valideert en slaat alles op; geen toegang tot sleutels of plaintext.

3.3.2 Downloadproces

1. Gebruiker vraagt bestand aan.

2. Server levert de versleutelde blob + metadata.
3. Client ontsleutelt via lokaal aanwezige sleutels.
4. Event wordt gelogd in het auditrail-systeem.

3.4 Beveiligingsgrenzen

Component	Vertrouwensniveau	Toegang tot sleutels
Client	Vertrouwd	Ja – volledige sleutelruimte
Server	Niet-vertrouwd (semi-trust)	Nee
Opslaglaag	Niet-vertrouwd	Nee

Belangrijk:
De **server** mag nooit sleutelmateriaal loggen of in cache houden.
Alle cryptografische berekeningen moeten plaatsvinden in de browser of desktop-client.
Audit-logs worden uitsluitend gebruikt voor detectie en naleving (bijv. ISO 27001 / NIS2).

3.5 Codevoorbeeld – Client encryptie (vereenvoudigd)

```
// Next.js (TypeScript) – bestand encryptie
async function encryptFile(file: ArrayBuffer, publicKey: Uint8Array) {
  const dek = crypto.getRandomValues(new Uint8Array(32));
  const iv  = crypto.getRandomValues(new Uint8Array(12));

  // 1. Bestand versleutelen
  const ciphertext = await crypto.subtle.encrypt(
    { name: "AES-GCM", iv },
    await crypto.subtle.importKey("raw", dek, "AES-GCM", false, ["encrypt"]),
    file
  );

  // 2. Sleutel encapsuleren
  const { kem_ct, shared } = mlkem1024.encap(publicKey);
  const wrapKey = await hkdfSha512(shared, "keypact|file|v1");
  const sealedDek = await aesGcmEncrypt(wrapKey, dek, "keypact|dek|v1");

  return { ciphertext, iv, kem_ct, sealedDek };
}
```

3.6 Codevoorbeeld – Server opslag (Laravel PHP)

```
// Laravel Controller – UploadFileController.php
public function store(Request $request)
{
    $validated = $request->validate([
        'file' => 'required|string',
        'kem_ct' => 'required|string',
        'sealedDek' => 'required|string',
        'iv' => 'required|string',
        'file_hash' => 'required|string',
    ]);

    // Controleer integriteit (hash-validatie)
    if (!hash_equals(
        $validated['file_hash'],
        hash('blake2b512', $validated['file'], false)
    )) {
        abort(400, 'Ongeldige bestands-hash');
    }

    // Sla versleutelde data en metadata op
    EncryptedFile::create([
        'owner_uid' => auth()->id(),
        'data' => $validated['file'],
        'kem_ct' => $validated['kem_ct'],
        'sealed_dek' => $validated['sealedDek'],
        'iv' => $validated['iv'],
        'file_hash' => $validated['file_hash'],
    ]);

    return response()->json(['status' => 'stored'], 201);
}
```

3.7 Samenvatting

- Alle encryptie en sleutelbeheer vinden plaats op de **client**.
- De server functioneert als **neutrale transportlaag** en bewaakt de integriteit via hashes en logs.
- Door gebruik van **ML-KEM-1024** en **AES-256-GCM** is het systeem bestand tegen kwantumaanvallen.

- De architectuur ondersteunt toekomstige uitbreidingen zoals *ratchets* en *key capsules* zonder herziening van de kern.

Deel 2 – Cryptografie & Sleutelbeheer

(Hoofdstukken 4 t/m 5)

Inhoud

---

4. Cryptografische keuzes

4.1 Algoritmenoverzicht

Functie	**Algoritme**	**Beschrijving**
Sleuteluitwisseling	**ML-KEM-1024**	Post-quantum KEM gebaseerd op *lattice-crypto*.
Data-encryptie	**AES-256-GCM**	Symmetrisch AEAD-schema met authenticatie-tag.
Sleutelafleiding	**HKDF-SHA-512**	Deterministische afleiding uit master-sleutels.
Integriteit	**BLAKE2b**	Snelle cryptografische hash voor checksums en HMAC.

Waarom deze combinatie:

- ML-KEM-1024 biedt **kwantumweerstand** tot ca. 256-bit veiligheid.
- AES-256-GCM is nog steeds de wereldwijde standaard voor AEAD-encryptie.

- HKDF-SHA-512 levert sterke afgeleide sleutels zonder hergebruik van mastermaterial.
- BLAKE2b is veiliger en sneller dan SHA-3 voor server- en client-doeleinden.

4.2 Waarom ML-KEM-1024 en AES-256-GCM

ML-KEM-1024

- Gebaseerd op het CRYSTALS-Kyber algoritme dat door NIST is gestandaardiseerd (2024).
- Resistent tegen het Shor-algoritme en Grover's zoekaanval.
- Uitvoeringstijd < 2 ms op moderne CPU's.
- Sleutelpaar: ≈ 1,6 KB public / ≈ 3,2 KB secret.
- Ondersteund door de **liboqs** bibliotheek, beschikbaar via PHP FFI en Node.js WASM.

AES-256-GCM

- Symmetrische encryptie met 256-bit sleutel en 12-byte nonce.
- Authenticated Encryption with Associated Data (**AEAD**) – waarbij de authenticiteit van zowel de data als de metagegevens (AAD) wordt gecontroleerd.
- Bewezen veilig tegen moderne aanvallen mits nooit hetzelfde (key, nonce) paar wordt hergebruikt.

Combinatievoordeel

ML-KEM beveiligt de uitwisseling van sleutelmateriaal, terwijl AES-GCM de daadwerkelijke gegevens versleutelt. Hierdoor is de verwerking snel en het sleutelbeheer volledig client-side.

4.3 Hash- en KDF-mechanismen

- **HKDF (HMAC-based Key Derivation Function)** gebruikt `HMAC-SHA-512` om nieuwe sleutels af te leiden uit één bron (IKM = Input Key Material).

- HKDF wordt toegepast voor:

    - Afleiden van de Master Key uit passphrase of seed.
    - Roteren van sessie- en bestandsleutels.
    - Mixen van nieuwe entropie tijdens Double Ratchet-stappen.

Belangrijk: Elke afleiding krijgt een eigen `info`-string, zoals `"keypact|master|v1"` of `"keypact|file|v1"`. Hiermee wordt voorkomen dat verschillende ketens dezelfde sleutel genereren.

4.4 Veiligheidsniveaus en sleutelgroottes

Type sleutel	Lengte	Functie
ML-KEM-Public	≈ 1600 bytes	Sleutel voor ontvanger; mag gedeeld worden.
ML-KEM-Secret	≈ 3200 bytes	Private sleutel; blijft op client.
AES-256 Key	32 bytes	Data-encryptie.
HKDF Salt	32 bytes	Random salt per omgeving.
IV (Initial Vector)	12 bytes	Uniek per encryptie-actie.
Auth Tag	16 bytes	Verificatie van de ciphertext.

4.5 Integriteit en authenticatie

Elke versleuteling levert een authenticatietag op (`tag`) van 16 bytes. Deze tag wordt gecontroleerd bij elke decryptie; mislukking leidt tot afwijzing van de ciphertext. Extra metadata (wordt meegenomen in de AEAD `aad` parameter) zoals `file_id`, `user_id` en `timestamp` zorgen voor sterke binding tussen data en context.

---

5. Sleutelbeheer en afleiding

5.1 Master Key en Root Key hiërarchie

Het platform gebruikt een hiërarchische sleutelstructuur:

Root Key ──┬──► Master Key (gebruiker)

├──► Service Keys (per app)

└──► Audit / Capsule Keys

- **Root Key:** interne basis, gecreëerd bij eerste installatie van de app (lokaal of mobiel).
- **Master Key:** uniek per gebruiker; wordt via HKDF afgeleid uit Root Key en een salt.
- **Derived Keys:** tijdelijke sleutels voor bestanden, berichten, ratchets en capsules.

5.2 Gebruik van HKDF met unieke salts

De afleiding van sleutels gebeurt met **HKDF-SHA-512** en unieke salts. Gebruik **nooit** de Laravel `APP_KEY` als salt voor HKDF. In plaats daarvan wordt een specifieke variabele toegevoegd aan `.env`:

MASTER_DERIVE_SALT=Vm9sa2tvb2tzZ3ViZQp0ZWxqZS1xdWFudHVtLXNlY3VyZQ==

PHP-voorbeeld:

$salt = base64_decode(env('MASTER_DERIVE_SALT'));

$ikm = random_bytes(32); // input key material

$info = "keypact|master|user={$userId}|v1";

$masterKey = hash_hkdf('sha512', $ikm, 32, $info, $salt);

De afgeleide sleutel ($masterKey) is 32 bytes en wordt uitsluitend in geheugen bewaard. Na gebruik wordt deze overschreven met null-bytes.

5.3 PHP implementatie – FFI (liboqs)

De PHP-backend roept liboqs aan via FFI om ML-KEM te gebruiken:

// /app/Services/Crypto/MLKemService.php

class MLKemService

{

    private FFI $oqs;

    public function __construct()

    {

        $this->oqs = FFI::cdef("

            typedef struct { ...; } OQS_KEM;

            OQS_KEM *OQS_KEM_ml_kem_1024_new(void);

            int OQS_KEM_keypair(OQS_KEM *kem, uint8_t *pk, uint8_t *sk);

            int OQS_KEM_encaps(OQS_KEM *kem, uint8_t *ct, uint8_t *ss, const uint8_t *pk);

```
        int OQS_KEM_decaps(OQS_KEM *kem, uint8_t *ss, const uint8_t *ct, const uint8_t
*sk);

        ", "/usr/local/lib/liboqs.so");

    }

    public function keypair(): array

    {

        $kem = $this->oqs->OQS_KEM_ml_kem_1024_new();

        $pk = FFI::new("uint8_t[1632]");

        $sk = FFI::new("uint8_t[3168]");

        $this->oqs->OQS_KEM_keypair($kem, $pk, $sk);

        return [

            'public' => base64_encode(FFI::string($pk, 1632)),

            'secret' => base64_encode(FFI::string($sk, 3168))

        ];

    }

}
```

De client roept `/api/keys/new` aan om dit te triggeren, maar de private sleutel wordt nooit opgeslagen in de database.

5.4 Next.js implementatie (client-side)

```
// /lib/crypto/hkdf.ts

export async function hkdfSha512(input: Uint8Array, info: string): Promise<Uint8Array> {

  const key = await crypto.subtle.importKey("raw", input, { name: "HMAC", hash:
"SHA-512" }, false, ["sign"]);

  const result = await crypto.subtle.sign("HMAC", key, new TextEncoder().encode(info));

  return new Uint8Array(result).slice(0, 32);
```

```
}

// /lib/crypto/aes.ts

export async function aesGcmEncrypt(key: Uint8Array, data: Uint8Array, aad: string) {

  const iv = crypto.getRandomValues(new Uint8Array(12));

  const cryptoKey = await crypto.subtle.importKey("raw", key, "AES-GCM", false, ["encrypt"]);

  const encodedAad = new TextEncoder().encode(aad);

  const ciphertext = await crypto.subtle.encrypt({ name: "AES-GCM", iv, additionalData: encodedAad }, cryptoKey, data);

  return { iv, ciphertext };

}
```

De Next.js-client combineert deze functies voor alle encryptie-operaties, zoals bestandversleuteling, ratchets en key capsules.

5.5 Sleutelrotatie en vernietiging

Elke sleutel heeft een levenscyclus:

Fase	Actie	Beschrijving
Generatie	`random_bytes(32)` of afleiding via HKDF.	Sleutel aangemaakt met sterke entropie.
Gebruik	Encryptie/decryptie van specifieke payload.	Nooit hergebruik van nonce of key.
Vernietiging	`sodium_memzero($key)` of overschrijven in browser-heap.	Geen residuele data.

De client houdt een tijdelijke cache van actieve sleutels in `IndexedDB`. Bij uitloggen of tab-sluiting worden de entries meteen verwijderd. De server kan slechts zien dat een actie heeft plaatsgevonden, niet welke sleutel werd gebruikt.

5.6 Samenvatting

- Sleutels worden uitsluitend afgeleid via **HKDF-SHA-512** met unieke salts.
- **APP_KEY** van Laravel wordt nooit gebruikt voor cryptografische afleiding.
- ML-KEM wordt alleen gebruikt voor veilige key encapsulation; alle data encryptie gebeurt met AES-256-GCM.
- Sleutels worden na gebruik vernietigd; de server heeft geen toegang tot het sleutelmateriaal.

Geweldig — hieronder volgt **Deel 3 van het Nederlandstalige projectdocument**, in dezelfde Markdown-stijl als de vorige delen. Dit deel behandelt alles van **data-encryptie en ratchets** tot en met de **1Password-integratie** en de **sleuteloverdrachtscapsule**.

Deel 3 – Data-encryptie & Sleuteloverdracht

Inhoud

Integratie met Laravel / Next.js / 1Password

8.1 Opslag van sleutels in 1Password
8.2 SDK-gebruik (service-accounts)
8.3 Client-implementatie

Sleuteloverdrachtscapsule (Key Transfer Capsule)

9.1 Doelstelling
9.2 Ontwerp-overzicht
9.3 Cryptografisch proces
9.4 Laravel-gateway (API)
9.5 Beveiliging en operationele aandachtspunten
9.6 Toekomstige uitbreidingen

---

6. Data-encryptie

6.1 Data-at-Rest

Alle opgeslagen bestanden worden client-side versleuteld met **AES-256-GCM**. Elke encryptie gebruikt:

- een **nieuwe 32-byte sleutel (DEK)**,
- een **unieke 12-byte IV (Initial Vector)**,
- en een **authenticatietag (16 bytes)**.

// Next.js: encryptie van bestand vóór upload

const dek = crypto.getRandomValues(new Uint8Array(32));

const iv = crypto.getRandomValues(new Uint8Array(12));

const cryptoKey = await crypto.subtle.importKey("raw", dek, "AES-GCM", false, ["encrypt"]);

const ciphertext = await crypto.subtle.encrypt({ name: "AES-GCM", iv }, cryptoKey, fileBuffer);

De versleutelde DEK wordt vervolgens verpakt met **ML-KEM-1024** en **HKDF-SHA-512** om veilig te delen met de beoogde ontvanger(s).

6.2 Data-in-Transit

Bij verzending tussen clients of services wordt het verkeer beschermd door:

- TLS 1.3 met **FIPS-goedgekeurde suites**,
- en daarnaast een **extra applicatielaag-encryptie** met **AES-GCM**, zodat zelfs een gecompromitteerde TLS-sessie geen plaintext onthult.

De sleutel voor deze applicatielaag wordt per sessie afgeleid via HKDF en ML-KEM.

6.3 Bestands- en berichtstructuren

Bestandsenvelop

```
{

  "version": 1,

  "file_id": "f_12345",

  "dek_enc": "b64u(...)",     // DEK versleuteld via ML-KEM

  "ciphertext": "b64u(...)",  // bestand zelf

  "iv": "b64u(...)",          // 12 B IV

  "tag": "b64u(...)",         // 16 B auth-tag

  "aad": "file|v1|owner=u42",

  "hash": "blake2b512(ciphertext)"

}
```

Bericht-envelop

Zelfde structuur maar met `msg_id`, `timestamp` en `ratchet_ctr`.

6.4 Integriteits- en foutafhandeling

- Elke decryptie valideert de **authenticatietag**.
- Fouten leveren uitsluitend de melding `Decryption Failed (Integrity Check Error)` om lekken te voorkomen.
- BLAKE2b-hashes worden vergeleken via `hash_equals()` om timing-aanvallen te vermijden.

---

7. Double Ratchet-mechanisme

7.1 Doel en principes

Het **Double Ratchet-mechanisme** zorgt voor continue sleutelvernieuwing per bericht of bestand. Elke stap gebruikt een eenrichtingsfunctie; oude sleutels worden vernietigd. Voordelen:

- **Forward Secrecy (FS)** – oude berichten blijven veilig,
- **Post-Compromise Security (PCS)** – na herstel blijven nieuwe berichten vertrouwelijk.

7.2 Sleutelketens

Elke gebruiker heeft twee ketens:

- **sendChain** en **recvChain**. Elke ratchet-stap gebruikt `HMAC-SHA-512`:

sendCK' = HMAC(sendCK, "step")

msgKey = HMAC(sendCK', "msg")

Na gebruik worden `msgKey` en de vorige `sendCK` overschreven.

7.3 Forward Secrecy & Post-Compromise Security

- FS: oude sleutels worden nooit bewaard.
- PCS: bij elke nieuw ontvangen ML-KEM-mix-in (`shared`) wordt de root-key hernieuwd. Zodra een aanvaller geen live toegang meer heeft, verliest hij toekomstige decrypties.

7.4 Implementatie-overzicht

- Ketens en tellers worden lokaal in **IndexedDB** bewaard.
- Er wordt geen server-sync gebruikt, behalve optionele versienummers (`epoch`, `ctr`).
- Alle verificatie gebeurt client-side via `aad`-bindings.

---

8. Integratie met Laravel / Next.js / 1Password

8.1 Opslag van sleutels in 1Password

De *Master Private Key* (**MPK**) blijft lokaal, maar de gebruiker kan deze veilig **exporteren naar 1Password** als back-up. Omdat 1Password geen post-quantum-sleuteltypes ondersteunt, wordt de sleutel opgeslagen als **aangepast item** (Secure Note / API Credential).

8.2 SDK-gebruik (service-accounts)

```
npm install @1password/sdk

export OP_SERVICE_ACCOUNT_TOKEN=<token>

Voorbeeld in JavaScript:

import { createClient } from "@1password/sdk";

const client = await createClient({

  auth: process.env.OP_SERVICE_ACCOUNT_TOKEN,

  integrationName: "KeyPact Quantum App",

  integrationVersion: "v1.0.0"

});

await client.items.create({

  vault: { id: "vault-id" },

  category: "SECURE_NOTE",

  title: "ML-KEM-1024 Sleutel",

  fields: [

    { label: "Public Key", type: "STRING", value: publicKeyB64 },

    { label: "Ciphertext", type: "STRING", value: kemCtB64 },

    { label: "Algorithm", type: "STRING", value: "ML-KEM-1024" }

  ]
```

});

De sleutel blijft end-to-end-versleuteld binnen 1Password.

8.3 Client-implementatie

- De browser gebruikt de **WebCrypto API** voor AES-GCM.
- ML-KEM-handelingen kunnen via een **WASM-binding** van liboqs.
- De MPK kan optioneel worden geëxporteerd als versleuteld blob voor back-up.

---

9. Sleuteloverdrachtscapsule (Key Transfer Capsule)

9.1 Doelstelling

Een veilige **off-site back-up en hersteloptie** voor de Master Private Key, zonder concessies aan end-to-end-encryptie. De capsule kan door één of meerdere trustees (1 : n) worden opgeslagen; alleen de eigenaar kan ontsleutelen met de **Recovery Secret Key (RSK)**.

9.2 Ontwerp-overzicht

Rol	Verantwoordelijkheid
Eigenaar	Genereert RSK + RPK (ML-KEM-1024) en maakt capsule.
Trustees (1 : n)	Bewaren de capsule; kunnen niet ontsleutelen.
Server (Laravel)	Valideert, logt en slaat capsule immutabel op.

Principe:

> Verlies van RSK = definitief verlies van data. Geen sleutel-escrow of server-herstel.

9.3 Cryptografisch proces

1. Genereer tijdelijke 32-byte RK.
2. `mpk_ct = AES-256-GCM(RK, MPK)`

3. `(kem_ct, shared) = ML-KEM-1024.encap(RPK)`
4. `wrapKey = HKDF-SHA-512(shared, info="keypact|rkwrap|v1")`
5. `rk_ct = AES-256-GCM(wrapKey, RK)`
6. Combineer alle waarden tot capsule + BLAKE2b-hash.


```
{
  "v": 1,
  "scheme": "ml-kem-1024|aes256gcm",
  "owner_uid": "u_42",
  "created": 1759257600,
  "rp_fingerprint": "b64u(blake2b(RPK))",
  "kem_ct": "b64u(...)",
  "rk_ct": "b64u(...)",
  "iv_rk": "b64u(12B)",
  "tag_rk": "b64u(16B)",
  "mpk_ct": "b64u(...)",
  "iv_mpk": "b64u(12B)",
  "tag_mpk": "b64u(16B)",
  "policy": { "expires": 1780793600, "label": "Owner MPK Recovery v1" },
  "capsule_hash": "b64u(blake2b(canonical_json_without_this_field))"
}
```

9.4 Laravel-gateway (API)

Endpoint	**Doel**
POST /api/keytransfer/capsules	Valideer schema + `capsule_hash`; sla immutabel op.
GET /api/keytransfer/capsules?	Toon metadata van capsules.

Endpoint	Doel
owner_uid=…	
GET /api/keytransfer/capsules/{hash}	Haal capsule op (bij retour).
POST /api/keytransfer/dispatch	Registreer verzending naar trustee (audit).
POST /api/keytransfer/return/{hash}	Registreer retour van capsule.

Opslag is **append-only**. Rollback-detectie via `HMAC(server_secret, capsule_hash || metadata)`. Server verwerkt nooit RSK, RK of MPK.

9.5 Beveiliging en operationele aandachtspunten

- Server blijft **cryptografisch blind**.
- Eén capsule kan veilig door meerdere trustees worden bewaard.
- ML-KEM-1024 → post-quantum-bestendig.
- Simpel herstel: één RSK → één capsule.
- **Verlies van RSK = onomkeerbaar verlies van data.**
- Auditspoor via `capsule_hash` en optionele hash-keten.

9.6 Toekomstige uitbreidingen

- **Device-attestatie** bij retour van capsules.
- **Multi-factor unlock** (RSK + WebAuthn-signatuur).
- **QR-weergave** voor offline/papieren back-ups.

Samenvatting Deel 3

- Bestanden en berichten zijn volledig versleuteld **client-side**.
- De **Double Ratchet** garandeert forward- en post-compromise security.
- **1Password-integratie** biedt veilige sleutelopslag.
- De **Sleuteloverdrachtscapsule** maakt off-site herstel mogelijk zonder risico op sleutel-exfiltratie.

Perfect 🔥 — dan maken we dit laatste deel **volledig en geïntegreerd**, inclusief alle 8 onderdelen die hierboven zijn uitgewerkt, maar nu als **één samenhangend Markdown-hoofdstuk** dat je zo kunt plakken in je projectdocument.

Bestanden openen door de eigenaar (Client-side decryptie)

10.1 Overzicht

Dit hoofdstuk beschrijft stap voor stap hoe de **eigenaar** van een bestand een op de server opgeslagen **versleuteld bestand** opnieuw kan openen en ontsleutelen, zonder dat de server ooit toegang krijgt tot sleutelmateriaal.

Het proces is volledig **client-side** en maakt gebruik van dezelfde **cryptografische bouwstenen** als de rest van het platform: **ML-KEM-1024**, **HKDF-SHA-512**, **AES-256-GCM**, en optioneel de **Double Ratchet-keten**.

De sleutel tot succes: ieder bestand bevat in zijn *envelope* altijd een **self-recipient**-entry, zodat de eigenaar zichzelf ook als ontvanger kan ontsleutelen.

10.2 Stappenoverzicht

Stap	Beschrijving	Component
1	Envelope en ciphertext ophalen bij de Laravel API.	Server
2	MPK (Master Private Key) laden of ontsleutelen.	Client
3	Self-recipient entry zoeken in de envelope.	Client
4	ML-KEM-decaps uitvoeren met MPK → `shared`.	Client
5	`wrapKey` afleiden via HKDF-SHA-512.	Client
6	`MK` + `IV` genereren via Ratchet- of HMAC-afleiding.	Client
7	DEK (Data Encryption Key) ontsleutelen.	Client
8	Bestand of chunks decrypten met DEK.	Client

10.3 Envelope-structuur (vereenvoudigd)

{

```
"v": 3,
"file_id": "f_12345",
"data_algo": "AES-256-GCM",
"kdf": "HKDF-SHA512",
"epoch": 7,
"aad": {
  "owner_id": "u_42",
  "created": 1759257600,
  "algo_id": "aes256gcm|hkdf512|mlkem1024"
},
"dr_nonce": "b64u(12B)",
"ciphertext": "b64u(...)",
"tag": "b64u(16B)",
"recipients": [
  {
    "uid": "u_42",
    "kem": "ml-kem-1024",
    "kem_ct": "b64u(...)",
    "kem_ct_hash": "b64u(blake2b(kem_ct))",
    "w_ctr": 17,
    "sealed": "b64u(...)",
    "tag": "b64u(16B)"
  }
],
```

```
  "prev_header_hash": "b64u(...)"
}
```

Elke gebruiker die toegang heeft, heeft een eigen entry in `recipients`. De eigenaar heeft altijd een **self-recipient** (eigen `uid`), noodzakelijk voor lokaal herstel.

10.4 Laravel-API – ophalen van data

Endpoints

GET /api/files/{fileId}/envelope

GET /api/files/{fileId}/blob

Controller-voorbeeld

```
public function showEnvelope(string $fileId)
{
    $env = Envelope::where('file_id', $fileId)->firstOrFail();
    $this->authorize('view', $env);
    return response()->json($env->json_payload);
}
public function showBlob(string $fileId)
{
    $file = EncryptedFile::where('file_id', $fileId)->firstOrFail();
    $this->authorize('view', $file);
    return response($file->cipher_blob, 200, [
        'Content-Type' => 'application/octet-stream',
        'Content-Disposition' => 'attachment; filename="cipher.bin"',
        'Cache-Control' => 'no-store',
    ]);
```

}

De server controleert enkel de toegang en levert de opgeslagen data. Decryptie gebeurt **uitsluitend aan de clientzijde**.

10.5 Client-side decryptie (monolithisch bestand)

Voorwaarden

- De **MPK** (Master Private Key) is beschikbaar in RAM.
- De **envelope** bevat een `self-recipient`.
- Er is een **WASM-binding** voor `mlkem1024Decap()` aanwezig.

Voorbeeld (Next.js / TypeScript)

```
export async function openOwnedFile(fileId: string, mpkRaw: Uint8Array) {

  const env = await (await fetch(`/api/files/${fileId}/envelope`)).json();

  const blobB64u = await (await fetch(`/api/files/${fileId}/blob`)).text();

  const cipher = b64uDec(blobB64u);

  const me = env.recipients.find((r: any) => r.uid === env.aad.owner_id);

  if (!me) throw new Error("Self-recipient ontbreekt");

  const kemCt = b64uDec(me.kem_ct);

  const shared = await mlkem1024Decap(mpkRaw, kemCt);

  const wrapKey = await hkdfSha512(shared, "keypact|rkwrap|v1");

  const mkKey = await crypto.subtle.importKey("raw", wrapKey, { name: "HMAC", hash:
"SHA-512" }, false, ["sign"]);

  const mk = new Uint8Array(await crypto.subtle.sign("HMAC", mkKey, new
TextEncoder().encode(`mk|${me.w_ctr}`))).slice(0, 32);

  const iv = new Uint8Array(await crypto.subtle.sign("HMAC", mkKey, new
TextEncoder().encode(`nonce|${me.w_ctr}`))).slice(0, 12);

  const sealed = b64uDec(me.sealed);
```

```
  const dek = await aesGcmDecrypt(mk, iv, sealed, `wrap|${me.kem}|uid=${me.uid}|file=${env.file_id}`);

  const drIv = b64uDec(env.dr_nonce);

  const pt = await aesGcmDecrypt(dek, drIv, b64uDec(env.ciphertext), JSON.stringify(env.aad));

  return pt; // Uint8Array met de originele bytes

}
```

10.6 Grote bestanden – chunked/streaming decryptie

Manifest (server-side opgeslagen)

```
{

  "v": 1,

  "file_id": "f_12345",

  "data_algo": "AES-256-GCM",

  "chunk_size": 4194304,

  "total_size": 987654321,

  "chunks": [

    { "i": 0, "iv": "b64u(12B)", "tag": "b64u(16B)", "len": 4194304 },

    { "i": 1, "iv": "b64u(12B)", "tag": "b64u(16B)", "len": 4194304 }

  ],

  "aad": { "owner_id": "u_42", "algo_id": "aes256gcm|hkdf512|mlkem1024" },

  "recipients": [ /* self + anderen */ ],

  "manifest_hash": "b64u(blake2b(canonical_json_without_hash))"

}
```

Client-streamdecryptie

```
async function openOwnedFileStreamed(fileId: string, mpkRaw: Uint8Array) {

  const man = await (await fetch(`/api/files/${fileId}/manifest`)).json();

  const me = man.recipients.find((r: any) => r.uid === man.aad.owner_id);

  if (!me) throw new Error("Self-recipient ontbreekt");

  const shared = await mlkem1024Decap(mpkRaw, b64uDec(me.kem_ct));

  const wrapKey = await hkdfSha512(shared, "keypact|rkwrap|v1");

  const dek = await aesGcmDecrypt(wrapKey, b64uDec(me.iv), b64uDec(me.sealed), `wrap|
${me.kem}|uid=${me.uid}|file=${man.file_id}`);

  const cryptoKey = await crypto.subtle.importKey("raw", dek, "AES-GCM", false, ["decrypt"]);

  for (const c of man.chunks) {

    const resp = await fetch(`/api/files/${fileId}/chunks/${c.i}`);

    const ctChunk = new Uint8Array(await resp.arrayBuffer());

    const ptChunk = new Uint8Array(await crypto.subtle.decrypt(

      { name: "AES-GCM", iv: b64uDec(c.iv), additionalData: new
TextEncoder().encode(JSON.stringify(man.aad)), tagLength: 128 },

      cryptoKey,

      ctChunk

    ));

    // schrijf chunk naar disk, geheugen of preview

  }

}
```

Chunked decryptie voorkomt geheugenproblemen en maakt hervatten of parallelle downloads mogelijk.

10.7 Multi-tab & incognito sessies

- Elke tab heeft zijn eigen **crypto-context**; de MPK mag nooit persistent worden opgeslagen.
- Sleutel-uitwisseling tussen tabbladen kan via een **BroadcastChannel**, maar enkel binnen één actieve sessie.
- **IndexedDB** mag alleen cipher-blobs opslaan (nooit sleutels).
- In **incognito-modus** werkt hetzelfde, maar data wordt verwijderd bij het sluiten.

10.8 Foutafhandeling en veiligheid

Regel	Omschrijving
Strict AAD-binding	Gebruik bij elke decryptie dezelfde `aad`-structuur (`file_id`, `owner_id`, `epoch`, enz.).
Rollback-detectie	Controleer `prev_header_hash` of `manifest_hash` – afwijking → blokkeren.
Unieke IV's	Elk bestand of chunk heeft zijn eigen 12-byte IV; nooit hergebruiken.
Zeroisation	Overschrijf buffers (`MK`, `DEK`, `shared`, `MPK`) zodra decryptie voltooid is.
Foutmeldingen	Alleen generieke foutmelding tonen: *"Decryption failed (integrity)"* – geen technische details aan gebruiker.
Audit-logging	Log enkel gebeurtenisniveau (download, decrypt, retour) – geen sleutels of hashes.

10.9 Teststrategie

Testtype	Doel	Verwacht resultaat
KAT – AES-GCM	Controle van encryptie/decryptie tegen NIST-vectors.	Identieke plaintext-herstel.
KAT – ML-KEM-1024	Test vectorvalidatie van liboqs.	Gelijke `shared`-waarde.
Integratie (Node <-> Browser)	Bestand encrypten in browser, decrypten in Node,	Cross-platform consistentie.

Testtype	Doel	Verwacht resultaat
	en omgekeerd.	
Chunk corruptie	Tag of byte wijzigen.	“Decryption failed (integrity)”.
Rollback test	Oude manifest opnieuw uploaden.	Detectie en afwijzing.

10.10 Implementatie-checklist

- [ ] Self-recipient aanwezig in elke envelope.
- [ ] Endpoints `/envelope`, `/blob`, `/chunks/{i}` geïmplementeerd.
- [ ] Clientfuncties `openOwnedFile()` en `openOwnedFileStreamed()`.
- [ ] Unlock-flow voor MPK via passphrase.
- [ ] WASM-binding `mlkem1024Decap()`.
- [ ] AAD-binding en hash-controle.
- [ ] Zeroisation van gevoelige buffers.
- [ ] Rollback-detectie bij mismatches.

Helemaal goed — hieronder staat de **referentie-implementatie** die aansluit op Hoofdstuk 10. Alles is **framework-klaar**, minimalistisch maar production-waardig qua structuur. Je kunt dit direct in je mono-repo plakken en iteratief uitbouwen.

📦 Laravel (API/gateway) – Referentie-implementatie

Doel: **geen cryptografie** op de server — alleen autoriseren, valideren, opslaan en terugleveren.

1) Migrations

// database/migrations/2025_10_04_000001_create_envelopes_table.php

use Illuminate\Database\Migrations\Migration;

use Illuminate\Database\Schema\Blueprint;

use Illuminate\Support\Facades\Schema;

```
return new class extends Migration {
    public function up(): void {
        Schema::create('envelopes', function (Blueprint $t) {
            $t->uuid('file_id')->primary();
            $t->string('owner_uid', 64)->index();
            $t->unsignedBigInteger('epoch')->default(1);
            $t->json('json_payload');                    // volledige envelope JSON
            $t->binary('prev_header_hash')->nullable();     // binaire hash
            $t->timestamps();
        });
    }
    public function down(): void { Schema::dropIfExists('envelopes'); }
};
// database/migrations/2025_10_04_000002_create_encrypted_files_table.php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
return new class extends Migration {
    public function up(): void {
        Schema::create('encrypted_files', function (Blueprint $t) {
            $t->uuid('file_id')->primary();
            $t->string('owner_uid', 64)->index();
            $t->string('storage_disk')->default('s3');      // of 'local'
            $t->string('storage_path');                     // pad in object store
```

```
            $t->unsignedBigInteger('size');
            $t->string('cipher_hash', 128);                 // blake2b512 hex
            $t->timestamps();
        });
    }
    public function down(): void { Schema::dropIfExists('encrypted_files'); }
};
// database/migrations/2025_10_04_000003_create_encrypted_chunks_table.php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
return new class extends Migration {
    public function up(): void {
        Schema::create('encrypted_chunks', function (Blueprint $t) {
            $t->id();
            $t->uuid('file_id')->index();
            $t->unsignedInteger('index');                   // chunk index
            $t->string('storage_disk')->default('s3');
            $t->string('storage_path');                     // pad per chunk
            $t->unsignedInteger('len');
            $t->string('iv_b64u', 32);
            $t->string('tag_b64u', 32);
            $t->timestamps();
            $t->unique(['file_id','index']);
```

```
        });
    }
    public function down(): void { Schema::dropIfExists('encrypted_chunks'); }
};
```

2) Models

```
// app/Models/Envelope.php
namespace App\Models;
use Illuminate\Database\Eloquent\Model;
class Envelope extends Model {
    protected $fillable = ['file_id','owner_uid','epoch','json_payload','prev_header_hash'];
    protected $casts = ['json_payload' => 'array'];
    public $incrementing = false; protected $keyType = 'string';
}
// app/Models/EncryptedFile.php
namespace App\Models;
use Illuminate\Database\Eloquent\Model;
class EncryptedFile extends Model {
    protected $fillable = ['file_id','owner_uid','storage_disk','storage_path','size','cipher_hash'];
    public $incrementing = false; protected $keyType = 'string';
}
// app/Models/EncryptedChunk.php
namespace App\Models;
use Illuminate\Database\Eloquent\Model;
class EncryptedChunk extends Model {
```

```
    protected $fillable = ['file_id','index','storage_disk','storage_path','len','iv_b64u','tag_b64u'];
}
```

3) Policies (optioneel, maar aanbevolen)

```
// app/Policies/FilePolicy.php
namespace App\Policies;
use App\Models\User;
use App\Models\EncryptedFile;
class FilePolicy {
    public function view(User $user, EncryptedFile $file): bool {
        // Basis: eigenaar of expliciet gedeeld; breid uit met ACLs
        return $user->id === $file->owner_uid || $user->can("files.view.{$file->file_id}");
    }
}
```

Registreer in `AuthServiceProvider`.

4) Controllers

```
// app/Http/Controllers/FileReadController.php
namespace App\Http\Controllers;
use App\Models\Envelope;
use App\Models\EncryptedFile;
use App\Models\EncryptedChunk;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Storage;
class FileReadController extends Controller
{
```

```
public function envelope(string $fileId) {
    $env = Envelope::where('file_id', $fileId)->firstOrFail();
    $file = EncryptedFile::where('file_id',$fileId)->firstOrFail();
    $this->authorize('view', $file);
    return response()->json($env->json_payload);
}
// monolithische blob (kleinere bestanden)
public function blob(string $fileId) {
    $file = EncryptedFile::where('file_id',$fileId)->firstOrFail();
    $this->authorize('view', $file);
    return Storage::disk($file->storage_disk)->download(
        $file->storage_path,
        basename($file->storage_path),
        ['Cache-Control'=>'no-store, no-cache, must-revalidate']
    );
}
// chunked – levert één chunk
public function chunk(string $fileId, int $index) {
    $file = EncryptedFile::where('file_id',$fileId)->firstOrFail();
    $this->authorize('view', $file);
    $chunk = EncryptedChunk::where(['file_id'=>$fileId, 'index'=>$index])->firstOrFail();
    $stream = Storage::disk($chunk->storage_disk)->readStream($chunk->storage_path);
    return response()->stream(function() use ($stream) {
        fpassthru($stream);
```

```
        }, 200, [
            'Content-Type' => 'application/octet-stream',
            'Content-Length' => $chunk->len,
            'Cache-Control' => 'no-store',
        ]);
    }
    // optioneel: manifest voor chunked decryptie
    public function manifest(string $fileId) {
        $env = Envelope::where('file_id', $fileId)->firstOrFail();
        $file = EncryptedFile::where('file_id',$fileId)->firstOrFail();
        $this->authorize('view', $file);
        $chunks = EncryptedChunk::where('file_id',$fileId)->orderBy('index')->get(['index','iv_b64u','tag_b64u','len']);
        $payload = $env->json_payload;
        $payload['chunk_size'] = optional($chunks->first())->len ?? null;
        $payload['chunks'] = $chunks->map(fn($c)=>['i'=>$c->index,'iv'=>$c->iv_b64u,'tag'=>$c->tag_b64u,'len'=>$c->len])->values();
        return response()->json($payload);
    }
}
```

5) Routes

```
// routes/api.php
use App\Http\Controllers\FileReadController;
Route::middleware('auth:sanctum')->group(function () {
    Route::get('/files/{fileId}/envelope', [FileReadController::class, 'envelope']);
```

```
    Route::get('/files/{fileId}/blob',     [FileReadController::class, 'blob']);

    Route::get('/files/{fileId}/manifest', [FileReadController::class, 'manifest']);

    Route::get('/files/{fileId}/chunks/{index}', [FileReadController::class, 'chunk'])

        ->whereNumber('index');

});
```

> **Opslag**: configureer `filesystems.php` voor `s3` of `local`. **Belangrijk**: **geen** plaintext, sleutels of KEM-artefacten worden berekend op de server.

🌐 Next.js (client) – Referentie-implementatie

1) WASM-adapter (liboqs) – interface

> Je kunt ieder WASM-pakket gebruiken dat ML-KEM-1024 decapsulation aanbiedt. We definiëren een dunne adapter zodat de rest van de code stabiel blijft.

```
// lib/wasm/mlkem.ts

export interface MlKem {

  ready(): Promise<void>;

  decap(secretKey: Uint8Array, kemCt: Uint8Array): Promise<Uint8Array>; // returns shared (32B)

}

let _impl: MlKem | null = null;

export function setMlKemAdapter(impl: MlKem) { _impl = impl; }

export function mlkem(): MlKem {

  if (!_impl) throw new Error("ML-KEM adapter not initialised");

  return _impl;

}
```

(In je app bootstrap stel je `_impl` in met de echte WASM binding.)

2) Utilities

```
// lib/crypto/base64url.ts

export const b64uDec = (s: string) =>

  new Uint8Array(Buffer.from(s.replace(/-/g, '+').replace(/_/g, '/') + '==='.slice((s.length + 3) %
4), 'base64'));

export const b64uEnc = (b: Uint8Array) =>

  Buffer.from(b).toString('base64').replace(/\+/g,'-').replace(/\//g,'_').replace(/=+$/,'');

// lib/crypto/hkdf.ts (HKDF-SHA-512 -> 32B)

export async function hkdfSha512(input: Uint8Array, info: string): Promise<Uint8Array> {

  const k = await crypto.subtle.importKey('raw', input, { name: 'HMAC', hash: 'SHA-512' },
false, ['sign']);

  const res = new Uint8Array(await crypto.subtle.sign('HMAC', k, new
TextEncoder().encode(info)));

  return res.slice(0, 32);

}

// lib/crypto/aes.ts

export async function aesGcmDecrypt(key: Uint8Array, iv12: Uint8Array, ctWithTag:
Uint8Array, aad?: string) {

  const k = await crypto.subtle.importKey('raw', key, 'AES-GCM', false, ['decrypt']);

  const opts: AesGcmParams = { name: 'AES-GCM', iv: iv12, tagLength: 128 };

  if (aad) (opts as any).additionalData = new TextEncoder().encode(aad);

  const pt = await crypto.subtle.decrypt(opts, k, ctWithTag);

  return new Uint8Array(pt);

}
```

3) Hook: **useOpenOwnedFile** (monolithisch)

```
// hooks/useOpenOwnedFile.ts

import { b64uDec } from '@/lib/crypto/base64url';

import { hkdfSha512 } from '@/lib/crypto/hkdf';

import { aesGcmDecrypt } from '@/lib/crypto/aes';

import { mlkem } from '@/lib/wasm/mlkem';

type Envelope = {

  v:number; file_id:string; dr_nonce:string; ciphertext:string; aad:any;

  recipients: Array<{ uid:string; kem:string; kem_ct:string; w_ctr:number; sealed:string }>;

};

export function useOpenOwnedFile() {

  async function open(fileId: string, mpk: Uint8Array): Promise<Uint8Array> {

    const env: Envelope = await (await fetch(`/api/files/${fileId}/envelope`)).json();

    const me = env.recipients.find(r => r.uid === env.aad.owner_id);

    if (!me) throw new Error('Self-recipient ontbreekt');

    // KEM decap → shared

    await mlkem().ready();

    const shared = await mlkem().decap(mpk, b64uDec(me.kem_ct));

    // wrapKey & per-wrap MK/IV

    const wrapKey = await hkdfSha512(shared, 'keypact|rkwrap|v1');

    const mkKey = await crypto.subtle.importKey('raw', wrapKey, { name: 'HMAC', hash:
'SHA-512' }, false, ['sign']);

    const mk = new Uint8Array(await crypto.subtle.sign('HMAC', mkKey, new
TextEncoder().encode(`mk|${me.w_ctr}`))).slice(0,32);

    const iv = new Uint8Array(await crypto.subtle.sign('HMAC', mkKey, new
TextEncoder().encode(`nonce|${me.w_ctr}`))).slice(0,12);
```

```
    // Unwrap DEK

    const dek = await aesGcmDecrypt(mk, iv, b64uDec(me.sealed), `wrap|${me.kem}|uid=${me.uid}|file=${env.file_id}`);

    // Decrypt file

    const pt = await aesGcmDecrypt(dek, b64uDec(env.dr_nonce), b64uDec(env.ciphertext), JSON.stringify(env.aad));

    // Zeroisation (best-effort)

    dek.fill(0); mk.fill(0); wrapKey.fill(0); shared.fill(0);

    return pt;

  }

  return { open };

}
```

4) Hook: **useOpenOwnedFileStreamed** (chunked)

```
// hooks/useOpenOwnedFileStreamed.ts

import { b64uDec } from '@/lib/crypto/base64url';

import { hkdfSha512 } from '@/lib/crypto/hkdf';

import { aesGcmDecrypt } from '@/lib/crypto/aes';

import { mlkem } from '@/lib/wasm/mlkem';

type Manifest = {

  v:number; file_id:string; aad:any;

  chunks: Array<{ i:number; iv:string; tag:string; len:number }>;

  recipients: Array<{ uid:string; kem:string; kem_ct:string; w_ctr:number; sealed:string }>;

};

export function useOpenOwnedFileStreamed() {
```

```
  async function open(fileId: string, mpk: Uint8Array, onChunk:(pt:Uint8Array,
idx:number)=>Promise<void>) {

    const man: Manifest = await (await fetch(`/api/files/${fileId}/manifest`)).json();

    const me = man.recipients.find(r => r.uid === man.aad.owner_id);

    if (!me) throw new Error('Self-recipient ontbreekt');

    await mlkem().ready();

    const shared = await mlkem().decap(mpk, b64uDec(me.kem_ct));

    const wrapKey = await hkdfSha512(shared, 'keypact|rkwrap|v1');

    // MK/IV zoals in monolithisch pad, of bewaar DEK direct in mk/iv in envelope:

    const mkKey = await crypto.subtle.importKey('raw', wrapKey, { name: 'HMAC', hash:
'SHA-512' }, false, ['sign']);

    const mk = new Uint8Array(await crypto.subtle.sign('HMAC', mkKey, new
TextEncoder().encode(`mk|${me.w_ctr}`))).slice(0,32);

    const ivw = new Uint8Array(await crypto.subtle.sign('HMAC', mkKey, new
TextEncoder().encode(`nonce|${me.w_ctr}`))).slice(0,12);

    const dek = await aesGcmDecrypt(mk, ivw, b64uDec(me.sealed), `wrap|${me.kem}|uid=$
{me.uid}|file=${man.file_id}`);

    const dekKey = await crypto.subtle.importKey('raw', dek, 'AES-GCM', false, ['decrypt']);

    const aad = new TextEncoder().encode(JSON.stringify(man.aad));

    for (const c of man.chunks) {

      const resp = await fetch(`/api/files/${fileId}/chunks/${c.i}`);

      const ctChunk = new Uint8Array(await resp.arrayBuffer());

      const ptChunk = new Uint8Array(await crypto.subtle.decrypt(

        { name:'AES-GCM', iv: b64uDec(c.iv), additionalData: aad, tagLength:128 }, dekKey,
ctChunk

      ));

      await onChunk(ptChunk, c.i);
```

```
    }

    // Zeroisation

    dek.fill(0); mk.fill(0); wrapKey.fill(0); shared.fill(0);

  }

  return { open };

}
```

5) MPK-unlock (voorbeeld)

```
// lib/mpk/unlock.ts

import { aesGcmDecrypt } from '@/lib/crypto/aes';

import { b64uDec } from '@/lib/crypto/base64url';

export async function unlockMpkFromVaultBlob(blob: any, passphrase: string):
Promise<Uint8Array> {

  // Voorbeeld met PBKDF2; vervang door Argon2id voor productie

  const salt = b64uDec(blob.kdf.salt);

  const baseKey = await crypto.subtle.importKey('raw', new
TextEncoder().encode(passphrase), 'PBKDF2', false, ['deriveKey']);

  const kek = await crypto.subtle.deriveKey(

    { name:'PBKDF2', hash:'SHA-256', salt, iterations: blob.kdf.iterations },

    baseKey, { name:'AES-GCM', length:256 }, false, ['decrypt']

  );

  const iv = b64uDec(blob.iv);

  const ct = b64uDec(blob.ct);

  const k = new Uint8Array(await crypto.subtle.exportKey('raw', kek));

  const mpk = await aesGcmDecrypt(k, iv, ct, blob.aad);

  k.fill(0);
```

```
  return mpk;

}
```

□ Extra: Minimal upload-pad (client) voor **self-recipient**

> Zodat elk bestand later door de eigenaar geopend kan worden, **moet** de client bij upload een **self-recipient** entry opnemen.

```
// lib/upload/makeSelfRecipient.ts

import { mlkem } from '@/lib/wasm/mlkem';

import { hkdfSha512 } from '@/lib/crypto/hkdf';

import { b64uEnc } from '@/lib/crypto/base64url';

export async function makeSelfRecipient(ownerUid: string, ownerPubKey: Uint8Array, dek: Uint8Array, fileId: string, wCtr: number) {

  await mlkem().ready();

  const { kemCt, shared } = await (async () => {

    // aanname: je WASM adapter heeft ook encap; zo niet, doe encap server-side (pubkey-only) en doe uitsluitend unwrap client-side

    const kemCt = await (mlkem() as any).encap(ownerPubKey); // of ander pad

    const shared = (mlkem() as any).sharedFromLastEncap as Uint8Array; // afhankelijk van implementatie

    return { kemCt, shared };

  })();

  const wrapKey = await hkdfSha512(shared, 'keypact|rkwrap|v1');

  const mkKey = await crypto.subtle.importKey('raw', wrapKey, { name: 'HMAC', hash: 'SHA-512' }, false, ['sign']);

  const mk = new Uint8Array(await crypto.subtle.sign('HMAC', mkKey, new TextEncoder().encode(`mk|${wCtr}`))).slice(0,32);
```

```
  const iv = new Uint8Array(await crypto.subtle.sign('HMAC', mkKey, new
TextEncoder().encode(`nonce|${wCtr}`))).slice(0,12);

  // wrap DEK

  const k = await crypto.subtle.importKey('raw', mk, 'AES-GCM', false, ['encrypt']);

  const aad = new TextEncoder().encode(`wrap|ml-kem-1024|uid=${ownerUid}|file=${fileId}`);

  const sealed = new Uint8Array(await crypto.subtle.encrypt({ name:'AES-GCM', iv,
additionalData: aad, tagLength:128 }, k, dek));

  return {

    uid: ownerUid,

    kem: 'ml-kem-1024',

    kem_ct: b64uEnc(kemCt),

    w_ctr: wCtr,

    sealed: b64uEnc(sealed),

  };

}
```

> **Opmerking**: Als je **encap** server-side wilt doen (met alléén de publieke sleutel), blijft E2E intact zolang **decap + unwrap** altijd client-side gebeuren.
