

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ  
імені Ігоря СІКОРСЬКОГО»**

**Навчально-науковий фізико-технічний інститут  
Кафедра математичних методів захисту інформації**

**Звіт до  
лабораторної роботи за темою:  
Дослідження сучасних алгебраїчних криптосистем  
на прикладі постквантових  
криптографічних алгоритмів.  
Алгоритм TiGER**

**Оформлення звіту:**  
Юрчук Олексій, ФІ-52мн  
Дигас Богдан, ФІ-52мн

16 листопада 2025 р.  
м. Київ

# ЗМІСТ

<b>1 Вступне слово</b>	<b>1</b>
<b>2 Загальне теоретичне дослідження</b>	<b>2</b>
2.1 Постквантова криптографія	2
2.2 Передумови створення TiGER	2
2.3 Участь у KpqC та злиття з SMAUG	3
2.3.1 Злиття TiGER та SMAUG	3
2.3.2 Результати KpqC 2023	4
<b>3 Теоретична база алгоритму TiGER</b>	<b>5</b>
3.1 Алгебраїчні структури	5
3.1.1 Кільця та многочлени	5
3.1.2 Факторкільця многочленів	6
3.1.3 Кільце многочленів у TiGER	6
3.1.4 Операції в кільці $R_q$	6
3.2 Задачі на решітках	7
3.2.1 Решітки та базові задачі	7
3.2.2 Задача Learning With Errors (LWE)	7
3.2.3 Задача Ring Learning With Errors (RLWE)	8
3.2.4 Задача Learning With Rounding (LWR)	8
3.2.5 Задача Ring Learning With Rounding (RLWR)	9
3.3 "Сімейство" алгоритмів на базі RLWE/RLWR	9
3.3.1 Lizard	9
3.3.2 RLizard	10
3.3.3 CRYSTALS-Kyber	10
3.3.4 Saber	11
3.3.5 Порівняння алгоритмів, і що з них взяв TiGER	12
3.4 Криптографічні примітиви	12
3.4.1 Схема шифрування з відкритим ключем (PKE)	12
3.4.2 Механізм інкапсуляції ключа (KEM)	13
3.4.3 Побудова KEM з PKE	13
3.4.4 Криптографічні геш-функції	14
3.4.5 Зв'язок PKE та KEM у TiGER	14
3.5 Основні поняття безпеки, що стосуються TiGER	14
3.5.1 IND-CPA безпека	14
3.5.2 IND-CCA безпека	15
3.5.3 Ймовірність помилки розшифрування (DFP/R)	16
3.5.4 Квантова безпека а.к.а. QROM	17
3.6 Перетворення Fujisaki-Okamoto	17
3.6.1 Класичне перетворення FO	17
3.6.2 Перетворення $FO_m^f$ з неявним відхиленням	18
3.6.3 Застосування у алгоритмі TiGER	19

<b>4</b>	<b>Повний опис алгоритму TiGER</b>	<b>20</b>
4.1	Загальна структура алгоритму	20
4.1.1	Модульна побудова алгоритму	20
4.1.2	Взаємодія між компонентами TiGER	21
4.2	Публічні параметри	21
4.3	Допоміжні алгоритми	23
4.4	TiGER.PKE	25
4.4.1	KeyGen – генерація ключів	25
4.4.2	Encryption – шифрування	26
4.4.3	Decryption – розшифрування	27
4.5	TiGER.KEM	29
4.5.1	KeyGen – генерація ключів	29
4.5.2	Encapsulation – інкапсуляція	29
4.5.3	Decapsulation – декапсуляція	30
4.6	Аналіз коректності	32
4.7	Особливості реалізації	34
<b>5</b>	<b>Аналіз безпеки</b>	<b>37</b>
5.1	Теоретичні основи безпеки	37
5.1.1	IND-CPA безпека TiGER.PKE	37
5.1.2	IND-CCA безпека TiGER.KEM	37
5.1.3	Важливість моделі QROM	38
5.2	Квантова стійкість	38
5.3	Рівні безпеки NIST	40
5.3.1	Категорії безпеки згідно NIST	40
5.3.2	Відповідність TiGER стандартам NIST	41
<b>6</b>	<b>Порівняння з іншими алгоритмами</b>	<b>42</b>
6.1	TiGER vs RLizard	42
6.2	TiGER vs Kyber	44
6.3	TiGER vs Saber	47
6.4	TiGER vs SMAUG	49
6.5	Порівняння усіх перелічених алгоритмів	52
<b>7</b>	<b>Аналіз атак на TiGER</b>	<b>54</b>
7.1	Аналіз слабких місць алгоритму	54
7.2	Можливі вразливості через DFP/R	57
7.3	Meet-LWE атака	59
7.4	ССА атаки на PKE	61
7.5	Атака з доступом до проміжного виводу декодування	64
7.6	Атака з урахуванням XEf корекції помилок	67
7.7	Side-channel attacks	68
7.8	Перенесення атак з RLWE/RLWR схем на TiGER	73
<b>8</b>	<b>Практичне застосування та висновки</b>	<b>75</b>
8.1	Можливі сценарії використання TiGER	75
8.2	Підсумки дослідження	75



# Розділ 1

## Вступне слово

**Мета роботи** (власне, для чого ми тут зібралися):

Дослідити особливостей реалізації сучасних алгебраїчних криптосистем на прикладі учасників першого раунду національного конкурсу з постквантової криптографії в Кореї (KpqC).

**Наші задачі на комп'ютерний практикум та порядок їх виконання:**

- 1) Роздітися на бригади. Визначили хто за що відповідатиме. Богдан – займається реалізацією алгоритму TiGER, Олексій – теоретичною частиною і звітом загалом.
- 2) Провести теоретичне дослідження теми, надавши вичерпний та повний опис теоретичної сторони алгоритму з усіма деталями та відомими результатами досліджень; провести аналіз вже існуючих атак на алгоритм TiGER, а також загалом можливих атак; виконати порівняльний аналіз нашого алгоритму зі схожими та дослідити можливість перенесення та застосування відомих атак на нього.
- 3) Реалізувати алгоритм програмно та всі(не, ну ми постараємося) можливі варіанти цього алгоритму;
- 4) Перевірити коректність – підтвердити правильність реалізації за допомогою тестів, використавши тестові дані з офіційної реалізації;
- 5) Зробити аналіз продуктивності алгоритму та, знову ж таки, провести порівняння та аналіз швидкодії за різних умов, дослідити вплив модифікацій окремих його складових частин на ефективність.

# Розділ 2

## Загальне теоретичне дослідження

### 2.1 Постквантова криптографія

Сучасна криптографія з відкритим ключем, зокрема RSA та криптографія на еліптичних кривих – Elliptic Curve Cryptography (ECC), базується на обчислювальній складності задач факторизації великих чисел та дискретного логарифмування. Однак у 1994 році Пітер Шор [1] продемонстрував квантові алгоритми, здатні розв'язувати ці задачі за поліноміальний час на достатньо потужному квантовому комп'ютері. Це створює критичну загрозу для існуючої криптографічної інфраструктури.

Постквантова криптографія (Post-Quantum Cryptography, PQC) – це галузь криптографії, що розробляє алгоритми, стійкі як до класичних, так і до квантових атак. Серед основних напрямків PQC виділяють криптографію на решітках, криптографію на кодах виправлення помилок, багатовимірну поліноміальну(квадратичну) криптографію та криптографію на основі геш-функцій [2].

### 2.2 Передумови створення TiGER

Механізм інкапсуляції ключа (Key Encapsulation Mechanism, KEM) є одним з найважливіших криптографічних примітивів для захищеного обміну ключами. У контексті заміни класичних протоколів, таких як Diffie-Hellman (DH) або Elliptic Curve Diffie-Hellman ECDH, постквантові KEM повинні забезпечувати не лише високий рівень безпеки, але й бути ефективними за розміром даних та залишатися обчислювано складними для зламу злоюмисником.

Криптографія на решітках, зокрема алгоритми на основі задач Learning With Errors (LWE) [3] та Ring Learning With Errors (RLWE) [4], продемонструвала перспективність у створенні ефективних постквантових схем. Розвиток цього напрямку призвів до появи сімейства алгоритмів, що використовують детермінований варіант – Learning With Rounding (LWR) [5], який замінює випадкову помилку округленням, що покращує як продуктивність, так і довжину шифротексту.

Серед попередніх розробок слід відзначити алгоритми Lizard [6] та RLizard [7], які комбінували RLWE для генерації ключів з RLWR для шифрування, досягаючи балансу між безпекою та ефективністю. Однак ці схеми мали певні обмеження щодо розміру відкритого ключа та шифротексту, що ускладнювало їх інтеграцію в існуючі протоколи.

TiGER (Tiny bandwidth key encapsulation mechanism for easy miGration based on RLWE(R)) [8] був розроблений командою дослідників з метою створення компактного та ефективного KEM, придатного для легкої інтеграції в існуючі системи безпеки. Основні задачі, які ставили перед собою науковці це:

- **Мінімізація розміру шифротексту та відкритого ключа**
- **Висока обчислювальна ефективність** — використання в якості модуля число, яке є степенем двійки ( $q = 2^k$ ) (для оптимізації операцій округлення через побітові зсуви);
- **Відмова від NTT** — алгоритм не використовує Number Theoretic Transform, що спрощує реалізацію;
- **Використання розріджених секретів (з малою вагою Геммінга)** — зменшення розміру секретного ключа та прискорення множення многочленів;
- **Корекція помилок** — застосування кодів XEf та D2 для зниження ймовірності помилки розшифрування.

Конструкція TiGER базується на комбінації RLWR для генерації відкритого ключа та RLWE для шифрування, з подальшим застосуванням перетворення Fujisaki-Okamoto [9, 10] для досягнення IND-CCA безпеки.

## 2.3 Участь у KpqC та злиття з SMAUG

У 2022 році Національна служба розвідки Республіки Корея ініціювала Korean Post-Quantum Cryptography Competition скорочено – KpqC [11]. Це національний конкурс для стандартизації постквантових криптографічних алгоритмів.

Обраний нами для аналізу алгоритм TiGER був поданий на перший раунд конкурсу KpqC у категорії механізмів інкапсуляції ключа (KEM) і був одним з чотирьох алгоритмів, які пройшли до другого раунду.

### 2.3.1 Злиття TiGER та SMAUG

Команди TiGER та SMAUG об'єдналися для створення спільного алгоритму SMAUG-T [12]. Метою злиття було поєднання переваг обох підходів:

- Від **TiGER**: Компактність шифротексту, використання RLWE/RLWR на кільцевому рівні, корекція помилок через D2 кодування (для параметра TiMER);
- Від **SMAUG**: Модульна структура (MLWE/MLWR), розріджені секрети через використання гаусівського шуму, покращена безпека за рахунок збільшення розмірності.

Результатом злиття став алгоритм SMAUG-T версії 3.0 (лютий 2024), який включає в себе:

- Три основні набори параметрів: **SMAUG-T128**, **SMAUG-T192**, **SMAUG-T256** (відповідають рівням безпеки NIST 1, 3, 5);
- Додатковий набір параметрів **TiMER** (Tiny SMAUG using Error Reconciliation) – оптимізований для IoT(Internet of Thing)-пристроїв з мінімальним шифротекстом завдяки використанню D2 кодування з TiGER.

### 2.3.2 Результати КрґС 2023

У січні 2025 року було оголошено фінальні результати конкурсу КрґС. Переможцями стали:

- У категорії КЕМ: **SMAUG-T** та **NTRU+**;
- У категорії цифрового підпису: **НАЕТАЕ** (до речі, також від команди SMAUG).

Таким чином, ідеї та технології TiGER увійшли до складу національного стандарту постквантової криптографії Кореї через алгоритм SMAUG-T.

# Розділ 3

## Теоретична база алгоритму TiGER

### 3.1 Алгебраїчні структури

Криптографія на решітках (Lattice-based cryptography) використовує алгебраїчні структури для забезпечення ефективності обчислень та компактності представлення даних. У цьому розділі я розпишу основні алгебраїчні об'єкти, що застосовуються в алгоритмі TiGER.

#### 3.1.1 Кільця та многочлени

**Означення 3.1.1** (Кільце).

Кільце  $(R, +, \cdot)$  – це множина з двома операціями: додавання  $(+)$  та множення  $(\cdot)$ , що задовольняє наступні властивості:

1.  $(R, +)$  є абелевою групою за додаванням. Нейтральний елемент  $0$ ;
2. Множення є асоціативним:  $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ ,  $\forall a, b, c \in R$ ;
3. Дистрибутивність:  $a \cdot (b + c) = a \cdot b + a \cdot c$  та  $(b + c) \cdot a = b \cdot a + c \cdot a$ ,  $\forall a, b, c \in R$ ;
4. Існує нейтральний елемент за множенням:  $1 \in R$  такий, що  $1 \cdot a = a \cdot 1 = a$ ,  $\forall a \in R$ .

Якщо ще  $a \cdot b = b \cdot a$ ,  $\forall a, b \in R$ , то таке кільце називається комутативним.

**Означення 3.1.2** (Кільце многочленів).

Нехай  $R$  – комутативне кільце з одиницею. Кільце многочленів  $R[x]$  складається з усіх виразів виду

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0,$$

де  $\forall i : a_i \in R$ ,  $a_n \neq 0$  та  $n \in \mathbb{Z}$ .

Число  $n$  називається степенем многочлена  $f(x)$ , позначається  $\deg(f)$ .

Операції додавання та множення многочленів наступним чином:

- $(f + g)(x) = \sum_{i=0}^{\max(\deg(f), \deg(g))} (a_i + b_i) x^i$ , де  $a_i, b_i$  – коефіцієнти  $f$  та  $g$  відповідно;
- $(f \cdot g)(x) = \sum_{k=0}^{\deg(f) + \deg(g)} c_k x^k$ , де  $c_k = \sum_{i=0}^k a_i b_{k-i}$ .

### 3.1.2 Факторкільця многочленів

**Означення 3.1.3** (Факторкільце).

Нехай  $R$  – кільце та  $I$  – його ідеал. Факторкільце  $R/I$  складається з класів еквівалентності  $a + I = \{a + r : r \in I\}$  для  $a \in R$ , з операціями:

$$(a + I) + (b + I) = (a + b) + I, \quad (a + I) \cdot (b + I) = (a \cdot b) + I.$$

У Lattice-based cryptography найчастіше використовується факторкільце многочленів за ідеалом, що породжений циклотомічним многочленом.

**Означення 3.1.4** (Циклотомічний многочлен).

$n$ -ий циклотомічний многочлен (Cyclotomic polynomial)  $\Phi_n(x)$  визначається як мінімальний многочлен над  $\mathbb{Q}$ , коренями якого є примітивні корені  $n$ -го степеня з одиниці:

$$\Phi_n(x) = \prod_{\substack{1 \leq k \leq n \\ \gcd(k, n) = 1}} (x - e^{2\pi i k/n}).$$

**Лема 3.1.1.** Для  $n = 2^k$ , де  $k \in \mathbb{N}$ , циклотомічний многочлен має вигляд:

$$\Phi_n(x) = x^{n/2} + 1.$$

*Доведення:* При  $n = 2^k$  примітивними коренями  $n$ -го степеня з одиниці є  $e^{2\pi i m/n}$  для непарних  $m$ . З  $(x^{n/2} + 1) = (x^n - 1)/(x^{n/2} - 1)$  випливає твердження леми.  $\square$

### 3.1.3 Кільце многочленів у TiGER

В алгоритмі TiGER використовується кільце многочленів виду:

$$R_q = \frac{\mathbb{Z}_q[x]}{(x^n + 1)},$$

де  $n$  – степінь двійки (зазвичай  $n = 512$  або  $n = 1024$ ), а  $q$  – модуль, що також є степенем двійки ( $q = 256$  у всіх варіаціях алгоритму TiGER).

Елементами  $R_q$  є многочлени степеня не вище  $n - 1$  з коефіцієнтами з  $\mathbb{Z}_q$ :

$$f(x) = \sum_{i=0}^{n-1} a_i x^i, \quad a_i \in \mathbb{Z}_q.$$

Вибір саме такого  $n$  та многочлена  $x^n + 1$  забезпечує:

- Ефективність множення многочленів (без потреби в NTT);
- Редукцію за модулем  $x^n + 1$ , що спрощує обчислення;
- Зв'язок з циклотомічними многочленами та решітковими задачами.

### 3.1.4 Операції в кільці $R_q$

Для  $f(x), g(x) \in R_q$  операції в кільці визначаються наступним чином:

**Додавання:** покоефіцієнтне за модулем  $q$ :

$$(f + g)(x) = \sum_{i=0}^{n-1} ((f_i + g_i) \bmod q) x^i.$$

**Множення:** спочатку виконується звичайне множення многочленів, потім взяття за модулем  $(x^n + 1)$  та  $q$ . Оскільки  $x^n \equiv -1 \pmod{x^n + 1}$ , то маємо:

$$h_i = \left( \sum_{j=0}^i f_j g_{i-j} - \sum_{j=i+1}^{n-1} f_j g_{n+i-j} \right) \bmod q,$$

де  $h(x) = (f \cdot g)(x) = \sum_{i=0}^{n-1} h_i x^i$ .

## 3.2 Задачі на решітках

Безпека TiGER базується на обчислювальній складності певних задач на решітках. У цьому підпункті зазначимо основні решіткові задачі, що лежать в основі постквантової криптографії.

### 3.2.1 Решітки та базові задачі

**Означення 3.2.1** (Решітка).

Нехай  $\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_m \in \mathbb{R}^n$  – лінійно незалежні вектори. Решітка  $\Lambda$ , породжена цими векторами, визначається як:

$$\Lambda = \Lambda(\mathbf{b}_1, \dots, \mathbf{b}_m) = \left\{ \sum_{i=1}^m a_i \mathbf{b}_i : a_i \in \mathbb{Z} \right\}.$$

Вектори  $\mathbf{b}_1, \dots, \mathbf{b}_m$  називаються базисом решітки, а  $m$  – розмірністю решітки.

**Означення 3.2.2** (Мінімальна відстань решітки(shortest vector)).

Мінімальна відстань решітки  $\Lambda$  визначається як:

$$\lambda_1(\Lambda) = \min_{\mathbf{v} \in \Lambda \setminus \{\mathbf{0}\}} \|\mathbf{v}\|,$$

де  $\|\cdot\|$  – евклідова норма.

**Означення 3.2.3** (SVP).

*Shortest Vector Problem (SVP):* Для заданого базису решітки  $\Lambda$  знайти ненульовий вектор  $\mathbf{v} \in \Lambda$  такий, що  $\|\mathbf{v}\| = \lambda_1(\Lambda)$ .

SVP є NP-складною задачею [13]. Для криптографічних цілей часто використовується наступна версія:

**Означення 3.2.4** (Апроксимаційна задача SVP).

$\gamma$ -approximate SVP ( $\gamma$ -SVP): Для заданого базису решітки  $\Lambda$  та параметра наближення  $\gamma \geq 1$ , знайти ненульовий вектор  $\mathbf{v} \in \Lambda$  такий, що  $\|\mathbf{v}\| \leq \gamma \cdot \lambda_1(\Lambda)$ .

### 3.2.2 Задача Learning With Errors (LWE)

Задача Learning With Errors була введена Одедом Регевим у 2005 році [3] і стала основою для багатьох постквантових криптосистем.

**Означення 3.2.5** (LWE задача (search version)).

Нехай  $n, q \geq 1$  – цілі числа,  $\chi$  – розподіл ймовірностей на  $\mathbb{Z}_q$ . Пошукова задача  $LWE_{n,q,\chi}$  визначається наступним чином:

Для невідомого секрету  $\mathbf{s} \in \mathbb{Z}_q^n$  та заданої послідовності пар  $(\mathbf{a}_i, b_i) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ , де

$$b_i = \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i \pmod{q},$$

з випадково обраними  $\mathbf{a}_i \in \mathbb{Z}_q^n$ ,  $\langle \cdot, \cdot \rangle$  – операція векторного добутку та  $e_i \xleftarrow{p} \chi$ , знайти секрет  $\mathbf{s}$ .

**Означення 3.2.6** (LWE задача (detection version)).

Розпізнавальна задача  $Decision-LWE_{n,q,\chi}$  полягає у розрізненні наступних двох розподілів:

- $(\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e \pmod{q})$ , де  $\mathbf{a} \leftarrow \mathbb{Z}_q^n$ ,  $\mathbf{s} \in \mathbb{Z}_q^n$  фіксоване,  $e \xleftarrow{p} \chi$ ;
- $(\mathbf{a}, u)$ , де  $\mathbf{a}, u$  – рівномірно розподілені на  $\mathbb{Z}_q^n$  та  $\mathbb{Z}_q$  відповідно.

**Теорема 3.2.1** (Регєва).

Для певних параметрів  $n, q, \chi$ , розв'язання задачі  $Decision-LWE$  за поліноміальний час у середньому випадку еквівалентно розв'язанню наближеної задачі  $\gamma$ -SVP за квантовий поліноміальний час у найгіршому випадку для деякого  $\gamma = \tilde{O}(n/\sigma)$ .

**3.2.3 Задача Ring Learning With Errors (RLWE)**

Ring-LWE є алгебраїчною версією LWE, що вже використовує кільця многочленів для більшої ефективності [4].

**Означення 3.2.7** (RLWE задача).

Нехай  $R = \mathbb{Z}[x]/(x^n + 1)$ ,  $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ , та  $\chi$  – розподіл ймовірностей на  $R_q$ . Розпізнавальна задача  $Decision-RLWE_{n,q,\chi}$  полягає у розрізненні наступних розподілів:

- $(a, a \cdot s + e)$ , де обрано  $a \in R_q$  (рівномірний розподіл),  $s \in R_q$  фіксоване,  $e \xleftarrow{p} \chi$ ;
- $(a, u)$ , де  $a, u$  обрані рівномірно з  $R_q$ .

Важливим є те, що RLWE дозволяє представляти  $n$  секретів (LWE-зразків) у вигляді одного многочлена в  $R_q$ , що значно зменшує розмір ключів та шифротекстів. Для TiGER використовується  $n \in \{512, 1024\}$  та  $q = 256$ .

**3.2.4 Задача Learning With Rounding (LWR)**

Learning With Rounding є детермінованим варіантом LWE, де замість додавання випадкової помилки використовується округлення [5].

**Означення 3.2.8** (LWR задача).

Нехай  $n, q, p$  – цілі числа,  $p < q$ . Визначимо функцію округлення  $\lfloor \cdot \rfloor_p : \mathbb{Z}_q \rightarrow \mathbb{Z}_p$  як

$$\lfloor x \rfloor_p = \left\lfloor \frac{p}{q} \cdot x \right\rfloor \pmod{p},$$

де  $\lfloor \cdot \rfloor$  позначатиме округлення до найближчого цілого.

Розпізнавальна задача  $Decision-LWR_{n,q,p}$  полягає у розрізненні наступних розподілів:

- $(\mathbf{a}, \lfloor \langle \mathbf{a}, \mathbf{s} \rangle \rfloor_p)$ , де  $\mathbf{a} \leftarrow \mathbb{Z}_q^n$ ,  $\mathbf{s} \in \mathbb{Z}_q^n$  – фіксоване;
- $(\mathbf{a}, u)$ , де  $\mathbf{a} \leftarrow \mathbb{Z}_q^n$ ,  $u \leftarrow \mathbb{Z}_p$ .

**Теорема 3.2.2** (Редукція (взяття за модулем) LWR до LWE [5]). Для відповідних параметрів  $n, q, p$  та достатньо малого розподілу помилок  $\chi$ , задача  $\text{Decision-LWR}_{n,q,p}$  зводиться до задачі  $\text{Decision-LWE}_{n,q,\chi}$ .

**Зауваження.** (Ідея теореми) При достатньо великому відношенні  $q/p$ , округлення  $\lfloor \langle \mathbf{a}, \mathbf{s} \rangle \rfloor_p$  стає еквівалентно до додавання малої помилки округлення, яка розподілена майже рівномірно на інтервалі  $(-q/(2p), q/(2p)]$ .

### 3.2.5 Задача Ring Learning With Rounding (RLWR)

RLWR поєднує переваги RLWE (компактність – за рахунок алгебраїчної структури) та LWR (детермінованість, та відсутність потреби у відборі помилок(sampling)).

**Означення 3.2.9** (RLWR задача).

Нехай  $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ ,  $R_p = \mathbb{Z}_p[x]/(x^n + 1)$ , де  $p < q$ . Функція округлення застосовуються для кожного коефіцієнта окремо:

$$\lfloor f \rfloor_p = \sum_{i=0}^{n-1} \left\lfloor \frac{p}{q} \cdot f_i \right\rfloor x^i \bmod p.$$

Розпізнавальна задача  $\text{Decision-RLWR}_{n,q,p}$  полягає у розрізненні:

- $(a, \lfloor a \cdot s \rfloor_p)$ , де  $a \in R_q$  обрано рівномірно,  $s \in R_q$  фіксоване;
- $(a, u)$ , де  $a \in R_q$ ,  $u \in R_p$  обрані рівномірно.

**Твердження 3.2.1.**

При певних параметрах  $n, q, p$ , задача  $\text{RLWR}_{n,q,p}$  є не легшою за задачу  $\text{RLWE}_{n,q,\chi}$  з розподілом помилок  $\chi$ , що відповідає помилці округлення.

У TiGER використовується комбінація: RLWR – для генерації відкритого ключа (компактність) та RLWE – для шифрування (гнучкості у контролі помилок). Модулі обираються як степені двійки:  $q = 256$ ,  $p \in \{64, 128\}$ , що дозволяє реалізувати округлення через побітові зсуви. Про це поговоримо детальніше наступному, 4 розділі.

## 3.3 "Сімейство" алгоритмів на базі RLWE/RLWR

TiGER належить до сімейства алгоритмів на решітках, які базуються на задачах RLWE та RLWR. Опишемо основні алгоритми цього сімейства, які вплинули на дизайн TiGER (далі). Далі їх буде порівняно у розділі 6 та попередньо у таблиці 3.1.

### 3.3.1 Lizard

Lizard [6] був одним з перших алгоритмів, що комбінував у собі LWE та LWR для досягнення балансу між безпекою та ефективністю.

**Основні характеристики:**

- **Структура:** LWE для генерації відкритого ключа, LWR для шифрування;

- **Простір:** Цілочисельні решітки над  $\mathbb{Z}_q^n$  без використання кільцевої структури;
- **Модуль:** Малий модуль  $q$  для покращення коректності;
- **Розміри:** Великі ключі (через відсутність алгебраїчної структури).

**Переваги:**

- Консервативна безпека (базується на стандартній LWE);
- Низька ймовірність помилки розшифрування.

**Недоліки:**

- Великі розміри ключів та шифротекстів;
- Повільніше множення векторів (порівняно з кільцевими варіантами).

### 3.3.2 RLizard

RLizard [7] є кільцевою версією попереднього алгоритму, оптимізованою для IoT-пристроїв.

**Основні характеристики:**

- **Структура:** RLWE для генерації ключів, RLWR для шифрування;
- **Простір:**  $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ ,  $n = 512$  або  $n = 1024$ ;
- **Модуль:** Малий модуль  $q$  (наприклад,  $q = 1024$ );
- **Помилки:** Дискретний гаусівський розподіл помилок з високою точністю (CDT-sampling).

**Переваги:**

- Компактні ключі та шифротексти завдяки кільцевій структурі;
- Швидке шифрування/розшифрування;
- Висока коректність;
- Підходить для пристроїв з обмеженим ресурсом.

**Недоліки:**

- Відносно велика пропускну здатність порівняно з найкомпактнішими схемами;
- Залежність від якісного(справді випадкового) семплювання гаусівських помилок.

### 3.3.3 CRYSTALS-Kyber

Kyber [14] є одним з фіналістів конкурсу NIST PQC і базується на Module-LWE (MLWE).

**Основні характеристики:**

- **Структура:** MLWE для обох операцій – генерації ключів та шифрування повідомлення;
- **Простір:** Модульні решітки  $R_q^k$ , де  $k \in \{2, 3, 4\}$  (залежно від рівня безпеки);

- **Модуль:**  $q = 3329$  (просте число для ефективного NTT);
- **Оптимізація:** Number Theoretic Transform (NTT) для швидкого множення багаточленів (хоча, тут порівнюючи з чим саме і що вважати швидко);
- **Компресія:** Агресивне "стиснення" шифротексту.

**Переваги:**

- Стандартизовано під NIST (FIPS 203);
- Непоганий баланс між розмірами, швидкістю та безпекою;
- Ефективна реалізація з NTT.

**Недоліки:**

- Використання простого модуля ускладнює деякі оптимізації;
- Більший шифротекст порівняно з деякими MLWR-схемами.

### 3.3.4 Saber

Saber [15] є Module-LWR схемою, фіналістом NIST PQC Round 3.

**Основні характеристики:**

- **Структура:** MLWR для обох операцій;
- **Простір:** Модульні решітки  $R_q^k$ ,  $k \in \{2, 3, 4\}$ ;
- **Модуль:**  $q = 8192 = 2^{13}$ ;
- **Оптимізація:** Відсутність NTT (округлення завдяки побітовим операціям);
- **Помилки:** Детерміновані (через застосування округлення).

**Переваги:**

- Компактний шифротекст;
- Простота реалізації (без потреби в NTT чи гаусівському семплюванні);
- Ефективні побітові операції;
- Хороша стійкість до side-channel атак.

**Недоліки:**

- Більший(за розміром) відкритий ключ порівняно з Kyber;

### 3.3.5 Порівняння алгоритмів, і що з них взяв TiGER

Характеристика	Lizard	RLizard	Kyber	Saber
Структура	LWE/LWR	RLWE/RLWR	MLWE	MLWR
Розмірність	$n$	$n$	$n \times k$	$n \times k$
Модуль $q$	малий	малий	3329	8192
NTT	Ні	Ні	Так	Ні
Семплювання	Гаусс	Гаусс/CDT	Centered binomial	Округлення
Компресія	Помірна	Помірна	Агресивна	Помірна

Таблиця 3.1: Порівняння підходів у сімействі RLWE/RLWR алгоритмів

TiGER об'єднує в собі найкращі ідеї з попередніх доробок:

#### Позиція TiGER:

- Базується на **RLWE/RLWR** (як RLizard);
- Використовує **степені двійки** для  $q, p$  (як Saber);
- **Розріджені секрети** для ефективності;
- **Корекція помилок** (XEf, D2) для мінімізації DFP/R;
- **Без NTT** для простоти;
- Фокусування на **мінімальному шифротексті** для легшого впровадження в існуючі протоколи.

## 3.4 Криптографічні примітиви

У цій частині розділу розглянемо базові криптографічні примітиви, що використовуються для безпосередньо при побудові TiGER: схеми шифрування з відкритим ключем (PKE) та механізми інкапсуляції ключа (KEM).

### 3.4.1 Схема шифрування з відкритим ключем (PKE)

**Означення 3.4.1** (PKE схема).

Схема шифрування з відкритим ключем (*Public Key Encryption, PKE*) складається з трьох алгоритмів:

- $\text{KeyGen}(1^\lambda) \rightarrow (\text{pk}, \text{sk})$  — ймовірнісний алгоритм генерації ключів, що на вході приймає параметр безпеки  $\lambda$  і повертає пару: відкритий ключ  $\text{pk}$  та секретний ключ  $\text{sk}$ ;
- $\text{Enc}(\text{pk}, m; r) \rightarrow c$  — ймовірнісний алгоритм шифрування, що приймає відкритий ключ  $\text{pk}$ , повідомлення  $m$  з простору повідомлень  $\mathcal{M}$  та випадкове число  $r$ , і повертає шифротекст  $c$ ;

- $\text{Dec}(\text{sk}, c) \rightarrow m'$  — детермінований алгоритм розшифрування, що приймає секретний ключ  $\text{sk}$  і шифротекст  $c$ , та повертає повідомлення  $m'$  або помилку  $\perp$ .

#### Означення 3.4.2 (Коректність PKE).

PKE схема є  $(1 - \delta)$ -коректною, якщо для будь-якої пари ключів  $(\text{pk}, \text{sk}) \leftarrow \text{KeyGen}(1^\lambda)$  та будь-якого повідомлення  $m \in \mathcal{M}$  виконується:

$$\mathbb{P}[\text{Dec}(\text{sk}, \text{Enc}(\text{pk}, m)) \neq m] \leq \delta,$$

де ймовірність визначається через випадковість алгоритму  $\text{Enc}$ . Параметр  $\delta$  називається ймовірністю помилки розшифрування (*Decryption Failure Probability/Rate, DFP/R*)

У задачах на решітках, через наявність помилок у RLWE/RLWR, коректність не завжди є ідеальною. Тому для практичних застосувань необхідно, щоб  $\delta$  було незначним ( $\delta \leq 2^{-128}$ ).

### 3.4.2 Механізм інкапсуляції ключа (KEM)

#### Означення 3.4.3 (KEM схема).

Механізм інкапсуляції ключа (*Key Encapsulation Mechanism, KEM*) складається з трьох кроків:

- $\text{KeyGen}(1^\lambda) \rightarrow (\text{pk}, \text{sk})$  — алгоритм генерації ключів (аналогічно до PKE);
- $\text{Encaps}(\text{pk}) \rightarrow (c, K)$  — ймовірнісний алгоритм інкапсуляції, що приймає відкритий ключ  $\text{pk}$  і повертає шифротекст  $c$  та спільний секретний ключ  $K \in \mathcal{K}$ ;
- $\text{Decaps}(\text{sk}, c) \rightarrow K'$  — детермінований алгоритм декапсуляції, що приймає секретний ключ  $\text{sk}$  і шифротекст  $c$ , та повертає прихований секретний ключ  $K'$  або символ помилки  $\perp$ .

#### Означення 3.4.4 (Коректність KEM).

KEM схема є  $(1 - \delta)$ -коректною, якщо для будь-якої пари ключів  $(\text{pk}, \text{sk}) \leftarrow \text{KeyGen}(1^\lambda)$  справджується таке:

$$\mathbb{P}[\text{Decaps}(\text{sk}, c) \neq K : (c, K) \leftarrow \text{Encaps}(\text{pk})] \leq \delta.$$

### 3.4.3 Побудова KEM з PKE

Стандартний спосіб побудови KEM з PKE полягає у шифруванні випадкового повідомлення та використанні геш-функції для отримання спільного ключа.

#### Твердження 3.4.1 (Нативна побудова KEM).

Нехай  $\text{PKE} = (\text{KeyGen}, \text{Enc}, \text{Dec})$  — PKE це ось така трійка, що містить у собі простір повідомлень  $\mathcal{M}$ , та  $H : \mathcal{M} \rightarrow \mathcal{K}$  — деяка геш-функція. Тоді можна побудувати KEM таким способом:

- Спосіб генерації  $\text{KeyGen}$  такий же, як у PKE;
- $\text{Encaps}(\text{pk})$ : обирають  $m \in \mathcal{M}$ , а далі обчислюють  $c \leftarrow \text{Enc}(\text{pk}, m)$  та  $K \leftarrow H(m)$ . На виході отримано пару:  $(c, K)$ ;

- $\text{Decaps}(\text{sk}, c)$ : обчислюють  $m' \leftarrow \text{Dec}(\text{sk}, c)$  та на виході отримуємо  $K' \leftarrow H(m')$ .

Така побудова не забезпечує IND-CCA безпеки навіть якщо базова PKE схема є IND-CPA безпечною, але для нас головне щоб було зрозуміло як цей механізм працює). А для досягнення IND-CCA безпеки вже необхідно застосувати перетворення Fujisaki-Okamoto (розглянемо його у секції 3.6).

### 3.4.4 Криптографічні геш-функції

**Означення 3.4.5** (Криптографічна геш-функція).

Функція  $H : \{0, 1\}^* \rightarrow \{0, 1\}^n$  називається криптографічною геш-функцією, якщо вона задовольняє наступні умови:

1. **Стійкість до знаходження прообразу:** Для випадкового  $y \in \{0, 1\}^n$  обчислювально важко знайти  $x$  такий, що  $H(x) = y$ ;
2. **Стійкість до знаходження другого прообразу:** Для заданого  $x$  обчислювально важко знайти  $x' \neq x$  такий, що  $H(x') = H(x)$ ;
3. **Стійкість до колізій:** Обчислювально важко знайти дві різні величини  $x, x'$  такі, що  $H(x) = H(x')$ .

У TiGER використовуються геш-функції з сімейства SHA-3 – SHAKE256 (або SHA3-256) для генерації випадковості, обчислення спільних ключів та інших криптографічних операцій. SHAKE256 є функцією з розширеним виходом (XOF), що дозволяє генерувати вихід довільної довжини.

### 3.4.5 Зв'язок PKE та KEM у TiGER

TiGER складається з двох рівнів:

1. **TiGER.PKE** – являє собою базову IND-CPA безпечну схему шифрування, що базується на RLWE(R);
2. **TiGER.KEM** – KEM, отриманий застосуванням перетворення Fujisaki-Okamoto (FO) до TiGER.PKE для досягнення IND-CCA безпеки.

Це дозволяє:

- Окремо аналізувати безпеку PKE (на базі RLWE/RLWR);
- Використовувати загальні результати про перетворення FO для доведення IND-CCA безпеки KEM;

## 3.5 Основні поняття безпеки, що стосуються TiGER

### 3.5.1 IND-CPA безпека

**Означення 3.5.1** (IND-CPA (Indistinguishability under Chosen-Plaintext Attack) гра для PKE).

Розглянемо наступну "гру" між претендентом (challenger)  $\mathcal{C}$  та супротивником (adversary)  $\mathcal{A}$ :

1.  $\mathcal{C}$  генерує  $(pk, sk) \leftarrow \text{KeyGen}(1^\lambda)$  і передає  $pk$  супротивнику  $\mathcal{A}$ ;
2.  $\mathcal{A}$  обирає два повідомлення  $m_0, m_1 \in \mathcal{M}$  однакової довжини і передає їх  $\mathcal{C}$ ;
3.  $\mathcal{C}$  Випадковим чином обирає біт  $b \xleftarrow{p} \{0, 1\}$ , обчислює  $c \leftarrow \text{Enc}(pk, m_b)$  і передає  $c$  супротивнику  $\mathcal{A}$ ;
4.  $\mathcal{A}$  виводить біт  $b'$ .

Перевага супротивника визначається наступним чином:

$$\text{Adv}_{\text{PKE}}^{\text{IND-CPA}}(\mathcal{A}) = \left| \mathbb{P}[b' = b] - \frac{1}{2} \right| + \varepsilon(\lambda).$$

РКЕ схема є IND-CPA безпечною, якщо для будь-якого ефективного (PPT) супротивника  $\mathcal{A}$  перевага  $\text{Adv}_{\text{PKE}}^{\text{IND-CPA}}(\mathcal{A})$  є незначною функцією від  $\lambda$ .

IND-CPA безпечність означає нерозрізненість шифротекстів: супротивник маючи доступ до відкритого ключа (а отже, має можливість шифрувати будь-які повідомлення), не може визначити, яке з двох повідомлень було зашифроване. Це вимагає від шифрування, щоб воно було ймовірнісним.

**Означення 3.5.2** (IND-CPA (Indistinguishability under Chosen-Plaintext Attack) безпека для КЕМ).

Для КЕМ "гра" IND-CPA визначається аналогічно:

1.  $\mathcal{C}$  генерує пару  $(pk, sk) \leftarrow \text{KeyGen}(1^\lambda)$  і передає  $pk$  супротивнику  $\mathcal{A}$ ;
2.  $\mathcal{C}$  обирає  $b \xleftarrow{p} \{0, 1\}$ . Якщо обрано  $b = 0$ , то обчислює  $(c, K_0) \leftarrow \text{Encaps}(pk)$ ; а якщо  $b = 1$ , обчислює  $(c, K_0) \leftarrow \text{Encaps}(pk)$  та  $K_1 \xleftarrow{p} \mathcal{K}$ . Опісля передає пару  $(c, K_b)$  супротивнику;
3.  $\mathcal{A}$  виводить біт  $b'$ .

Перевага супротивника  $\text{Adv}_{\text{КЕМ}}^{\text{IND-CPA}}(\mathcal{A})$  визначається аналогічно як і в РКЕ.

Безпека КЕМ тісно пов'язана з:

- IND-CPA безпекою базової РКЕ схеми;
- Ймовірністю помилки розшифрування  $\varepsilon$ ;
- Параметрами випадкових оракулів (про трохи далі).

Важливим є те, що навіть за наявності помилок розшифрування (що неминуче для схем на решітках), можна довести IND-CCA безпеку за умови достатньо малого  $\varepsilon(\lambda)$ .

### 3.5.2 IND-CCA безпека

**Означення 3.5.3** (IND-CCA для РКЕ).

"Гра" IND-CCA відрізняється від IND-CPA тим, що противник  $\mathcal{A}$  має додатково доступ до оракула дешифратора (decryption oracle)  $\text{Dec}(sk, \cdot)$ :

1.  $\mathcal{C}$  генерує  $(pk, sk) \leftarrow \text{KeyGen}(1^\lambda)$  і передає  $pk$  супротивнику  $\mathcal{A}$ ;

2.  $\mathcal{A}$  робить запити до оракула дешифрування, подаючи на вхід довільні шифротексти  $c_i$  і отримувати  $m_i = \text{Dec}(\text{sk}, c_i)$ ;
3.  $\mathcal{A}$  обирає  $m_0, m_1$  і отримує challenge шифротекст  $c^* \leftarrow \text{Enc}(\text{pk}, m_b)$  для випадкового  $b$ ;
4.  $\mathcal{A}$  продовжує робити запити до оракула дешифрування, але не може запитувати  $c^*$  – випадок **IND-CCA2** (а якщо запити заборонені вже після отримання  $c^*$  – це **IND-CCA1**);
5.  $\mathcal{A}$  виводить біт  $b'$ .

Перевага  $\text{Adv}_{\text{PKE}}^{\text{IND-CCA}}(\mathcal{A})$  визначається аналогічно.

### Означення 3.5.4 (IND-CCA безпека для KEM).

Для KEM схеми противник має доступ до оракула декапсуляції  $\text{Decaps}(\text{sk}, \cdot)$  і **не може** запитувати challenge шифротекст  $c^*$  після його отримання.

IND-CCA (Indistinguishability under Chosen Ciphertext Attack) безпека є значно сильнішою, ніж IND-CPA, оскільки моделює активного противника, який може маніпулювати шифротекстами та спостерігати результати дешифрування. Для практичних застосувань зазвичай потрібна IND-CCA2 безпека.

## 3.5.3 Ймовірність помилки розшифрування (DFP/R)

### Означення 3.5.5 (Decryption Failure Probability/Rate).

Для PKE схеми ймовірність помилки розшифрування (DFP/R) визначається як:

$$\delta = \max_{m \in \mathcal{M}} \mathbb{P}_{(\text{pk}, \text{sk}), r}[\text{Dec}(\text{sk}, \text{Enc}(\text{pk}, m; r)) \neq m],$$

де ймовірність береться за випадковістю генерації ключів та шифрування.

Для KEM схеми:

$$\delta = \mathbb{P}_{(\text{pk}, \text{sk}), (c, K)}[\text{Decaps}(\text{sk}, c) \neq K].$$

У решіткових схемах помилки розшифрування виникають природнім шляхом (наявність шуму у RLWE/RLWR), тому параметри схеми (розміри модулів, розподіл помилок, коефіцієнт стиснення) мають бути підібрані так, щоб забезпечити мале  $\delta$ .

(!) Висока ймовірність помилки розшифрування може призвести до наступних атак:

- **Failure boosting attacks [16]:** Супротивник може ітеративно створювати шифротексти, що мають високу ймовірність помилки, щоб витягувати поступово інформацію про секретний ключ;
- **Multi-target attacks [17]:** При наявності багатьох публічних ключів або сесій, навіть відносно мала DFP/R може стати проблемою.

Для реалізацій TiGER цільова DFP/R становить:

- TiGER128:  $\delta \approx 2^{-120}$ ;
- TiGER192:  $\delta \approx 2^{-136}$ ;
- TiGER256:  $\delta \approx 2^{-167}$ .

P.S. Ці значення вважаються достатньо малими для практичного застосування.

### 3.5.4 Квантова безпека а.к.а. QROM

**Означення 3.5.6** (Quantum Random Oracle Model).

Модель квантового випадкового оракула (скорочено QROM) – це розширення класичної моделі випадкового оракула (ROM), де противник має квантовий доступ до геи-функцій, тобто може ще робити запити у суперпозиції.

Для TiGER необхідно, щоб перетворення Fujisaki-Okamoto забезпечувало IND-ССА безпеку у QROM, це гарантуватиме стійкість проти квантових атак.

## 3.6 Перетворення Fujisaki-Okamoto

Перетворення Fujisaki-Okamoto (скорочено FO) [9, 10] є загальним методом перетворення IND-CPA безпечної PKE схеми у IND-ССА безпечну KEM схему. В цьому підпункті наведемо просте класичне FO перетворення та його модифікацію – варіант з неявним відхиленням, що використовується в TiGER.

### 3.6.1 Класичне перетворення FO

**Теорема 3.6.1** (Fujisaki-Okamoto).

Нехай  $PKE = (\text{KeyGen}, \text{Enc}, \text{Dec})$  – IND-CPA безпечна PKE схема з однозначним детермінованим розшифруванням. Нехай  $G : \mathcal{M} \rightarrow \mathcal{R} \times \mathcal{K}$  та  $H : \mathcal{M} \times \mathcal{C} \rightarrow \mathcal{K}$  – випадкові оракули. Тоді наступна конструкція є IND-ССА безпечною KEM у моделі випадкового оракула [9]:

#### Algorithm 1 KeyGen()

- 1: Generate  $(pk, sk) \leftarrow PKE.\text{KeyGen}(1^\lambda)$
- 2: **return**  $(pk, sk)$

#### Algorithm 2 Encaps( $pk$ )

- 1: Choose  $m \xleftarrow{p} \mathcal{M}$
- 2: Calculate  $(r, K) \leftarrow G(m)$
- 3: Calculate  $c \leftarrow PKE.\text{Enc}(pk, m; r)$
- 4: **return**  $(c, K)$

#### Algorithm 3 Decaps( $sk, c$ )

- 1: Calculate  $m' \leftarrow PKE.\text{Dec}(sk, c)$
- 2: Calculate  $(r', K') \leftarrow G(m')$
- 3: Calculate  $c' \leftarrow PKE.\text{Enc}(pk, m'; r')$
- 4: **if**  $c = c'$  **then**
- 5:     **return**  $K'$
- 6: **else**
- 7:     **return**  $\perp$
- 8: **end if**

**Ключова ідея перетворення FO** полягає у **повторному шифруванні** (re-encryption); після розшифрування повідомлення  $m'$  воно повторно шифрується з тією ж випадковістю, і результат порівнюється з отриманим шифротекстом. Це дозволяє виявити модифікації шифротексту.

### 3.6.2 Перетворення $FO_m^\perp$ з неявним відхиленням

Для схем на решітках з ненульовою ймовірністю помилки розшифрування було розроблено модифікацію – варіант FO з **неявним відхиленням** (implicit rejection) [18].

**Означення 3.6.1** (Перетворення  $FO_m^\perp$ ).

Нехай  $PKE$  – IND-CPA безпечна  $PKE$  схема. Конструкція  $FO_m^\perp$  відрізняється від класичного FO у додатковій декапсуляції:

#### Algorithm 4 KeyGen()

- 1: Generate  $(pk, sk') \leftarrow PKE.KeyGen(1^\lambda)$
- 2: Choose  $z \xleftarrow{p} \mathcal{Z}$  ▷ Додатковий випадковий ключ
- 3: **return**  $(pk, sk = (sk', z))$

#### Algorithm 5 Encaps( $pk$ )

- 1: Choose  $m \xleftarrow{p} \mathcal{M}$
- 2: Calculate  $r \leftarrow G(m, pk)$
- 3: Calculate  $c \leftarrow PKE.Enc(pk, m; r)$
- 4: Calculate  $K \leftarrow H(m, c)$
- 5: **return**  $(c, K)$

#### Algorithm 6 Decaps( $sk, c$ )

- 1: Split  $sk = (sk', z)$
- 2: Calculate  $m' \leftarrow PKE.Dec(sk', c)$
- 3: Calculate  $r' \leftarrow G(m', pk)$
- 4: Calculate  $c' \leftarrow PKE.Enc(pk, m'; r')$
- 5: **if**  $c = c'$  **then**
- 6:     **return**  $K' \leftarrow H(m', c)$
- 7: **else**
- 8:     **return**  $\bar{K} \leftarrow H(z, c)$  ▷ Неявне відхилення
- 9: **end if**

**Ключова відмінність:** замість повернення  $\perp$  при невдалій перевірці, алгоритм повертає псевдовипадковий ключ  $\bar{K} = H(z, c)$ , де  $z$  – секретний випадковий ключ. Це має дві переваги:

- **Захист від side-channel атак:** Оскільки зовнішньому спостерігачу стає важче визначити, чи відбулася помилка при декапсуляції;
- **Постійний час виконання:** Обидва варіанти (успіх/невдача) тепер виконують однакові операції гешування.

### 3.6.3 Застосування у алгоритмі TiGER

TiGER.KEM побудовано із застосуванням варіанту перетворення  $\text{FO}_m^\perp$  до TiGER.PKE:

- **TiGER.PKE:** Базується на RLWR (для відкритого ключа) та RLWE (для шифрування), забезпечує IND-CPA безпеку;
- **Геш-функції:** Використовуються  $G = H = \text{SHAKE256}$  (функція з розширеним виходом) для генерації випадковості та спільних ключів;
- **Додаткове гешування:** Відкритий ключ  $\text{pk}$  гешується разом з повідомленням:  $r \leftarrow G(m, H(\text{pk}))$ , що забезпечує захист від multi-target атак;
- **Неявне відхилення:** При невдалій декапсуляції повертається  $\bar{K} = H(z, c)$ , що захищає від витoku інформації про помилки та розкритті хоч і мізерної, та інформації, про ключ.

# Розділ 4

## Повний опис алгоритму TiGER

### 4.1 Загальна структура алгоритму

TiGER має модульну архітектуру, що складається з двох рівнів: базової схеми шифрування з відкритим ключем (TiGER.PKE) та механізму інкапсуляції ключа (TiGER.KEM), отриманого застосуванням перетворення Fujisaki-Okamoto до PKE схеми.

#### 4.1.1 Модульна побудова алгоритму

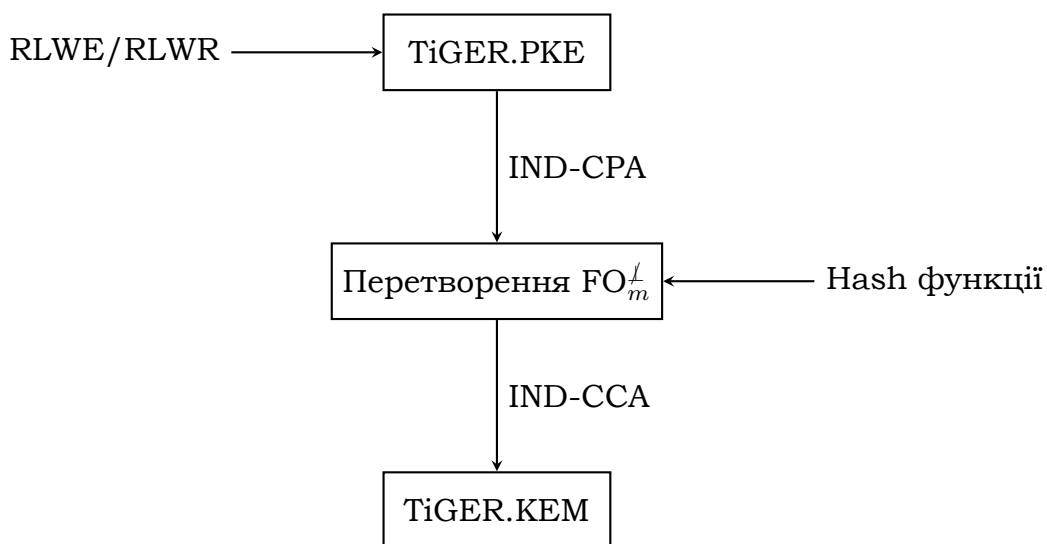


Рис. 4.1: Модульна структура TiGER

Конструкція TiGER базується на трьох наступних принципах:

1. **Перший крок – TiGER.PKE:** Схема шифрування з відкритим ключем, що використовує комбінацію RLWR для генерації відкритого ключа та RLWE для шифрування повідомлень. На припущення складності RLWE та RLWR задач маємо IND-CPA безпеку.
2. **Перетворення  $\text{FO}_m^f$ :** Варіант перетворення Fujisaki-Okamoto з неявним відхиленням, що перетворює IND-CPA безпечну PKE схему в IND-CCA безпечну KEM схему використовує геш-функції SHAKE256 та SHA3-256 як випадкові оракули  $H : \mathcal{M} \times \mathcal{C} \rightarrow \mathcal{K}$ .

3. **Вихід – TiGER.KEM:** Повний механізм інкапсуляції ключа з IND-CCA безпекою у моделі квантового випадкового оракула (QROM), придатний для практичного використання у постквантових протоколах.

#### 4.1.2 Взаємодія між компонентами TiGER

Взаємодія між компонентами відбувається наступним чином:

**Генерація ключів:**

- TiGER.PKE генерує пару  $(pk, sk')$  використовуючи RLWR;
- TiGER.KEM додає випадковий ключ  $z$  до секретного ключа:  $sk = (sk', z)$ ;
- Відкритий ключ  $pk$  гешується для захисту від multi-target атак.

**Інкапсуляція:**

- Генерується випадкове повідомлення  $m$ ;
- геш-функція  $G$  обчислює детерміновану випадковість  $r$  з  $m$  та гешу  $pk$ ;
- TiGER.PKE шифрує  $m$  з випадковістю  $r$  і видає шифротекст  $c$ ;
- Спільний ключ  $K$  обчислюється як  $H(m, c)$ .

**Декапсуляція:**

- TiGER.PKE розшифровує шифротекст  $c$  та повертає повідомлення  $m'$ ;
- Виконується повторне шифрування (re-encryption)  $m'$  для перевірки цілісності;
- Якщо перевірка успішна, повертається  $K' = H(m', c)$ ;
- Якщо перевірка невдала, повертається псевдовипадковий ключ  $\bar{K} = H(z, c)$  (неявне відхилення).

## 4.2 Публічні параметри

TiGER визначає три набори параметрів, що відповідають трьом рівням безпеки згідно з класифікацією NIST: TiGER128 (рівень 1), TiGER192 (рівень 3) та TiGER256 (рівень 5). Розглянемо детальніше значення кожного параметра та обґрунтування їх вибору.

Параметр	TiGER128	TiGER192	TiGER256
$n$	512	1024	1024
$q$	$2^{14}$	$2^{15}$	$2^{16}$
$p$	$2^{10}$	$2^{11}$	$2^{11}$
$k_1$ (бітів для <b>b</b> )	10	11	11
$k_2$ (бітів для <b>c</b> <sub>1</sub> )	4	4	5
$h_s$	274	284	274
$h_r$	274	284	274
$h_e$	274	284	274
$d$ (для XEf)	8	8	9
$f$ (для D2)	2	2	2
$ pk $ (кількість байт)	804	1568	1568
$ sk $ (кількість байт)	1876	3680	3680
$ ct $ (кількість байт)	804	1408	1600
Рівень безпеки NIST	1	3	5
Класична безпека (кількість біт)	143	207	272
Квантова безпека (кількість біт)	128	192	256
DFP	$2^{-120}$	$2^{-136}$	$2^{-167}$

Таблиця 4.1: Параметри TiGER для різних рівнів безпеки

### Структурні параметри:

- $n$  — степінь многочлена, що визначає розмірність поліноміального кільця  $R = \mathbb{Z}[x]/(x^n + 1)$ . Більше  $n$  забезпечує вищу безпеку, але і водночас збільшує обчислювальну складність та розміри ключів;
- $q$  — основний модуль, що визначає кільце  $R_q = R/qR$ . В даному алгоритмі обрано як степінь двійки для оптимізації операцій модульної арифметики.
- $p$  — модуль для RLWR округлення при генерації відкритого ключа. Задовольняє  $p < q$  та також є степенем двійки. Відношення  $q/p$  визначає рівень "шуму" від округлення.

### Параметри для стиснення:

- $k_1$  — кількість біт для представлення компонент вектора **b** у відкритому ключі. Стиснення з  $\log_2 q$  до  $k_1$  біт зменшує розмір відкритого ключа;
- $k_2$  — кількість біт для представлення першої частини шифротексту **c**<sub>1</sub>. Агресивніша компресія зменшує розмір шифротексту, але збільшує ймовірність помилки розшифрування(!).

### Параметри розподілів помилок:

- $h_s$  — вага Геммінга (Hamming weight) секретного ключа **s**. TiGER використовує розріджені тернарні многочлени з коефіцієнтами  $\{-1, 0, 1\}$ , де рівно  $h_s$  коефіцієнтів є ненульовими;
- $h_r$  — вага Геммінга помилки **r** при генерації відкритого ключа (RLWR);
- $h_e$  — вага Геммінга для помилок **e**<sub>1</sub>, **e**<sub>2</sub> при шифруванні (RLWE).

### Параметри кодів корекції помилок:

- $d$  — параметр коду XEf (extended XOR-based error correction). Визначає кількість біт повідомлення, які кодуються разом. Чим більше  $d$ , тим краща корекція помилок, але при цьому збільшуються обчислення;
- $f$  — параметр коду D2 (duplication code). Визначає частоту дублювання бітів повідомлення для додаткового захисту від помилок DFP/R.

Фактичні бітові розміри криптографічних об'єктів обчислюються наступним чином:

**Відкритий ключ**  $pk = (\rho, \mathbf{b})$ :

$$|pk| = 32 + n \cdot k_1 / 8 \text{ байт},$$

де 32 байти – розмір seed  $\rho$ , який використовується для генерації випадкового многочлена  $\mathbf{a} \in R_q$ . Замість зберігання повного многочлена  $\mathbf{a}$  (що займало б  $n \log_2 q$  біт) пам'яті, зберігається лише seed, з якого  $\mathbf{a}$  відновлюється за потреби. Компонента  $\mathbf{b}$  – це результат RLWR обчислення  $\mathbf{b} = \lfloor \mathbf{a} \cdot \mathbf{s} + \mathbf{r} \rfloor_p$ , стиснута до  $k_1$  біта на кожен коефіцієнт.

**Секретний ключ**  $sk = (sk', z, h, pk)$ :

$$|sk| = |sk'| + 32 + 32 + |pk| \text{ байт},$$

де  $sk'$  – зжате представлення розрідженого секретного ключа  $\mathbf{s}$ ,  $z$  та  $h$  — 32-байтові значення для FO перетворення.

**Шифротекст**  $ct = (\mathbf{c}_1, \mathbf{c}_2)$ :

$$|ct| = n \cdot k_2 / 8 + n / 8 \text{ байт}.$$

Параметри TiGER були підібрані розробниками з урахуванням наступних критеріїв:

1. **Безпека:** Забезпечують стійкість до відомих атак на RLWE/RLWR, включаючи атаки з використанням решіткових алгоритмів (BKZ, LatticeSieve) на класичних та квантових комп'ютерах. Запас безпеки становить 10-15 біт понад нормово;
2. **Коректність:** Ймовірність помилки розшифрування (DFP/R) не перевищує  $2^{-120}$  для всіх наборів параметрів, що є достатнім для практичного використання;
3. **Ефективність:** Вибір степенів двійки для модулів дозволяє використовувати швидкі побітові операції, а розріджені секрети прискорюють множення многочленів;
4. **Компактність:** Параметри компресії  $(k_1, k_2)$  обрані так, щоб мінімізувати розміри при збереженні прийнятного рівня DFP/R. Коди корекції помилок дозволяють застосовувати агресивнішу компресію без погіршення коректності.

## 4.3 Допоміжні алгоритми

TiGER використовує набір допоміжних алгоритмів для генерації випадкових величин, розширення seed-значень та корекції помилок.

1. **Алгоритм**  $\text{HWT}_h$  відповідає за генерацію розрідженого тернарного многочлена з фіксованою вагою Геммінга.

**Algorithm 7**  $\text{HWT}_h$ 

- 1: Input: Seed  $\sigma \in \{0, 1\}^{256}$ , weight  $h \leq n$ .
- 2: Output: Поліном  $\mathbf{s} \in R$  з коефіцієнтами  $\{-1, 0, 1\}$  та рівно  $h$  ненульовими коефіцієнтами.

Алгоритм використовує SHAKE256 для генерації послідовності випадкових індексів та знаків. Спочатку обираються  $h$  різних позицій в діапазоні  $[0, n - 1]$ , потім для кожної позиції генерується випадковий знак  $\pm 1$ . Використовується метод rejection sampling для забезпечення рівномірного розподілу індексів. Виконується за константний час (з точністю до rejection sampling).

2. **Алгоритм expandA** — генерація псевдовипадкового многочлена  $\mathbf{a}$  з seed.

**Algorithm 8**  $\text{HWT}_h$ 

- 1: Input: Seed  $\rho \in \{0, 1\}^{256}$ .
- 2: Output: Многочлен  $\mathbf{a} \in R_q$  з рівномірно розподіленими коефіцієнтами.

Тут також використовується SHAKE256 як додаткова псевдовипадковість (PRG). Seed  $\rho$  розширюється до послідовності байтів, які інтерпретуються як коефіцієнти многочлена в  $\mathbb{Z}_q$ . Застосовується rejection sampling для забезпечення рівномірності: якщо згенероване значення  $\geq q$ , воно відкидається і генерується нове. Многочлен  $\mathbf{a}$  використовується у RLWR для обчислення відкритого ключа:  $\mathbf{b}$  та у RLWE при шифруванні.

Цей підхід дозволяє зберігати у відкритому ключі лише 32 байти seed замість повного многочлена, що займав би  $n \log_2 q$  біт — для TiGER256 це  $1024 \times 16 = 16384$  біт = 2048 байт). Обидві сторони (відправник і отримувач) можуть незалежно відновити  $\mathbf{a}$  з  $\rho$ .

3. **Алгоритми корекції помилок eccENC та eccDEC** — кодування та декодування повідомлення для зниження DFP/R.

- Код XEf (XOR-based error correction with extension): Повідомлення розбивається на блоки по  $d$  біт. Для кожного блоку обчислюється біт парності як ксор (XOR) усіх біт блоку та додається до закодованого повідомлення. Це дозволяє виявити та виправити одиночні помилки в кожному блоці. При декодуванні для кожного блоку перевіряється цей біт парності. Якщо він не збігається, алгоритм намагається виправити помилку перебором всіх  $d$  позицій у блоці.
- Код D2 (Duplication code): Додатковий рівень захисту, що дублює кожен біт повідомлення  $f$  разів. При декодуванні використовується мажоритарне голосування (majority voting): якщо більшість копій біта мають значення 1, результат буде 1, інакше — 0.

TiGER застосовує спочатку код XEf, потім код D2, що дає двоетапну систему корекції помилок. Це дозволяє використовувати агресивнішу компресію шифротексту при збереженні низької DFP/R.

4. **Геш-функції** — використовуються для генерації випадковостей та обчислення спільних ключів у перетворенні FO.

- **SHAKE256**: Функція з розширеним виходом (XOF) зі стандарту SHA-3. Використовується першочергово як генератор псевдовипадковості (PRG) у  $\text{expandA}$  та  $\text{HWT}_h$ , функція  $G$  у перетворенні FO для отримання детермінованої випадковості з повідомлення та для генерації довгих послідовностей псевдовипадкових байтів.
- **SHA3-256**: Криптографічна геш-функція зі стандарту SHA-3. Використовується як функція  $H$  у перетворенні Fujisaki-Okamoto для обчислення спільного ключа  $K = H(m, c)$ , гешуванні відкритого ключа для захисту від multi-target атак та для генерації фінального спільного ключа фіксованого розміру (256 біт).

Обидві ці функції є частиною стандарту FIPS 202 (2015 р) та мають формальний аналіз безпеки. У моделі квантового випадкового оракула (QROM) вони моделюються як ідеальні випадкові функції та доступними супротивнику.

## 4.4 TiGER.PKE

TiGER.PKE це базова схема шифрування з відкритим ключем, що забезпечує IND-CPA безпеку. Схема використовує комбінацію RLWR для генерації відкритого ключа та RLWE для шифрування повідомлень.

### 4.4.1 KeyGen – генерація ключів

Алгоритм генерації ключів створює пару відкритий&секретний ключ на основі RLWR.

#### Algorithm 9 TiGER.PKE.KeyGen()

```

1: Input: Параметри  $(n, q, p, h_s, h_r, k_1)$ 
2: Output: Відкритий ключ  $pk$ , секретний ключ  $sk$ 
3:
4: Generate seed  $\rho \xleftarrow{p} \{0, 1\}^{256}$ 
5: Generate seed  $\sigma_s \xleftarrow{p} \{0, 1\}^{256}$ 
6: Generate seed  $\sigma_r \xleftarrow{p} \{0, 1\}^{256}$ 
7:
8:  $\mathbf{a} \leftarrow \text{expandA}(\rho)$                                 ▷ Expand seed в многочлен
9:  $\mathbf{s} \leftarrow \text{HWT}_{h_s}(\sigma_s)$                             ▷ Sparse секретний ключ
10:  $\mathbf{r} \leftarrow \text{HWT}_{h_r}(\sigma_r)$                             ▷ Sparse error
11:
12:  $\mathbf{b}' \leftarrow \mathbf{a} \cdot \mathbf{s} + \mathbf{r} \in R_q$                                 ▷ RLWR обчислення
13:  $\mathbf{b}' \leftarrow \lfloor \mathbf{b}' \cdot (p/q) \rfloor \in R_p$                             ▷ Округлення за mod p
14:  $\mathbf{b} \leftarrow \text{Compress}(\mathbf{b}', k_1)$                                 ▷ Компресія до  $k_1$  біт
15:
16:  $pk \leftarrow (\rho, \mathbf{b})$ 
17:  $sk \leftarrow \sigma_s$                                 ▷ Save secret seed(!)
18: return  $(pk, sk)$ 

```

Кроки алгоритму (якщо словами):

1. **Генерація seeds:** Спершу створюються три незалежні 256-бітні seeds:

- $\rho$  — для генерації публічного многочлена  $\mathbf{a}$ ;
- $\sigma_s$  — для генерації секретного ключа  $\mathbf{s}$ ;
- $\sigma_r$  — для генерації помилки  $\mathbf{r}$ .

2. **Expand a:** Використовується алгоритм `expandA` для детермінованої генерації рівномірно випадкового многочлена  $\mathbf{a} \in R_q$  з seed  $\rho$ .

### 3. Генерація розріджених многочленів:

- Секретний ключ  $\mathbf{s}$  генерується як тернарний многочлен з вагою Геммінга  $h_s$ ;
- Помилка  $\mathbf{r}$  аналогічно генерується з вагою  $h_r$ .

4. **RLWR:** Обчислюється  $\mathbf{b}' = \mathbf{a} \cdot \mathbf{s} + \mathbf{r} \in R_q$ , після чого виконується округлення до найближчого цілого модуля  $p$ :  $\lfloor \mathbf{b}' \cdot (p/q) \rfloor$ . Операція округлення вводить детермінований "шум", що замінює явну помилку в класичному RLWE.

5. **Компресія:** Многочлен  $\mathbf{b}'$  стискається з  $\log_2 p$  біт до  $k_1$  біт на коефіцієнт для зменшення розміру відкритого ключа.

6. **Формування ключів:** Відкритий ключ містить seed  $\rho$  та стиснений  $\mathbf{b}'$ , а секретний ключ зберігається у вигляді seed  $\sigma_s$ , з якого можна відновити  $\mathbf{s}$  за потреби.

## 4.4.2 Encryption – шифрування

Алгоритм шифрування перетворює повідомлення в шифротекст використовуючи RLWE.

### Algorithm 10 TiGER.PKE.Enc( $pk, m; r$ )

```

1: Input: Відкритий ключ  $pk = (\rho, \mathbf{b})$ , повідомлення  $m \in \{0, 1\}^{256}$ , випадковість  $r$ 
2: Output: Шифротекст  $ct = (\mathbf{c}_1, \mathbf{c}_2)$ 
3:
4: Parse seeds from  $r$ :  $(\sigma_{e_1}, \sigma_{e_2}) \leftarrow r$ 
5:
6:  $\mathbf{a} \leftarrow \text{expandA}(\rho)$  ▷ Відновлення  $\mathbf{a}$  з seed
7:  $\mathbf{b}' \leftarrow \text{Decompress}(\mathbf{b}, k_1)$  ▷ Декомпресія відкритого ключа
8:
9:  $\mathbf{e}_1 \leftarrow \text{HWT}_{h_e}(\sigma_{e_1})$  ▷ Генерація "помилки"
10:  $\mathbf{e}_2 \leftarrow \text{HWT}_{h_e}(\sigma_{e_2})$ 
11:
12:  $m' \leftarrow \text{eccENC}(m)$  ▷ Кодування з корекцією помилок
13:  $\mathbf{m} \leftarrow \text{Encode}(m')$  ▷ Перетворення біт в многочлен
14:
15:  $\mathbf{c}'_1 \leftarrow \mathbf{a} \cdot \mathbf{e}_1 + \mathbf{e}_2 \in R_q$  ▷ RLWE компонента
16:  $\mathbf{c}_1 \leftarrow \text{Compress}(\mathbf{c}'_1, k_2)$  ▷ Агресивна компресія
17:
18:  $\mathbf{c}'_2 \leftarrow \mathbf{b}' \cdot \mathbf{e}_1 + \mathbf{m} \cdot \lfloor q/2 \rfloor \in R_q$  ▷ Повідомлення + шум
19:  $\mathbf{c}_2 \leftarrow \text{Compress}(\mathbf{c}'_2, 1)$  ▷ Компресія до 1 біт
20:
21: return  $ct = (\mathbf{c}_1, \mathbf{c}_2)$ 

```

Кроки алгоритму:

1. **Парсинг випадковості:** З детермінованої випадковості  $r$  (згенерованої через ген-функцію  $G$  у FO) отримуємо seeds для генерації помилок  $\mathbf{e}_1, \mathbf{e}_2$ .
2. **Відновлення відкритого ключа:**
  - Многочлен  $\mathbf{a}$  відновлюється з  $\rho$  seed;
  - Компресований  $\mathbf{b}$  декомпресується до  $\mathbf{b}' \in R_p$ .
3. **Генерація "помилки":** Створюються два розріджених тернарних многочлена  $\mathbf{e}_1, \mathbf{e}_2$  з вагою Геммінга  $h_e$  кожен.
4. **Підготовка повідомлення:**
  - Повідомлення  $m$  кодується через eccENC (коди XEf та D2) для захисту від помилок;
  - Закодоване повідомлення перетворюється в многочлен  $\mathbf{m} \in R$  з коефіцієнтами з  $\{0, 1\}$ ;
  - Множиться на  $\lfloor q/2 \rfloor$  для розміщення в "середині" модуля  $q$ .
5. **RLWE шифрування:**
  - $\mathbf{c}'_1 = \mathbf{a} \cdot \mathbf{e}_1 + \mathbf{e}_2$  — "маскування" помилки  $\mathbf{e}_1$ ;
  - $\mathbf{c}'_2 = \mathbf{b}' \cdot \mathbf{e}_1 + \mathbf{m} \cdot \lfloor q/2 \rfloor$  — зашифроване повідомлення.
6. **Компресія шифротексту:**
  - $\mathbf{c}_1$  стискається до  $k_2$  біт на коефіцієнт (агресивна компресія);
  - $\mathbf{c}_2$  стискається до 1 біт на коефіцієнт (зберігається лише знак).

#### 4.4.3 Decryption – розшифрування

Алгоритм розшифрування відновлює повідомлення з шифротексту використовуючи секретний ключ.

**Algorithm 11** TiGER.PKE.Dec( $sk, ct$ )

```

1: Input: Секретний ключ  $sk = \sigma_s$ , шифротекст  $ct = (\mathbf{c}_1, \mathbf{c}_2)$ 
2: Output: Повідомлення  $m \in \{0, 1\}^{256}$  або  $\perp$ 
3:
4:  $\mathbf{s} \leftarrow \text{HWT}_{h_s}(\sigma_s)$  ▷ Відновлення  $sk$  з seed
5:
6:  $\mathbf{c}'_1 \leftarrow \text{Decompress}(\mathbf{c}_1, k_2)$  ▷ Декомпресія шифротексту
7:  $\mathbf{c}'_2 \leftarrow \text{Decompress}(\mathbf{c}_2, 1)$ 
8:
9:  $\mathbf{v} \leftarrow \mathbf{c}'_2 - \mathbf{c}'_1 \cdot \mathbf{s} \in R_q$  ▷ Видалення маски
10:
11:  $\mathbf{m}' \leftarrow \text{Round}(\mathbf{v})$  ▷ Округлення до  $\{0, 1\}$ 
12:  $m' \leftarrow \text{Decode}(\mathbf{m}')$  ▷ Розбиття многочлена на біти
13:
14:  $m \leftarrow \text{eccDEC}(m')$  ▷ Декодування з корекцією помилок
15:
16: if  $m = \perp$  then
17:   return  $\perp$  ▷ Помилка розшифрування
18: else
19:   return  $m$ 
20: end if

```

Покроково маємо:

1. **Відновлення секретного ключа:** З seed  $\sigma_s$  відновлюється розріджений многочлен  $\mathbf{s}$ .
2. **Декомпресія шифротексту:** Обидві компоненти  $\mathbf{c}_1, \mathbf{c}_2$  декомпресуються до повного розміру в  $R_q$ .
3. **Видалення маски:** Обчислюється  $\mathbf{v} = \mathbf{c}'_2 - \mathbf{c}'_1 \cdot \mathbf{s}$ , що має бути близьким до  $\mathbf{m} \cdot \lfloor q/2 \rfloor$  (за наявності малої помилки).
4. **Округлення:** Функція Round округлює кожен коефіцієнт  $\mathbf{v}$  до найближчого з чисел: 0 або  $\lfloor q/2 \rfloor$ , потім нормалізує до  $\{0, 1\}$ :

$$\text{Round}(v_i) = \begin{cases} 0, & \text{якщо } |v_i| < q/4 \\ 1, & \text{якщо } |v_i - \lfloor q/2 \rfloor| < q/4 \\ \text{error}, & \text{інакше} \end{cases}$$

5. **Декодування:** Многочлен  $\mathbf{m}'$  перетворюється в послідовність біт  $m'$ , після чого застосовується eccDEC для виправлення помилок та отримання повідомлення  $m$ .
6. **Перевірка коректності:** Якщо eccDEC виявляє unexrest error, повертається  $\perp$ .

Джерела помилок розшифрування можуть виникнути завдяки:

- Помилка  $\mathbf{r}$  від RLWR при генерації  $\mathbf{b}$ ;
- Помилки  $\mathbf{e}_1, \mathbf{e}_2$  від RLWE при шифруванні;

- Помилки від компресії/декомпресії  $\mathbf{b}, \mathbf{c}_1, \mathbf{c}_2$  відповідно;

Параметри TiGER підбрані так, щоб сумарна помилка залишалась в межах  $\pm q/4$  з високою ймовірністю, забезпечуючи коректне розшифрування. Коди корекції помилок XEf та D2 додатково підвищують надійність.

## 4.5 TiGER.KEM

TiGER.KEM – механізм інкапсуляції ключа, отриманий застосуванням перетворення Fujisaki-Okamoto з неявним відхиленням до TiGER.PKE. Забезпечує IND-CCA безпеку у моделі квантового випадкового оракула.

### 4.5.1 KeyGen – генерація ключів

Алгоритм генерації ключів розширює TiGER.PKE.KeyGen додатковою компонентою – FO перетворенням.

#### Algorithm 12 TiGER.KEM.KeyGen()

```

1: Input: Параметри  $(n, q, p, h_s, h_r, k_1)$ 
2: Output: Відкритий ключ  $pk$ , секретний ключ  $sk$ 
3:
4:  $(pk', sk') \leftarrow \text{TiGER.PKE.KeyGen}()$  ▷ Базова генерація ключів
5:
6:  $z \xleftarrow{p} \{0, 1\}^{256}$  ▷ Випадковий ключ для implicit rejection
7:  $h \leftarrow \text{SHA3-256}(pk')$  ▷ Геш відкритого ключа
8:
9:  $pk \leftarrow pk'$ 
10:  $sk \leftarrow (sk', z, h, pk')$  ▷ Розширений секретний ключ
11:
12: return  $(pk, sk)$ 

```

#### Компоненти секретного ключа:

- $sk'$  — секретний ключ TiGER.PKE (правду кажучи це seed  $\sigma_s$ );
- $z$  — випадковий 256-бітний ключ для обчислення псевдовипадкового спільного ключа при невдалій декапсуляції (неявне відхилення);
- $h$  — геш відкритого ключа, використовується геш-функцією  $G$  для генерації випадковості шифрування;
- $pk'$  — копія відкритого ключа (для можливості повторного шифрування при декапсуляції).

### 4.5.2 Encapsulation – інкапсуляція

Алгоритм інкапсуляції генерує спільний ключ та його зашифровану форму.

**Algorithm 13** TiGER.KEM.Encaps( $pk$ )

```
1: Input: Відкритий ключ  $pk$ 
2: Output: Шифротекст  $ct$  та спільний ключ  $K \in \{0, 1\}^{256}$ 
3:
4:  $m \xleftarrow{p} \{0, 1\}^{256}$  ▷ Вибираємо випадкове повідомлення
5:
6:  $h \leftarrow \text{SHA3-256}(pk)$  ▷ Гешування відкритого ключа
7:  $r \leftarrow \text{SHAKE256}(m \| h)$  ▷ Детермінована випадковість
8:
9:  $ct \leftarrow \text{TiGER.PKE.Enc}(pk, m; r)$  ▷ Шифрування повідомлення
10:
11:  $K \leftarrow \text{SHA3-256}(m \| ct)$  ▷ Спільний ключ обчислюється через геш
12:
13: return ( $ct, K$ )
```

**Ключові аспекти:**

1. **Випадкове повідомлення:** Генерується рівномірно (розподілене) випадкове  $m \in \{0, 1\}^{256}$ , яке буде зашифроване.
2. **Детермінована випадковість:** Замість використання нової випадковості для шифрування,  $r$  обчислюється детермінована як  $r = \text{SHAKE256}(m \| h)$ . Це критично для можливості повторного шифрування при декапсуляції.
3. **Гешування відкритого ключа:** Включення  $h = \text{SHA3-256}(pk)$  в обчислення  $r$  забезпечує взаємозв'язок шифротексту до конкретного відкритого ключа, що захищає від multi-target атак.
4. **Спільний ключ:** Обчислюється як  $K = \text{SHA3-256}(m \| ct)$ , як можна бачити є залежність як від повідомлення так і від шифротексту. Це є важливим для безпеки алгоритму загалом: противник не може обчислити  $K$  без знання  $m$ , навіть якщо перехопив шифротекст  $ct$ .

### 4.5.3 Decapsulation – декапсуляція

Декапсуляція застосовується для відновлення спільного ключ з шифротексту, виконуючи перевірку цілісності через повторне шифрування.

**Algorithm 14** TiGER.KEM.Decaps( $sk, ct$ )

```

1: Input: Секретний ключ  $sk = (sk', z, h, pk)$ , шифротекст  $ct$ 
2: Output: Спільний ключ  $K \in \{0, 1\}^{256}$ 
3:
4:  $m' \leftarrow \text{TiGER.PKE.Dec}(sk', ct)$  ▷ Розшифрування
5:
6: if  $m' = \perp$  then
7:   return  $\bar{K} \leftarrow \text{SHA3-256}(z \| ct)$  ▷ Implicit rejection
8: end if
9:
10:  $r' \leftarrow \text{SHAKE256}(m' \| h)$  ▷ Створення "випадковості"
11:  $ct' \leftarrow \text{TiGER.PKE.Enc}(pk, m'; r')$  ▷ Повторне шифрування
12:
13: if  $ct' = ct$  then
14:   return  $K \leftarrow \text{SHA3-256}(m' \| ct)$  ▷ Успішна декапсуляція
15: else
16:   return  $\bar{K} \leftarrow \text{SHA3-256}(z \| ct)$  ▷ Implicit rejection
17: end if

```

1. **Розшифрування:** Спершу – звичайне розшифрування через  $\text{TiGER.PKE.Dec}$ . Якщо розшифрування видало помилку ( $m' = \perp$ ), одразу повертається псевдовипадковий ключ.
2. **Re-encryption:** Розшифроване повідомлення  $m'$  повторно шифрується з тією ж визначеною випадковістю  $r' = \text{SHAKE256}(m' \| h)$  (є критичном кроком для забезпечення IND-CCA безпеки).
3. **Перевірка цілісності:** Отриманий  $ct'$  порівнюється з оригінальним  $ct$ :
  - Якщо  $ct' = ct$ , це означає, що шифротекст не був модифікований противником, і тоді повертається справжній спільний ключ  $K = \text{SHA3-256}(m' \| ct)$ ;
  - Якщо  $ct' \neq ct$ , це сигналізує про атаку або помилку, і повертається псевдовипадковий ключ, щоб заплутати злоумисника.
4. **Неявне відхилення (implicit rejection):** Замість явного повернення помилки  $\perp$ , алгоритм повертає псевдовипадковий ключ  $\bar{K} = \text{SHA3-256}(z \| ct)$ . Це має дві такі переваги:
  - **Захист від side-channel атак:** Зовнішньому спостерігачу важко визначити, чи відбулась успішна декапсуляція чи ні, оскільки в обох випадках повертається деякий 256-бітний ключ;
  - **Константний час:** Що при успішному output, що при failed виконується однакова кількість геш-операцій.
5. **Роль секретного ключа  $z$ :** Випадковий ключ  $z$  є унікальним для кожної пари ключів та невідомим противнику. Це гарантує, що  $\bar{K}$  буде непередбачуваним для противника навіть за наявності багатьох невдалих декапсуляцій.

**Ключові моменти безпеки або ж чому це безпечно**

Перетворення  $\text{FO}_m^\perp$  з повторним шифруванням перетворює будь-яку IND-CPA безпечну PKE схему в IND-CCA безпечну KEM.

- **Детермінована випадковість:** Використання  $r = \text{SHAKE256}(m\|h)$  замість нової випадковості робить шифрування детермінованим для даного  $m$  та  $pk$ , що дозволяє виконати перевірку через повторне шифрування;
- **Гешування спільного ключа:**  $K = \text{SHA3-256}(m\|ct)$  робить спільний ключ непередбачуваним без знання  $m$ , навіть якщо  $ct$  скомпроментовано супротивником;
- **Захист від атак вибраного шифротексту:** Протівник може подавати довільні модифіковані шифротексти  $ct^*$  до оракула декапсуляції, але:
  - Якщо  $ct^*$  не є валідним шифруванням деякого  $m^*$ , перевірка умови  $ct' \neq ct^*$  виявить це;
  - Протівник отримає лише  $\bar{K} = \text{SHA3-256}(z\|ct^*)$ , що не надає жодної інформації про справжній спільний ключ через випадковість  $z$  та властивості геш-функції.

## 4.6 Аналіз коректності

Коректність вимагає від TiGER, того щоб розшифрування майже завжди повертало правильне повідомлення. У цій секції проаналізуємо джерела помилок та обчислимо ймовірність помилки розшифрування.

### Математичний аналіз помилок

При розшифруванні повідомлення обчислюється величина:

$$\mathbf{v} = \mathbf{c}'_2 - \mathbf{c}'_1 \cdot \mathbf{s} = \mathbf{m} \cdot \lfloor q/2 \rfloor + \mathbf{err},$$

де  $\mathbf{err}$  – сумарна помилка з декількох джерел.

**Джерела помилок є наступними:**

1. **Помилка від RLWR:** При генерації відкритого ключа:

$$\mathbf{b} = \lfloor (\mathbf{a} \cdot \mathbf{s} + \mathbf{r}) \cdot (p/q) \rfloor,$$

що вносить помилку округлення  $\mathbf{err}_{\text{RLWR}}$ , де  $\mathbf{err}_{\text{RLWR}} \approx \mathbf{r} \cdot (p/q)$ .

2. **Помилка від RLWE:** При шифруванні:

$$\mathbf{c}'_1 = \mathbf{a} \cdot \mathbf{e}_1 + \mathbf{e}_2, \quad \mathbf{c}'_2 = \mathbf{b}' \cdot \mathbf{e}_1 + \mathbf{m} \cdot \lfloor q/2 \rfloor,$$

що вносить помилку  $\mathbf{err}_{\text{RLWE}} = -\mathbf{e}_2 \cdot \mathbf{s} + \mathbf{err}_{\text{RLWR}} \cdot \mathbf{e}_1$ .

3. **Помилка від компресії відкритого ключа:** Компресія  $\mathbf{b}$  ( $\log_2 p \rightarrow k_1$ ) біт вносить наступну помилку округлення:

$$\mathbf{err}_{\text{comp}, \mathbf{b}} \approx \mathbf{U}([-p/2^{k_1+1}, p/2^{k_1+1}]).$$

4. **Помилка від компресії шифротексту:** Компресія  $\mathbf{c}_1$  до  $k_2$  біт та  $\mathbf{c}_2$  до 1 біт:

$$\mathbf{err}_{\text{comp}, \mathbf{c}_1} \approx \mathbf{U}([-q/2^{k_2+1}, q/2^{k_2+1}]), \quad \mathbf{err}_{\text{comp}, \mathbf{c}_2} \approx \mathbf{U}([-q/4, q/4]).$$

**Сумарна помилка складатиме:** помилку в кожному коефіцієнті многочлена  $\mathbf{v}$ :

$$\begin{aligned} \mathbf{err} = & \mathbf{err}_{\text{comp}, \mathbf{c}_2} + \mathbf{err}_{\text{comp}, \mathbf{c}_1} \cdot \mathbf{s} \\ & + (\mathbf{err}_{\text{comp}, \mathbf{b}} \cdot \mathbf{e}_1 - \mathbf{e}_2 \cdot \mathbf{s} + \mathbf{r} \cdot \mathbf{e}_1 \cdot (p/q)). \end{aligned}$$

Помилка розшифрування виникає, коли  $|\mathbf{err}_i| \geq q/4$  для якогось коефіцієнта  $i$ , і це призводить до неправильного округлення.

## Оцінка ймовірності помилки

Для оцінки DFP/R необхідно обчислити ймовірність того, що хоча б один коефіцієнт має помилку  $\geq q/4$ .

### Статистичний аналіз:

1. **Розподіл помилок:** Кожна компонента помилки має рівномірний розподіл в заданих межах.
2. **Дисперсія:** Для розрідженого тернарного многочлена  $\mathbf{s}$  з вагою  $h_s$ :

$$\text{Var}(\mathbf{s}) \approx h_s, \quad \text{Var}(\mathbf{s} \cdot \mathbf{e}) \approx h_s \cdot h_e/3,$$

де ділення враховує коефіцієнти  $\{-1, 0, 1\}$ .

3. **Максимальна помилка складає:**

$$\sigma_{\text{err}}^2 \approx h_s \cdot h_e/3 + h_r \cdot h_e \cdot (p/q)^2 + (q/2^{k_2+1})^2 + (q/2)^2 + \dots$$

4. **DFP/R для одного коефіцієнта:** Ймовірність помилки в одному з коефіцієнтів:

$$P(\text{помилка в } i\text{-му коефіцієнті}) \approx 2 \cdot Q(q/4/\sigma_{\text{err}}),$$

де  $Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty e^{-t^2/2} dt$  – функція розподілу помилок для нормального розподілу.

5. **Тоді DFP/R для всього повідомлення:**

$$\text{DFP/R} \leq n \cdot P(\text{помилка в одному коефіцієнті}) = n \cdot 2 \cdot Q(q/4/\sigma_{\text{err}}).$$

### Різні параметри мають різний вплив на DFP/R:

1. **Модулі  $q$  та  $p$ :**

- Більше  $q$  збільшує інтервал ( $q/4$  стає більшим), зменшуючи DFP/R;
- Менше відношення  $q/p$  зменшує помилку від RLWR, але при цьому збільшує розмір відкритого ключа.

2. **Ваги Геммінга  $h_s, h_r, h_e$ :**

- Більші ваги збільшують дисперсію помилки, підвищуючи DFP/R;
- Менші ваги зменшують безпеку (легше атакувати розріджені ключі);

3. **Параметри компресії  $k_1, k_2$ :**

- Менші  $k_1, k_2$  краще зменшують розміри ключів/шифротексту, збільшуючи помилку компресії;

4. **Коди корекції помилок (XEf, D2):**

- Коди XEf можуть виправити одиночні помилки в блоках по  $d$  біт, знижуючи доволі DFP/R доволі добре (у  $d$  разів) для таких помилок;
- Код D2 з  $f = 2$  дублює кожен біт, дозволяючи виправляти помилки через мажоритарне голосування.

## Експериментальні результати

Автори TiGER провели обчислювальні експерименти для верифікації теоретичних оцінок DFP/R і отримали:

- **TiGER128:**  $DFP/R \approx 2^{-120}$  (теоретична),  $< 2^{-128}$  (експериментальна після  $2^{40}$  тестів);
- **TiGER192:**  $DFP/R \approx 2^{-136}$  (теоретична), не виявлено помилок після  $2^{45}$  тестів;
- **TiGER256:**  $DFP/R \approx 2^{-167}$  (теоретична), не виявлено помилок після  $2^{48}$  тестів.

Експерименти підтверджують, що практична DFP/R не перевищує теоретичних оцінок і є достатньо малою для будь-яких реалістичних сценаріїв використання.

## 4.7 Особливості реалізації

TiGER розроблявся з ухилом на ефективність та безпеку.

Конструкція TiGER має наступні риси, що забезпечують компактність, ефективність та безпеку реалізації.

**Компактність** досягається через використання RLWR та агресивної компресії, що дозволяє отримати малі розміри відкритого ключа та шифротексту порівняно з аналогами. Наприклад, TiGER128 має відкритий ключ розміром лише 804 байти та шифротекст 804 байти, що менше ніж у Kyber512 (800 + 768 байт) та Saber (672 + 736 байт). Економія місця особливо важлива для протоколів з обмеженою пропускну здатністю, таких як TLS, де кожен байт має значення. Також зберігання seed замість повних многочленів (як у а) економить до 2 КБ для TiGER256.

**Висока обчислювальна ефективність** забезпечується через використання модулів які є степенями двійки ( $q = 2^k$ ), що дозволяє виконувати операції модульної арифметики через прості побітові зсуви:

- Округлення  $\lfloor x \cdot (p/q) \rfloor$  виконується як побітовий зсув вправо: оскільки  $q = 2^{14}$  та  $p = 2^{10}$  для TiGER128, ділення на  $q/p = 2^4$  еквівалентне зсуву на 4 біти;
- Операції за  $\text{mod } q$  виконуються через побітовий AND з числом  $q - 1$ . Це значно швидше ніж модульні операції, що використовуються в деяких інших схемах.
- Розріджені секрети ( $h_s \ll n$ ) прискорюють множення многочленів: замість  $O(n^2)$  операцій для множення, потрібно лише  $O(h_s \cdot n)$  операцій, що дає прискорення приблизно в  $n/h_s \approx 2$  рази.

**Простота в реалізації** є важливою перевагою TiGER. Відмова від використання Number Theoretic Transform (NTT) спрощує імплементацію, оскільки NTT вимагає специфічних модулів виду  $q = 1 \bmod 2n$  та bit-reversal permutation (якийсь жах, я так не вдупив що це). TiGER використовує множення многочленів, яке, хоч і має вищу асимптотичну складність  $O(n^2)$  порівняно з  $O(n \log n)$  для NTT, але є простішим для людського сприйняття та реалізації. Для розріджених многочленів складність знижується до  $O(h \cdot n)$ , що робить це множення конкурентоспроможним. При реалізації важливо уникати умовних переходів та операцій індексації масивів, які залежать від секретних значень, оскільки це може призвести до витоку інформації через timing або cache атаки.

**Захист від атак** забезпечується кількома механізмами:

- Неявне відхилення (implicit rejection) у TiGER.KEM захищає від витоку інформації через помилки розшифрування. Це протидіє failure boosting attacks, де противник ітеративним підходом підбирає шифротексти з високою ймовірністю помилки для витягування інформації про секретний ключ. (адаптивна атака на основі вибраного шифротекста)
- Гешування відкритого ключа ( $r = G(m, H(pk))$ ) забезпечує стійкість до multi-target атак, прив'язуючи кожен шифротекст до конкретного відкритого ключа.
- Детермінована випадковість шифрування через  $G$  дозволяє виконати перевірку цілісності через повторне шифрування, що є основою IND-CCA безпеки.

**Коди корекції помилок** є ключовою інновацією в TiGER. Застосування кодів XEf та D2 значно знижує ймовірність помилки розшифрування без збільшення розміру шифротексту. Код XEf може виправити одиночні помилки в блоках по  $d$  біт та відповідно підвищує стійкість у стільки ж разів. Без кодів корекції довелося або використовувати менш агресивне стиснення (більший шифротекст), або змиритися з вищим DFP/R, що неприйнятно для практичних застосувань.

**Константний час виконання** є критичним для захисту від атак через побічні канали (side-channel attacks). Генерація розріджених многочленів через  $HWT_h$  використовує rejection sampling, при цьому загальна кількість ітерацій обмежена та не залежить від секретних даних. Операції з многочленами виконуються за фіксований час незалежно від значень коефіцієнтів. Плюс неявне відхилення гарантує, що (не)успішна декапсуляція виконуються з використанням однакової кількості геш-операцій.

**Практичні переваги** Модульна структура (PKE + FO) дозволяє окремо тестувати та оптимізувати різні компоненти. Відсутність складних математичних операцій (як pairing у криптографії на еліптичних кривих) робить TiGER перспективним для впровадження на пристроях з обмеженим ресурсом (embedded systems, IoT).

У таблиці 4.2 підсумуємо ключові технічні рішення TiGER та їх вплив на характеристики алгоритму.

Таблиця 4.2: Порівняльна таблиця покращень TiGER

Технічне рішення	Переваги	Компроміси
<b>RLWR замість RLWE</b> (відкритий ключ)	+ Менший розмір $pk$ через детерміноване округлення	- Необхідність балансування $q/p$
<b>Модулі – степені двійки</b> ( $q = 2^k, p = 2^k$ )	+ Швидкі побітові операції + Простота в реалізації + Const час виконання	- Трохи менша безпека на біт порівняно з простим $q$
<b>Розріджені секрети</b> ( $h_s, h_r, h_e \ll n$ )	+ Швидке множення: $O(h \cdot n)$ замість $O(n^2)$ + Менша дисперсія помилок	- Потенційна вразливість до комбінованих атак - Невипадковий вибір $h$ для збереження safety balance
<b>Відсутність NTT</b>	+ Свобода у виборі модулів $p, q$ + Менше ризиків для side-channel attacks	- Повільніше множення: $O(n^2)$ vs $O(n \log n)$ - Компенсується розрідженістю

Continued on next page

Таблиця 4.2: Порівняльна таблиця покращень TiGER (Continued)

Технічне рішення	Переваги	Компроміси
<b>Агресивна компресія</b> ( $k_1 = 10-11$ , $k_2 = 4-5$ )	+ Компактні розміри $pk$ і $ct$ + Ефективність у протоколах з обмеженою пропускнуою здатністю	- Збільшення помилок компресії - Вища DFP без корекції помилок
<b>Коди корекції помилок</b> (XEf + D2)	+ Дуже низька DFP ( $\leq 2^{-120}$ ) + Дозволяє агресивнішу компресію + Надійність decryption	- Невелике збільшення обчислень - Складніша логіка реалізації коду
<b>Неявне відхилення</b> (implicit rejection)	+ Захист від failure boosting атак + Константний час (успіх = невдача)	- Складніша декапсуляція
<b>Seed-based генерація</b> ( $\rho$ для $\mathbf{a}$ , $\sigma$ для $\mathbf{s}$ )	+ Малі розміри ключів (256 біт seed замість $n \log q$ біт) + Детермінованість і повторюваність	- Потреба регенерувати многочлени при використанні - Вимоги до якісного PRNG
<b><math>\text{FO}_m^f</math> перетворення</b>	+ IND-CCA безпека з IND-CPA PKE + Стандартизований підхід + Забезпечення безпеки в QROM	- Потреба повторного шифрування - Додаткові геш-обчислення
<b>Детермінована випадковість</b> ( $r = G(m, h)$ )	+ Можливість перевірки через re-encryption + Відсутність потреби у новій випадковості при Enc	- Критична залежність від стійкості самих геш-функцій - І вразливість при компрометації $G$

Як видно з таблиці, більшість технічних рішень TiGER представляють собою компроміси між різними характеристиками. Ключовою перевагою є досягнення балансу між безпекою, ефективністю та компактністю при збереженні практичності реалізації. Особливо цінною є комбінація агресивної компресії з кодами корекції помилок – це унікальність TiGER порівняно з багатьма конкурентами.

# Розділ 5

## Аналіз безпеки

### 5.1 Теоретичні основи безпеки

TiGER.PKE та TiGER.KEM забезпечують різні рівні криптографічної безпеки завдяки застосуванню добре вивчених криптографічних перетворень.

#### 5.1.1 IND-CPA безпека TiGER.PKE

Базова схема TiGER.PKE забезпечує безпеку проти атаки з використанням відкритого тексту – IND-CPA (Indistinguishability under Chosen Plaintext Attack). Формально, це означає, що противник, може обирати повідомлення та отримувати їх шифрування, але при цьому не може відрізнити шифротексти двох обраних ним повідомлень з ймовірністю суттєво більшою за  $1/2$ .

**Твердження 5.1.1** (про IND-CPA безпеку).

*У припущенні складності RLWE та RLWR проблем, TiGER.PKE є IND-CPA безпечною схемою шифрування з відкритим ключем у моделі випадкового оракула.*

*Ідея доведення:* Безпека TiGER.PKE зводиться до складності двох задач: RLWR (для відкритого ключа) та RLWE (для шифротексту). Відкритий ключ  $\mathbf{b} = \lfloor (\mathbf{a} \cdot \mathbf{s} + \mathbf{r}) \cdot (p/q) \rfloor$  є RLWR зразком, який не розкриває інформацію про  $\mathbf{s}$  за припущення складності RLWR. А шифротекст  $(\mathbf{c}_1, \mathbf{c}_2)$  складається з RLWE зразків:  $\mathbf{c}'_1 = \mathbf{a} \cdot \mathbf{e}_1 + \mathbf{e}_2$  є RLWE зразком відносно  $\mathbf{e}_1$ , а  $\mathbf{c}'_2 = \mathbf{b}' \cdot \mathbf{e}_1 + \mathbf{m} \cdot \lfloor q/2 \rfloor$  маскує повідомлення  $\mathbf{m}$  через RLWE. Зловмисник, який здатний розрізняти шифротексти, міг би вирішити RLWE/RLWR задачі, а це суперечить припущенню їх складності.

#### 5.1.2 IND-CCA безпека TiGER.KEM

TiGER.KEM досягає значно кращої безпеки. Він має стійкість до атак з виборним шифротекстом – IND-CCA (Indistinguishability under Chosen Ciphertext Attack). У цій моделі противник має доступ до оракула декапсуляції і може подавати довільні шифротексти для розшифрування (за винятком цільового шифротексту).

**Твердження 5.1.2** (про IND-CCA безпеку).

*За припущенням IND-CPA безпеки TiGER.PKE та моделювання геш-функцій  $G$ ,  $H$ , як квантових випадкових оракулів, TiGER.KEM є IND-CCA безпечною в моделі квантового випадкового оракула (QROM).*

*Ідея доведення:* Безпека досягається через перетворення Fujisaki-Okamoto з неявним відхиленням ( $\text{FO}_m^\perp$ ). Доведення використовує послідовність game-hopping (гібридних

експериментів), де в кожному наступному експерименті протокол модифікується незначним чином, і в результаті зломисник не може відрізнити сусідні експерименти. Заключний експеримент відповідає ситуації, де спільний ключ є повністю випадковим і незалежним від шифротексту.

### 5.1.3 Важливість моделі QROM

Класичні доведення безпеки FO перетворення використовують модель випадкового оракула (ROM), де геш-функції моделюються як справжні випадкові функції, доступні лише через класичні запити(?). Однак зломисник з квантовим комп'ютером може виконувати *квантові запити* до оракулів, що являють собою суперпозиції багатьох вхідів одночасно, що потенційно дає переваги в атаках.

Модель квантового випадкового оракула (QROM) враховує цю загрозу, моделюючи геш-функції як квантові оракули, доступні для квантових запитів. TiGER.KEM має формальне доведення безпеки в QROM, що підтверджує його стійкість навіть проти зломисників з доступом до квантових обчислень геш-функцій.

## 5.2 Квантова стійкість

Основна мотивація постквантової криптографії – захист від появи квантових комп'ютерів, які становлять екзистенційну і доволі реальну загрозу для сучасних криптосистем на основі розв'язання задач факторизації та дискретного логарифму.

### Квантові загрози класичній криптографії

Квантові алгоритми: Шора (1994) та Гровера (1996) радикально змінили ландшафт криптографічної безпеки:

**Алгоритм Шора** вирішує задачі факторизації цілих чисел та дискретного логарифму за поліноміальний час на квантовому комп'ютері. Для числа  $N$  складність факторизації становить  $\exp(O((\log N)^{1/3}))$  (субекспоненційна), тоді як квантовий алгоритм Шора досягає  $O((\log N)^3)$  (поліноміальна). Це означає повну компрометацію таких алгоритмів як:

- **RSA:** Квантовий комп'ютер з  $\sim 2000$  логічних кубітів може факторизувати 2048-бітне число за лічені години (трохи перебільшую, але все ж);
- **ECDSA/ECDH:** Дискретний логарифм на еліптичних кривих вирішується аналогічно ефективно;
- **Diffie-Hellman:** Класичний варіант DH також стає вразливим через полегшення обрахування дискретного логарифму у скінченних полях.

**Алгоритм Гровера** забезпечує квадратичне(!) прискорення задачі пошуку в несортованих базах даних. Для звичаного пошуку розміру  $N$ , класична складність становила  $O(N)$ , тоді як квантовий алгоритм досягає  $O(\sqrt{N})$ . Це має свій вплив на симетричну криптографію:

- **AES-128:** Ефективна безпека знижується до  $2^{64}$  операцій, що вже недостатньо;
- **SHA-256:** Collision resistance (стійкість до колізій) зменшується з  $2^{128}$  до  $2^{85}$ , а preimage resistance (стійкість до знаходження прообразу) – з  $2^{256}$  до  $2^{128}$ ;

Як наслідок - треба подвоювати розмірів ключів (AES-256, SHA-512) для збереження еквівалентного рівня безпеки.

В той же час решіткові задачі є стійкими до квантових атак. На відміну від факторизації та дискретного логарифму, задачі на решітках (включно з RLWE та RLWR) не мають відомих ефективних квантових алгоритмів.

Ключові причини цього:

1. **Відсутність прихованої підгрупової структури:** Алгоритм Шора використовує квантове перетворення Фур'є (QFT) для виявлення періодичності в підгрупах кільця  $\mathbb{Z}_N^*$  або серед точок еліптичної кривої. Решіткові задачі не мають такої регулярної алгебраїчної структури, яка б дозволяла QFT виявляти корисну інформацію.
2. **Геометрична проблема:** Задачі на решітках є геометричними проблемами – пошук найближчого вектора решітки або найкоротшого ненульового вектора. Ці задачі залишаються складними навіть для квантових алгоритмів, оскільки не існує приведення до задач з експоненційною кількістю періодичних розв'язків.
3. **Відомі квантові атаки неефективні:** Найкращі з відомих квантових алгоритми для решіткових задач це квантові версії класичних BKZ (Block Korkine-Zolotarev) та sieving алгоритмів. Вони надають лише *поліноміально-логарифмічне* прискорення (наприклад, з  $2^{0.292\beta}$  до  $2^{0.265\beta}$  для просіювання, де  $\beta$  – block dimension), що далеко від експоненційного прискорення алгоритму Шора.

## Оцінка квантової складності атак на TiGER

Для оцінки безпеки TiGER проти квантових атак використовується модель Core-SVP (Short Vector Problem), що вимірює складність вирішення SVP на решітці розмірності  $\beta$  (block dimension):

В класичній моделі обчислень найкращими класичними алгоритми (BKZ + sieving) досягається:

$$T_{\text{classical}} \approx 2^{0.292\beta + o(\beta)}$$

В квантовому всесвіті, при застосуванні квантових версії sieving алгоритмів (див. Laarhoven et al., 2015), складність досягає:

$$T_{\text{quantum}} \approx 2^{0.265\beta + o(\beta)}$$

Для TiGER параметри обиралися так, щоб при достатньо великому  $\beta$  забезпечувався бажаний рівень безпеки:

- **TiGER128:**  $\beta_{\text{quantum}} \approx 483$ , це дає  $2^{128}$  квантових операцій для атаки;
- **TiGER192:**  $\beta_{\text{quantum}} \approx 724$ , що дає  $2^{192}$  квантових операцій;
- **TiGER256:**  $\beta_{\text{quantum}} \approx 966$ , що дає  $2^{256}$  квантових операцій.

Ці оцінки консервативні і враховують можливі майбутні покращення квантових алгоритмів. Навіть за наявності велитенського квантового комп'ютера/ів з мільйонами логічних кубітів, злам того ж TiGER128 вимагав би  $2^{128}$  квантових операцій, що наразі є практично нереалістичним.

## Стійкість головного у TiGER – геш-функції

TiGER використовує геш-функції SHA3-256 та SHAKE256, які також мають бути квантово стійкими:

- **SHA3 (Кессак):** Базується на губчастій конструкції (sponge construction), яка не має відомих квантових атак, ефективніших за атаку із застосуванням алгоритму Гровера. Для SHA3-256 квантова collision resistance становить порядку  $\approx 2^{85}$  операцій, а preimage resistance –  $2^{128}$  операцій. Це є терпимим для використання у TiGER.KEM.
- **SHAKE256:** Як XOF (extendable-output function) з змінною довжиною виходу, SHAKE256 має аналогічну як і SHA3. Використання  $G$  для генерації детермінованої випадковості не створює додаткових квантових вразливостей.

## 5.3 Рівні безпеки NIST

NIST (National Institute of Standards and Technology) визначив п'ять рівнів безпеки для стандартизації постквантових алгоритмів. Ці рівні дозволяють порівнювати стійкість різних алгоритмів з існуючими класичними еталонними схемами.

### 5.3.1 Категорії безпеки згідно NIST

Рівні безпеки NIST визначаються через мінімальну обчислювальну складність зламу, еквівалентну пошуку ключа в симетричних криптосистемах або зламу класичних асиметричних схем:

Рівень	Класичний еквівалент	Квантовий еквівалент
1	Пошук ключа AES-128	$2^{128}$ класичних або $2^{64}$ квантових операцій
2	Collision пошук в SHA-256	$2^{128}$ класичних або $2^{64}$ квантових операцій (для колізій)
3	Пошук ключа AES-192	$2^{192}$ класичних або $2^{96}$ квантових операцій
4	Collision пошук в SHA-384	$2^{192}$ класичних або $2^{96}$ квантових операцій (для колізій)
5	Пошук ключа AES-256	$2^{256}$ класичних або $2^{128}$ квантових операцій

Таблиця 5.1: Рівні безпеки NIST

Насправді є 3 NIST anchor layers (1, 3, 5), а інші два – містять проміжні значення.

1. **TiGER128 (Рівень 1):** Призначений для загального використання, де потрібен баланс між безпекою та ефективністю. Перший рівень достатній для захисту більшості даних на найближчі 10-15 років, враховуючи поточний стан квантових технологій.

2. **TiGER192 (Рівень 3):** Підвищена безпека для конфіденційних даних, що вимагають довгострокового захисту (20-30 років). Квантова безпека  $2^{192}$  операцій робить атаку практично неможливою навіть за наявності значних прогресів у квантових обчисленнях.
3. **TiGER256 (Рівень 5):** Максимальна безпека для критично важливих застосувань (державні секрети, військові документи, фінансові інфраструктури і т.п.). Квантова безпека в  $2^{256}$  операцій гарантує захист навіть від гіпотетичних квантових комп'ютерів майбутнього з безпрецедентною обчислювальною потужністю, але ми ще подивимось!).

#### Ключові принципи NIST:

1. Рівні безпеки NIST визначаються за складністю атаки квантовим комп'ютером. Наприклад, рівень 1 вимагає, щоб квантова атака на алгоритм вимагала не менше ресурсів, ніж квантовий пошук ключа AES-128 (що становить  $2^{64}$  квантових операцій через алгоритм Гровера є еквівалентно  $2^{128}$  класичним).
2. NIST вимагає консервативних оцінок безпеки, що враховують можливі майбутні покращення алгоритмів для атак. Алгоритм вважається відповідним рівню безпеки, якщо найкраща відома атака (з урахуванням можливих покращень) вимагатиме не менше ресурсів, ніж еталонна задача.
3. Рівні NIST фокусуються також на практичній складності атак, враховуючи не лише теоретичну складність алгоритмів, але й реальні обчислювальні обмеження.

### 5.3.2 Відповідність TiGER стандартам NIST

TiGER має три набори (див. табл 4.1) параметрів, що відповідають відповідно трьом рівням безпеки NIST:

Характеристика	TiGER128	TiGER192	TiGER256
Рівень NIST	<b>1</b>	<b>3</b>	<b>5</b>
Класична безпека (біт)	143	207	272
Квантова безпека (біт)	128	192	256
Core-SVP $\beta$ (квантовий)	483	724	966
Складність атаки (кв. опер.)	$2^{128}$	$2^{192}$	$2^{256}$
Еквівалент NIST	AES-128	AES-192	AES-256

Таблиця 5.2: Відповідність параметрів TiGER рівням NIST

Важливою характеристикою також є *запас безпеки* – різниця між заявленим рівнем безпеки та фактичною складністю найкращої відомої атаки. TiGER має значний запас безпеки:

- **TiGER128:** Класична безпека  $\approx 143$  біт при заявлених 128 біт (запас +15 біт);
- **TiGER192:** Класична безпека  $\approx 207$  біт при заявлених 192 біт (запас +15 біт);
- **TiGER256:** Класична безпека  $\approx 272$  біт при заявлених 256 біт (запас +16 біт).

Цей запас забезпечує захист від можливих майбутніх покращень алгоритмів атак та надає впевненість у довгостроковій безпеці TiGER. Навіть якщо з'являться нові методи атак, що знижують складність на порядки  $\sim 2^{10}$  операцій, TiGER лишатиметься безпечним на заявлених рівнях.

# Розділ 6

## Порівняння з іншими алгоритмами

### 6.1 TiGER vs RLizard

RLizard є безпосереднім попередником алгоритму TiGER, розробленим тією ж командою в 2018 році. TiGER успадковує багато ідей від RLizard і вносить значні покращення, що роблять його більш компактним та ефективним.

#### Спільні риси

TiGER та RLizard мають багато спільного:

1. **Поєднання RLWE/RLWR:** Обидва алгоритми використовують RLWR для генерації відкритого ключа та RLWE для шифрування. Це дозволяє зменшити розмір відкритого ключа через округлення.
2. **Модулі у вигляді степенів двійки:** Використання модуля виду  $q = 2^k$  для оптимізації модульних операцій через побітові операції зсуву та bitwise AND.
3. **Відсутність NTT:** Обидва алгоритми не використовують Number Theoretic Transform, що спрощує програмну реалізацію, і при цьому ж зменшує ризики side-channel атак.
4. **Розріджені секрети:** Секретні ключі та помилки є тернарними многочленами з малою вагою Геммінга ( $h \ll n$ ). Це прискорює множення многочленів.
5. **FO перетворення:** Обидві схеми використовують перетворення Fujisaki-Okamoto для досягнення IND-CCA безпеки KEM з IND-CPA безпечної PKE.

#### Ключові відмінності

TiGER вносить кілька суттєвих покращень порівняно з RLizard:

1. **Зменшення степені многочлена ( $n = 1024 \rightarrow 512$ ):**
  - TiGER використовує половинний  $n$  для рівня 1, що зменшує складність множення многочленів з  $O(1024^2) \approx 10^6$  до  $O(512^2) \approx 2.6 \times 10^5$  операцій (прискорення у  $\sim 4\times$ );
  - Зменшення  $n$  також зменшує розміри всіх інших многочленів у 2 рази;
  - Безпека не на гіршому рівні за рахунок компенсації – використання кодів корекції помилок.

## 2. Зменшення модуля ( $q = 2^{20} \rightarrow 2^{14}$ ):

- Менший  $q$  означає менше біт на кожен коефіцієнт многочлена: з 20 до 14 біт;
- Як наслідок – зменшення розмірів всіх компонентів многочленів з  $R_q$ ;
- Менший  $q$  також прискорює операції модульної арифметики (менші числа).

## 3. Агресивніша компресія:

- TiGER застосовує агресивну компресію  $k_2 = 4$  біт для  $\mathbf{c}_1$  та 1 біт для  $\mathbf{c}_2$ ;
- Агресивна компресія збільшує помилки, але TiGER компенсує це знову ж таки через коди корекції помилок.

## 4. Застосування кодів корекції помилок (XEf + D2):

- Ключова інновація TiGER — використання двох методів для корекції помилок;
- Код XEf виправляє одиночні помилки в блоках по  $d = 8$  біт, збільшуючи надійність приблизно в  $d$  разів;
- Код D2 дублює кожен біт ( $f = 2$ ), дозволяючи додатково виправляти помилки через мажоритарне голосування;
- Комбінація дозволяє TiGER використовувати агресивнішу компресію при збереженні дуже низької DFP/R ( $2^{-120}$ );
- В RLizard що того, що того немає.

## 5. Неявне відхилення (implicit rejection):

- TiGER використовує  $\text{FO}_m^f$  варіант з неявним відхиленням: замість повернення  $\perp$  при невдалій декапсуляції, повертається псевдовипадковий ключ  $\bar{K} = H(z, ct)$  – захист від failure boosting атак
- RLizard використовує стандартний FO варіант з явним відхиленням, що є вразливішим до таких атак.

## 6. Підвищення безпеки при менших параметрах:

- Незважаючи на менший  $n$  та  $q$ , TiGER128 має вищий Core-SVP  $\beta = 483$  порівняно з RLizard  $\beta = 450$  – забезпечення кращої безпеки при значно менших розмірах;
- Це досягається через оптимізацію розподілу помилок, вибір ваг Геммінга, та консервативніші оцінки безпеки;

Це можна підсумувати таблицею:

Характеристика	RLizard	TiGER128
<i>Параметри</i>		
Степінь многочлена $n$	1024	512
Основний модуль $q$	$2^{20}$	$2^{14}$
Модуль округлення $p$	$2^{10}$	$2^{10}$
Вага секрету $h_s$	256	274
<i>Розміри (байт)</i>		
Відкритий ключ $ pk $	1472	804
Шифротекст $ ct $	1312	804
Секретний ключ $ sk $	1536	1876
Загалом	4320	3484
<i>Безпека та надійність</i>		
Core-SVP $\beta$	450	483
DFP/R	$2^{-128}$	$2^{-120}$
Коди корекції помилок	Немає	XEf + D2
Implicit rejection	Відсутнє	Є

Таблиця 6.1: Порівняння RLizard та TiGER

## Практичні наслідки

Покращення TiGER мають значний практичний сенс:

- **Компактність:** Зменшення  $|pk| + |ct|$  на  $\sim 35\%$  робить TiGER ефективнішим для протоколів з обмеженою пропускнуою здатністю таких як TLS чи IoT;
- **Продуктивність:** Менший  $n$  та  $q$  прискорюють усі операції: генерацію ключів, шифрування, дешифрування.
- **Надійність:** Коди корекції помилок роблять TiGER більш надійним у реальних умовах, де можливі апаратні помилки;
- **Безпечність:** Implicit rejection та вищий Core-SVP  $\beta$  роблять TiGER стійкішим до сучасних атак.

## 6.2 TiGER vs Kyber

Kyber (нині стандартизований ML-KEM) є переможцем конкурсу NIST з постквантової криптографії та де-факто слугує стандартом для KEM на основі решіток. Порівняння TiGER з Kyber дає розуміння переваг та компромісів обох підходів.

### Архітектурні відмінності

Kyber та TiGER базуються на різних варіантах решіткових задач та мають різне математичне підґрунтя:

#### 1. MLWE vs RLWE:

- *Kyber* використовує Module-LWE — узагальнення алгоритму LWE на вектори многочленів. Це компромісний варіант між стандартним LWE та RLWE;

- *TiGER* використовує чистий RLWE/RLWR на одному многочлені з кільцевої структури. Ця структура дозволяє зменшити розміри, але потенційно зменшує запас безпеки;

## 2. Модуль просте число vs степінь двійки:

- *Kyber* використовує  $q = 3329$  — спеціально обране просте число виду  $q = 1 \bmod 2n$ , що дозволяє використовувати NTT;
- *TiGER* використовує  $q = 2^{14}$ , що дозволяє швидкі побітові операції, але унеможливає NTT (оскільки  $2^{14}$  не має примітивних коренів степеня  $2n$ );

## 3. RLWR vs RLWE для відкритого ключа:

- *Kyber* генерує  $\mathbf{pk} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e}$  з явною помилкою  $\mathbf{e}$ ;
- *TiGER* використовує RLWR:  $\mathbf{pk} = \lfloor \mathbf{a} \cdot \mathbf{s} + \mathbf{r} \rfloor_{p/q}$  з округленням, що вводить неявну помилку та економить місце (але це усього порядку декількох кілобайт);

## 4. Секретні ключі:

- *Kyber* використовує Centered Binomial Distribution (CBD) з параметром  $\eta = 3$ . Коефіцієнти многочленів беруться з множини  $\{-3, -2, -1, 0, 1, 2, 3\}$ . Це дає "природний" розподіл помилок;
- *TiGER* використовує розріджені тернарні многочлени: коефіцієнти з  $\{-1, 0, 1\}$ , де рівно  $h_s = 274$  ненульових коефіцієнтів. Це прискорює множення, але розріджені секрети потенційно більш вразливі до комбінаторних атак. Необхідний консервативний вибір  $h_s$ .

Зведемо до таблиці:

Характеристика	Kyber512	TiGER128
Базова проблема	MLWE (Module-LWE) Вектори многочленів	RLWE + RLWR Один многочлен
Розмірність	$k = 2$ модулів $n = 256$ на модуль	$k = 1$ (тільки кільце) $n = 512$
Модуль $q$	3329 (просте число)	$2^{14} = 16384$ (ступінь 2)
Множення	NTT (FFT в $\mathbb{Z}_q$ ) $O(kn \log n)$	Пряме множення $O(hn)$ (розріджене)
Відкритий ключ	MLWE: $\mathbf{A} \cdot \mathbf{s} + \mathbf{e}$ Явна помилка $\mathbf{e}$	RLWR: $\lfloor \mathbf{a} \cdot \mathbf{s} + \mathbf{r} \rfloor_{p/q}$ Округлення замість помилки
Шифрування	MLWE на векторах	RLWE на многочленах
Секрети	CBD розподіл ( $\eta = 3$ ) Щільні (всі коеф. $\neq 0$ )	Розріджені тернарні Вага Геммінга $h_s = 274$
Коди корекції	Немає	XEf + D2
ФО варіант	Стандартний	Implicit rejection

Таблиця 6.2: Архітектурні відмінності Kyber та TiGER

Також порівняємо їхні характеристик згідно (NIST рівень 1)

Метрика	Kyber512	TiGER128
<i>Безпека</i>		
Квантова безпека (біт)	128	128
Core-SVP $\beta$	512	483
Класична безпека (біт)	143	143
DFP/R	$2^{-139}$	$2^{-120}$
<i>Розміри в байтах</i>		
Відкритий ключ $ pk $	800	804
Шифротекст $ ct $	768	804
Секретний ключ $ sk $	1632	1876
$ pk  +  ct $	1568	1608
<i>Продуктивність на x86</i>		
KeyGen	55K	40K
Encaps	75K	65K
Decaps	80K	70K

Таблиця 6.3: Порівняння Kyber512 та TiGER128 згідно NIST 1

Ми бачимо, що:

#### 1. По безпеці:

- Kyber512 має трохи вищий Core-SVP  $\beta = 512$  порівняно з TiGER128 ( $\beta = 483$ ). Це дає невеликий запас безпеки ( $\sim 6\%$ );
- Kyber має значно нижчу DFP/R ( $2^{-139}$  vs  $2^{-120}$ ), що робить його надійнішим.

#### 2. По пам'яті:

- TiGER128 має майже однакові розміри з Kyber512:  $|pk| + |ct| = 1608$  vs 1568 байт;
- Kyber трохи компактніший завдяки використанню модульної структури MLWE та оптимізований компресії;
- Секретний ключ TiGER більший (+15%), що є компромісом за зберігання додаткових seed для відновлення секретного ключа та ключа  $z$  для implicit rejection;

#### 3. По швидкодії:

- TiGER показує кращу продуктивність ( $\sim 10$ -27% швидше), особливо на генерації ключів. Це завдячуючи використанню побітових операцій за модулем  $2^{14}$ ;
- Kyber з NTT асимптотично швидший для доволі великих  $n$ , але для  $n = 256$  (Kyber) vs  $n = 512$  (TiGER) різниця є невеликою;

## Переваги та недоліки

### Переваги Kyber:

- **Стандартизація:** Затверджений NIST як ML-KEM;

- **Консервативність:** MLWE менш структурований за RLWE, вищий Core-SVP  $\beta$ , нижча DFP/R;
- **Досвід:** Багато оптимізаційних реалізацій (більше років "на сцені" алгоритмів);
- **NTT:** Асимптотично швидше множення для великих розмірностей (чесно, ну така собі перевага).

#### Переваги TiGER:

- **Простота:** Відсутність NTT спрощує реалізацію;
- **Ефективність:** Швидші побітові операції з модулем призводять до кращих показників на практиці для малих  $n$ ;
- **Застосування кодів коректування:** Інтеграція XEf + D2 дозволяє агресивніше стискання та підвищує надійність;
- **Implicit rejection:** Додатковий захист від failure boosting атак;

#### Недоліки TiGER:

- **Відсутність стандартизації:** Поки що затверджений NIST;
- **Нижчий  $\beta$ :** Трохи менший запас безпеки порівняно з Kyber;
- **Вища DFP/R:**  $2^{-120}$  vs  $2^{-139}$  (по факту змагання "хто ближче до нуля", бо і так це майже нуль);
- **Менш досліджений:** Менше публічних аудитів та оптимізованих реалізацій (дуже молодий алгоритм).

Kyber та TiGER представляють різну, так би мовити, філософію реалізації у постквантових KEM. Kyber обирає консервативний підхід (MLWE, вищий  $\beta$ ), а TiGER фокусується на практичній ефективності (ступені двійки, коди корекції).

## 6.3 TiGER vs Saber

Saber є фіналістом конкурсу NIST, що базується на Module-LWR (Learning With Rounding) – детермінованому варіанті MLWE. Порівняння TiGER з Saber є важливим, оскільки обидва алгоритми використовують округлення (rounding) замість використання методу явних помилок.

За концепцією TiGER та Saber є доволі схожими:

1. **LWR підхід:** Обидва використовують округлення для введення т.з. "шуму":
  - *Saber:* MLWR на векторах многочленів;
  - *TiGER:* RLWR на одному многочлені (для відкритого ключа).
2. **Модулі степенів двійки:** Обидва використовують  $q = 2^k$ , що дозволяє ефективні побітові операції для округлення та модульної арифметики.
3. **Відсутність NTT:** Жоден з алгоритмів не використовує NTT. Це спрощує реалізацію та знижує складність коду.
4. **Компактність:** Обидва досягають малих розмірів ключів порівняно з алгоритмами на основі LWE.

Ключові відмінності

Незважаючи на схожість у використанні LWR, TiGER та Saber мають і певні архітектурні відмінності:

1. **MLWR vs RLWR:**
  - *Saber* використовує чистий MLWR для всього: шифрування і відкритий ключ генерується через MLWR на векторах ( $k = 2$  для LightSaber), щось середнє між LWE та RLWE;
  - *TiGER* комбінує RLWR (для відкритого ключа) та RLWE (для шифрування). Такий гібридний підхід *TiGER* може бути складним з точки зору аналізу безпеки, але дає більше можливостей для оптимізації окремих параметрів.
2. **Модулі та точність округлень:**
  - *Saber* використовує більше біт для кодування повідомлення ( $T = 2^4$ ), що дає більший "запас" помилки, але збільшує розмір шифротексту;
  - *TiGER* компенсує меншу точність ( $T = 2$ ) використанням кодів корекції помилок.
3. **Агресивність компресії:**
  - *Saber* використовує помірну компресію з параметрами  $\varepsilon_p$  та  $\varepsilon_T$ . Це дає достатньо бітів про запас для надійного дешифрування;
  - *TiGER* застосовує дуже агресивну компресію:  $k_2 = 4$  біт для  $\mathbf{c}_1$  (з оригінальних 14 біт) та 1 біт для  $\mathbf{c}_2$  (зберігає лише знак). Така агресивна компресія *TiGER* можлива завдяки використанню корекції помилок XEf + D2.

Характеристика	LightSaber	TiGER128
Базова проблема	MLWR (Module-LWR)	RLWR + RLWE (гібрид)
Структура	Вектори многочленів $k = 2$ модулів, $n = 256$	Один многочлен $k = 1$ , $n = 512$
Модулі	$q = 2^{13}$ , $p = 2^{10}$ $T = 2^4$ (повідомлення)	$q = 2^{14}$ , $p = 2^{10}$ $T = 2^1$ (повідомлення)
Відкритий ключ	MLWR: $[\mathbf{A} \cdot \mathbf{s}]_{p/q}$ Чисте округлення	RLWR: $[\mathbf{a} \cdot \mathbf{s} + \mathbf{r}]_{p/q}$ Округлення + мала помилка
Шифрування	MLWR: $[\mathbf{A}^T \cdot \mathbf{s}']$ Округлення	RLWE: $\mathbf{a} \cdot \mathbf{e}_1 + \mathbf{e}_2$ Явні помилки
Секрети	Біноміальний розподіл $\mu = 10$	Розріджені тернарні $h_s = 274$
Коди корекції	Немає	XEf + D2
Компресія	Помірна ( $\varepsilon_p, \varepsilon_T$ )	Агресивна ( $k_2 = 4$ , 1 біт)

Таблиця 6.4: Порівняння Saber та TiGER

Порівняємо їхні характеристики згідно першого рівня NIST:

Метрика	LightSaber	TiGER128
<i>Безпека</i>		
Квантова безпека (біт)	128	128
Core-SVP $\beta$	511	483
DFP/R	$2^{-136}$	$2^{-120}$
<i>Розміри (байт)</i>		
Відкритий ключ $ pk $	672	804
Шифротекст $ ct $	736	804
Секретний ключ $ sk $	1568	1876
$ pk  +  ct $	1408	1608
<i>Особливості</i>		
Коди корекції	Немає	XEf + D2
Implicit rejection	Немає	Є
Симетричність розмірів	Ні (672/736)	Так (804/804)

Таблиця 6.5: NIST 1 порівняння LightSaber та TiGER128

Можна спостерігати, що:

- 1) LightSaber є *найкомпактнішим* серед провідних кандидатів NIST першого рівня. А TiGER на 14% більший за загальними розмірами  $|pk| + |ct|$  (1608 vs 1408 байт).
- 2) Обидва мають схожі параметри безпеки, але LightSaber має трохи вищий Core-SVP  $\beta$  (+5.5%) та значно нижчий DFP/R ( $2^{-136}$  vs  $2^{-120}$ ). Saber виграє в консервативності параметрів.
- 3) Але TiGER має унікальні переваги, такі як:
  - **Коди корекції помилок** роблять TiGER стійкішим до апаратних помилок та fault injection атак;
  - **Implicit rejection** забезпечує додатковий захист від failure boosting атак;
  - **Симетричність розмірів ключів** ( $|pk| = |ct|$ ) спрощує управління пам'яттю (при реалізації).
- 4) А "під капотом" обох алгоритмів маємо наступне:
  - *Saber* компактність і простота реалізації через застосування чистого MLWR підходу;
  - *TiGER* жертвує компактністю заради додаткових функцій безпеки та надійності.

Saber та TiGER можна назвати "родичами" сім'ї LWR алгоритмів, які мають з різні пріоритети. Для застосувань, де критична мінімізація розмірів, Saber має перевагу. Для сценаріїв з високими вимогами до відмовостійкості та захисту від fault injection вже TiGER має певні переваги.

## 6.4 TiGER vs SMAUG

SMAUG (Speedy Module-lAttice-based mUltimedia Ghost) є найближчим "родичем" для TiGER, оскільки обидва алгоритми мають спільну історію розробки та в результаті

були об'єднані в єдину специфікацію для участі у фінальному раунді конкурсу KpqC. Порівняння TiGER з SMAUG дає глибше розуміння вигоди об'єднання двох підходів.

## Історія об'єднання

TiGER та SMAUG розроблялися незалежно, але з однією ціллю – створити ефективний KEM на основі RLWE/RLWR з малими розмірами (але підвищеною безпекою) та високою продуктивністю:

- **TiGER** (2020): Розроблений як еволюція RLizard, фокус на кодах виправлення помилок та гібридному (RLWE/RLWR) підході;
- **SMAUG** (2021): Розроблений фокусуючись на модульній структурі та оптимізації для мультимедійних застосувань;
- **Об'єднання** (2022): Задля перемоги у конкурсі проекти об'єдналися, створивши єдину специфікацію "TiGER/SMAUG" та названою SMAUG-T з параметрами обох алгоритмів.

## Архітектурні схожості та відмінності

Ключові спостереження, які можна помітити:

### 1. Module vs Ring:

- *SMAUG* використовує модульну структуру ( $k = 2$ ) аналогічно до Kyber/Saber, що дає консервативніший підхід до безпеки. Цей модульний підхід дозволяє гнучкіше масштабувати безпеку через зміну  $k$ .
- *TiGER* використовує чисту кільцеву структуру ( $k = 1$ ), максимізуючи ефективність та компактність;

### 2. Коди корекції помилок:

- *SMAUG* використовує тільки XEf код для корекції одиночних помилок;
- *TiGER* додає другий рівень корекції (D2), що дублює біти для додаткової надійності;
- Подвійна корекція помилок в *TiGER* дозволяє використовувати агресивнішу компресію при збереженні низького рівня DFP/R.

### 3. Розміри модулів:

- *SMAUG* використовує менші модулі:  $q = 2^{11}$ ,  $p = 2^8$  – вимагають більшої обережності з відслідковуванням "помилки";
- В той час як *TiGER* використовує більші модулі:  $q = 2^{14}$ ,  $p = 2^{10}$ , але компенсує їх через використання чистої кільцевої структури та кодів корекції.

### 4. Цільове застосування:

- *SMAUG* оптимізований для мультимедійних застосувань та streaming, де важлива низька латентність;
- *TiGER* призначений для широкого використання, де цінується баланс між безпекою, розмірами та швидкістю.

Згребемо це все до компактного табличного вигляду:

Характеристика	SMAUG128	TiGER128
Базові алгоритми	MLWE + MLWR (гібрид на векторах)	RLWE + RLWR (гібрид на кільці)
Структура	Module: $k = 2, n = 256$	Ring: $k = 1, n = 512$
Модулі	$q = 2^{11}, p = 2^8$	$q = 2^{14}, p = 2^{10}$
Секрети	Біноміальний розподіл $\eta = 2$	Розріджені тернарні $h_s = 274$
Коди корекції	XEf	XEf + D2
Компресія	Помірна	Агресивна
Implicit rejection	Є	Є
Target usage	Мультимедіа, streaming	Загальне застосування (KEM, PKE)

Таблиця 6.6: Порівняння SMAUG та TiGER

Метрика	SMAUG128	TiGER128
<i>Безпека</i>		
Квантова безпека (біт)	128	128
Core-SVP $\beta$	492	483
DFP/R (орієнтовна)	$2^{-130}$	$2^{-120}$
<i>Розміри (байт)</i>		
Відкритий ключ $ pk $	672	804
Шифротекст $ ct $	800	804
$ pk  +  ct $	1472	1608

Таблиця 6.7: Числові характеристики SMAUG128 та TiGER128

Що можна сказати виходячи з цих цифр? Обидва алгоритми мають дуже близькі параметри безпеки (Core-SVP  $\beta$  відрізняється лише на 2%). SMAUG має трохи нижчу DFP/R завдяки консервативнішим параметрам. SMAUG є більш компактним приблизно на  $\sim 9\%$ . Це досягається через використання менших модулів та оптимізовану компресію. Також TiGER має перевагу через подвійну систему корекції помилок (XEf & D2), що робить його стійкішим до різних атак та апаратних помилок.

## Стратегія об'єднання

Об'єднана специфікація TiGER/SMAUG (SMAUG-T) включає в себе:

- **Єдина кодова база:** Спільна реалізація базових операцій (множення многочленів, гешування), а також застосування Implicit rejection для обох варіантів, уніфікованих геш-функцій (SHA3-256, SHAKE256) та спільна структура FO перетворення.
- **Акцентування на параметрах та напрямку використання:** Користувач може обирати між TiGER (з акцентом на надійність) або SMAUG (акцент на компактність) залежно від сфери застосування, оскільки TiGER: Ring-based з подвійною корекцією – для загального використання, в той час як SMAUG: Module-based з одинарною корекцією помилок – для мультимедіа.

**Переваги від цього об'єднання такі:**

1. В першу чергу це гнучкість. Користувачі можуть обирати оптимальний варіант (зміна параметрів) для конкретного цілього застосування;
2. Друга перевага – це "спільний аудит" – тобто безпека обох алгоритмів аналізується заразом, що підвищує довіру до нього;
3. По третє – економія часу розробки, бо спільна кодова база зменшує дублювання зусиль;
4. І наостанок – конкурентноспроможність. Об'єднана заявка сильніша через різноманітність у підходах.

Як підсумок вищенаведеного, SMAUG та TiGER є комплементарними підходами в рамках родини RLWE/RLWR алгоритмів. Об'єднання двох алгоритмів створює гнучку екосистему, що покриває широкий спектр вимог постквантової криптографії.

## 6.5 Порівняння усіх перелічених алгоритмів

Тут вже зведемо результати і зробимо комплексне порівняння TiGER з провідними постквантовими КЕМ алгоритмами за ключовими характеристиками: безпека, розміри, продуктивність та особливості реалізації, ну загалом як і робили до того.

### Порівняння безпеки та розмірів:

Всі дані взято для рівня безпеки NIST 1 (128 біт квантової безпеки).

1. Усі алгоритми забезпечують 128-бітну квантову безпеку (рівень NIST 1), але з різними значеннями Core-SVP  $\beta$  (block size):
  - *Kyber512* та *LightSaber* мають  $\beta \approx 510$ . Це дає невеликий запас порівняно з пороговим  $\beta = 512$  для 128-бітної безпеки;
  - *TiGER128* має  $\beta = 483$ , що є нижчим за еталонне значення, але це компенсується більш консервативними оцінками та використанням D2, XEf і implicit rejection.
2. *TiGER128* має конкурентні розміри у порівнянні з іншими та володіє унікальною властивістю – однакові розміри  $|pk|$  та  $|ct|$  (обидва 804 байт):
  - Відкритий ключ (804 байт) близький до *Kyber512* (800 байт) і трохи більший за *LightSaber* (672 байт);
  - Шифротекст (804 байт) трохи більший за *Kyber512* (768 байт) та *LightSaber* (736 байт);

Алгоритм	Core-SVP $\beta$	$ pk $ (байт)	$ ct $ (байт)
<b>Kyber512</b>	512	800	768
<b>LightSaber</b>	511	672	736
<b>TiGER128</b>	483	804	804

Таблиця 6.8: Порівняння постквантових КЕМ (рівень NIST 1)

Для вищих рівнів безпеки (NIST 3 та NIST 5) TiGER також демонструє конкурентно спроможні характеристики:

Рівень	Алгоритм	$ pk  +  ct $ (байт)	Відносна компактність
<b>NIST 1</b>	Kyber512	1568	baseline(certificated)
	LightSaber	1408	-10%
	TiGER128	1608	+2.5%
<b>NIST 3</b>	Kyber768	2400	baseline(certificated)
	Saber	2304	-4%
	TiGER192	2976	+24%
<b>NIST 5</b>	Kyber1024	3168	baseline(certificated)
	FireSaber	3040	-4%
	TiGER256	3168	same

Таблиця 6.9: Порівняння розмірів для різних рівнів NIST

Можна бачити що: TiGER128 та TiGER256 мають конкурентні розміри порівняно з Kyber та Saber. TiGER192 має помітно більші розміри (+24% порівняно з Kyber768). Це компроміс з вищою надійністю дешифрування (визначається по показнику DFP/R);

## Узагальнююче порівняння характеристик

Характеристика	Kyber512	LightSaber	TiGER128	RLizard
<i>Параметри безпеки</i>				
Core-SVP $\beta$	512	511	483	450
DFP/R	$2^{-139}$	$2^{-136}$	$2^{-120}$	$2^{-128}$
<i>Розміри (байт)</i>				
$ pk $	800	672	804	1472
$ ct $	768	736	804	1312
$ sk $	1632	1568	1876	1536
$\Sigma$	3200	2976	3484	4320
<i>Особливості</i>				
Базова проблема	MLWE	MLWR	RLWE+RLWR	RLWE+RLWR
NTT	Так	Ні	Ні	Ні
Error correction codes	Ні	Ні	XEf+D2	Ні
Модулі	Простий	$2^k$	$2^k$	$2^k$
Статус	Standard NIST	Фіналіст	Кандидат	Попередник

Таблиця 6.10: Комплексне порівняння постквантових KEM (NIST 1)

З таблиці можна бачити, що Kyber є затвердженим NIST як ML-KEM стандарт, що робить його пріоритетним вибором для більшості застосувань. А TiGER є єдиним алгоритмом з інтегрованими кодами корекції помилок (XEf+D2), він також демонструє конкурентні розміри, близькі до Kyber та Saber, і значно кращі за RLizard та залишається поки лиш цікавою альтернативою з унікальними характеристиками у задачах, де наведені його параметри становитимуть вагоме критичне значення.

# Розділ 7

## Аналіз атак на TiGER

### 7.1 Аналіз слабких місць алгоритму

У цьому розділі ми висвітлимо потенційно слабкі місця алгоритму TiGER та оцінимо їх вплив на експлуатацію загалом. Хоч TiGER базується на добре вивчених задачах RLWE/RLWR, його окремі реалізації мають свої певні особливості.

#### 1. Атаки на базову задачу RLWE/RLWR

Решіткові атаки а.k.a. Lattice reduction attack є основним класом атак на RLWE/RLWR, що використовують алгоритми BKZ (Block Korkine-Zolotarev) для пошуку shortest вектору у решітці (SVP) та вектору решітки, найближчого до заданої точки (CVP).

##### Побудова решітки:

Маючи  $m$  зразків (пар) RLWE ( $\mathbf{a}_i, \mathbf{b}_i = \mathbf{a}_i \cdot \mathbf{s} + \mathbf{e}_i$ ), будуємо решітку розмірності:

$$d = n(m + 1) = 512 \cdot 2 = 1024 \quad \left. \vphantom{\begin{matrix} d \\ n \\ m \end{matrix}} \right\} \text{для TiGER128 } m = 1$$

##### Цільовий вектор:

Для розріджених тернарних секретів з вагою Геммінга  $h_s = 274$ , норма складає:

$$\|\mathbf{s}\| = \sqrt{h_s} = \sqrt{274} \approx 16.55$$

Комбінований вектор помилок і секрету:

$$\|\mathbf{v}\| = \sqrt{h_e + h_s} = \sqrt{274 + 274} = \sqrt{548} \approx 23.41$$

##### Необхідний розмір блоку BKZ:

BKZ (Block Korkine-Zolotarev) [19] є алгоритмом редукції для базису решітки, який послідовно покращує базис, роблячи вектори коротшими. Параметр  $\beta$  (розмір блоку) визначає якість редукції: більший  $\beta$  знаходить коротші вектори, але зі значно більшою обчислювальною складністю.

Згідно до Core-SVP,  $\beta = 483$  є порогом необхідним для здійснення успішної атаки на TiGER128 [20].

##### Складність:

Складність BKZ оцінюється за формулами [21]:

$$\text{Класична: } 2^{0.292\beta + 16.4} = 2^{0.292 \cdot 483 + 16.4} \approx 2^{157} \text{ операцій}$$

$$\text{Квантова: } 2^{0.265\beta + 16.4} = 2^{0.265 \cdot 483 + 16.4} \approx 2^{144} \text{ операцій}$$

Параметр	Значення
Розмірність решітки $d$	1024
Core-SVP $\beta$	483
Класична складність	$\approx 2^{157}$ операцій
Квантова складність	$\approx 2^{144}$ операцій
Класична безпека	$\approx 143$ біт
Квантова безпека (NIST 1)	128 біт

Таблиця 7.1: Складність решіткової атаки на TiGER128

Як можна бачити, решіткові атаки на RLWE/RLWR, яка лежить в основі TiGER, є практично нездійсненними.

## 2. Комбінаторні атаки на розріджені секрети

TiGER використовує розріджені тернарні секрети ( $h_s = 274$  ненульових коефіцієнтів з загальної кількості  $n = 512$ ). Це потенційно зменшує ентропію порівняно зі щільними секретами (такі, які використовуються в Kyber512).

### Простір ключів $\mathcal{S}$ :

Кількість можливих секретних ключів обчислюється як:

$$|\mathcal{S}| = \binom{n}{h_s} \cdot 2^{h_s}$$

де  $C_{h_s}^n$  – комбінація (вибірка позицій ненульових коефіцієнтів), а  $2^{h_s}$  – вибір знаку ( $\pm 1$ ).

Для нашого TiGER128 маємо:

$$|\mathcal{S}| = \binom{512}{274} \cdot 2^{274}$$

Використавши апроксимацію Стірлінга для  $n!$  маємо:

$$\binom{n}{k} \approx \frac{2^{nH(k/n)}}{\sqrt{2\pi k(1-k/n)}}$$

де  $H(p) = -p \log_2 p - (1-p) \log_2 (1-p)$  – бінарна ентропія (binary entropy function).

Ймовірність дорівнює:  $p = h_s/n = 274/512 \approx 0.535$ , тоді

$$H(0.535) = -0.535 \log_2 (0.535) - 0.465 \log_2 (0.465) \approx 0.999$$

Логарифмічна оцінка довжини ключа складає:

$$\log_2 |\mathcal{S}| \approx 512 \cdot 0.999 + 274 = 511.5 + 274 = 785.5 \text{ біт}$$

Якщо здійснювати атаку перебором (brutforce), навіть з застосуванням квантового прискорення Гровера отримуємо:

$$\text{Complexity}_{\text{Grover}} = 2^{785.5/2} = 2^{392.75} \gg 2^{128}$$

Як підсумок, простір ключів є достатньо великим, щоб бути стійким до атак перебором.

### 3. Атаки через помилки округлення (RLWR specific)

RLWR вводить детерміновану помилку з використанням округлення (див. розділ 3.2.5):

$$\mathbf{e}_{\text{rounding}} = \mathbf{a} \cdot \mathbf{s} - \lfloor \mathbf{a} \cdot \mathbf{s} \rfloor_{p/q} \cdot \frac{q}{p}$$

Кожен коефіцієнт  $\mathbf{e} \in [-\frac{q}{2p}, \frac{q}{2p}]$ . Для TiGER:

$$\|\mathbf{e}_{\text{rounding}}\|_{\infty} \leq \frac{2^{14}}{2 \cdot 2^{10}} = 8$$

Риторичне питання: Чи може детермінованість може давати додаткову інформацію зловмиснику? Округлення є відомою для всіх функцією від секретного  $\mathbf{s}$  та публічного  $\mathbf{a}$ . Проте, без знання  $\mathbf{s}$ , розподіл помилки залишається рівномірним і не дає практичної переваги атакуючому. Редукції з RLWR до RLWE [22] показують, що RLWR не слабший за RLWE при правильному виборі параметрів (див. твердження 3.2.1). Отже, використання такого округлення не створює практичної вразливості.

### 4. Атаки на кільцеву структуру алгоритму

Кільце  $R = \mathbb{Z}[x]/(x^{512} + 1)$  має спеціальну структуру. Чи можна її якось експлуатувати, щоб здійснити атаку?

Існують так звані "підкільцеві атаки" (Subfield lattice attacks). Ці атаки працюють, коли кільце має нетривіальні підкільця, що дозволяє редукувати (зменшувати) розмірність решітки. Для циклотомічного кільця  $\mathbb{Z}[x]/(\Phi_m(x))$ , проблеми виникають у тому випадку, якщо  $m$  має багато дільників. Для TiGER,  $m = 2 \cdot 512 = 1024 = 2^{10}$ . Хоча  $m$  має велику кількість дільників, проте для степенів двійки відомі підкільцеві атаки [23] потребують експоненційної кількості зразків або мають не кращу складність за звичайні решіткові атаки при  $n \geq 512$ . Тобто використання кільцевої структури не дає практичної переваги для атаки на TiGER.

### 5. Side-channel вразливості

Є три основні підвиди атак на бічні канали (детальніша класифікація у розділі 7.7):

#### Timing атаки:

- **Помножень на розріджені многочлени:** Множення на многочлен  $\mathbf{s}$  займає час  $O(h_s \cdot n)$ , який може варіюватися, як наслідок – потенційний витік часу виконання.  
*Захист:* є дуже банальним – константний час реалізації через використання ключів однакової довжини або dummy операцій (??).
- **Rejection sampling:** Генерація секретів з точною вагою  $h_s$  через rejection може привести до витоку інформації про кількість ітерацій.  
*Захист:* Використання seed expansion з постійною кількістю спроб, незалежно від успіху. (отут прикола я не поняв)

#### Power analysis атаки:

Споживання енергії під час множення многочленів може корелюватися з секретними даними.

Як захист застосовується маскування (masking) першого та другого порядку – тобто проведення розділення секретних даних на випадкові частки.

### Fault injection атаки:

Індукування помилок під час дешифрування може викрити інформацію про секретний ключ через failure boosting.

Використання implicit rejection у TiGER ( $FO_m^L$ ) слугує захистом, оскільки при помилці декапсуляції повертається псевдовипадковий ключ замість  $\perp$ , що унеможливає відрізнення успішної декапсуляції від невдалої.

Висновок: Side-channel атаки є реальною загрозою, але TiGER має вбудований захисний механізм implicit rejection, а також константна реалізація та маскування є необхідним для забезпечення повного захисту.

### Підсумкова таблиця по слабких місцях TiGER

Тип атаки	Складність	Рівень загрози	Захист
BKZ	$2^{144}$ (квант.)	Безпечно	Достатнє значення $\beta$
Brutforce	$2^{393}$ (Grover)	Безпечно	Large key space
RLWR specific	?	Безпечно	Редукція до RLWE
Subfield	$\geq 2^{144}$	Безпечно	$m$ має мало дільників
Timing	Варіюється	Помірний	Const-time implementation
Power analysis	Варіюється	Помірний	Mask using
Fault injection	Варіюється	Безпечно	Implicit rejection

Таблиця 7.2: Слабкі місця TiGER

**Загальний висновок:** Як бачимо, TiGER не має критичних математичних вразливостей. Основні його ризики пов'язані з side-channel атаками, які потребують обережної реалізації. Його "Cryptographic Core Architecture" є стійкою до всіх відомих теоретичних атак при заданих оптимальних параметрах.

## 7.2 Можливі вразливості через DFP/R

Decapsulation Failure Probability (DFP) та Decryption Failure Rate (DFR) є критичними параметрами для KEM (Key Encapsulation Mechanism) на основі решіток. Навіть дуже мала ймовірність помилки дешифрування може бути використана зломисником для витягування інформації про секретний ключ. Проаналізуємо вразливості через DFP/R та існуючі механізми захисту в TiGER.

DFP – це ймовірність того, що правильно сформований шифротекст декапсулюється некоректно, тобто:

$$DFP = \Pr[\text{Decaps}(sk, \text{Encaps}(pk)) \neq K]$$

Ця помилка виникає, коли сума помилок від шифрування та компресії перевищує границю, яку зазвичай виправляє код корекції. Низька ймовірність DFP є критичною, оскільки дуже мала DFP (від  $2^{-30}$  до  $2^{-100}$ ) може бути експлуатована зломисником при *failure boosting* атаці для витягування секретного ключа.

### Джерела помилок у TiGER та оцінка DFP для TiGER128

У алгоритмі TiGER помилки накопичуються з кількох джерел:

1. **Помилки при шифруванні.** В процесі шифрування генеруються навмисні помилки  $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$ :

$$\mathbf{c}_1 = \mathbf{a} \cdot \mathbf{e}_1 + \mathbf{e}_2, \quad \mathbf{c}_2 = \mathbf{b} \cdot \mathbf{e}_1 + \mathbf{e}_3 + \mathbf{Encode}(\mu)$$

Кожна помилка має свою вагу Геммінга  $h_e$ . Для TiGER128 вони всі однакові:  $h_{e_1} = h_{e_2} = h_{e_3} = 274$ .

2. **Помилки округлення відкритого ключа.** Нагадаємо, що публічний ключ має вигляд:

$$\mathbf{b} = \lfloor \mathbf{a} \cdot \mathbf{s} + \mathbf{r} \rfloor_{p/q}$$

Округлення вносить неявну помилку:

$$\mathbf{e}_{\text{round}} = (\mathbf{a} \cdot \mathbf{s} + \mathbf{r}) - \mathbf{b} \cdot \frac{q}{p}$$

Для TiGER128:  $\|\mathbf{e}_{\text{round}}\|_{\infty} \leq \frac{q}{2p} = 8$ .

3. **Помилки компресії:** Шифротекст зазнає стиснення з 14 біт до  $k_2 = 4$  біт для  $\mathbf{c}_1$  та до  $k_3 = 1$  біт (лишається лише знак) для  $\mathbf{c}_2$ :

$$\mathbf{e}_{\text{comp},1} = \mathbf{c}_1 - \text{Decompress}(\text{Compress}(\mathbf{c}_1, k_2), k_2)$$

TiGER використовує рівномірну компресію (uniform compression), розділяючи на рівні інтервали. Довжина кожного інтервалу  $q/2^k$ . Тому помилка компресії має таке верхнє обмеження:

$$\|\mathbf{e}_{\text{comp}}\|_{\infty} \leq \frac{q}{2^{k+1}}$$

Чисельні значення є наступними:

$$\begin{aligned} \|\mathbf{e}_{\text{comp},1}\|_{\infty} &\leq \frac{2^{14}}{2^{4+1}} = \frac{16384}{32} = 512 \\ \|\mathbf{e}_{\text{comp},2}\|_{\infty} &\leq \frac{2^{14}}{2^{1+1}} = \frac{16384}{4} = 4096 \end{aligned}$$

4. **Сумарна помилка при дешифруванні:** При дешифруванні обчислюється вектор  $\mathbf{v}$ :

$$\mathbf{v} = \mathbf{c}_2 - \mathbf{c}_1 \cdot \mathbf{s}$$

Сумарна помилка при дешифруванні складатиме:

$$\mathbf{e}_{\text{total}} = \mathbf{r} \cdot \mathbf{e}_1 + \mathbf{e}_{\text{round}} \cdot \mathbf{e}_1 - \mathbf{e}_2 \cdot \mathbf{s} + \mathbf{e}_3 + \mathbf{e}_{\text{comp},1} + \mathbf{e}_{\text{comp},2}$$

Кожен член суми  $\mathbf{e}_{\text{total}}$  має свій розподіл. Найбільший внесок має  $\mathbf{e}_{\text{comp},2}$ . В TiGER використовуються коди корекції помилок (XEf + D2), які можуть виправити обмежену кількість помилок. Нехай цей поріг виправлення складає  $\tau$  помилкових біт на один блок.

Припустимо, що після застосування кодів корекції, DFP на один біт (ймовірність помилки для одного біта) дорівнює  $p_{\text{bit}}$ . Для повідомлення довжиною в 256 біт це становитиме:

$$\text{DFP} \approx 1 - (1 - p_{\text{bit}})^{256} \approx 256 \cdot p_{\text{bit}} \quad (\text{якщо } p_{\text{bit}} \ll 1)$$

З специфікації TiGER (заявлена DFP):  $\text{DFP}_{\text{TiGER128}} = 2^{-120}$ . Це означає буде 1 failure на  $2^{120}$  спроб декапсуляції.

Алгоритм	DFP
Kyber512	$2^{-139}$
Saber (LightSaber)	$2^{-136}$
TiGER128	$2^{-120}$
SMAUG128	$2^{-130}$

Таблиця 7.3: Порівняння DFP різних постквантових КЕМ для рівня NIST 1

TiGER має трохи вищу DFP порівняно з Kyber/Saber, але все ще достатньо низьку для практичного використання.

## Failure boosting attack

Загальна концепція класичного варіанту атаки полягає в тому, що атакуючий може ітеративно створювати спеціально підібрані шифротексти з підвищеною ймовірністю failure (замість  $2^{-120}$ , досягає  $2^{-10}$  або більше) та спостерігає за відповіддю алгоритму, чи відбудеться помилка. За достатньої кількості спостережень можна витягнути інформацію про секретний ключ [24], а з використанням статистичних методів реконструювати і коефіцієнти секретного ключа.

Для здобуття (extraction) одного біту інформації про секретний ключ  $s$  потрібно  $\approx 1/\text{DFP}_{\text{boosted}}$  запитів. Якщо атакуючий може досягти  $\text{DFP}_{\text{boosted}} = 2^{-10}$ , то для витягання 256 бітного ключа необхідно здійснити:

$$\text{Queries} \approx 256 \cdot 2^{10} = 2^{18} \approx 262,000 \text{ запитів.}$$

Тобто маємо практично здійснену атаку! Якщо використовується стандартний FO з explicit rejection (див. алгоритм 3.6.1), то зловмисник в результаті бачить  $\perp$ , він знає, що відбулася помилка декапсуляції.

Який захист від цього має TiGER? Він використовує варіант Fujisaki-Okamoto з *implicit rejection* (див. алгоритм 3.6.2). Ефект від *implicit rejection* є наступним:

- Атакуючий завжди отримує деякий ключ  $K$  ( $\bar{K}$  або  $K'$ ), незалежно від того, була помилка чи ні.  $K'$  (при успіху) та  $\bar{K}$  (при помилці) – обидва виглядають випадковими;
- Атакуючий не може відрізнити успішну декапсуляцію від невдалої без знання секретного  $z$  і failure boosting атака стає неможливою, оскільки немає спостережувального сигналу про failure.

*Implicit rejection* забезпечує IND-CCA безпеку навіть за наявності ненульової DFP. Навіть якщо декапсуляція іноді помиляється, атакуючий не може це відстежити та використати проти системи. Показник DFP для TiGER складає  $2^{-120}$ , і є достатньо низьким для практичного використання. Але критично важливим є те, що TiGER використовує *implicit rejection*, що робить failure boosting атаки практично неможливими. За умови коректної константної реалізації ці вразливості через DFP у алгоритмі TiGER можуть бути належним чином мітиговані.

## 7.3 Meet-LWE атака

Meet-in-the-Middle (MitM) атаки є потужним криптоаналітичним інструментом та "грозою" для багатьох криптопримітивів. У цій секції детально проаналізуємо цю

атаку та її застосовність до TIGER.

## Концепція Meet-in-the-Middle

Почнемо одразу з класичного прикладу застосування – атака на подвійне шифрування.

Нехай  $E_k(m)$  – шифрування повідомлення  $m$  ключем  $k$ . Подвійне шифрування:  $c = E_{k_2}(E_{k_1}(m))$ . Наївна атака (перебір) складатиме  $2^{2n}$  операцій для ключів довжини  $n$  біт кожен, в той час як MitM атака:

### Algorithm 15 Meet-in-the-Middle атака на подвійне шифрування

```
1: Input: Відоме  $(m, c)$ , де  $c = E_{k_2}(E_{k_1}(m))$ 
2: Output: Ключі  $(k_1, k_2)$ 
3:
4: // Фаза 1: Forward (побудова таблиці)
5: Ініціалізувати геш-таблицю  $T$ 
6: for  $k_1 = 0$  to  $2^n - 1$  do
7:    $v \leftarrow E_{k_1}(m)$ 
8:    $T[v] \leftarrow k_1$  ▷ Зберігаємо пару (значення, ключ)
9: end for
10:
11: // Фаза 2: Backward (пошук колізії)
12: for  $k_2 = 0$  to  $2^n - 1$  do
13:    $v \leftarrow D_{k_2}(c)$ 
14:   if  $v \in T$  then ▷ Перевірка наявності в таблиці
15:      $k_1 \leftarrow T[v]$ 
16:     if  $E_{k_2}(E_{k_1}(m)) = c$  then ▷ Верифікація
17:       return  $(k_1, k_2)$ 
18:     end if
19:   end if
20: end for
21: return Failure
```

По складності маємо оцінку:  $2^n$  час +  $2^n$  пам'ять замість  $2^{2n}$  лише часу(!).

## Meet-LWE атака: основна ідея

Alexander May розробив спеціалізований варіант MitM для задачі LWE, відомий як Meet-LWE атака [25].

Для задачі LWE маємо:

$$\mathbf{b} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e} \pmod{q}$$

де  $\mathbf{s} \in \mathbb{Z}_q^n$  – секрет,  $\mathbf{e}$  – помилка. Секрет, в свою чергу, розділений на дві частини:

$$\mathbf{s} = (\mathbf{s}_1, \mathbf{s}_2) \quad \text{де } \mathbf{s}_1 \in \mathbb{Z}_q^{n_1}, \mathbf{s}_2 \in \mathbb{Z}_q^{n_2}, \quad n_1 + n_2 = n$$

### Складність атаки на TIGER128:

Для розріджених секретів TiGER з  $h_s = 274$ , оптимізована версія Meet-LWE має складність:

Complexity<sub>classical</sub> ≈ 2<sup>0.8·274</sup> ≈ 2<sup>219</sup>

Complexity<sub>quantum</sub> ≈ 2<sup>109.5</sup>

Тип атаки	Класична	Квантова
Brutforce	2 <sup>785</sup>	2 <sup>392</sup>
BKZ	2 <sup>157</sup>	2 <sup>144</sup>
Meet-LWE (оптим.)	2 <sup>219</sup>	2 <sup>109.5</sup>
Цільовий рівень	2 <sup>143</sup>	2 <sup>128</sup>

Таблиця 7.4: Порівняння складності атак на TiGER128

Як бачимо, Meet-LWE має квантову складність 2<sup>109.5</sup>, що є нижчою цільового рівня, але з урахуванням практичних обмежень на ресурси загроза є обмеженою. TiGER192/256 мають складність > 2<sup>190</sup> (тут все добре).

## 7.4 ССА атаки на РКЕ

Атаки з вибраним шифротекстом а.к.а. ССА (Chosen Ciphertext Attack) є одними з найнебезпечніших атак на криптосистеми. В 2023 році von Berg та його колеги опублікували роботу, що аналізує ССА атаку на РКЕ схеми на основі решіток, включаючи варіанти наближені до TiGER. Розглянемо нижче ці атаки і як від них захиститися.

### ССА безпека

Завдання атакуючого – відрізнити шифротекст реального повідомлення від шифротексту випадкового повідомлення. Сама атака на основі вибраного шифротексту відбувається за рахунок того, що зловмисник має доступ до:

1. **Оракула шифрування:** – може отримувати шифротексти довільних повідомлень;
2. **Оракула дешифрування:** – може дешифрувати довільні шифротексти (крім цільового).

Існують три рівні безпеки (залежно від наявних засобів у зловмисника):

- **IND-CPA** (Indistinguishability under Chosen Plaintext Attack): Атакуючий має доступ лише до оракула шифрування;
- **IND-CCA1** (Non-adaptive CCA): Має обмежений доступ до оракула дешифрування (зазвичай до отримання цільового шифротексту);
- **IND-CCA2** (Adaptive CCA): Має необмежений доступ до оракула дешифрування, але не може з його допомогою дешифрувати сам цільовий ШТ.

Ця безпека є дуже важливою, бо у таких протоколах як TLS або SSH атакуючий часто може "підсунути" серверу модифіковані певним чином шифротексти та спостерігати за його реакцією (executing time, error messages). Це дає можливість ССА атак.

## Базова РКЕ схема TiGER та її CPA безпека

TiGER.РКЕ (базова схема без FO) була розглянута у розділі 4.4. Також навели твердження 5.1.1, яке стосується оцінки безпеки.

Класична ССА атака (атака через модифікацію шифротексту) на стандартний РКЕ виглядає так:

Нехай атакуючий має цільовий шифротекст  $(ct_1, ct_2)$  для повідомлення  $\mu$  та хоче дізнатися власне  $\mu$ , маючи доступ до оракула дешифрування.

1. **Крок 1:** Створює модифікований шифротекст:

$$ct'_2 = ct_2 + \Delta,$$

де  $\Delta$  – невелике відхилення.

2. **Крок 2:** Подає  $(ct_1, ct'_2)$  до оракула дешифрування і отримує  $\mu'$ .
3. **Крок 3:** Маючи  $\mu'$  та знаючи  $\Delta$ , можна витягнути інформацію про  $\mu$  або навіть про секретний ключ  $\mathbf{s}$ .

### Приклад конкретної атаки (bit-flipping attack):

Якщо  $\Delta$  обрано так, що вона впливає лише на один біт декодованого повідомлення, то зломисник може біт-за-бітом витягувати  $\mu$ :

$$\mu'_i = \mu_i \oplus f(\Delta_i, \mathbf{s}), \quad \text{де } f - \text{деяка функція.}$$

Спостерігаючи за зміною  $\mu'_i$  для різних  $\Delta_i$ , можна реконструювати всі біти оригінального повідомлення  $\mu$ .

## Атака von Berg на РКЕ з частковим декодуванням

У роботі Каспер фон Берга [26] запропоновано більш витончену атаку, що експлуатує проміжний стан під час дешифрування.

Сценарій атаки полягає в наступному:

Припустимо, що атакуючий може спостерігати не лише фінальне декодоване повідомлення  $\mu'$  (після застосування кодів корекції), але й проміжний результат  $\mathbf{v}_{\text{raw}}$  до застосування кодів корекції помилок:

$$\mathbf{v}_{\text{raw}} = \mathbf{c}_2 - \mathbf{c}_1 \cdot \mathbf{s}$$

Саме значення  $\mathbf{v}_{\text{raw}}$  містить закодоване оригінальне повідомлення  $\mu$  і плюс всі накопичені помилки (від RLWE, RLWR та компресії), тобто є лінійною комбінацією коефіцієнтів секретного ключа  $\mathbf{s}$ .

### Ідея атаки:

1. Зломисник створює шифротекст  $(ct_1, ct_2)$  зі спеціально підібраними параметрами  $(\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3)$ ;
2. Спостерігає  $\mathbf{v}_{\text{raw}}$  через power analysis (різновидність side-channel атаки);
3. Збираючи багато таких спостережень з різними  $ct_1$ , будує систему лінійних рівнянь:

$$\mathbf{V} = \mathbf{C}_1 \cdot \mathbf{s} + \mathbf{E},$$

де  $\mathbf{V}$  – вектор спостережень  $\mathbf{v}_{\text{raw}}$ ,  $\mathbf{E}$  – вектор помилок;

4. Розв'язує систему з урахуванням шуму для знаходження  $\mathbf{s}$  (використовуючи LWE-solving methods).

#### Складність атаки:

За оцінками авторів, для витягування повного  $\mathbf{s}$  потрібно:

$$\text{Queries} \approx O(n \cdot \log q) \approx 512 \cdot 14 = 7168$$

запитів до оракула "часткового дешифрування" (IND-CCA1).

Ця атака потребує доступу до  $\mathbf{v}_{\text{raw}}$  і до кодів корекції. У реалізації TiGER дешифрування повертає лише фінальне  $\mu$  після всіх перетворень (проміжні значення не витікають назовні), навіть і якщо є side-channel attacks (timing, power), витягнути саме  $\mathbf{v}_{\text{raw}}$  дуже складно.

### Захист через Fujisaki-Okamoto перетворення

TiGER використовує не базову PKE, а КЕМ з перетворенням Fujisaki-Okamoto (FO), яке перетворює IND-CPA безпечну PKE в IND-CCA2 безпечний КЕМ (див. розділ 3.6.3). Це перетворення захищає від CCA атак, бо:

1. **Re-encryption check:** Дешифроване  $\mu'$  використовується для повторного шифрування. Якщо результат не збігається з  $ct$ , значить  $ct$  був модифікований атакуючим;
2. **Детермінованість:** Шифрування використовує  $H'(\mu)$  як seed для помилок, тому одне  $\mu$  завжди видає той самий  $ct$  (для даного  $pk$ ). Ключ  $K = H(\mu)$  повністю визначається повідомленням, тому модифікація  $ct$  дає інший  $\mu'$  та, відповідно, інший  $K'$ , що виглядає як випадкове значення для атакуючого.
3. **Implicit rejection:** Навіть якщо перевірка провалилася, атакуючий не дізнається про це, оскільки отримує псевдовипадковий  $\tilde{K}$ ;

Існує доведення [18], що  $\text{FO}_m^\perp$  перетворення забезпечує IND-CCA2 безпеку за умов:

- Базова PKE є IND-CPA безпечною;
- Геш-функції  $H, H'$  моделюються як випадкові оракули (ROM);
- DFP/R схеми є достатньо малим значенням.

Для TiGER всі ці умови виконуються і TiGER КЕМ з  $\text{FO}_m^\perp$  перетворенням є IND-CCA2 безпечним за умови коректної його реалізації. Атака von Berg (повна назва: CCA attack on PKE with access to intermediate decryption) на базову PKE не застосовна до повної схеми КЕМ. Основні ризики пов'язані з side-channel атаками, які потребують обережної імплементації з константним часом та маскуванням. Незважаючи на застосування модифікованого FO перетворення, деякі практичні атаки все ще можливі:

Вектор атаки	Підґрунтя атаки	Захист у TiGER
Модифікація $ct$	Відсутність перевірки цілісності	Re-encryption check
Failure boosting	Помилки декапсуляції	Implicit rejection
Timing side-channel	Різний час обробки для різних $ct$	Const-time реалізація
Fault injection	Індукування помилок під час дешифрування	Використання кодів корекції, $FO_m^\perp$ перетворення
Leak $\mathbf{v}_{\text{raw}}$	Power/EM side-channel	Masking

Таблиця 7.5: ССА вектори атак на TiGER KEM

Як бачимо по таблиці, алгоритм математично доволі добре захищений від атак, інше – залежить від програмної реалізації.

## 7.5 Атака з доступом до проміжного виводу декодування

Ще доцільно поговорити про специфічний клас атак, коли противник може спостерігати або маніпулювати проміжними значеннями під час процесу дешифрування. У розділі 7.4 ми вже поверхнево зачіпали і наводили приклад такої атаки (von Berg). Також приділимо увагу унікальній структурі TiGER, що використовує коди корекції помилок XEf та D2.

### Етапи дешифрування в TiGER

Визначимо спершу позначення:

- $\mu$  – оригінальне повідомлення (зашифроване відправником)
- $\tilde{\mu}$  – необроблені біти після екстракції (можуть містити помилки)
- $\hat{\mu}$  – після XEf корекції
- $\mu'$  – після D2 корекції (фінальний результат)

Процес декапсуляції TiGER складається з кількох етапів та відповідних проміжних значень:

1) Декомпресія шифротексту:

$$\mathbf{c}'_1 = \text{Decompress}(ct_1, k_2), \quad \mathbf{c}'_2 = \text{Decompress}(ct_2, k_3)$$

На цьому моменті з'являється помилка компресії  $\mathbf{e}_{\text{comp}}$ .

2) Обчислення проміжного значення  $\mathbf{v}$ :

$$\mathbf{v}_{\text{raw}} = \mathbf{c}'_2 - \mathbf{c}'_1 \cdot \mathbf{s}$$

Це значення містить закодоване оригінальне повідомлення  $\mu$  плюс накопичені помилки з минулих від етапів (RLWE, RLWR та компресії).

3) Екстракція помилкових бітів:

$$\tilde{\mu}_i = \text{ExtractBit}(\mathbf{v}_{\text{raw}}, i), \quad i = 0, \dots, 255$$

(!) Кожен біт  $\tilde{\mu}_i$  може бути помилковим через надмірні попередні помилки. Це так звані необроблені біти.

4) Корекція помилок за допомогою XEf:

$$\hat{\mu}_i = \text{Xef.Decode}(\tilde{\mu}_i), \quad \text{для блоків по 8 біт}$$

XEf виправляє до 1 помилки на кожен блок з 8 біт.  $\hat{\mu}$  – результат після першої корекції.

5) Корекція помилок за допомогою D2:

$$\mu' = \text{D2.Decode}(\hat{\mu}), \quad (\text{мажоритарне голосування для кожного біту})$$

D2 виправляє помилки, використовуючи дуплікат інформації. Результатом є  $\mu'$  – декодоване повідомлення (в ідеалі має дорівнювати оригінальному  $\mu$ ).

6) Повторне шифрування (FO check):

$$ct' = \text{PKE.Encrypt}(pk, \mu'; H(\mu'))$$

Також робиться перевірка:  $ct' \stackrel{?}{=} ct$ . Якщо рівність правильна, то повертається  $K = H(\mu')$ , інакше –  $\bar{K} = H(z, ct)$  (застосовується implicit rejection).

## Вразливість через витік проміжних значень

Припустимо, що атакуючий може спостерігати  $\tilde{\mu}$  (біти після екстракції, але до застосування кодів корекції) через side-channel (power analysis, ЕМ, timing) – все як в атаці von Berg. Постає логічне питання: Що дає атакуючому  $\tilde{\mu}$ ? З рівняння, яке записували для  $\mathbf{v}_{\text{raw}}$  маємо:

$$\mathbf{v}_{\text{raw}} = \mathbf{c}'_2 - \mathbf{c}'_1 \cdot \mathbf{s} = \underbrace{\mathbf{b} \cdot \mathbf{e}_1 + \mathbf{e}_3}_{\text{відомі або контрольовані}} + \underbrace{\text{Encode}(\mu)}_{\text{(не)відомий}} - \mathbf{c}'_1 \cdot \mathbf{s}$$

Перегрупуємо:

$$\mathbf{c}'_1 \cdot \mathbf{s} = \mathbf{b} \cdot \mathbf{e}_1 + \mathbf{e}_3 + \text{Encode}(\mu) - \mathbf{v}_{\text{raw}}$$

Можливі два сценарії атаки:

- *Сценарій 1 ("пасивний")*: Атакуючий перехоплює чужий шифротекст, де  $\mu$  невідомий. Його мета – витягнути  $\mu$  з спостережень  $\mathbf{v}_{\text{raw}}$ .
- *Сценарій 2 ("активний")*: Атакуючий сам створює шифротексти з відомими йому  $\mu^{(i)}$  і спостерігає відповідні  $\mathbf{v}_{\text{raw}}^{(i)}$ . Його мета вже змінюється, він хоче витягнути  $\mathbf{s}$  розв'язавши систему лінійних рівнянь.

Розглянемо другий випадок:

1. Атакуючий створює  $t$  шифротекстів, самостійно обираючи помилки  $\mathbf{e}_1^{(i)}, \mathbf{e}_3^{(i)}$  та повідомлення  $\mu^{(i)}$ ;

2. Для кожного шифротексту отримує  $\tilde{\mu}^{(i)}$ , з яких може обчислити вже  $\mathbf{v}_{\text{raw}}^{(i)}$ ;
3. Оскільки атакуючий знає всі  $\mu^{(i)}$ , то він може обрахувати і кожен  $\text{Encode}(\mu^{(i)})$ .  
Отримується система рівнянь:

$$\mathbf{c}_1'^{(i)} \cdot \mathbf{s} = \underbrace{(\mathbf{b} \cdot \mathbf{e}_1^{(i)} + \mathbf{e}_3^{(i)} + \text{Encode}(\mu^{(i)}) - \mathbf{v}_{\text{raw}}^{(i)})}_{\text{відомі, оскільки атакуючий сам обрав } \mu^{(i)}} + \mathbf{e}_{\text{noise}}^{(i)}$$

де  $\mathbf{e}_{\text{noise}}^{(i)}$  – залишкова помилка від RLWR та округлень;

4. Маючи  $m \geq n$  рівнянь виду  $\mathbf{C}_1 \cdot \mathbf{s} = \mathbf{y} + \mathbf{E}_{\text{noise}}$ , розв'язує їх будь-яким зручним LWE-solving методом, знаходячи  $\mathbf{s}$ .

Необхідна кількість запитів до оракула для повного витягування  $\mathbf{s}$  з  $n = 512$  коефіцієнтами складає приблизно:

$$m \approx n + O(\sqrt{n}) \approx 512 + 50 = 562$$

Після збору  $m$  зразків, розв'язування системи рівнянь матиме складність:

$$\text{Complexity} \approx O(m^3) \approx O(512^3) \approx 2^{27} \text{ операцій}$$

Це *практично здійсненна атака*, якщо є витік  $\tilde{\mu}$ .

В першому ж випадку  $\mu$  є невідомим і атака була б набагато складнішою.

TiGER має закладений захист: коди корекції XEf і D2 приховують зв'язок між  $\tilde{\mu}$  та фінальним  $\mu$ . Проблема для атакуючого полягає в тому, що щоб використати спостережені  $\tilde{\mu}_{\text{received}}$  для витягування  $\mathbf{s}$ , потрібно знати *оригінальне*  $\mu$ . Але зв'язок між  $\mu$  та  $\tilde{\mu}_{\text{encoded}}$  є нетривіальним через те, що помилки змінюють кожен біт передбачувано, а коди XEf та D2 не обов'язково виправляють усі існуючі помилки (про це у наступному розділі 7.6). Тому без знання точного  $\mu$ , атакуючий не може побудувати систему рівнянь.

Але що буде, якщо атакуючий дізнався частину  $\mu$ ? Чи допоможе це йому? Припустимо, що через якусь іншу вразливість зломисник дізнався  $k$  біт з  $\mu$ . Чи полегшить це атаку? Частково так, але залишається  $256 - k$  невідомих біт, а коди корекції розподіляють інформацію, тому навіть знаючи  $k$  біт  $\mu$ , важко точно передбачити всі біти  $\tilde{\mu}$ . Нехай  $k = 128$  біт,  $\mu = 256$ , тоді залишкова ентропія складає  $2^{128}$ . Це все ще занадто велике для перебору і алгоритм залишається захищеним.

Міра захисту	Ефект
XEf + D2 codes	Приховування зв'язку між $\mu$ та $\tilde{\mu}$
FO re-encryption check	Атакуючий не може підтвердити коректність спостережених значень
Implicit rejection	Немає observable signal про успіх/провал маніпуляції
Memory wiping	Очищення буферів (temp keys) після використання
Hardware security	Ізоляція обчислень від можливих side-channels attacks

Таблиця 7.6: Захист TiGER від атак на проміжні значення

Атака через витік проміжних значень декодування є теоретично можливою, але TiGER має багаторівневий захист. Коректна реалізація (з усіма мірами захисту) алгоритму майже унеможливорює цю атаку.

## 7.6 Атака з урахуванням XEf корекції помилок

Використання коду корекції помилок XEf є особливістю TiGER, що дозволяє здійснювати агресивнішу компресію шифротексту. Тут, в цій секції, розглянемо чи можна якось експлуатувати структуру XEf для атак на TiGER.

Параметри XEf, які застосовуються для TiGER:

- Розмір блоку:  $d = 8$  біт
- Кількість блоків:  $256/8 = 32$  блоки
- Можливість корекції:  $t = 1$  помилка на блок

### Алгоритм застосування XEf

Спершу відбувається кодування:

Для оригінального блоку даних  $\mathbf{m} = (m_0, m_1, \dots, m_7) \in \{0, 1\}^8$  обчислюється так званий синдром  $s$ , шляхом застосування операції XOR:

$$s = m_0 \oplus m_1 \oplus \dots \oplus m_7 = \bigoplus_{i=0}^7 m_i$$

Декодування проходить наступним чином:

Маємо отриманий блок (received block)  $\mathbf{y} = (y_0, \dots, y_7) = \mathbf{m} \oplus \mathbf{e}$  може містити помилки  $\mathbf{e}$ . Обчислюємо значення:

$$s' = \bigoplus_{i=0}^7 y_i$$

Якщо  $s' \neq s$ , то присутня помилка. XEf визначає позицію помилки та виправляє її.

### Теоретична вразливість XEf

#### Проблема 1: Обмежена здатність корекції

XEf виправляє лише  $t = 1$  помилку на блок. Якщо в блоці  $\geq 2$  помилки, то:

$$\mathbb{P}[\text{блок декодується неправильно}] \approx C_8^2 \cdot p^2 = 28p^2$$

де  $p$  – ймовірність помилки одного біту.

#### Проблема 2: Детермінована структура

Оскільки синдром є лінійною функцією від бітів блоку  $f(\mathbf{m})$ , то знаючи деякі біти, можна обчислити синдром  $s$  та використати цю інформацію для звуження простору можливих  $\mathbf{m}$ .

### Атака через маніпуляцію синдромами

Нехай зломисник вміє перехоплювати шифротекст  $ct$ , модифікувати його для індукування помилок, а також спостерігати, чи була декапсуляція успішною.

Ідея атаки полягає в наступному:

1. Зломисник flip-ує (модифікує) один біт у блоці шифротексту:

$$ct'_i = ct_i \oplus (1 \ll j), \quad j \in \{0, \dots, 7\}$$

2. Це призводить до зміни одного біту в певному блоці після дешифрування;

3. XEf код спробує виправити цю помилку: якщо в блоці більше немає помилок, XEf успішно виправить і декапсуляція буде успішною;
4. Якщо в блоці вже була 1 помилка (наприклад від компресії/RLWE), то тепер їх 2. XEf виправляє одну, а інші лишаються і декапсуляція є неуспішною;
5. Спостерігаючи success/failure для різних flip-ів, атакуючий може дізнатися про розподіл помилок у блоках. Це дає приблизно  $\log_2 3 \approx 1.58$  біт інформації на кожен блок (три стани: 0 помилок, 1 помилка,  $\geq 2$  помилок).

Для 256 бітного повідомлення (займає 32 блоки), маємо:

$$\text{Total leaked info} \approx 32 \cdot 1.58 \approx 50 \text{ біт}$$

Цього для компрометації і витягування 785-бітного секретного ключа або 256-бітного повідомлення не достань, проте це зменшує простір пошуку з  $2^{256}$  до  $2^{206}$ .

Оскільки TiGER використовує  $\text{FO}_m^\perp$  з implicit rejection, що унеможливає спостереження success/failure, це зводить цю атаку нанівець.

## Атака спираючись на алгебраїчні властивості XEf

Синдром це лінійна операція XOR над  $\mathbb{F}_2$ , а це означає, що:

$$s(\mathbf{m}_1 \oplus \mathbf{m}_2) = s(\mathbf{m}_1) \oplus s(\mathbf{m}_2)$$

Припустимо, що зловмисник знає частину бітів повідомлення  $\mu$  та хоче знайти решту, використовуючи лінійність синдромів. Він зіштовхнеться з наступними проблемами:

1. Синдроми є внутрішньою частиною коду корекції та не передаються разом з шифротекстом;
2. Навіть якщо атакуючому вдасться відтворити процес кодування, йому потрібно знати *точні помилки* від RLWE/компресії (а вони випадкові);
3. Без знання помилок, система лінійних рівнянь на синдроми має занадто багато невідомих:
  - 32 синдроми (по одному на блок)  $\rightarrow$  32 рівняння;
  - Невідомі: 256 біт  $\mu$  + помилки від RLWE (кожна має ентропію  $\approx n \cdot h_e = 512 \cdot 274 \approx 140,000$  біт)  $\rightarrow$  система сильно недовизначена.

Атаки, що спрямовані на XEf код корекції помилок є небезпечними, але для TiGER практично нездійсненними завдяки використанню:

1. **Implicit rejection** – унеможливає спостереження success/failure декапсуляції;
2. **Подвійна корекція (XEf + D2)** — навіть якщо XEf зламано, D2 надає додатковий захист;
3. **FO перетворення** — re-encryption check виявляє модифіковані шифротексти.

## 7.7 Side-channel attacks

Side-channel атаки експлуатують фізичні характеристики при реалізації алгоритму: такі як час виконання, споживання енергії, електромагнітне випромінювання замість безпосередньої атаки на математичну структуру. Для TiGER, як і для всіх постквантових алгоритмів, side-channel атаки становлять серйозну практичну загрозу.

Тип атаки	Що відстежується	Цільова інформація
Timing attack	Час виконання операцій	Секретний ключ <b>s</b> , проміжні значення $\mu$
Power analysis (SPA/DPA)	Споживання енергії	Біти секретного ключа <b>s</b> при множенні
EM analysis	Електромагнітне випромінювання	Аналогічно до power analysis
Cache timing	Паттерни доступу до кешу	Індекси ненульових коефіцієнтів <b>s</b>
Fault injection	Індукування помилок	Обхід перевірок Decaps

Таблиця 7.7: Типи side-channel атак

## Класифікація side-channel атак

У пункті 5 розділу 7.1 ми вже частково згадували за них. Зараз більш детально розглянемо кожен з них:

Пройдемося по порядку та розглянемо кожен тип окремо:

### Timing атаки на TiGER

Перше, що є важливим джерелом variable timing – це множення многочленів на розрізнені секрети. TiGER використовує секрет **s** з вагою Геммінга  $h_s = 274$ . Схематично в реалізації це виглядає так:

```
for i in 0..n-1:
    if s[i] != 0:
        result += a[i] * s[i] // Executed h_{s} times
```

Час виконання:  $T \approx h_s \cdot T_{mul}$  варіюється залежно від позицій ненульових коефіцієнтів.

Друге – це rejection sampling при генерації секретного ключа з вагою рівною  $h_s$ :

```
do:
    s = sample_random()
    while hamming_weight(s) != h_s // Amount of iterations varies
```

Тут час виконання є випадковим.

Третє, але не менш важливе це умовні перевірки у декодуванні за допомогою кодів корекції XEf:

```
if syndrome_matches:
    correct_error() // Additional operations
else:
    keep_as_is()
```

Час залежатиме від кількості помилок у блоках.

Як використати цю інформацію? Зловмисник виміряв час декапсуляції для різних шифротекстів та побудував статистичну модель:

$$T_{\text{decaps}}(ct) \approx T_{\text{base}} + \sum_i w_i \cdot s_i + \text{noise}, \quad \text{де}$$

- $T_{\text{base}}$  – базовий час виконання операцій, які не залежать від секретного ключа;
- $s_i$  –  $i$ -тий коефіцієнт секретного ключа (attack target values);
- $w_i$  – ваги, що корелюють з операціями на  $s_i$ , іншими словами це скільки додаткового часу додає коефіцієнт  $s_i$  до загального часу виконання;
- noise – шум вимірювань.

Маючи достатню кількість вимірювань ( $\approx 10^4$ - $10^6$ ), можна відновити  $\mathbf{s}$  методами machine learning або кореляційного аналізу.

Захищатися від цього можна маючи константний час виконання (додавання dummy operations/masking) або (а краще і) випадковізація (shuffle) порядку обчислень (при тому ж множенні наприклад)

## Power analysis (DPA/CPA)

1. Differential power analysis (DPA): Зловмисник бачить на екрані осцилографа трейси споживання енергії під час множення  $\mathbf{a} \cdot \mathbf{s}$ :

- 1) Він збирає  $N$  трейсів споживання енергії для різних  $\mathbf{a}^{(i)}$ ;
- 2) Будує по три гіпотези для кожного біту  $s_j \in \{-1, 0, 1\}$ ;
- 3) Обчислює кореляцію між гіпотетичним споживанням та виміряним:

$$\rho_{j,\text{hyp}} = \text{corr}(\mathbb{P}_{\text{measured}}, \mathbb{P}_{\text{model}}(s_j = \text{hyp}))$$

- 4) Вибирає гіпотезу з найвищою кореляцією як правильну.

2. Correlation Power Analysis (CPA): Використовуючи Hamming weight model:

$$P_{\text{model}}(x) \approx \alpha + \beta \cdot HW(x), \quad \text{де}$$

- 1)  $P_{\text{model}}(x)$  – очікуване споживання енергії пристрою під час обробки значення  $x$ , вимірюється в одиниці потужності (Вт) або напругою на осцилографі
- 2)  $\alpha$  – базове споживання енергії пристроєм
- 3)  $\beta$  – коефіцієнт, що показує залежність від даних (скільки додаткової енергії споживає один біт, встановлений в 1)
- 4)  $HW(x)$  – кількість одиниць у двійковому представленні  $x$ .

Яка ефективність всього цього проти TiGER? Оскільки  $\mathbf{s}$  розріджений, атакуючий може зосередитися на виявленні позицій ненульових коефіцієнтів. Оскільки, грубо кажучи:

- Якщо  $s_i = 0$ : операція  $a_i \cdot s_i$  майже не споживає енергії;
- Якщо  $s_i = \pm 1$ : помітне споживання.

За допомогою ідентифікації позиції ненульових  $s_i$  можна трохи зменшити простір пошуку справжнього ключа. А захиститися від цього можна шляхом Masking 1/2-го порядку: розділивши  $\mathbf{s} = \mathbf{s}_0 \oplus \mathbf{s}_1$  на дві/більше випадкових частинок.

## Cache timing атаки

Механізм атаки закручений на архітектурі сучасних процесорів. Вони мають ієрархічну пам'ять (L1/L2/L3 кеш, RAM). Доступ до кешу відбувається набагато швидше за RAM, тому зловмисник може:

1. Заповнити кеш своїми даними (FLUSH);
2. Викликати функцію TiGER, що працює з секретом;
3. Виміряти, які кеш-лінії були витіснені (зробити RELOAD);
4. З паттернів доступу дізнатися інформацію про секрет.

Вразливість TiGER заключається в тому, що якщо індекси ненульових коефіцієнтів  $s$  використовуються для доступу до таблиць (наприклад, multiplication lookup tables), то паттерн доступу спричиняє витік інформації:

```
for i in sparse_indices(s): // Indices of non-zero s_i
    result += lookup_table[a[i]] // Access at index i
```

Як результат атакуючий бачить, які  $i$  були використані і дізнається позиції ненульових коефіцієнтів. Захиститися від цього дуже просто:

- Завжди обробляти всі індекси, використовуючи conditional move:

```
for i in 0..n-1:
    mask = (s[i] != 0)
    result += mask * lookup_table[a[i]]
```

- Уникати звертання по індексу, натомість використовувати лише арифметичні операції;
- Випадковізувати порядок доступу до пам'яті (Memory access randomization).

## Fault injection атаки

Fault injection attacks поділяються ще на чотири підкатегорії:

1. **Voltage glitching:** Короткочасне зниження напруги → пропуск операції;
2. **Clock glitching:** Маніпуляція тактовою частотою CPU → неправильні обчислення;
3. **Laser injection:** Цілеспрямоване випромінювання → bit flips у регістрах/пам'яті;
4. **EM injection:** Електромагнітні імпульси → індукування сторонніх помилок.

Мета цієї атаки на TiGER полягає в наступному – обійти re-encryption check у FO перетворенні, щоб змусити алгоритм повернути детерміністичний ключ  $K = H(m, ct)$  замість псевдовипадкового  $\bar{K} = H(z, ct)$  для модифікованого шифротексту  $ct'$ .

1. Зловмисник модифікує шифротекст  $ct$  (наприклад, змінює один біт) і має  $ct'$ ;
2. Індукує fault під час виконання re-encryption check:

```

m' = Dec(sk, ct)
ct'' = Enc(pk, m'; G(m', H(pk)))
if ct'' == ct: // Fault: skip this comparison
    return K = H(m', ct) // Determinated
else:
    return K_bar = H(z, ct) // Pseudorandom

```

Якщо fault успішний (перевірка не виявила підхову), то алгоритм повернув  $K = H(m', ct)$  – детермінований ключ, що залежить від декодованого  $m'$ . Для того самого  $ct'$ , ключ  $K$  буде однаковим при повторних запитах.

3. Зловмисник може експлуатувати це так:

- Подає  $ct'$  з різними fault injection багато разів;
- Якщо отримує однакові відповіді для  $ct'$ , то fault успішний, отримано детермінований  $K$ ;
- Якщо отримує різні відповіді, то fault не спрацював, отримував  $\bar{K} = H(z, ct')$  з випадковим  $z$ ;
- Маючи детермінований  $K$  для модифікованого  $ct'$ , він може отримати якусь інформацію про  $m'$  через side-channel аналіз або статистичні атаки.

Захист у TiGER є наступним:

1. Використання implicit rejection ускладнює fault атаку, але *не* є повноцінним захистом(!)
2. Використання кодів корекції (XEf + D2): Можуть виправити деякі індуковані помилки у даних  $ct$  до початку декапсуляції;

З того, що бажано б додати для додаткового захисту, це:

1. Виконувати re-encryption check двічі незалежними способами:

```

ct'' = Enc(pk, m'; G(m', H(pk)))
check1 = (ct'' == ct)
check2 = check_independently(ct'', ct) // Other implementation
if check1 AND check2:
    return K = H(m', ct)
else:
    return K_bar = H(z, ct)

```

Ймовірність, що fault вплине на обидві перевірки одночасно, набагато менша.

2. Проводити моніторинг скачків напруги, тактової частоти для виявлення fault injection спроб;

Підсумовуючи, можна сказати що: side-channel атаки є найсерйознішою практичною загрозою для TiGER. Математична конструкція алгоритму стійка, але паршива реалізація може призводити до витоків секретів через фізичні канали. Для безпечного використання TiGER критично важливі:

1. Константний час усіх операцій;
2. Masking секретних даних (принаймні 1-го порядку);
3. Уникнення secret-dependent memory access (див. cache timing attacks);
4. Захист від fault injection через *подвійні перевірки*;
5. Апаратні контрзаходи (HSM, secure enclaves).

## 7.8 Перенесення атак з RLWE/RLWR схем на TiGER

TiGER є частиною великої родини криптосистем на основі RLWE/RLWR. Багато атак, розроблених для інших схем (Kyber, Saber, NTRU, NewHope), можуть потенційно бути адаптовані до TiGER.

### 1. Решіткові атаки (Primal/Dual Lattice Attack)

**Статус:** Застосовна, але неефективна

Всі схеми на основі RLWE/RLWR вразливі до цих решіткових атак, але оскільки ця атака універсальна вона не є специфічною. Та й TiGER використовує консервативно обрані параметри з урахуванням найновіших (на 2022 рік) оцінок складності BKZ (Core-SVP  $\beta = 483$ ).

### 2. Meet-LWE атака

**Статус:** Частково застосовна

Розроблена для LWE з розрідженими секретами, прямо застосовна до TiGER. але Квантова складність  $\approx 2^{109.5}$  хоч і нижче цільового рівня 128 біт, проте, потребує нереалістичних ресурсів (пам'ять) для самої атаки.

### 3. Failure boosting атака

**Статус** не застосовна

Оригінальна атака [24] застосовувалася до Kyber, Saber та LAC схем без implicit rejection. Оскільки TiGER використовує  $\text{FO}_m^{\perp}$  з implicit rejection, то немає observable  $\perp$  і атакуючий не знає, чи відбувалися failure.

### 4. ССА атаки на базову РКЕ

**Статус:** не застосовна

Атаки по типу von Berg базуються на відсутності верифікації цілісності  $ct$  у базових РКЕ схемах. А TiGER використовує повне FO перетворення з re-encryption check, тому будь-яка модифікація  $ct$  виявляється та атака блокується.

### 5. Signal leakage атаки

**Статус:** Теоретично застосовна (якщо реалізація хромає), але практично складна

Оригінальна атака [27] застосовувалась до схем Round5, LAC, де проміжні "сигнали" могли витікати через side-channels і за великої кількості спостережень можна витягнути  $\mathbf{s}$ .

Якщо  $\mathbf{v}_{\text{raw}}$  витікає через power/ЕМ side-channel, атака можлива (див. секцію 7.5). Проте належна реалізація з masking мітигує цю атаку.

## 6. NTRU-specific атаки (не RLWR/RLWE схема, але ладно)

**Статус:** не застосовна

NTRU використовує структуру кільця  $\mathbb{Z}[x]/(x^n - 1)$  та інші методи множення (FFT/NTT), в той час як TiGER використовує циклотомічне кільце  $\mathbb{Z}_q[x]/(x^n + 1)$ .

### Lesson learned from predecessors:

1. **Від Kyber:** Консервативний вибір параметрів безпеки, аналіз DFP;
2. **Від Saber:** LWR підхід для компактності, уникнення NTT;
3. **Від LAC/Round5:** Код корекції помилок (XEf);
4. **Від досліджень failure boosting:** Implicit rejection;
5. **Від NewHope:** Importance of reconciliation / error correction for reliability.

Основні ризики залишаються в площині side-channel атак, які потребують лише обережної реалізації.

# Розділ 8

## Практичне застосування та висновки

### 8.1 Можливі сценарії використання TiGER

TiGER, як постквантовий KEM алгоритм, може інтегруватися у широкий спектр криптографічних протоколів та застосувань:

- TLS/HTTPS: Використання TiGER для встановлення спільного ключа в TLS 1.3 може потенційно захистити веб-комунікації від квантових атак.
- VPN та IPsec: Інтеграція TiGER в IKEv2/IPsec протоколи забезпечуватиме квантово-стійке шифрування для VPN-з'єднань.

### 8.2 Підсумки дослідження

Ми комплексно дослідили (хочеться вмерти після цього) постквантовий криптографічний алгоритм TiGER, що базується на проблемах RLWE та RLWR, його теоретичне підґрунтя та параметри безпеки.

#### Основні результати:

- Побачили, що TiGER забезпечує всі рівні безпеки NIST;
- Подвійна система корекції помилок (XEf + D2) та implicit rejection забезпечують стійкість до більшості відомих атак;
- Компактні розміри ключів та шифротекстів роблять TiGER привабливим для практичних застосувань;
- Основні ризики пов'язані з side-channel атаками, вимагають обережної реалізації системи.

TiGER демонструє збалансоване поєднання безпеки, ефективності та практичності, що підтверджує його потенціал як постквантової криптографічної системи. Злиття з SMAUG у алгоритм SMAUG-T та його визнання фіналістом KpqC підкреслює перспективність підходу.

Найближчі роки будуть вирішальними для формування постквантового криптографічного ландшафту, тому своєчасна міграція критичної інфраструктури є необхідною для захисту від майбутніх квантових загроз.

---

## Розділ 9

### Результати власної реалізації

Тут Богдан вже щось допише

# Bibliography

- [1] Peter W. Shor. «Algorithms for quantum computation: discrete logarithms and factoring». In: *Proceedings 35th Annual Symposium on Foundations of Computer Science* (1994), pp. 124–134. DOI: [10.1109/SFCS.1994.365700](https://doi.org/10.1109/SFCS.1994.365700).
- [2] Wikipedia. *Post-quantum cryptography* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 7-October-2025]. URL: [https://en.wikipedia.org/wiki/Post-quantum\\_cryptography](https://en.wikipedia.org/wiki/Post-quantum_cryptography).
- [3] Oded Regev. «On lattices, learning with errors, random linear codes, and cryptography». In: *Journal of the ACM* 56.6 (2009), pp. 1–40. DOI: [10.1145/1568318.1568324](https://doi.org/10.1145/1568318.1568324).
- [4] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. «On Ideal Lattices and Learning with Errors over Rings». In: *Advances in Cryptology – EUROCRYPT 2010*. 2010, pp. 1–23. DOI: [10.1007/978-3-642-13190-5\\_1](https://doi.org/10.1007/978-3-642-13190-5_1).
- [5] Abhishek Banerjee, Chris Peikert, and Alon Rosen. «Pseudorandom Functions and Lattices». In: *Advances in Cryptology – EUROCRYPT 2012*. 2012, pp. 719–737. DOI: [10.1007/978-3-642-29011-4\\_42](https://doi.org/10.1007/978-3-642-29011-4_42).
- [6] Jung Hee Cheon et al. «Lizard: Cut off the Tail! A Practical Post-Quantum Public-Key Encryption from LWE and LWR». In: *Security and Cryptography for Networks – SCN 2018*. 2018, pp. 160–177. DOI: [10.1007/978-3-319-98113-0\\_9](https://doi.org/10.1007/978-3-319-98113-0_9).
- [7] Joohee Lee et al. «RLizard: Post-quantum Key Encapsulation Mechanism for IoT Devices». In: *IEEE Access* 7 (2019), pp. 2080–2091. DOI: [10.1109/ACCESS.2018.2886964](https://doi.org/10.1109/ACCESS.2018.2886964).
- [8] Seunghwan Park et al. *TiGER: Tiny bandwidth key encapsulation mechanism for easy miGration based on RLWE(R)*. Cryptology ePrint Archive, Paper 2022/1651. <https://eprint.iacr.org/2022/1651>. 2022.
- [9] Eiichiro Fujisaki and Tatsuaki Okamoto. «Secure Integration of Asymmetric and Symmetric Encryption Schemes». In: *Advances in Cryptology – CRYPTO ’99*. 1999, pp. 537–554. DOI: [10.1007/3-540-48405-1\\_34](https://doi.org/10.1007/3-540-48405-1_34).
- [10] Eiichiro Fujisaki and Tatsuaki Okamoto. «Secure Integration of Asymmetric and Symmetric Encryption Schemes». In: *Journal of Cryptology* 26.1 (2013), pp. 80–101. DOI: [10.1007/s00145-011-9114-1](https://doi.org/10.1007/s00145-011-9114-1).
- [11] KpqC Team. *Korean Post-Quantum Cryptography Competition*. <https://www.kpqc.or.kr>. 2023.
- [12] Jung Hee Cheon et al. *SMAUG-T: the Key Exchange Algorithm based on Module-LWE and Module-LWR*. KpqC Round 2 Submission. Version 3.0. 2024.

- [13] Miklós Ajtai. «The shortest vector problem in  $L_2$  is NP-hard for randomized reductions». In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 1998, pp. 10–19. DOI: [10.1145/276698.276705](https://doi.org/10.1145/276698.276705).
- [14] Joppe Bos et al. «CRYSTALS-Kyber: A CCA-Secure Module-Lattice-Based KEM». In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2018, pp. 353–367. DOI: [10.1109/EuroSP.2018.00032](https://doi.org/10.1109/EuroSP.2018.00032).
- [15] Jan-Pieter D’Anvers et al. «Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM». In: *Progress in Cryptology – AFRICACRYPT 2018*. 2018, pp. 282–305. DOI: [10.1007/978-3-319-89339-6\\_16](https://doi.org/10.1007/978-3-319-89339-6_16).
- [16] Siemen Dhooghe and Svetla Nikova. «My Gadget Just Cares For Me - How NINA Can Prove Security Against Combined Attacks». In: *Progress in Cryptology – INDOCRYPT 2018*. 2018, pp. 35–55. DOI: [10.1007/978-3-030-05378-9\\_3](https://doi.org/10.1007/978-3-030-05378-9_3).
- [17] Jan-Pieter D’Anvers, Siemen Dhooghe, and Frederik Vercauteren. «On the Impact of Decryption Failures on the Security of NTRU Encryption». In: *Advances in Cryptology – ASIACRYPT 2019*. 2019, pp. 565–598. DOI: [10.1007/978-3-030-34621-8\\_20](https://doi.org/10.1007/978-3-030-34621-8_20).
- [18] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. «A Modular Analysis of the Fujisaki-Okamoto Transformation». In: *Theory of Cryptography – TCC 2017*. 2017, pp. 341–371. DOI: [10.1007/978-3-319-70500-2\\_12](https://doi.org/10.1007/978-3-319-70500-2_12).
- [19] C. P. Schnorr and M. Euchner. «Lattice basis reduction: Improved practical algorithms and solving subset sum problems». In: *Mathematical Programming*. Vol. 66. 1994, pp. 181–199. DOI: [10.1007/BF01581144](https://doi.org/10.1007/BF01581144).
- [20] MATZOV. *Report on the Security of LWE: Improved Dual Lattice Attack*. Submitted to NIST. <https://zenodo.org/record/6412487>. Apr. 2022.
- [21] Martin R. Albrecht, Rachel Player, and Sam Scott. «On the concrete hardness of Learning with Errors». In: *Journal of Mathematical Cryptology* 9.3 (2015), pp. 169–203. DOI: [10.1515/jmc-2015-0016](https://doi.org/10.1515/jmc-2015-0016).
- [22] Andrej Bogdanov et al. «On the Hardness of Learning with Rounding over Small Modulus». In: *Theory of Cryptography – TCC 2016-A*. 2016, pp. 209–224. DOI: [10.1007/978-3-662-49096-9\\_9](https://doi.org/10.1007/978-3-662-49096-9_9).
- [23] Ronald Cramer, Léo Ducas, and Benjamin Wesolowski. «Short Stickelberger Class Relations and Application to Ideal-SVP». In: *Advances in Cryptology – EUROCRYPT 2017*. 2017, pp. 324–348. DOI: [10.1007/978-3-319-56620-7\\_12](https://doi.org/10.1007/978-3-319-56620-7_12).
- [24] Jan-Pieter D’Anvers et al. «Decryption Failure Attacks on IND-CCA Secure Lattice-Based Schemes». In: *Public-Key Cryptography – PKC 2019*. 2019, pp. 565–598. DOI: [10.1007/978-3-030-17253-4\\_19](https://doi.org/10.1007/978-3-030-17253-4_19).
- [25] Alexander May. «How to Meet Ternary LWE Keys». In: *Advances in Cryptology – CRYPTO 2021* 12826 (2021), pp. 701–731. DOI: [10.1007/978-3-030-84245-1\\_24](https://doi.org/10.1007/978-3-030-84245-1_24).
- [26] Casper von Berg and Serge Fehr. «Parallel Isogeny Path Finding with Limited Memory». In: *Progress in Cryptology – INDOCRYPT 2023*. 2023, pp. 462–492. DOI: [10.1007/978-3-031-56235-8\\_22](https://doi.org/10.1007/978-3-031-56235-8_22).
- [27] Qian Guo, Thomas Johansson, and Jing Yang. «A Novel CCA Attack Using Decryption Errors Against LAC». In: *Advances in Cryptology – ASIACRYPT 2019*. 2019, pp. 82–111. DOI: [10.1007/978-3-030-34578-5\\_4](https://doi.org/10.1007/978-3-030-34578-5_4).