BAB 2

LANDASAN TEORI

2.1 Algoritma Post-Quantum Cryptography CRYSTALS-Kyber

CRYSTALS-Kyber adalah satu dari dua algoritma kriptografi yang diusulkan oleh Cryptographic Suite for Algebraic Lattices (CRYSTALS) pada NIST post-quantum *standardization effort* tahun 2017 lalu (*Cryptology ePrint Archive: Report 2017/634*, 2017). CRYSTALS-Kyber masuk sebagai finalis algoritma *public-key encryption and key-establishment* pada tahap putaran ketiga dari *call for proposals* yang dilakukan oleh NIST (*PQC Standardization Process: Third Round Candidate Announcement*, 2020). CRYSTALS-Kyber adalah sebuah *key-encapsulation mechanism* (KEM) yang tahan dari IND-CCA2 (*Indistinguishability under adaptive Chosen Ciphertext Attack*) dengan basis keamanan *hardness of solving the learning-with-errors problem in module lattices* (Avanzi *et al.*, 2020). Konstruksi CRYSTALS-Kyber terdiri dari dua bagian, yaitu (Avanzi *et al.*, 2020).

1. Skema a*symmetric-encryption* yang tahan dari IND-CPA (KYBER.CPAPKE) dilakukan dengan melakukan enkripsi pesan yang memiliki ukuran tetap yaitu 32 *bytes* (Avanzi *et al.*, 2020).
2. Skema *key-encapsulation mechanism* yang tahan dari IND-CCA2 (KYBER.CCAKEM) dilakukan dengan menggunakan Fujisaki-Okamoto (FO) Transform (Avanzi et al., 2020). FO Transform membuat enkripsi simetrik dan asimetrik menjadi skema hybrid yang tahan dari *chosen ciphertext attack.*

2.1.1 Key Encapsulation Mechanism

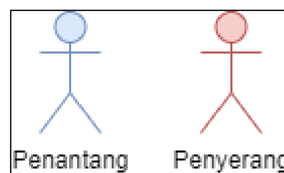
Key-encapsulation mechanism (KEM) bertujuan untuk mengenkapsulasi *bit-strings* (*keys*) yang dipilih secara *random*, sehingga keluaran yang dihasilkan dari KEM berupa sebuah *key* yang dipilih secara *random* dan hasil enkapsulasi *key* tersebut. *Key* ini selanjutnya akan digunakan kembali untuk mengenkapsulasi data dengan skema *symmetric encryption*. Enkapsulasi data ini dikenal dengan sebutan *Data Encapsulation Mechanism* (DEM). Skema inipun dikenal dengan sebutan *hybrid encryption* atau paradigma KEM-DEM (Albrecht et al., 2019). Sebuah *Key Encapsulation Mechanism* terdiri dari tiga algoritma, yaitu algoritma:

1. *Key Generation*, menghasilkan keluaran berupa sepasang *public* dan *private key* (pk, sk) (Albrecht *et al.*, 2019).
2. *Encapsulation*, menerima masukan sebuah *public key* pk, lalu menghasilkan keluaran berupa sebuah *random key* dan sebuah *ciphertext* (K, C) (Albrecht *et al.*, 2019).
3. *Decapsulation*, menerima masukan sebuah *ciphertext* C dan sebuah *private key* sk, lalu menghasilkan keluaran berupa sebuah *key* K yang sebelumnya dienkapsulasi di dalam *ciphertext* C (Albrecht, Cid, Paterson, Tjhai, & Tomlinson, 2019).

2.1.2 Indistinguishability Under Adaptive Chosen Ciphertext Attack

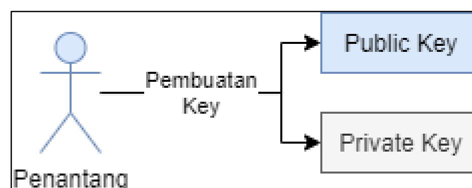
Indistinguishability under Adaptive Chosen Ciphertext Attack atau disingkat sebagai IND-CCA2 merupakan *security notions* terkuat untuk skema *public key encryption* (PKE) (Zeng, Chen and Choo, 2019). Pemodelan IND-CCA2 dapat digambarkan dalam bentuk permainan yang menggunakan *random oracle model* untuk memanggil fungsi enkripsi dan dekripsi algoritma kriptografi terkait yang diuji. Berikut adalah skenario bentuk permainan tersebut (SEJPM, 2015).

1. Terdapat dua buah entitas, yaitu penantang dan penyerang. Penantang bertugas memberikan tantangan kepada penyerang (berupa *ciphertext*). Penyerang bertugas untuk menebak isi dari *ciphertext* yang diberikan oleh penantang. Dalam *security notions* IND-CCA2, penyerang memiliki kemampuan untuk melakukan pemanggilan fungsi enkripsi dan dekripsi pada *random oracle model*.



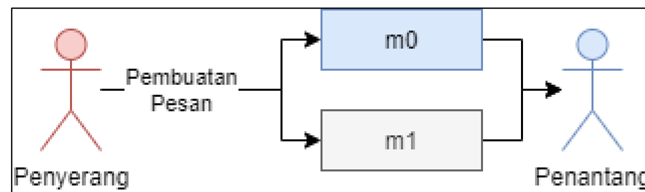
Gambar 2.1 Entitas penantang dan entitas penyerang

2. Penantang membuat *public key* dan *private key* menggunakan fungsi *key generation*.



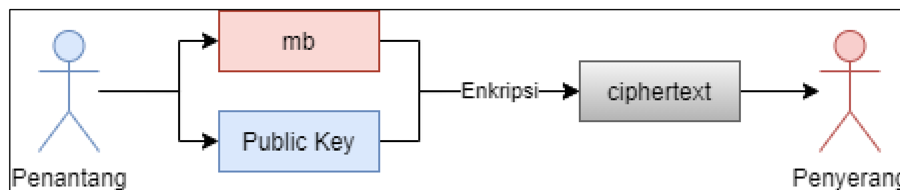
Gambar 2.2 Proses pembuatan *key*

3. Penyerang memilih dua pesan m_0 dan m_1. Di mana m_0 dan m_1 memiliki panjang yang sama dan memberikannya ke penantang.



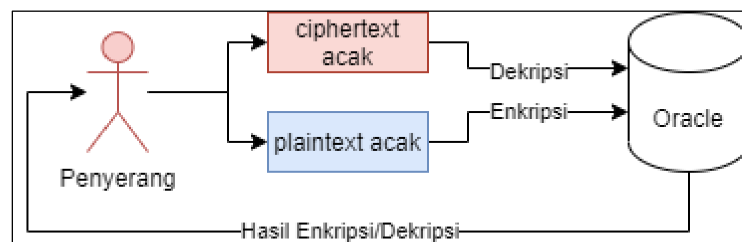
Gambar 2.3 Penyerang memberikan dua pesan *random* kepada penantang

4. Penantang memilih salah satu pesan secara *random* m_b lalu melakukan enkripsi menggunakan *public key* miliknya. Hasil enkripsi berupa *ciphertext* *C* akan diberikan kepada penyerang.



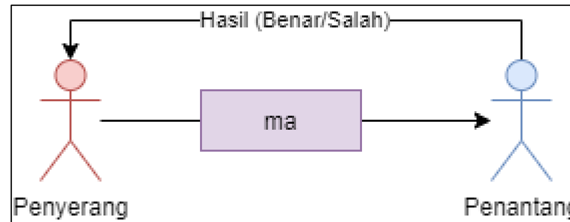
Gambar 2.4 Enkripsi pesan yang dipilih penantang

5. Penyerang dapat melakukan proses enkripsi dan dekripsi dengan memanggil *random oracle model*. Saat pemanggilan fungsi dekripsi pada *random oracle model*, penyerang tidak dapat meminta fungsi dekripsi pada *random oracle model* untuk mendekripsi *ciphertext* *C* yang diberikan oleh penantang.



Gambar 2.5 Pemanggilan fungsi enkripsi dan dekripsi pada Oracle

6. Penyerang menebak pesan mana yang dienkripsi oleh penantang dengan cara memberikan pesan yang dipilihnya m_a. Apabila hasil tebakan sesuai ($m_a = m_b$), maka penyerang memenangkan permainan.



Gambar 2.6 Penyerang menebak pesan yang dienkripsi penantang

2.1.3 KYBER.CPAPKE

Parameter yang digunakan pada skema KYBER.CPAPKE terdiri dari bilangan *integer* $\eta_1, \eta_2, d_u, d_v, k, q, n$ di mana nilai n dan q secara berturut selalu bernilai 256 dan 3329 (Avanzi *et al.*, 2020). Terdapat tiga fungsi yang digunakan pada KYBER.CPAPKE yaitu fungsi *key generation* (KYBER.CPAPKE.KeyGen()), fungsi *encryption* (KYBER.CPAPKE.Enc()), dan fungsi *decryption* (KYBER.CPAPKE.Dec()) (Avanzi *et al.*, 2020). Penjelasan lebih lengkap terkait skema KYBER.CPAPKE dapat dibaca pada penelitian Avanzi (Avanzi *et al.*, 2020).

A. KYBER.CPAPKE.KeyGen()

Fungsi KYBER.CPAPKE.KeyGen() digunakan untuk menghasilkan *public* dan *private key*. Fungsi KYBER.CPAPKE.KeyGen() dapat dilihat pada Gambar 2.7. *Output* dari fungsi ini adalah *secret key* (*sk*) yang merupakan elemen dari himpunan *bytes array* berukuran 12 * *k* * *n* / 8 ($\mathcal{B}^{12*k*n/8}$), dan *public key* (*pk*) yang merupakan elemen dari himpunan *bytes array* berukuran 12 * *k* * *n* / 8 + 32 ($\mathcal{B}^{12*k*n/8+32}$).

Pada langkah 1 sampai 3, dilakukan inisialisasi variabel *d*, ρ, σ, dan N.

Variabel *d* merupakan *random bytes array* dengan panjang 32 ($\mathcal{B}^{32}$). Variabel ρ dan σ merupakan *public* dan *noise seed*, yang didapatkan dari fungsi G(*d*). Fungsi G(*d*) merupakan fungsi *hash* SHA3-512 yang menghasilkan *output* 64 *bytes*. *Bytes* ke 0 sampai 31 diberikan kepada ρ, sedangkan *bytes* ke 32 sampai 64 diberikan kepada σ. Variabel *N* diinisialisasi dengan nilai 0.

Pada langkah 4 sampai 8, dibuat matriks $\hat{A}$ yang berukuran dua dimensi dan merupakan elemen dari himpunan *polynomial rings* R_q^{kxk} pada domain *number theoretic transform* (NTT). NTT digunakan untuk mengefisiensikan operasi perkalian matriks, yang mengurangi kompleksitas waktu operasi dari $O(n^2)$ menjadi $O(n \log n)$ (Sprenkels, 2020). Setiap elemen dari matriks $\hat{A}$ didapatkan dari hasil *parsing* terhadap *output* dari fungsi *extendable output function* (XOF) yang menerima inputan berupa konkatenasi dari *p*, *j*, dan *i* (Parse(XOF(*p,j,i*))). Variabel *j* dan *i* merupakan nilai indeks dari matriks. Fungsi Parse merupakan fungsi *uniform sampling* yang menerima atau menolak nilai yang dihasilkan dari *parsing byte array* yang diberikan. *Uniform sampling* adalah *sampling* yang dilakukan secara acak, di mana setiap elemen yang dipilih memiliki peluang kemunculan yang sama (*Explore With Uniform Sampling*, no date). Fungsi XOF merupakan fungsi *hash* SHAKE-128 yang menghasilkan *output* dengan panjang yang bervariasi. Pada *library* CRYSTALS-Kyber yang digunakan, *output* dari fungsi XOF(*p,j,i*) adalah sebuah 504 *byte array*. Pembuatan matriks $\hat{A}$ bersifat deterministik, di mana *output* akan selalu sama untuk inputan tertentu yang diberikan.

Pada langkah 9 sampai 12, dilakukan *sampling* terhadap *secret vectors s* dari sebuah *centered binomial distribution* (B_η), yang merupakan elemen dari

himpunan *polynomial rings* (R_q^k). Variabel *s* berbentuk matriks satu dimensi, di mana setiap elemennya didapatkan dari hasil fungsi CBD_η(PRF(σ, *N*)). Fungsi *pseudorandom function* (PRF) menerima inputan berupa σ dan *N*. PRF adalah fungsi yang menghasilkan *output* secara acak, yang tidak dapat dibedakan secara komputasi bahwa *output* tersebut benar-benar acak (Kelsey, Change and Perlner, 2016). Fungsi PRF merupakan fungsi *hash* SHAKE-256(σ || N) yang menghasilkan *output* berupa sebuah 128 *byte array*. *Byte array* tersebut dikonversi menjadi *array of bits* yang memiliki panjang 1024. Setiap 4 *bits* pada *array* dikalkulasikan menjadi suatu nilai, sehingga panjang *array* berkurang menjadi 256. Pada langkah 13 sampai 16, dilakukan *sampling* terhadap *noise vectors e*, seperti pada *secret vectors s*.

Pada langkah 17 sampai 18, dilakukan transformasi NTT pada *s* dan *e* yang disimpan pada variabel $\hat{s}$ dan $\hat{e}$. Pada langkah 19, varibel $\hat{t}$ dihitung berdasarkan rumus $\hat{t} = \hat{A} * \hat{s} + \hat{e}$. Pada langkah 20 sampai 21, dilakukan *encoding* untuk melakukan serialisasi *polynomials* menjadi *byte arrays* pada *pk* dan *sk*. Variabel *pk* merupakan hasil *encoding* dari $\hat{t}\ mod\ q$ dan dikonkatenasi dengan ρ. Variabel *sk* merupakan hasil *encoding* dari $\hat{s}\ mod\ q$. Pada langkah 22, *pk* dan *sk* dikembalikan.

```
Algorithm 4 KYBER.CPAPKE.KeyGen(): key generation
Output: Secret key sk ∈ B^{12·k·n/8}
Output: Public key pk ∈ B^{12·k·n/8+32}
 1: d ← B^32
 2: (ρ, σ) := G(d)
 3: N := 0
 4: for i from 0 to k − 1 do                    ▷ Generate matrix Â ∈ R_q^{k×k} in NTT domain
 5:     for j from 0 to k − 1 do
 6:         Â[i][j] := Parse(XOF(ρ, j, i))
 7:     end for
 8: end for
 9: for i from 0 to k − 1 do                                    ▷ Sample s ∈ R_q^k from B_η1
10:     s[i] := CBD_η1(PRF(σ, N))
11:     N := N + 1
12: end for
13: for i from 0 to k − 1 do                                    ▷ Sample e ∈ R_q^k from B_η1
14:     e[i] := CBD_η1(PRF(σ, N))
15:     N := N + 1
16: end for
17: ŝ := NTT(s)
18: ê := NTT(e)
19: t̂ := Â ∘ ŝ + ê
20: pk := (Encode_12(t̂ mod⁺ q)||ρ)                                          ▷ pk := As + e
21: sk := Encode_12(ŝ mod⁺ q)                                                   ▷ sk := s
22: return (pk, sk)
```

Gambar 2.7 Potongan kode fungsi KYBER.CPAPKE.KeyGen()
(Avanzi *et al.*, 2020)

B. KYBER.CPAPKE.Enc()

Fungsi KYBER.CPAPKE.Enc() digunakan untuk melakukan enkripsi. Fungsi KYBER.CPAPKE.Enc() dapat dilihat pada Gambar 2.8. Fungsi ini menerima inputan berupa *public key* (*pk*), *message* (*m*), dan *random coins* (*r*). Variabel *pk* merupakan elemen dari himpunan *bytes array* berukuran 12 * *k* * *n* / 8 + 32 ($\mathcal{B}^{12*k*n/8+32}$). Variabel *m* dan *r* merupakan elemen dari himpunan *bytes array* berukuran 32 ($\mathcal{B}^{32}$). *Output* dari fungsi ini adalah *ciphertext* (*c*) yang merupakan elemen dari himpunan *bytes array* berukuran $d_u * k * n\ /\ 8\ + d_v * n\ /\ 8$ ($\mathcal{B}^{d_u*k*n/8+d_v*n/8}$).

Pada langkah 1 sampai 3, dilakukan inisialisasi variabel *N*, $\hat{t}$, dan ρ. Variabel *N* diinisialisasi dengan nilai 0. Variabel $\hat{t}$ merupakan hasil *decoding* untuk mengubah *byte arrays* menjadi *polynomials*. Variabel ρ merupakan *public*

seed yang dihitung berdasarkan rumus ρ = $pk + 12 * k * n / 8$.

Pada langkah 4 sampai 8, dibuat matriks $\hat{A}^T$ yang merupakan matriks *transpose* dari $\hat{A}$. Setiap elemen dari matriks $\hat{A}^T$ didapatkan dari hasil *parsing* terhadap *output* dari fungsi *extendable output function* (XOF) yang menerima inputan berupa konkatenasi dari p, i, dan j (Parse(XOF(p,i,j))). Urutan konkatenasi dengan variabel j dan i ditukar menjadi i dan j agar dapat menghasilkan matriks *transpose* yang dibutuhkan untuk mengkalkulasi *inverse*.

Pada langkah 9 sampai 12, dilakukan *sampling* terhadap *random coins r* dari sebuah *centered binomial distribution* ($B_{\eta 1}$), yang merupakan elemen dari himpunan *polynomial rings* (R_q^k). Pembuatan *r* didapatkan secara deterministik, dari *message* m berdasarkan rumus $r = G(m)$. Setiap elemen *r* didapatkan dari hasil fungsi $CBD_{\eta 1}$(PRF(r, N)). Pada langkah 13 sampai 16, dilakukan *sampling* terhadap vektor *polynomials* e_1 dari sebuah *centered binomial distribution* ($B_{\eta 2}$), yang merupakan elemen dari himpunan *polynomial rings* (R_q^k). Setiap elemen e_1 didapatkan dari hasil fungsi $CBD_{\eta 2}$(PRF(r, N)). *Sampling* terhadap *r* dan e_1 yang dilakukan sama seperti *sampling* terhadap *s* dan *e* pada fungsi KYBER.CPAPKE.KeyGen().

Pada langkah 17, dilakukan *sampling* terhadap sebuah vektor *polynomial* e_2, yang didapatkan dari hasil fungsi $CBD_{\eta 2}$(PRF(r, N)). Pada langkah 18, dilakukan transformasi NTT pada *r* yang disimpan pada variabel $\hat{r}$. Pada langkah 19, dilakukan perhitungan pada vektor *polynomial u* berdasarkan rumus $u = A^T r + e_1$. Pada langkah 20, dilakukan perhitungan pada vektor *polynomial v* = $t^T r + e_2 + Decompress_q(m, 1)$. Fungsi $Decompress_q$ dilakukan untuk membuat toleransi *error*, di mana *bit* 0 menjadi 0 dan *bit* 1 menjadi *q/2*. Pada

langkah 21 sampai 22, dilakukan *encoding* terhadap hasil *compress* vektor *polynomial u* dan *v,* yang disimpan pada variabel c_1 dan c_2. Fungsi $Compress_q$ berfungsi untuk membuang beberapa *low-order bits* agar ukuran *ciphertext* lebih kecil. *Low-order bits* atau *least significant bit* (LSB) merupakan *bit* paling kanan yang menentukan sebuah bilangan bernilai ganjil atau genap. *Low-order bits* dapat dibuang karena tidak memiliki pengaruh yang besar pada peluang dekripsi bernilai benar. Pada langkah 23, *ciphertext c* didapatkan dari konkatenasi hasil *encoding* c_1 dan c_2, dan dikembalikan.

Algorithm 5 KYBER.CPAPKE.Enc(pk, m, r): encryption

Input: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
Input: Message $m \in \mathcal{B}^{32}$
Input: Random coins $r \in \mathcal{B}^{32}$
Output: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$

```
 1: N := 0
 2: t̂ := Decode_12(pk)
 3: ρ := pk + 12 · k · n/8
 4: for i from 0 to k − 1 do                      ▷ Generate matrix Â ∈ R_q^{k×k} in NTT domain
 5:     for j from 0 to k − 1 do
 6:         Â^T[i][j] := Parse(XOF(ρ, i, j))
 7:     end for
 8: end for
 9: for i from 0 to k − 1 do                      ▷ Sample r ∈ R_q^k from B_η1
10:     r[i] := CBD_η1(PRF(r, N))
11:     N := N + 1
12: end for
13: for i from 0 to k − 1 do                      ▷ Sample e1 ∈ R_q^k from B_η2
14:     e1[i] := CBD_η2(PRF(r, N))
15:     N := N + 1
16: end for
17: e2 := CBD_η2(PRF(r, N))                       ▷ Sample e2 ∈ R_q from B_η2
18: r̂ := NTT(r)
19: u := NTT^{-1}(Â^T ∘ r̂) + e1                   ▷ u := A^T r + e1
20: v := NTT^{-1}(t̂^T ∘ r̂) + e2 + Decompress_q(Decode_1(m), 1)   ▷ v := t^T r + e2 + Decompress_q(m, 1)
21: c1 := Encode_du(Compress_q(u, d_u))
22: c2 := Encode_dv(Compress_q(v, d_v))
23: return c = (c1||c2)                           ▷ c := (Compress_q(u, d_u), Compress_q(v, d_v))
```

Gambar 2.8 Potongan kode fungsi KYBER.CPAPKE.Enc()
(Avanzi *et al.*, 2020)

C. KYBER.CPAPKE.Dec()

Fungsi KYBER.CPAPKE.Dec() digunakan untuk melakukan dekripsi. Fungsi KYBER.CPAPKE.Dec() dapat dilihat pada Gambar 2.9. Fungsi ini menerima inputan berupa *secret key* (*sk*) dan *ciphertext* (*c*). Variabel *sk* merupakan elemen dari himpunan *bytes array* berukuran 12 * *k* * *n* / 8 ($\mathcal{B}^{12*k*n/8}$). Variabel *c* merupakan elemen dari himpunan *bytes array* berukuran $d_u * k * n / 8 + d_v * n / 8$ ($\mathcal{B}^{d_u*k*n/8+d_v*n/8}$). *Output* dari fungsi ini adalah *message* (*m*) yang merupakan elemen dari himpunan *bytes array* berukuran 32 ($\mathcal{B}^{32}$).

Pada langkah 1 sampai 2, dilakukan *decompress* pada *output decoding ciphertext c* untuk mendapatkan vektor *polynomial u* dan *v*. Pada langkah 3, dilakukan *decoding* terhadap *sk* untuk mendapatkan $\hat{s}$. Pada langkah 4, message *m* dikalkulasikan dengan rumus $m = Compress_q(v - s^T u, 1)$. Fungsi $Compress_q$ dilakukan untuk mendekripsi *m* menjadi 1 jika hasil $v - s^T u$ mendekati *q/2* daripada 0, dan mendekripsi *m* menjadi 0 jika sebaliknya. *Message m* didapatkan dari hasil *encoding* pada *output* fungsi $Compress_q$, dengan ukuran 32 *bytes array*. Pada langkah 5, *m* dikembalikan.

Algorithm 6 KYBER.CPAPKE.Dec(sk, c): decryption

Input: Secret key $sk \in \mathcal{B}^{12 \cdot k \cdot n/8}$
Input: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
Output: Message $m \in \mathcal{B}^{32}$

1: $\mathbf{u} := \mathsf{Decompress}_q(\mathsf{Decode}_{d_u}(c), d_u)$
2: $v := \mathsf{Decompress}_q(\mathsf{Decode}_{d_v}(c + d_u \cdot k \cdot n/8), d_v)$
3: $\hat{\mathbf{s}} := \mathsf{Decode}_{12}(sk)$
4: $m := \mathsf{Encode}_1(\mathsf{Compress}_q(v - \mathsf{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \mathsf{NTT}(\mathbf{u})), 1))$ $\triangleright\ m := \mathsf{Compress}_q(v - \mathbf{s}^T\mathbf{u}, 1))$
5: **return** m

Gambar 2.9 Potongan kode fungsi KYBER.CPAPKE.Dec()
(Avanzi *et al.*, 2020)

2.1.4 KYBER.CCAKEM

Potongan kode pada fungsi KYBER.CCAKEM menggunakan fungsi yang terdapat di KYBER.CPAPKE untuk memperoleh ketahanan dari IND-CCA2 (Avanzi *et al.*, 2020). Ketahanan dari IND-CCA2 didapatkan melalui Fujisaki-Okamoto (FO) Transform, yang membutuhkan penambahan beberapa fungsi *hash* dan terbukti aman ketika diuji pada *random oracle model* (Tutoveanu, 2021). Terdapat tiga fungsi pada KYBER.CCAKEM yaitu fungsi *key generation* (KYBER.CCAKEM-.KeyGen()), fungsi *encapsulation* (KYBER.CCAKEM.Enc()), dan fungsi *decapsulation* (KYBER.CCAKEM.Dec()) (Avanzi *et al.*, 2020). Penjelasan lebih lengkap terkait skema KYBER.CCAKEM dapat dibaca pada penelitian Avanzi (Avanzi *et al.*, 2020).

A. KYBER.CCAKEM.KeyGen()

Fungsi KYBER.CCAKEM.KeyGen() digunakan untuk menghasilkan *public (pk)* dan *private key (sk)* dari fungsi KYBER.CPAPKE.KeyGen(). Fungsi KYBER.CCAKEM.KeyGen() dapat dilihat pada Gambar 2.10. Variabel *pk* merupakan elemen dari himpunan *bytes array* berukuran 12 * k * n / 8 + 32 ($\mathcal{B}^{12*k*n/8+32}$). Variabel *sk* merupakan elemen dari himpunan *bytes array* berukuran 24 * k * n / 8 + 96 ($\mathcal{B}^{24*k*n/8+96}$). Pada langkah 1, dibuat *random bytes array* dengan panjang 32 ($\mathcal{B}^{32}$) yang disimpan pada variabel *z*. Pada langkah 2, didapatkan *pk* dan sk' yang merupakan *output* dari fungsi KYBER.-CPAPKE.KeyGen(). Pada langkah 3, *sk* baru didapatkan dengan konkatenasi *sk* dengan *pk*, H(*pk*), dan *z*. Fungsi H merupakan fungsi *hash* SHA3-256 yang menghasilkan *output* 32 *bytes*. Pada langkah 4, *pk* dan *sk* dikembalikan.

```
Algorithm 7 KYBER.CCAKEM.KeyGen()
Output: Public key pk ∈ B^{12·k·n/8+32}
Output: Secret key sk ∈ B^{24·k·n/8+96}
1: z ← B^32
2: (pk, sk') := KYBER.CPAPKE.KeyGen()
3: sk := (sk'||pk||H(pk)||z)
4: return (pk, sk)
```

Gambar 2.10 Potongan kode fungsi KYBER.CCAKEM.KeyGen()
(Avanzi *et al.*, 2020)

B. KYBER.CCAKEM.Enc()

Fungsi KYBER.CCAKEM.Enc() digunakan untuk menghasilkan *shared key* (*K*) beserta *ciphertext* enkapsulasinya (*c*), yang menggunakan fungsi KYBER.CPAPKE.Enc(). Fungsi KYBER.CCAKEM.Enc() dapat dilihat pada Gambar 2.11. Fungsi ini menerima inputan berupa *public key* (*pk*), yang merupakan elemen dari himpunan *bytes array* berukuran 12 * *k* * *n* / 8 + 32 ($\mathcal{B}^{12*k*n/8+32}$). *Output* dari fungsi ini adalah *c* dan *K*, di mana *c* merupakan elemen dari himpunan *bytes array* berukuran $d_u * k * n / 8 + d_v * n / 8$ ($\mathcal{B}^{d_u*k*n/8+d_v*n/8}$), dan *K* merupakan elemen dari himpunan *bytes array* ($\mathcal{B}^*$).

Pada langkah 1 sampai 2, dibuat sebuah *random byte array* berukuran 32 (*m*) yang di-*hash* pada fungsi H(*m*). Pada langkah 3, didapatkan $\bar{K}$ dan *r* yang merupakan *output* dari fungsi G(*m* || H(*pk*)). Pada langkah 4, didapatkan *c* yang merupakan *output* dari fungsi KYBER.CPAPKE.Enc(*pk, m, r*). Pada langkah 5, didapatkan *K* yang merupakan *output* dari fungsi KDF($\bar{K}$ || H(c)). KDF merupakan fungsi *key derivation function* yang menggunakan SHAKE-256 dengan *output* 32 *bytes*. Pada langkah 6, *c* dan *K* dikembalikan.

Algorithm 8 KYBER.CCAKEM.Enc(pk)

Input: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
Output: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
Output: Shared key $K \in \mathcal{B}^*$
1: $m \leftarrow \mathcal{B}^{32}$
2: $m \leftarrow \mathrm{H}(m)$ ▷ Do not send output of system RNG
3: $(\bar{K}, r) := \mathrm{G}(m \| \mathrm{H}(pk))$
4: $c :=$ KYBER.CPAPKE.Enc(pk, m, r)
5: $K :=$ KDF$(\bar{K} \| \mathrm{H}(c))$
6: **return** (c, K)

Gambar 2.11 Potongan kode fungsi KYBER.CCAKEM.Enc()
(Avanzi *et al.*, 2020)

C. KYBER.CCAKEM.Dec()

Fungsi KYBER.CCAKEM.Dec() digunakan untuk menghasilkan *shared key* (*K*), yang menggunakan fungsi KYBER.CPAPKE.Dec(). Fungsi KYBER.CCAKEM.Dec() dapat dilihat pada Gambar 2.12. Fungsi ini menerima inputan berupa *ciphertext* (*c*) dan *secret key* (*sk*). Variabel *c* merupakan elemen dari himpunan *bytes array* berukuran $d_u * k * n / 8 + d_v * n / 8$ ($\mathcal{B}^{d_u * k * n/8 + d_v * n/8}$). Variabel *sk* merupakan elemen dari himpunan *bytes array* berukuran 24 * *k* * *n* / 8 + 96 ($\mathcal{B}^{24 * k * n/8+96}$). *Output* dari fungsi ini adalah *K* yang merupakan elemen dari himpunan *bytes array* ($\mathcal{B}^*$).

Pada langkah 1 sampai 3, dilakukan inisialisasi variabel *pk*, h, dan z. Variabel *pk* dihitung berdasarkan rumus $pk = sk + 12 * k * n / 8$. Variabel *h* dihitung berdasarkan rumus $h = sk + 24 * k * n / 8 + 32$, di mana *h* merupakan elemen dari himpunan *bytes array* berukuran 32 ($\mathcal{B}^{32}$). Varibel *z* dihitung berdasarkan rumus $z = sk + 24 * k * n / 8 + 64$,

Pada langkah 4, didapatkan m' yang merupakan *output* dari fungsi KYBER.CPAPKE.Dec(*s*, (*u*, *v*)). Pada langkah 5, didapatkan $\bar{K}'$ dan r' yang merupakan *output* dari fungsi G(m' || *h*). Pada langkah 6, didapatkan c' yang merupakan *output* dari fungsi KYBER.CPAPKE.Enc(*pk*, m', r'). Pada langkah 7

sampai 11, dilakukan perbandingan nilai c dengan c'. Jika bernilai sama, maka K didapatkan dari *output* fungsi KDF($\bar{K}'$ || H(c)). Jika tidak bernilai sama, maka K didapatkan dari *output* fungsi KDF(z || H(c)). Pada langkah 12, K dikembalikan.

Algorithm 9 KYBER.CCAKEM.Dec(c, sk)

Input: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
Input: Secret key $sk \in \mathcal{B}^{24 \cdot k \cdot n/8 + 96}$
Output: Shared key $K \in \mathcal{B}^*$

```
1: pk := sk + 12 · k · n/8
2: h := sk + 24 · k · n/8 + 32 ∈ B^32
3: z := sk + 24 · k · n/8 + 64
4: m' := KYBER.CPAPKE.Dec(s, (u, v))
5: (K̄', r') := G(m'||h)
6: c' := KYBER.CPAPKE.Enc(pk, m', r')
7: if c = c' then
8:     return K := KDF(K̄'||H(c))
9: else
10:    return K := KDF(z||H(c))
11: end if
12: return K
```

Gambar 2.12 Potongan kode fungsi KYBER.CCAKEM.Dec()
(Avanzi *et al.*, 2020)

2.2 Aplikasi Autentikasi Produk Oricon

Oricon merupakan sebuah aplikasi yang menyimpan data produk suatu perusahaan (Philips, 2021). Aplikasi Oricon ini bertujuan untuk melakukan autentikasi pada produk-produk yang terdaftar di dalamnya (Philips, 2021). Autentikasi merupakan sebuah proses yang dilakukan untuk membuktikan kebenaran, orisinalitas, atau keaslian suatu produk (Rosencrance, 2018). Oricon dibangun menggunakan sebuah *framework blockchain* yang dikenal dengan sebutan Hyperledger Fabric (Philips, 2021). Terdapat beberapa fungsi yang dapat dijalankan pada aplikasi Oricon, yaitu (Philips, 2021).

1. Read Product, merupakan fungsi yang digunakan untuk melihat *detail* produk yang diinginkan oleh *user*.
2. Update Product, merupakan fungsi yang digunakan untuk melakukan perubahan data produk yang telah ada sebelumnya di *database*.

Pengecekan produk dilakukan terlebih dahulu sebelum mengubah data produk tersebut.

3. Delete Product, merupakan fungsi yang digunakan untuk menghapus data produk yang telah ada sebelumnya di *database*.
4. Product History, merupakan fungsi yang digunakan untuk melakukan pengecekan alur perubahan data produk yang pernah terjadi di dalam *blockchain*.
5. ERC20 token, merupakan fungsi yang diimplementasikan menggunakan standar *fungible token* ERC20. ERC20 ini melingkupi fungsionalitas seperti melihat nama token, simbol token, decimal yang digunakan token, jumlah persediaan token, saldo *user*. Fungsi ini juga dapat melakukan *transfer* token dari satu *user* ke *user* lain. Proses *transfer* token memerlukan persetujuan dari *user* yang menerima token. *User* yang menerima token dapat menentukan jumlah token yang hendak diterima dari *user* lain. Apabila persediaan token hampir habis, fungsi ERC20 ini dapat menambahkan jumlah persediaan seluruh token. Penambahan jumlah token ini hanya dapat dilakukan oleh akun yang diizinkan saja.

2.3 Hyperledger Caliper

Hyperledger Caliper merupakan sebuah *tools* yang didukung oleh Hyperledger Fabric untuk melakukan proses *benchmarking* sehingga performa dari *blockchain* yang telah diimplementasikan dapat diketahui (*Hyperledger Caliper*, 2020). Pengukuran performa pada Hyperledger Caliper memerlukan beberapa *use case* (*Hyperledger Caliper*, 2020). Pada penelitian ini Hyperledger Caliper akan digunakan untuk pengukuran performa dari empat segi, yaitu (Philips, 2021).

1. Transaksi sukses merupakan jumlah transaksi valid yang berhasil dimasukkan pada *ledger*. Sebuah transaksi dapat dikatakan valid apabila berhasil melewati tahap validasi dari *blockchain*.
2. Transaksi gagal merupakan jumlah transaksi yang gagal pada tahap *endorsement* maupun tahap *validation* pada *blockchain*. Transaksi yang gagal pada tahap *validation* tetap akan dimasukkan ke *ledger* namun memiliki status transaksi tidak valid sehingga tidak akan diterapkan pada *world state*.
3. *Latency*, merupakan waktu yang dibutuhkan untuk satu transaksi agar berhasil disimpan pada *ledger*.
4. *Throughput* merupakan jumlah transaksi valid yang berhasil dimasukkan ke *ledger* dalam rentang waktu tertentu.

Pada penelitian Wilson, terdapat 5 fungsi data produk dan 6 fungsi data token pada *smart contract* yang dilakukan pengujian *latency* dan *throughtput* menggunakan Hyperledger Caliper (Philips, 2021). Pengujian yang dilakukan pada penelitian tersebut terdiri dari 12 eksperimen yang dapat dilihat pada Tabel

2.1 di bawah ini (Philips, 2021).

Tabel 2.1 Tabel eksperimen Hyperledger Caliper Wilson

No	Parameter Pengujian	MaxMessageCount	BatchTimeout (s)	Jumlah Worker
1	MaxMessageCount	2	20	2
2		2	20	4
3		4	20	2
4		4	20	4
5		8	20	2
6		8	20	4
7	BatchTimeout	20	2	2
8		20	2	4
9		20	4	2
10		20	4	4
11		20	8	2
12		20	8	4

Hasil yang didapatkan dari pengukuran performa dari segi transaksi sukses, transaksi gagal, *latency* dan *throughput* dari penelitian tersebut selanjutnya akan menjadi acuan perbandingan performa sebelum dan sesudah diimplementasikannya CRYSTALS-Kyber pada *smart contract*. Terdapat dua hasil eksperimen yang diambil, yaitu eksperimen 7 dan 8 yang memiliki kriteria BatchTimeout sebesar 2 detik, MaxMessageCount sebesar 20, dan jumlah *workers* sebanyak 2 *workers* dan 4 *workers*. Sehingga jumlah iterasi yang dilakukan akan bersifat *random* karena iterasi akan berlangsung sampai salah satu dari kriteria BatchTimeout atau kriteria MaxMessageCount terpenuhi (Philips, 2021). Hasil kedua eksperimen tersebut dapat terlihat pada Tabel 2.2 dan Tabel 2.3 di bawah ini (Philips, 2021).

Tabel 2.2 Tabel hasil eksperimen Hyperledger Caliper 2 *workers*

Nama	Transaksi Sukses	Transaksi Gagal	Latency (s)	Throughput (TPS)
createProduct	21	86	1,4300	0,6000
readProduct	1186	1451	0,0400	39,5000
readProductHistory	3021	0	0,0200	100,7000
updateProduct	14	133	1,9900	0,4000
deleteProduct	11	89	2,2700	0,3000
Rata-rata	850,6000	351,8000	1,1500	28,3000

Tabel 2.3 Tabel hasil eksperimen Hyperledger Caliper 4 *workers*

Nama	Transaksi Sukses	Transaksi Gagal	Latency (s)	Throughput (TPS)
createProduct	3	33	11,6700	0,1000
readProduct	1628	24	0,0400	54,3000
readProductHistory	4780	0	0,0400	159,2000
updateProduct	7	60	3,7400	0,2000
deleteProduct	10	90	4,6800	0,2000
Rata-rata	1285,6000	41,4000	4,0340	42,8000