НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. Ігоря СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

Звіт з виконання лабораторної роботи

ДОСЛІДЖЕННЯ СУЧАСНИХ АЛГЕБРАЇЧНИХ КРИПТОСИСТЕМ

Виконала студентка
групи ФІ-52мн
Балацька Вікторія

Перевірив:
Фесенко А.В

Київ — 2025

ВСТУП

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

Постановка завдання:

1. Реалізація алгоритму: розробити програмну реалізацію обраного криптографічного алгоритму та всі можливі варіанти цього алгоритму.

2. Перевірка коректності: підтвердити правильність реалізації за допомогою тестів, використовуючи офіційні тестові вектори або офіційну реалізацію.

3. Аналіз продуктивності та порівняння: знайти схожі алгоритми та провести порівняльний аналіз швидкодії за різних умов, дослідити вплив модифікацій окремих складових частин на ефективність.

4. Теоретичне дослідження: надати повний теоретичних опис алгоритму з усіма деталями та відомими результатами досліджень; провести аналіз наявних атак на обраний алгоритм та описати власні дослідження атак; виконати порівняльний аналіз обраного алгоритму зі схожими та дослідити можливість перенесення відомих атак на нього.

Хід роботи: У своїй роботи я досліджуватиму алгоритм підпису SOLMAE. SOLMAE - це схема підпису на основі решіток, що відповідає парадигмі "хеш-і-підпис"і представляється над NTRU решітками.

1 ТЕОРЕТИЧНЕ ДОСЛІДЖЕННЯ

У цій частині звіту проведено теоретичне дослідження алгоритму підпису SOLMAE: формальне визначення та опис алгоритму; аналіз безпеки та відомі результати; порівняльний аналіз з іншими підписами; можливі атаки та їх перенесення; аналіз продуктивності.

1.1 Формальне визначення та опис алгоритму

SOLMAE - це схема підпису на основі решітки, та розшифровується як quantum-**S**ecure alg**O**rithm for **L**ong-term **M**essage **A**uthentication and **E**ncryption (пост-квантовий алгоритм для довгострокової автентифікації та шифрування повідомлень). Для ефективної реалізації, структура потребує класу решіток, що мають ефективно обчислювальні бази з односторонньою функцією з секретом (trapdoors) для процедури підпису, дослідивши існуючі класи решіток, автори зупинились на NTRU-решітках. У такому випадку підписання зводиться до вибірки коротких гаусових векторів у відкритій NTRU решітці. Сам алгоритм SOLMAE натхненний дизайном Falcon. Проте, порівнюючи з Falcon, тут є певні нові теоретичні основи: на високому рівні усувається властива процедура вибірки технічності і більша частина її індукованої складності з точки зору реалізації, без втрати ефективності. Простота конструкції перетворюється на швидшу роботу, але при цьому зберігаючи розміри підписів і ключів верифікації, а також надаючи додаткові функції такі як дешевше маскування та можливість паралелізації.

1.1.1 Принципи проектування

Алгоритм підпису SOLMAE побудований за парадигмою "Хешуємо та підписуємо"на решітках та покликаний покращити ефективність і безпеку

порівняно з попередніми схемами, зокрема Falcon. Його проектування ґрунтується на трьох ключових ідеях (зображені на рисунку 1.1):

- Гібридний семплер (Hybrid Sampler) — швидший, простіший і паралелізований спосіб генерувати гаусові вектори, що спрощує підписування та зменшує обчислювальні витрати.

- Оптимізований алгоритм створення ключів — спеціально налаштований для покращення якості односторонніх функцій з секретом у NTRU-решітках і підвищення рівня безпеки при збереженні продуктивності.

- Техніки стиснення даних — зменшують розмір підписів і ключів без впливу на безпеку, оптимізуючи використання пропускної здатності.

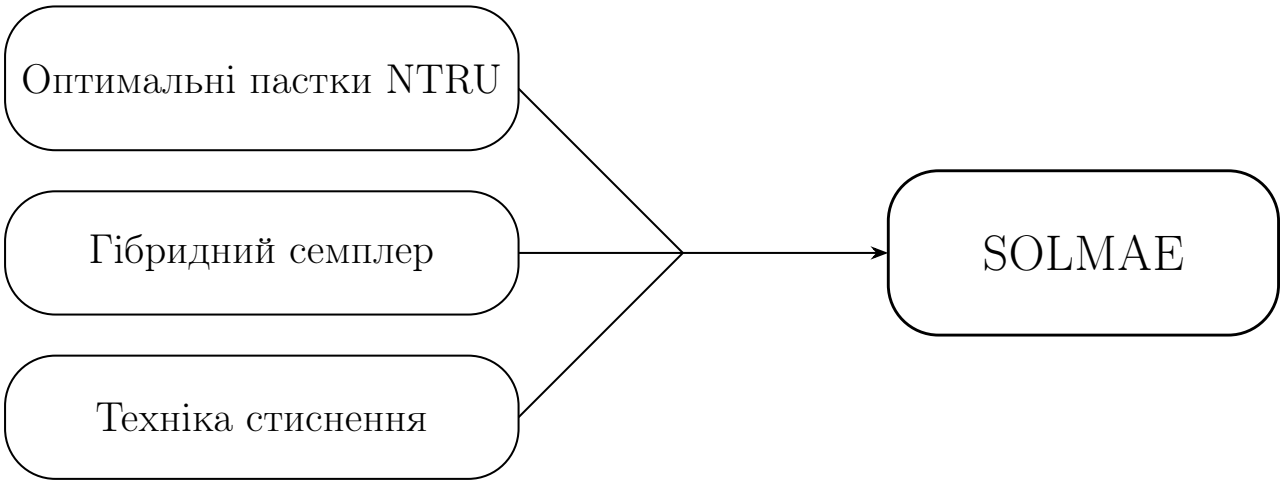

Рисунок 1.1 – Схематичне представлення основних компонентів алгоритму SOLMAE

SOLMAE використовує переваги алгебраїчної структури NTRU-решіток та поєднує сучасні підходи до побудови односторонніх функцій з секретом і вибору гаусових вибірок. Завдяки цьому схема досягає високої швидкодії, меншого розміру підписів та ключів, а також зберігає стійкість до відомих атак на базові задачі решіток.

1.1.2 Базові поняття та позначення

Вектори позначаються жирними малими літерами та розглядаються як стовпчикові. Матриці позначаються жирними великими літерами. Коли ми говоримо, що матриця є базисом простору, маємо на увазі, що стовпчики цієї матриці утворюють базис. ℓ_2-норма вектора $\mathbf{x} = (x_1, \ldots, x_d)$ визначається як $\|\mathbf{x}\| = \left(\sum_i |x_i|^2\right)^{1/2}$, а його ℓ_∞-норма — як $\|\mathbf{x}\|_\infty = \max_i |x_i|$.

Решітки. Решітками називають дискретну підгрупу $\mathbb{R}^n$. Іншими словами, решітка - це множина цілочисельних лінійних комбінацій, отриманих з базису $\mathbf{B} \in \mathbb{R}^n$. Об'єм решітки дорівнює $det(\mathbf{B})$ для будь-якого її базису.

Циклотомічні кільця степенів двійки. Для побудови схеми підпису SOLMAE використовується циклотомічне кільце $K = \mathbb{Q}[X]/(X^d + 1)$, де $d = 2^n$ — степінь двійки. Його цілочисельним підкільцем є $R = \mathbb{Z}[X]/(X^d + 1)$, а дійсне розширення позначається як $K_\mathbb{R} = \mathbb{R}[X]/(X^d + 1)$.

Поліном $f \in K_\mathbb{R}$ може бути представлений кількома способами:

1) Коефіцієнтне подання:

$$f = \sum_{i=0}^{d-1} f_i X^i \quad \longleftrightarrow \quad \mathbf{f} = (f_0, \ldots, f_{d-1}).$$

2) Канонічне вкладення (також відоме як дискретне перетворення Фур'є — DFT):

$$\varphi(f) = (\varphi_1(f), \ldots, \varphi_d(f)), \quad \varphi_j(f) = f(\zeta_j),$$

де $\zeta_j = e^{i(2j-1)\pi/d}$ — d-ті примітивні корені одиниці. У цьому поданні множення поліномів у кільці переходить у покомпонентне множення в $\mathbb{C}^d$.

3) Матриця множення:

$$[f] := \begin{pmatrix} f_0 & -f_{d-1} & \dots & -f_1 \\ f_1 & f_0 & \dots & -f_2 \\ \vdots & \vdots & \ddots & \vdots \\ f_{d-1} & f_{d-2} & \dots & f_0 \end{pmatrix}.$$

Для векторів $\mathbf{x} = (x_1, \dots, x_d)$ використовуються стандартні норми:

$$\|\mathbf{x}\|_2 = \left(\sum_i |x_i|^2 \right)^{1/2}, \qquad \|\mathbf{x}\|_\infty = \max_i |x_i|.$$

Алгебраїчний метод Грама-Шмідта. Для пар (f,g) та $(F,G) \in K_{\mathbb{R}}^{2\times 2}$ визначимо скалярний добуток: $\langle (f,g), (F,G) \rangle_K = f^*F + g^*G$, де f^* та g^* — спряжені елементи у $K_{\mathbb{R}}$.

Ортогоналізація Грама — Шмідта для пари (F,G) відносно (f,g) має вигляд:

$$(\widetilde{F}, \widetilde{G}) = (F, G) - \frac{\langle (f,g), (F,G) \rangle_K}{\langle (f,g), (f,g) \rangle_K} \cdot (f,g).$$

Легко перевірити, що $\langle (f,g), (\widetilde{F}, \widetilde{G}) \rangle_K = 0$, тобто вектори (f,g) та $(\widetilde{F}, \widetilde{G})$ є ортогональними.

Решітка NTRU. Нехай q — ціле число, а $f \in R$ таке, що f є оборотним за модулем q (еквівалентно, $\det[f]$ взаємно простий із q). Позначимо $h = g/f \bmod q$ та розглянемо $NTRU$-модуль, пов'язаний з h:

Ця решітка має об'єм q^d. Над R вона породжується парою (f,g) та будь-якими (F,G), що задовольняють $fG - gF = q$. У такому випадку $\mathcal{L}_{\text{NTRU}}$ має базис вигляду:

$$\mathbf{B}_{f,g} = \begin{bmatrix} [f] & [F] \\ [g] & [G] \end{bmatrix}.$$

Легко перевірити, що $([h], -\mathrm{Id}_d) \cdot \mathbf{B}_{f,g} = 0 \bmod q$, тому відкритим ключем є h.

Задача NTRU-search формулюється так: маючи $h = g/f \bmod q$, знайти будь-яку пару $(f' = x^i f,\ g' = x^i g)$.

У варіанті decision потрібно розрізнити $h = g/f \bmod q$ від рівномірно випадкового $h \in R_q := \mathbb{Z}[X]/(q, X^d + 1) = (\mathbb{Z}/q\mathbb{Z})[X]/(X^d + 1)$. Ці задачі вважаються складними при великому d.

Якість базису NTRU. Секретний базис $\mathbf{B}_{f,g}$ не може бути довільною парою, оскільки він повинен забезпечувати можливість відбору коротких гаусових векторів у ґратці $\mathcal{L}_{\mathrm{NTRU}}$ за допомогою гібридного семплінгу. Якість базису $\mathbf{B}_{f,g}$ визначається наступною величиною:

$$\mathcal{Q}(f,g) = \max_{1 \leqslant i \leqslant d/2} \max \left(\frac{|\varphi_i(f)|^2 + |\varphi_i(g)|^2}{q},\ \frac{q}{|\varphi_i(f)|^2 + |\varphi_i(g)|^2} \right)^{1/2}.$$

Гаусові розподіли. Гаусова функція, центрована в точці $\mathbf{c} \in \mathbb{R}^d$ з додатно визначеною коваріаційною матрицею Σ, визначається як $\rho_{\mathbf{c},\Sigma}(\mathbf{x}) = \exp\bigl(-\frac{1}{2}(\mathbf{x} - \mathbf{c})^t \Sigma^{-1} (\mathbf{x} - \mathbf{c})\bigr)$.

Нормальний розподіл $\mathcal{N}_{\mathbf{c},\Sigma}$ із центром у $\mathbf{c}$ та коваріацією Σ має щільність, пропорційну $\rho_{\mathbf{c},\Sigma}$.

Коли ми пишемо $\mathbf{x} \leftarrow \mathcal{N}_{\Sigma}^{K_{\mathbb{R}}}$, ми маємо на увазі, що відповідний d-вимірний вектор

$\frac{1}{\sqrt{d}}\bigl(\Re\varphi_1(\mathbf{x}),\ \Im\varphi_1(\mathbf{x}),\ \ldots,\ \Re\varphi_{d/2}(\mathbf{x}),\ \Im\varphi_{d/2}(\mathbf{x})\bigr)$ має розподіл $\mathcal{N}_{\Sigma}$, де $\Re z, \Im z$ — це дійсна та уявна частини комплексного числа z.

Для ґратки $\mathcal{L} \subset \mathbb{R}^d$ дискретний гаусовий розподіл із параметрами $\mathbf{c} \in \mathbb{R}^d$ та Σ визначається для всіх $\mathbf{x} \in \mathcal{L}$ як

$$D_{\mathcal{L},\mathbf{c},\Sigma}(\mathbf{x}) = \frac{\rho_{\mathbf{c},\Sigma}(\mathbf{x})}{\rho_{\Sigma}(\mathcal{L} - \mathbf{c})}.$$

Якщо центр $\mathbf{c} = 0$, його часто упускають. Коли $\Sigma = s^2 I$ (скалярна матриця), використовують позначення $\mathcal{N}_s$ або $D_{\mathcal{L},s}$.

1.1.3 Загальний огляд схеми підпису SOLMAE

Як і в будь-якій схемі підпису потрібно ввести три алгоритми: створення ключів (KeyGen), підпису (Sign) та верифікації (Verif). Алгоритм KeyGen

відповідає за створення ключів — відкритого та особистого. Алгоритм Sign використовується для створення підпису на основі особистого ключа, тоді як Verif забезпечує перевірку правильності підпису за допомогою відкритого ключа.

У межах GPV (Gentry–Peikert–Vaikuntanathan) до схеми додаються додаткові вимоги:

— визначається клас решіток, у якому алгоритм KeyGen обчислює пару, що складається з базису решітки та відповідної пари односторонніх функцій з секретом (trapdoor pair);

— алгоритм Sign застосовує спеціальну процедуру Sample, яка генерує випадкові вектори в решітці. При цьому розподіл вибірки побудований так, щоб не розкривати жодної інформації про одностороннюю функцію з секретом.

Таким чином, схема SOLMAE наслідує основну структуру GPV-підписів, забезпечуючи безпечне створення та перевірку підписів на базі решіткових припущень, зокрема завдяки використанню односторонніх функцій з секретом та стохастичного семплінгу, що гарантує приховування секретної інформації.

Конструкція KeyGen

Алгоритм KeyGen у схемі SOLMAE відповідає за побудову пар базисів решітки для односторонньої функції з секретом та попередню підготовку даних, необхідних для процедур підписування і семплінгу. Архітектурно KeyGen складається з трьох основних підзадач:

1) *PairGen* — створює початкові поліномні пари (f,g) у кільці R таким чином, щоб отримана якість базису $\mathcal{Q}(f,g) = \alpha$ відповідала цільовому значенню (цей параметр залежить від обраного рівня безпеки; за замовчуванням $\alpha_{512} = 1.17$, $\alpha_{1024} = 1.64$). PairGen працює в FFT-представленні: випадковим чином вибирає комплексні коефіцієнти у заданому кільці і округлює їх до цілих, контролюючи похибку через підібрані радіуси. Для цього використовується підпроцедура UnifCrown, яка забезпечує рівномірний розподіл у FFT-форматі.

2) *NtruSolve* — реалізує алгоритм Prest & Pornin. Вона приймає на вхід

пару (f,g) і модуль q, а на виході повертає доповнення (F,G), що задовольняє рівняння типу

$$fG - gF = q \quad \text{в кільці } R.$$

Цей алгоритм будує базис решітки $\mathcal{L}_{\text{NTRU}}$, асоційованої з публічним ключем $h = g/f \bmod q$. Реалізація NtruSolve оптимізована за допомогою рекурсивного використання підпільних перетворень і має практичну складність близьку до $O(d \log d)$ для параметра розміру d.

3) *Передобчислення для Sample* — після побудови базису обчислюються додаткові параметри, необхідні для стохастичного семплінгу підписів: матриці коваріацій Σ_1, Σ_2 для проміжних Gaussian-вибірок, а також проєкційні вектори β_1, β_2.

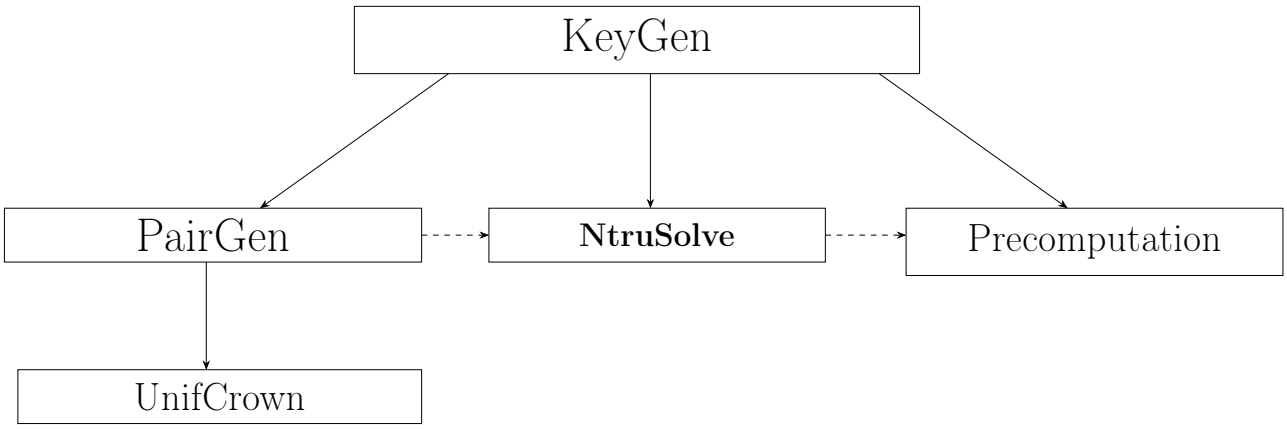

Рисунок 1.2 – Блок-схема KeyGen

Після виконання всіх підзадач алгоритм KeyGen повертає такі структури:

– Особистий ключ (sk):

$$\text{sk} = (b_1 = (f,g),\ b_2 = (F,G),\ \tilde{b}_2 = (\tilde{F}, \tilde{G}),\ \Sigma_1, \Sigma_2, \beta_1, \beta_2),$$

де $\tilde{b}_2$ — ортогоналізація (Gram–Schmidt) вектора (F,G) відносно (f,g); Σ_i та β_i — попередньо обчислені параметри для семплінгу.

– Відкритий ключ (pk):

$$\mathrm{pk} = (h, q, \sigma_{\mathrm{sig}}, \eta),$$

де $h = g/f \bmod q$, а σ_{sig} і η — стандартні відхилення, які визначають параметри Gaussian-семплінгу та межу параметр згладжування (smoothing parameter).

Якість базису та генерація пар: якість базису визначається через FFT-оцінки коефіцієнтів за формулою:

$$\alpha := \mathcal{Q}(f,g) = \max_{1 \le i \le d/2} \max \left(\frac{|\varphi_i(f)|^2 + |\varphi_i(g)|^2}{q}, \frac{q}{|\varphi_i(f)|^2 + |\varphi_i(g)|^2} \right)^{1/2}.$$

Базис має якість α, якщо для всіх $i \le d/2$ виконується нерівність:

$$\frac{q}{\alpha^2} \le |\varphi_i(f)|^2 + |\varphi_i(g)|^2 \le \alpha^2 q.$$

Щоб забезпечити стабільність якості базису, PairGen використовує UnifCrown, яка вибирає коефіцієнти у кільці з радіусами

$$R_- = \left(\frac{1}{\alpha} + \delta \right) \sqrt{q}, \quad R_+ = (\alpha - \delta)\sqrt{q},$$

де δ — мала корекційна константа, підібрана експериментально. Наприклад, $\delta_{512} \approx 0.065$, $\delta_{1024} \approx 0.3$.

Нижче представлено основний алгоритм генерації особистого та відкритого ключів для NTRU-решіткової схеми. Алгоритм ґрунтується на побудові базису $\mathcal{L}_{\mathrm{NTRU}}$ з контрольованою якістю α, що використовується для забезпечення балансу між безпекою та ефективністю.

Додаткові підпроцедури PairGen та NtruSolve відповідають за створення пар поліномів та знаходження розв'язків відповідного рівняння в кільці $\mathbb{R}_q$.

Algorithm 1.1 KeyGen

Input: modulus q, target quality $1 < \alpha$, parameters $\sigma_{\text{sig}}, \eta > 0$
Output: basis $((f,g),(F,G)) \in R^2$ of $\mathcal{L}_{\text{NTRU}}$ with $\mathcal{Q}(f,g) = \alpha$

▷ Secret basis computation

repeat
  $\mathbf{b}_1 \leftarrow (f,g) \leftarrow \text{PairGen}(q, \alpha, R_-, R_+)$
until f is invertible mod q
$\mathbf{b}_2 \leftarrow (F,G) \leftarrow \text{NtruSolve}(q,f,g)$

▷ Public key quantities

$h \leftarrow g/f \bmod q$
$\gamma \leftarrow 1.1 \cdot \sigma_{\text{sig}} \cdot \sqrt{2d}$ *// tolerance for signature length*

▷ Sampling data (Fourier domain)

$\beta_1 \leftarrow \frac{1}{\langle \mathbf{b}_1, \mathbf{b}_1 \rangle_K} \cdot \mathbf{b}_1$
$\Sigma_1 \leftarrow \sqrt{\frac{\sigma_{\text{sig}}}{\langle \mathbf{b}_1, \mathbf{b}_1 \rangle_K} - \eta^2}$
$\widetilde{\mathbf{b}}_2 \leftarrow \mathbf{b}_2 - \langle \mathbf{b}_1, \mathbf{b}_2 \rangle \cdot \mathbf{b}_1$
$\beta_2 \leftarrow \frac{1}{\langle \mathbf{b}_2, \mathbf{b}_2 \rangle_K} \cdot \mathbf{b}_2$
$\Sigma_2 \leftarrow \sqrt{\frac{\sigma_{\text{sig}}}{\langle \mathbf{b}_2, \mathbf{b}_2 \rangle_K} - \eta^2}$
$sk \leftarrow (\mathbf{b}_1, \mathbf{b}_2, \widetilde{\mathbf{b}}_2, \Sigma_1, \Sigma_2, \beta_1, \beta_2)$
$pk \leftarrow (q, h, \sigma_{\text{sig}}, \eta, \gamma)$
return (sk, pk)

Algorithm 1.2 PairGen

Input: modulus q, target quality $1 < \alpha$, radii $0 < R_- < R_+$
Output: pair (f,g) with $\mathcal{Q}(f,g) = \alpha$

for $i = 1$ **to** $d/2$ **do**
  $(x_i, y_i) \leftarrow \text{UnifCrown}(R_-, R_+)$ ▷ see UnifCrown
  $\theta_x, \theta_y \leftarrow \mathcal{U}(0,1)$
  $\varphi_{f,i} \leftarrow |x_i| e^{2\pi i \theta_x}, \quad \varphi_{g,i} \leftarrow |y_i| e^{2\pi i \theta_y}$
end for
$(f^{\mathbb{R}}, g^{\mathbb{R}}) \leftarrow \text{FFT}^{-1}\big((\varphi_{f,i})_{i \le d/2}\big), \text{FFT}^{-1}\big((\varphi_{g,i})_{i \le d/2}\big)$
$(f,g) \leftarrow \big(|f_i^{\mathbb{R}}|\big)_{i \le d/2}, \big(|g_i^{\mathbb{R}}|\big)_{i \le d/2}$
$(\varphi(f), \varphi(g)) \leftarrow (\text{FFT}(f), \text{FFT}(g))$
for $i = 1$ **to** $d/2$ **do**
  if $\frac{\alpha^2}{q} > |\varphi_i(f)|^2 + |\varphi_i(g)|^2$ **or** $\alpha^2 q < |\varphi_i(f)|^2 + |\varphi_i(g)|^2$ **then**
    restart ▷ reject and resample
  end if
end for
return (f,g)

Конструкція Sign

Як відомо, решітки NTRU живуть у просторі $\mathbb{R}^{2d}$. Їхня структура спрощує обчислення прообразу. Дійсно, підписувачу достатньо обчислити $\mathbf{m} = \mathrm{H}(M) \in \mathbb{R}^d$, оскільки $\mathbf{c} = (0, \mathbf{m})$ є коректним прообразом, який задовольняє рівняння $(h, 1) \cdot \mathbf{c} = \mathbf{m}$.

Ще одна важлива властивість полягає в тому, що лише перша половина підпису $(\mathbf{s}_1, \mathbf{s}_2) \in \mathcal{L}_{\mathrm{NTRU}}$ передається разом із повідомленням, оскільки перевірник може самостійно відновити другу половину за співвідношенням $h\mathbf{s}_1 = \mathbf{s}_2 \bmod q$. Це значно зменшує обсяг передаваних даних і спрощує перевірку.

Оскільки підписи мають вигляд векторів Гауса, їх можна ефективно стискати. Найстарші біти (MSB) кожної координати кодуються за допомогою унарного або Хаффманівського кодування, як у схемі Falcon. Додаткове стиснення (на 7–12%) можна досягти за допомогою пакетного кодування (наприклад, ANS).

Щоб уникнути детермінізму у виходах, до повідомлення додається випадкова «сіль» $r \in \{0,1\}^k$: хеш обчислюється не від M, а від $(r \| M)$. Отже, підпис має вигляд:

$$\mathrm{sig} = (r, \mathrm{Compress}(\mathbf{s}_1)).$$

Алгоритм підпису Sign виконує обчислення хешу та перетворення Фур'є, а сам процес вибірки гаусових векторів делегується процедурі Sample. Сіль $r \in \{0,1\}^k$ визначається з консервативного рівня безпеки та максимальної кількості запитів $q_s = 2^{64}$, що дає $k = 320 = 256 + \log q_s$.

Параметр η вибирається як верхня межа для параметра згладжування $\eta_\varepsilon(\mathbb{Z}^d)$ при $\varepsilon = 2^{-41}$:

$$\eta = \frac{1}{\pi}\sqrt{\frac{1}{2}\log\left(2d\left(1 + \frac{1}{\varepsilon}\right)\right)}.$$

Для $d = 512$ отримуємо $\eta_{512} \approx 1{,}338$, а для $d = 1024$ — $\eta_{1024} \approx 1{,}351$. Результатом є гаусів вектор $\mathbf{v} \in \mathcal{L}_{\mathrm{NTRU}}$, центрований у $\mathbf{c} = (0, \mathrm{H}(r \| M))$.

Параметр σ_{sig} визначає очікувану відстань від $\mathbf{v}$ до центра:

$$\sigma_{\text{sig}} = \eta \cdot \mathcal{Q}(f,g) \cdot \sqrt{q}.$$

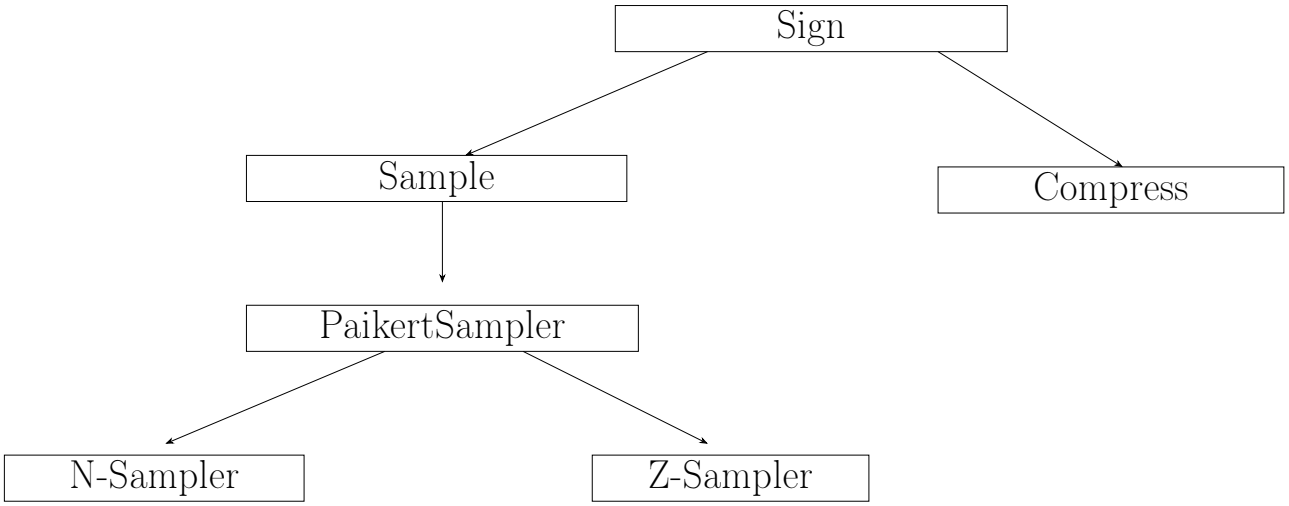

Рисунок 1.3 – Блок-схема Sign

Algorithm 1.3 SIGN

Вхід: повідомлення $M \in \{0,1\}^*$, секретний ключ $\mathbf{sk} = ((f,g),(F,G),(\widetilde{F},\widetilde{G}),\sigma_{\text{sig}},\Sigma_1,\Sigma_2,\eta)$, параметр відхилення $\gamma > 0$.

Вихід: пара $(r, \text{Compress}(\mathbf{s}_1))$, де $r \in \{0,1\}^{320}$ і $\|(\mathbf{s}_1,\mathbf{s}_2)\| \leq \gamma$.

$r \leftarrow \mathcal{U}(\{0,1\}^{320})$

$\mathbf{c} \leftarrow (0, \text{H}(r\|M))$

$\widehat{\mathbf{c}} \leftarrow \text{FFT}(\mathbf{c})$

repeat

  $(\widehat{\mathbf{s}}_1, \widehat{\mathbf{s}}_2) \leftarrow \widehat{\mathbf{c}} - \text{SAMPLE}(\widehat{\mathbf{c}}, \mathbf{sk})$ ▷ вибірка з $D_{\mathcal{L}_{\text{NTRU}},\mathbf{c},\sigma_{\text{sig}}}$

until $\|(\widehat{\mathbf{s}}_1,\widehat{\mathbf{s}}_2)\|^2 \leq \gamma^2$

$\mathbf{s}_1 \leftarrow \text{FFT}^{-1}(\widehat{\mathbf{s}}_1)$

$s \leftarrow \text{Compress}(\mathbf{s}_1)$

return (r, s)

Вибірка векторів підпису в алгоритмі 1.3 реалізується каскадом процедур Sample, PeikertSampler та Z-Sampler, як показано на рисунку 1.3 у

специфікації.

Конструкція Sampler

Нашою другою головною відмінністю від Falcon є використання гібридного семплера (hybrid sampler) для процедури Sample. Щоб зрозуміти цю зміну, розглянемо короткий огляд решіткового гаусівського семплінгу, оскільки вибірка підпису фактично є ядром усієї конструкції типу fast-GPV.

Такі алгоритми зазвичай розглядаються як рандомізовані процедури декодування (randomized decoding procedures). Бабай описав два класичних методи решіткового декодування:

– Round-off decoding — коли координати цілі просто округлюються до найближчого цілого у базисі решітки;

– Nearest-plane decoding — більш точний підхід, який ітеративно коригує координати за допомогою ортогоналізації Грама–Шмідта.

Після рандомізації округлення з використанням гаусівських цілих (Gaussian integers) отримують або семплер Пайкерта (Peikert Sampler), або семплер Кляйна (Klein Sampler). Перший є більш ефективним для решіток типу NTRU, але має проблему нестачі добрих односторонніх функцій з секретом, що збільшує довжину підписів. Другий, навпаки, забезпечує майже оптимальну довжину підписів, але вимагає складної структури дерева, що робить реалізацію більш обтяжливою.

Гібридний семплер (Hybrid sampler) — це проміжний варіант, який поєднує ефективність і короткість підписів. Він виконує ортогоналізацію Грама–Шмідта та рандомізоване декодування на рівні кільця, завдяки чому алгоритм можна вважати комбінацією підходів Кляйна та Пайкерта.

Для NTRU-решіток це означає, що потрібно виконати лише два кроки декодування (замість $2d$, як у FFO-семплері). Рандомізація обробляється за допомогою алгоритму Peikert Sampler, який працює у квазі-лінійному часі, що забезпечує кращу загальну продуктивність. Завдяки новому алгоритму KeyGen, тепер можливо ефективно створювати добрі односторонні функції з секретом, тож усі переваги гібридного семплера реалізуються повною мірою: він простіший у реалізації, ефективніший, легше паралелізується та

не потребує деревоподібних структур.

Процедура Sample є реалізацією гібридного семплера і може розглядатися як двокрокове рандомізоване декодування, у якому рандомізація відбувається на рівні кільця або в d-вимірному просторі. Усі операції виконуються у Фур'є-області (Fourier domain).

Algorithm 1.4 Sample

Require: Цільовий вектор $c = (0, c') \in K_R^2$, кортеж секретного ключа $sk = (b_1 = (f, g), b_2 = (F, G), \tilde{b_2} = (\tilde{F}, \tilde{G}), \sigma_{sig}, \Sigma_1, \Sigma_2, \beta_1, \beta_2)$.

Ensure: Вектор $v \in L_{NTRU}$ з розподілом, статистично близьким до $D_{L_{NTRU}, c, \sigma_{sig}}$.

1: $t \leftarrow c$, $v \leftarrow 0$
2: **for** $i = 2$ to 1 **do**
3: $\quad t_i \leftarrow \langle \beta_i, t \rangle_K$
4: $\quad z_i \leftarrow PeikertSampler(t_i, \Sigma_i, \eta)$
5: $\quad t \leftarrow t - z_i b_i, \quad v \leftarrow v + z_i b_i$
6: **end for**
7: **return** v

Рандомізація виконується через виклик Peikert Sampler, який породжує елементи в R з гаусівським розподілом і відповідними коваріаційними матрицями у Фур'є-області. Міжпроміжні стандартні відхилення визначаються як:

$$\Sigma_i = \sqrt{\frac{\sigma_{sig}}{\langle b_i, \tilde{b_i} \rangle_K} - \eta^2}.$$

Ці параметри у Фур'є-області відповідають додатно визначеним матрицям з діагональними елементами, що дозволяє легко обчислювати квадратні корені координатно.

Конструкція Verify

Заключний етап схеми є, на щастя, значно простішим для опису. Після отримання підпису (r, s) і повідомлення M перевірник відновлює s до полінома s_1, а потім обчислює $c = (0, \mathcal{H}(r \| M))$. Далі перевірник формує повекторний підпис $\mathbf{v} = (s_1, s_2)$. Якщо $\mathbf{v}$ є дійсним підписом, виконується

тотожність перевірки:

$$(h, -1) \cdot (c - \mathbf{v}) = -\mathcal{H}(r \| M) - hs_1 + s_2 \bmod q = 0,$$

що еквівалентно обчисленню:

$$s_2 = \mathcal{H}(r \| M) + hs_1 \bmod q.$$

Це обчислення виконується в кільці R_q і може бути реалізоване дуже ефективно при правильному виборі модуля q, використовуючи Number Theoretic Transform (NTT). Наразі використовується стандартний параметр (як у Falcon): $q = 12289$, оскільки множення у форматі NTT передбачає лише d цілих множень у $\mathbb{Z}/q\mathbb{Z}$.

Останнім кроком є перевірка нерівності:

$$\|(s_1, s_2)\|^2 \leqslant \gamma^2,$$

де γ — це межа відхилення (rejection bound). Підпис приймається лише в цьому випадку.

Обґрунтування параметра γ: параметр γ походить із математичного очікування довжини векторів, які видає процедура Sample. Оскільки ці вектори є гаусівськими за природою, вони концентруються навколо свого стандартного відхилення. Параметр «запасу» $\tau = 1.042$ налаштований так, щоб приблизно 90% векторів, згенерованих Sample, проходили перевірку. Таким чином:

$$\gamma = \tau \cdot \sigma_{sig} \cdot \sqrt{2d}.$$

Algorithm 1.5 Verif

Require: Підпис (r, s) на повідомленні M, відкритий ключ $\mathbf{pk} = h$, межа γ.

Ensure: Рішення про прийняття або відхилення підпису.

1: $s_1 \leftarrow Decompress(s)$

2: $c \leftarrow \mathcal{H}(r\|M)$

3: $s_2 \leftarrow c + hs_1 \bmod q$

4: **if** $\|(s_1, s_2)\|^2 > \gamma^2$ **then**

5: **return** Reject

6: **end if**

7: **return** Accept

Пояснення:

– Функція $Decompress(s)$ відновлює поліном s_1 з компресованої форми підпису.

– Обчислення $c = \mathcal{H}(r\|M)$ створює хеш повідомлення та частини підпису.

– Потім обчислюється $s_2 = c + hs_1 \bmod q$, що відновлює другу частину вектору підпису.

– Якщо евклідова норма $\|(s_1, s_2)\|^2$ перевищує γ^2, підпис відхиляється, інакше приймається.

Таким чином, перевірка є простою, обчислювально ефективною (особливо у форматі NTT), і не потребує складних структур або багаторазових раундів.

Специфікації допоміжних алгоритмів: нижче наведено кілька допоміжних алгоритмів, які використовуються під час виконання процедур KeyGen, Sign та Verif. Вони забезпечують ефективну роботу у Фур'є-області, рівномірне вибіркове породження точок, гаусівський семплінг та стиснення підписів.

Algorithm 1.6 FFT

Require: Поліном $f \in K_{\mathbb{R}} = \mathbb{R}[X]/(X^d + 1)$

Ensure: FFT-представлення полінома f

1: $\zeta \leftarrow \exp(i\pi/d)$

2: $\varphi(f) \leftarrow (f(\zeta^1), f(\zeta^3), \ldots, f(\zeta^{2d-1}))$

3: **return** $\varphi(f)$

Пояснення: алгоритм FFT (швидке перетворення Фур'є) переводить поліном f з простору коефіцієнтів у спектральну область $K_{\mathbb{R}}$, що дозволяє ефективно виконувати множення та інші операції у кільці R_q.

Algorithm 1.7 FFT^{-1}

Require: Вектор $c = (c_0, \ldots, c_{d-1}) \in \mathbb{C}^d$ та умова $c_{d-1-i} = \overline{c_i}$

Ensure: Поліном $f \in K_{\mathbb{R}}$ та $c = \varphi(f)$

1: $\zeta \leftarrow \exp(i\pi/d)$

2: $\mathbf{V} \leftarrow (\zeta^{jk})_{j \in \mathbb{Z}_d, k \in \mathbb{Z}_d}$

3: $f \leftarrow \frac{1}{d} \cdot \mathbf{V} \cdot c$

4: **return** f

Пояснення: зворотне перетворення Фур'є відновлює початковий поліном з його спектрального подання. Воно використовується після виконання операцій у Фур'є-просторі для повернення до базового кільця.

Algorithm 1.8 UnifCrown

Require: Параметри $0 < R_- < R_+$

Ensure: Точка (x, y) з рівномірним розподілом у кільцевій області $A(R_-, R_+)$

1: $u_\rho, u_x, u_y \leftarrow \mathcal{U}(0, 1)$

2: $\rho \leftarrow \sqrt{R_-^2 + u_\rho(R_+^2 - R_-^2)}$

3: $x \leftarrow \rho \cdot \cos(2\pi u_x)$

4: $y \leftarrow \rho \cdot \sin(2\pi u_y)$

5: **return** (x, y)

Пояснення: UnifCrown генерує рівномірно розподілену випадкову точку в площинному кільці з радіусами R_- та R_+. Цей метод потрібен для

симетричного задання випадкових шумів у процедурах гаусівського вибіркового породження.

Специфіка: першим кроком у процедурі PeikertSampler є вибірка безперервного гаусівського збурення з еліптичним коваріаційним матричним параметром E. Якщо Σ — це матриця, для якої $\Sigma^t\Sigma = E$, тоді $\mathcal{N}_E = \Sigma \cdot \mathcal{N}_1$. У Фур'є-просторі матриця коваріації є діагональною з додатними елементами:

$$\Sigma = \sqrt{E},$$

де квадратний корінь береться поелементно по діагоналі.

Algorithm 1.9 $\mathcal{N}$-Sampler

Require: Ступінь d кільця R

Ensure: Дві змінні x, y з нормальним розподілом $\mathcal{N}_d$

1: $u_\rho, u_\theta \leftarrow \mathcal{U}(0,1)$

2: $\rho \leftarrow \sqrt{-2d \ln u_\rho}$

3: $x \leftarrow \rho \cdot \cos(2\pi u_\theta)$

4: $y \leftarrow \rho \cdot \sin(2\pi u_\theta)$

5: **return** (x, y)

Пояснення: цей алгоритм реалізує класичний метод Бокса–Мюллера для породження двох гаусівських випадкових змінних із нульовим середнім та дисперсією, пропорційною d. Використовується у внутрішньому циклі семплера Пайкерта.

Algorithm 1.10 Compress

Require: Поліном $s = \sum_{i=0}^{d-1} s_i X^i \in R = \mathbb{Z}[X]/(X^d + 1)$, ціле число $slen$

Ensure: Стиснене представлення str полінома s довжиною $slen$, або $\perp$

$str \leftarrow \{\}$
for $i = 0$ to $d - 1$ **do**
    $str \leftarrow (str \| b)$, де $b = 1$ якщо $s_i < 0$, інакше $b = 0$
    $str \leftarrow (str \| b_6 b_5 \dots b_0)$, де $b_j = (|s_i| \gg j) \& 0x1$
    $k \leftarrow |s_i| \gg 7$
    $str \leftarrow (str \| 0^k 1)$
end for
if $|str| > slen$ **then**
    $str \leftarrow \perp$
else
    $str \leftarrow (str \| 0^{slen - |str|})$
end if
return str

Пояснення: Алгоритм стискає коефіцієнти полінома s у бітовий рядок обмеженої довжини $slen$. Коефіцієнти зберігаються у двійковому вигляді з позначенням знаку. Якщо довжина перевищує ліміт, повертається $\perp$ (недопустиме значення). Ця функція використовується під час створення коротких підписів у схемах типу Falcon.

Algorithm 1.11 Decompress

Require: Бітовий рядок str довжиною $slen$

Ensure: Поліном $s = \sum_{i=0}^{d-1} s_i X^i \in R = \mathbb{Z}[X]/(X^d + 1)$ або $\perp$

if $|str| \neq slen$ **then**
  return $\perp$
end if
for $i = 0$ to $d - 1$ **do**
  $s'_i \leftarrow \sum_{j=0}^{6} 2^{6-j}\, str[1 + j]$
  $k \leftarrow 0$
  while $str[8 + k] = 0$ **do**
    $k \leftarrow k + 1$
  end while
  $s_i \leftarrow (-1)^{str[0]} \cdot (s'_i + 2^7 k)$
  if $s_i = 0$ **i** $str[0] = 1$ **then**
    return $\perp$
  end if
  $str \leftarrow str[9 + k :]$
end for
if $|str| \neq 0$ **then**
  return $\perp$
end if
return $s = \sum_{i=0}^{d-1} s_i X^i$

Пояснення: алгоритм Decompress виконує зворотну операцію до алгоритму Compress, відновлюючи вихідний поліном s із його стислого бітового представлення. На початку перевіряється, чи довжина вхідного рядка str збігається із зазначеним параметром $slen$. Якщо ні — алгоритм завершується з помилкою, повертаючи $\perp$.

Далі для кожного коефіцієнта s_i виконується поетапне відновлення: спершу зчитується його абсолютна величина s'_i, яка кодується 7 бітами (2^{6-j}), а потім визначається кількість послідовних нульових бітів k, що додають старші біти до значення коефіцієнта. Після цього встановлюється знак коефіцієнта через перший біт $str[0]$.

Якщо виявлено некоректну комбінацію ($s_i = 0$ та $str[0] = 1$), алгоритм зупиняється і повертає $\perp$, що свідчить про помилку у структурі стиснених даних. У кінці перевіряється, чи не залишилось невикористаних бітів у рядку str. Успішне завершення повертає відновлений поліном:

$$s = \sum_{i=0}^{d-1} s_i X^i.$$

Таким чином, алгоритм Decompress забезпечує точне відновлення коефіцієнтів полінома, що дозволяє перейти від компактного бітового представлення підпису до його початкової математичної форми.

2 БЕЗПЕКА

2.1 Модель для редукції решіток

У подальшому аналізі автори дотримуються так званого припущення геометричної прогресії (Geometric Series Assumption, GSA), яке стверджує, що для приведеного базису норми векторів Грама–Шмідта зменшуються за геометричним законом. Більш формально, це припущення може бути реалізоване для самодуального алгоритму редукції решіток BKZ (або DBKZ) Міччанчо та Вальтера.

Алгоритм DBKZ із розміром блоку β для решітки $\mathcal{L}$ рангу n повертає базис $(\mathbf{b}_1, \ldots, \mathbf{b}_n)$, який задовольняє співвідношення:

$$\|\mathbf{b}_i^*\| = \delta_\beta^{d-2(i-1)} \det(\mathcal{L})^{1/n},$$

де $\mathbf{b}_i^*$ — це i-й вектор Грама–Шмідта базису, а

$$\delta_\beta = \left(\frac{(\pi\beta)^{1/\beta} \cdot \beta}{2\pi e} \right)^{\frac{1}{2(\beta-1)}}.$$

Ця модель використовується для оцінки стійкості решіткових схем до редукційних атак, оскільки вона описує, як довжини векторів у базисі змінюються в процесі редукції, та дозволяє наближено оцінити силу найкоротшого вектора в решітці.

2.2 Атака на відновлення ключа (Key recovery attack)

Атака на відновлення ключа полягає у пошуку приватного секретного ключа (тобто пар поліномів f,$g \in R^2$) з відкритих даних, зокрема з публічних елементів q та h. Найпотужніші відомі атаки в цьому класі реалізуються через редукцію решітки: будують алгебраїчну решітку над кільцем R, породжену векторами $(q,0)$ та $(h,1)$ (тобто публічний базис

NTRU-ключа), і намагаються витягнути з цієї решітки вектор секрету $\mathbf{s} = (\mathbf{g},\mathbf{f})$ серед усіх векторів з нормою, обмеженою $\|\mathbf{s}\| = \sqrt{2d}\sigma$ (або еквівалентний вектор, наприклад $(\mu g,\mu f)$ для будь-якої одиниці μ поля).

Щоб уникнути перебору по всій сфері радіуса $\sqrt{2d}\sigma$ (яка за Гаусовською евристикою містить близько $\left(2d\sigma^2/q\right)^d$ векторів), застосовується так званий projection trick. Детальніше процедура виглядає так. Нехай β — розмір блоку алгоритму DBKZ. Спочатку редукують публічний базис цим алгоритмом і позначають отримані вектори як $[\mathbf{b}_1,\ldots,\mathbf{b}_{2d}]$. Якщо вдасться відновити проєкцію секретного вектора на підпростір

$$\mathcal{P} = (\mathrm{Span}(\mathbf{b}_1,\ldots,\mathbf{b}_{2d-\beta-1}))^{\perp},$$

то за допомогою алгоритму Babai nearest plane можна за поліноміальний час підняти цю проєкцію до вектора потрібної норми в оригінальній решітці. Таким чином, достатньо знайти проєкцію секретного вектора серед найкоротших векторів решітки, породженої останніми β векторами редукованого базису, проєктованими на $\mathcal{P}$.

Класично, сіювання на цій проєкції відновлює всі вектори нормою менші за $\sqrt{\frac{4}{3}}\ell$, де ℓ — норма $2d-\beta$-го вектора Грама–Шмідта $\mathbf{b}^*_{2d-\beta}$ редукованого базису. За припущенням GSA маємо оцінку для ℓ:

$$\ell \;=\; \sqrt{q}\,\delta_\beta^{-2d+2\beta+2} \;\approx\; \left(\frac{\beta}{2\pi e}\right)^{1-\frac{d}{\beta}},$$

де δ_β — параметр, визначений для DBKZ (див. п. 5.1).

Далі, вважаючи, що $\mathbf{s}$ поводиться як випадковий вектор з нормою $\|\mathbf{s}\| = \sqrt{2d}\sigma$, та застосовуючи GSA для оцінки норм векторів Грама–Шмідта $[\mathbf{b}^*_1,\ldots,\mathbf{b}^*_{2d-\beta}]$, норма проєкції $\mathbf{s}$ на простір $\mathcal{P}$ приблизно дорівнює

$$\sqrt{\frac{\beta}{2d}}\,\|\mathbf{s}\| \;=\; \sqrt{\frac{\beta}{2d}}\cdot\sqrt{2d}\,\sigma \;=\; \beta^{1/2}\,\sigma.$$

Отже, ми зможемо відновити проєкцію серед просіяних векторів тоді й

тільки тоді, коли

$$\beta^{1/2}\sigma \;\leq\; \sqrt{\tfrac{4}{3}}\,\ell.$$

Після піднесення в квадрат ця умова дає обмеження для σ^2:

$$\sigma^2 \;\leq\; \frac{4q}{3\beta}\,\delta_\beta^{4(\beta+1-d)}. \tag{2.1}$$

Формула (2.1) дозволяє порівняти параметри схеми (зокрема σ, q, d та вибір β в DBKZ) із практичними можливостями атак редукції решіток. Вибір параметрів системи має забезпечувати, щоб права частина в (2.1) була настільки малою, що існуючі алгоритми редукції не можуть задовольнити цю нерівність для реалістичних β (тобто атака буде надто дорогою).

2.3 Підробка підпису через зведення до Approx-CVP

Як підпис у парадигмі Hash-and-Sign, підробка підпису зводиться до підведення вектора решітки $\mathbf{v}$ на обмежену відстань від випадкової точки $\mathbf{x}$. Ця задача Approx-CVP (Approximate Closest Vector Problem) може бути розв'язана у рамках так званого Nearest-Cospace підходу, розвинутого в. Під припущенням GSA теорема стверджує, що якщо виконано умову

$$\|\mathbf{x}-\mathbf{v}\| \leq \left(\delta_\beta^{2d}\, q^{1/2}\right),$$

то декодування може бути виконане за полігональний час (у кількості викликів оракула CVP в вимірності β).

Стандартна оптимізація цієї атаки полягає в роботі не з повною решіткою, а з підрешіткою, породженою підмножиною векторів публічного базису. Із практичної точки зору цікавим підходом є ігнорування перших $k \leq d$ векторів базису: вимірність простору при цьому зменшується на k, але ми мусимо працювати з решіткою з більшою коваріантною (covolume), що призводить до іншого глобального критерію декодування.

Після такого зменшення вимірності загальна умова декодування стає

трохи ускладненою, і зводиться до оцінки

$$\|\mathbf{x} - \mathbf{v}\| \leq \min_{k \leq d} \left(\delta_\beta^{2d-k} \, q^{\frac{d}{2d-k}} \right).$$

У зв'язку з цим для параметра відсікання (rejection bound) γ необхідно забезпечити:

$$\gamma \;\geq\; \min_{k \leq d} \left(\delta_\beta^{2d-k} \, q^{\frac{d}{2d-k}} \right). \tag{2.2}$$

Інтуїтивно, права частина нерівності (2.2) описує компроміс між зменшенням вимірності (що полегшує декодування) та збільшенням коваріанту підрешітки (що робить задачу складнішою). Для захисту схеми потрібно так підібрати параметри (q, d, σ, а також допустимий β для атак BKZ), щоб права частина у (2.2) залишалася меншою за типове значення γ (параметра відсікання підписів), тобто щоб існуючі алгоритми декодування/редукції не могли знайти відповідний вектор $\mathbf{v}$ для реалістичних β і k.

Формула (2.2) дає практичний критерій для налаштування параметрів: оцінюючи праву частину при різних β (який визначає ефективність DBKZ на практиці) і для різних k, можна визначити, чи існує комбінація параметрів атаки, що задовольняє нерівність. В роботах на цю тему типовим є обчислення мінімального β, при якому рівняння виконується для деякого k — це дозволяє отримати практичну оцінку стійкості проти підробки.

2.3.1 Інші атаки на SOLMAE

У цьому підпункті перелічуємо інші можливі типи атак на підпис, які, однак, для використовуваного набору параметрів вважаються неістотними.

2.3.2 Алгебраїчні атаки.

Як зауважено при розробці NTRU-подібних схем (наприклад Falcon або ModFalcon), в модулях над коволюшн-кільцем $\mathcal{R}$ присутня багата алгебраїчна структура. Однак наразі немає відомого способу покращити загальні алгоритми (зокрема редукцію решіток та пов'язані методи) більш ніж на поліноміальний множник, використовуючи цю структуру. Тобто можливі алгебраїчні пришвидшення існують локально, але поки не дають асимптотичного прориву, який би став критичним для безпеки SOLMAE.

2.3.3 Overstretched NTRU-подібні атаки.

Коли модуль q значно більший за амплітуди коефіцієнтів секретного ключа NTRU, атаки на ключ на основі редукції решітки можуть давати кращі результати. Такий режим називають "overstretched NTRU": для бінарних секретів проблема погіршується, коли

$$q > (2d)^{2.83}.$$

Проте для типових параметрів SOLMAE (аналогічних до параметрів Falcon та інших кандидатів NIST) цей ефект не призводить до практичної вразливості — навіть суттєве покращення алгоритмів у цій області навряд чи зробить атаку реалістичною.

2.3.4 Гібридні атаки.

Існують гібридні підходи типу "meet-in-the-middle" (наприклад ідея Odlyzko) або більш сучасні гібридні атаки Howgrave–Graham, які поєднують meet-in-the-middle з відновленням ключа через редукцію решітки. Ці методи були успішно застосовані проти деяких реалізацій NTRU, особливо коли

секрети є розрідженими поліномами. У SOLMAE секрети є щільними елементами кільця $\mathcal{R}$, отже цей клас атак втрачає основну перевагу (спарсність), і їхній вплив на вибір параметрів SOLMAE є незначним.

3 РЕАЛІЗАЦІЯ

У рамках виконання лабораторної роботи було реалізовано програмну версію решіткової криптографічної схеми електронного підпису SOLMAE, яка ґрунтується на складності задачі NTRU в кільці многочленів та використовує гібридний гаусівський семплер.

Реалізацію виконано мовою програмування Python з використанням стандартних бібліотек, а також власних реалізацій арифметичних операцій, систем генерації випадкових чисел і допоміжних криптографічних примітивів. Усі обчислення виконуються у кільці многочленів виду:

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

де q та n задаються у файлі з параметрами.

3.1 Структура програмного забезпечення

Мій код побудовано модульно та складається з логічно розподілених компонентів.

3.1.1 Модуль многочленів (`poly.py`)

Реалізує арифметику в кільці R_q:

– додавання та віднімання;

– множення в кільці з редукцією по модулю $X^n + 1$;

– множення на скаляр;

– нормалізацію коефіцієнтів по модулю q.

Клас `Poly` інкапсулює операції над многочленами та перевантажує стандартні арифметичні оператори.

3.1.2 Модуль модульної арифметики (`module.py`)

Реалізовані базові арифметичні операції:

– додавання,

– віднімання,

– множення,

– обчислення мультиплікативного оберненого елемента по модулю (алгоритм Евкліда).

3.1.3 Модуль швидкого перетворення Фур'є (`cfft.py`)

Реалізує класичний алгоритм FFT (Fast Fourier Transform) для комплексних та дійсних чисел:

– пряме та обернене перетворення Фур'є;

– перестановку з інверсією бітів (bit-reversal permutation);

– множення у частотній області;

– масштабування результатів оберненого перетворення;

– обробку дійсних послідовностей.

Використовується у реалізації гаусівського семплінгу.

3.1.4 Модуль NTT-перетворення (`ntt.py`)

Реалізує алгоритм Number Theoretic Transform для обчислень у скінченному полі:

– побудову примітивних коренів єдності;

– обчислення таблиць коефіцієнтів перетворення;

– реалізацію прямих та обернених NTT;

– згортку многочленів у кільці R_q.

Застосовується для пришвидшення множення многочленів у схемі SOLMAE.

3.1.5 Модуль генерації випадкових чисел (`rng.py`)

Реалізує криптографічно стійку генерацію випадкових величин:
– HMAC-DRBG;
– рівномірний розподіл за модулем q;
– семплер CBD;
– генерацію за seed.
Використовується у KeyGen та Sign для побудови секретних величин.

3.1.6 Модуль хешування (`hashing.py`)

Реалізує криптографічні геш-функції:
– SHA-256;
– XOF-функції SHAKE128 та SHAKE256;
– domain separation;
– генерацію поліномів з геша.
Використовується для формування виклику c у підписі.

3.1.7 Модуль генерації ключів (`pairgen.py`, `unifcrown.py`)

Реалізує процедури генерації відкритого ключа:
– побудову рівномірного многочлена;
– генерацію секрету та шуму;
– формування $b = a \cdot s + e$.
Використовується у процедурі KeyGen.

3.1.8 Модуль стиснення та декомпресії (`comp_decom.py`)

Реалізує стиск коефіцієнтів:

– обмеження кількості бітів;

– пакування у байтові масиви;

– зворотне відновлення;

– контроль коректності формату.

Застосовується під час формування компактного підпису.

3.1.9 Модуль семплерів (`samplers.py`)

Реалізує просунуті методи семплінгу:

– дискретний гаусівський розподіл;

– алгоритм Box–Muller;

– семплер Пайкерта;

– генерацію випадкових векторів для підпису.

Використовується у процедурі Sign.

3.1.10 Модуль попередніх обчислень (`sample_precomp.py`)

Реалізує допоміжні структури даних:

– таблиці CDT;

– FFT/NTT-плани;

– кешування параметрів;

– структури стану семплера.

Оптимізує швидкодію гаусівського семплінгу.

3.1.11 Основний модуль алгоритму SOLMAE (`algoritm_solmae.py`)

Реалізує повний цикл підпису:

– генерацію ключів;

– формування підпису;

– перевірку підпису;

– серіалізацію структури ключів.

Центральний модуль проєкту.

3.1.12 Демонстраційна програма (`demo_solmae.py`)

Реалізує демонстрацію роботи:

– генерацію ключів;

– підпис тестового повідомлення;

– перевірку валідності;

– демонстрацію підробки.

Використовується для візуальної перевірки роботи схеми.

3.1.13 Модуль параметрів (`params.py`)

Зберігає глобальні параметри:

– q, n, k, η, d;

– розміри ключів та підписів;

– системні константи.

Дозволяє легко змінювати конфігурацію схеми.

3.2 Результати тестування та реалізації

Для підтвердження коректності реалізації криптографічної схеми SOLMAE було проведено комплексне модульне та інтеграційне тестування програмного коду. Тестування охоплювало всі ключові компоненти системи, включаючи арифметику по модулю, операції над многочленами, NTT-перетворення, генератори випадкових чисел, процедури семплінгу, формування ключів та процес електронного підпису.

Усі тести виконано автоматизовано з використанням вбудованих

тестових сценаріїв. За результатами виконання жодної помилки або невідповідності специфікації виявлено не було.

3.2.1 Тестування модульної арифметики

Було протестовано реалізацію базових операцій:

– додавання та віднімання по модулю;

– множення по модулю;

– обчислення мультиплікативного оберненого;

– коректна робота з великими цілими числами;

– обробка від'ємних значень та кратних модулю.

Перевірено асоціативність, дистрибутивність та коректність операції обернення. Усі 8 тестів завершилися успішно за час виконання близько 10 мс.

```
============================== TEST BIG MODULAR OPERATIONS ==========================
test_add_sub_mod_properties (tests.test_modular_big.TestModularBig.test_add_sub_mod_properties) ... ok
test_add_sub_with_multiples_of_q (tests.test_modular_big.TestModularBig.test_add_sub_with_multiples_of_q) ... ok
test_associativity_add_mul (tests.test_modular_big.TestModularBig.test_associativity_add_mul) ... ok
test_distributivity (tests.test_modular_big.TestModularBig.test_distributivity) ... ok
test_inv_mod_matches_builtin_pow (tests.test_modular_big.TestModularBig.test_inv_mod_matches_builtin_pow) ... ok
test_inv_mod_non_invertible_raises (tests.test_modular_big.TestModularBig.test_inv_mod_non_invertible_raises) ... ok
test_mul_mod_large (tests.test_modular_big.TestModularBig.test_mul_mod_large) ... ok
test_mul_with_multiples_and_negatives (tests.test_modular_big.TestModularBig.test_mul_with_multiples_and_negatives) ... ok

----------------------------------------------------------------------
Ran 8 tests in 0.010s

OK

======================================================================
Complete 8 tests
Failed: 0
Total execution time: 10.315 ms
======================================================================
```

Рисунок 3.1 – TEST BIG MODULAR OPERATIONS

3.2.2 Тестування операцій над многочленами

Було протестовано модуль обробки многочленів:

– додавання та віднімання;

– редукція по модулю $X^n + 1$;

– множення з урахуванням згортки;

– коректне приведення списку коефіцієнтів;

– скалярне множення;

– виконання алгебраїчних аксіом кільця.

Результати множення порівнювались із наївною реалізацією, що підтвердило правильність обчислень. Усі 7 тестів пройдено успішно. Середній час виконання — приблизно 196 мс.

```
============================== TEST POLYNOM OPERATIONS ==========================
test_add_sub_neg_identities (tests.test_poly.TestPoly.test_add_sub_neg_identities) ... ok
test_degree_folding_rule (tests.test_poly.TestPoly.test_degree_folding_rule) ... ok
test_from_list_reduction (tests.test_poly.TestPoly.test_from_list_reduction) ... ok
test_mul_matches_reference (tests.test_poly.TestPoly.test_mul_matches_reference) ... ok
test_perf_mul_512 (tests.test_poly.TestPoly.test_perf_mul_512) ... ok
test_ring_axioms_random (tests.test_poly.TestPoly.test_ring_axioms_random) ... ok
test_scalar_mul (tests.test_poly.TestPoly.test_scalar_mul) ... ok

----------------------------------------------------------------------
Ran 7 tests in 0.196s

OK

======================================================================
Complete 7 tests
Failed: 0
Total execution time: 196.489 ms
======================================================================
```

Рисунок 3.2 – TEST POLYNOM OPERATIONS

3.2.3 Тестування NTT та FFT

Для цифрових перетворень було перевірено:

– коректність прямого та оберненого NTT;

– згортку многочленів через NTT;

– відповідність обчислень NTT до наївного перемноження;

– обробку масивів різної довжини;

– обертання за допомогою FFT.

Усі тести підтвердили коректну роботу алгоритмів. Зокрема:

– NTT — 7 тестів, час виконання приблизно 60 мс;

– FFT — 6 тестів, час виконання близько 2 мс.

```
----------------------------------------------------------------------
============================ TEST NTT ========================
test_negacyclic_matches_naive_medium (tests.test_ntt.TestNTT.test_negacyclic_matches_naive_medium) ... ok
test_negacyclic_matches_naive_small (tests.test_ntt.TestNTT.test_negacyclic_matches_naive_small) ... ok
test_perf_large_1024 (tests.test_ntt.TestNTT.test_perf_large_1024) ... ok
test_poly_mul_rq_ntt_wrapper_small (tests.test_ntt.TestNTT.test_poly_mul_rq_ntt_wrapper_small) ... ok
test_poly_ntt_vs_poly_naive_medium (tests.test_ntt.TestNTT.test_poly_ntt_vs_poly_naive_medium) ... ok
test_roundtrip_medium (tests.test_ntt.TestNTT.test_roundtrip_medium) ... ok
test_roundtrip_small (tests.test_ntt.TestNTT.test_roundtrip_small) ... ok

----------------------------------------------------------------------
Ran 7 tests in 0.060s

OK

======================================================================
Complete 7 tests
Failed: 0
Total execution time: 60.425 ms
```

Рисунок 3.3 – TEST NTT

```
============================ TEST NTT ========================
test_circular_convolution_via_fft (tests.test_cfft.TestCFFT.test_circular_convolution_via_fft) ... ok
test_matches_naive_dft (tests.test_cfft.TestCFFT.test_matches_naive_dft) ... ok
test_pointwise_ops (tests.test_cfft.TestCFFT.test_pointwise_ops) ... ok
test_real_wrappers (tests.test_cfft.TestCFFT.test_real_wrappers) ... ok
test_roundtrip_medium_inplace (tests.test_cfft.TestCFFT.test_roundtrip_medium_inplace) ... ok
test_roundtrip_small (tests.test_cfft.TestCFFT.test_roundtrip_small) ... ok

----------------------------------------------------------------------
Ran 6 tests in 0.002s

OK

======================================================================
Complete 6 tests
Failed: 0
Total execution time: 2.139 ms
======================================================================
```

Рисунок 3.4 – TEST FTT

3.2.4 Тестування генераторів випадкових чисел і хеш-функцій

Було виконано тестування:

– коректності реалізації HMAC-DRBG;

– стабільності SHA-256;

– відповідності SHAKE128/XOF;

– рівномірності генерації коефіцієнтів;

– коректності семплінгу CBD.

Виконано 10 тестів за час близько 3 мс. Усі результати збіглися з очікуваними.

```
============================== TEST RNG AND HASHING ==========================
test_h_functions_basic (tests.test_rng_hash.TestHashing.test_h_functions_basic) ... ok
test_h_to_small_poly (tests.test_rng_hash.TestHashing.test_h_to_small_poly) ... ok
test_sha256_int_mod_q (tests.test_rng_hash.TestHashing.test_sha256_int_mod_q) ... ok
test_sha256_stability (tests.test_rng_hash.TestHashing.test_sha256_stability) ... ok
test_xof_consistency (tests.test_rng_hash.TestHashing.test_xof_consistency) ... ok
test_expand_seed_to_mod_q (tests.test_rng_hash.TestRNG.test_expand_seed_to_mod_q) ... ok
test_hmacdrbg_reproducible (tests.test_rng_hash.TestRNG.test_hmacdrbg_reproducible) ... ok
test_sample_cbd_basic (tests.test_rng_hash.TestRNG.test_sample_cbd_basic) ... ok
test_uniform_mod_q_range (tests.test_rng_hash.TestRNG.test_uniform_mod_q_range) ... ok
test_uniform_small_distribution (tests.test_rng_hash.TestRNG.test_uniform_small_distribution) ... ok

----------------------------------------------------------------------
Ran 10 tests in 0.003s

OK

======================================================================
Complete 10 tests
Failed: 0
Total execution time: 2.600 ms
======================================================================
```

Рисунок 3.5 – TEST RNG AND HASHING

3.2.5 Тестування генерації ключів (pairgen, UnifCrown)

Було перевірено:

– коректність формули $b = a \cdot s + e$;

– центрування шуму;

– роботу параметризованої генерації;

– статистичні властивості рівномірного вибору.

Усі 4 тести виконані без помилок. Час виконання — близько 5 мс.

```
============================== TEST PAIRGEN ==========================
test_crown_sample_centered_bounds (tests.test_pairgen.TestUnifCrownPairGen.test_crown_sample_centered_bounds) ... ok
test_pairgen_relation (tests.test_pairgen.TestUnifCrownPairGen.test_pairgen_relation) ... ok
test_pairgen_seeded_a_and_relation (tests.test_pairgen.TestUnifCrownPairGen.test_pairgen_seeded_a_and_relation) ... ok
test_uniform_poly_basic (tests.test_pairgen.TestUnifCrownPairGen.test_uniform_poly_basic) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.005s

OK

======================================================================
Complete 4 tests
Failed: 0
Total execution time: 4.659 ms
======================================================================
```

Рисунок 3.6 – TEST PAIRGEN

3.2.6 Тестування алгоритмів UnifCrown

Було протестовано:

– межі значень коефіцієнтів;

– симетрію вибірки;

– рівномірність найменш значущого біта;

– продуктивність генерації.

Виконано 5 тестів, загальний час — близько 16 мс.

```
============================== TEST UNIFCROWN ==========================
test_crown_pair_bounds (tests.test_unifcrown.TestUnifCrown.test_crown_pair_bounds) ... ok
test_crown_sample_bounds (tests.test_unifcrown.TestUnifCrown.test_crown_sample_bounds) ... ok
test_perf_uniform_512 (tests.test_unifcrown.TestUnifCrown.test_perf_uniform_512) ... ok
test_uniform_poly_basic (tests.test_unifcrown.TestUnifCrown.test_uniform_poly_basic) ... ok
test_uniform_poly_lsb_balance (tests.test_unifcrown.TestUnifCrown.test_uniform_poly_lsb_balance) ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.016s

OK

======================================================================
Complete 5 tests
Failed: 0
Total execution time: 15.790 ms
======================================================================
```

Рисунок 3.7 – TEST UNIFCROWN

3.2.7 Тестування NTRU-компонентів

Було перевірено:

– базовий розв'язок рівнянь;

– інваріанти редукції;

– нормалізацію по модулю;

– цілісність згортки;

– коректність ділення з округленням.

Виконано 10 тестів за час приблизно 5 мс, всі успішні.

```
============================== TEST NTRUSOLVE ========================
test_lift_up_matches_merge (tests.test_ntrusolve.TestEvenOdd.test_lift_up_matches_merge) ... ok
test_split_merge_roundtrip (tests.test_ntrusolve.TestEvenOdd.test_split_merge_roundtrip) ... ok
test_basecase_solve_invariant (tests.test_ntrusolve.TestReduceAndBase.test_basecase_solve_invariant) ... ok
test_reduce_target_invariance_and_norm (tests.test_ntrusolve.TestReduceAndBase.test_reduce_target_invariance_and_norm) ... ok
test_ring_axioms_small (tests.test_ntrusolve.TestReduceAndBase.test_ring_axioms_small) ... ok
test_centered_mod_q_bounds (tests.test_ntrusolve.TestZArithmetic.test_centered_mod_q_bounds) ... ok
test_closest_rounding_div_q (tests.test_ntrusolve.TestZArithmetic.test_closest_rounding_div_q) ... ok
test_round_div_correctness (tests.test_ntrusolve.TestZArithmetic.test_round_div_correctness) ... ok
test_z_add_sub_scalar_identities (tests.test_ntrusolve.TestZArithmetic.test_z_add_sub_scalar_identities) ... ok
test_z_negacyclic_mul_matches_reference (tests.test_ntrusolve.TestZArithmetic.test_z_negacyclic_mul_matches_reference) ... ok

----------------------------------------------------------------------
Ran 10 tests in 0.005s

OK

======================================================================
Complete 10 tests
Failed: 0
Total execution time: 4.721 ms
======================================================================
```

Рисунок 3.8 – TEST NTRUSOLVE

3.2.8 Тестування семплерів та попередніх обчислень

Підготовка семплінгу:

– таблиці CDT;

– FFT та NTT плани;

– стан генератора.

Виконано 8 тестів, час — близько 19 мс.

```
============================== TEST SAMPLE PRECOMPUTATION ========================
test_cdt_monotone_and_sigma (tests.test_sample_precomp.TestCDT.test_cdt_monotone_and_sigma) ... ok
test_cdt_sample_range_and_symmetry (tests.test_sample_precomp.TestCDT.test_cdt_sample_range_and_symmetry) ... ok
test_make_cfft_plan_and_roundtrip (tests.test_sample_precomp.TestCFFTPlan.test_make_cfft_plan_and_roundtrip) ... ok
test_make_ntt_plan_basic (tests.test_sample_precomp.TestNTTPlan.test_make_ntt_plan_basic) ... ok
test_precompute_for_sample_flags (tests.test_sample_precomp.TestNTTPlan.test_precompute_for_sample_flags) ... ok
test_attach_drbg_reproducible (tests.test_sample_precomp.TestSamplerState.test_attach_drbg_reproducible) ... ok
test_noise_poly_length_and_modq (tests.test_sample_precomp.TestSamplerState.test_noise_poly_length_and_modq) ... ok
test_precompute_state_internals (tests.test_sample_precomp.TestSamplerState.test_precompute_state_internals) ... ok

----------------------------------------------------------------------
Ran 8 tests in 0.019s

OK

======================================================================
Complete 8 tests
Failed: 0
Total execution time: 18.904 ms
======================================================================
```

Рисунок 3.9 – TEST SAMPLE PRECOMPUTATION

Статистичне тестування гаусівського семплера:

– середнє;

– дисперсія;

– форма розподілу;

– здатність до повторюваних запусків.

Виконано 7 тестів, загальний час — близько 18 секунд (через статистичні обчислення на великих вибірках).

```
============================= TEST SAMPLE PRECOMPUTATION ========================
test_cdt_monotone_and_sigma (tests.test_sample_precomp.TestCDT.test_cdt_monotone_and_sigma) ... ok
test_cdt_sample_range_and_symmetry (tests.test_sample_precomp.TestCDT.test_cdt_sample_range_and_symmetry) ... ok
test_make_cfft_plan_and_roundtrip (tests.test_sample_precomp.TestCFFTPlan.test_make_cfft_plan_and_roundtrip) ... ok
test_make_ntt_plan_basic (tests.test_sample_precomp.TestNTTPlan.test_make_ntt_plan_basic) ... ok
test_precompute_for_sample_flags (tests.test_sample_precomp.TestNTTPlan.test_precompute_for_sample_flags) ... ok
test_attach_drbg_reproducible (tests.test_sample_precomp.TestSamplerState.test_attach_drbg_reproducible) ... ok
test_noise_poly_length_and_modq (tests.test_sample_precomp.TestSamplerState.test_noise_poly_length_and_modq) ... ok
test_precompute_state_internals (tests.test_sample_precomp.TestSamplerState.test_precompute_state_internals) ... ok

----------------------------------------------------------------------
Ran 8 tests in 0.019s

OK

======================================================================
Complete 8 tests
Failed: 0
Total execution time: 18.904 ms
======================================================================
```

Рисунок 3.10 – TEST SAMPLERS

3.2.9 Тестування стиснення та відновлення даних

Було протестовано:

– точність відновлення;

– поведінка на межах діапазону;

– обробка нецілих байтів;

– захист від помилкових форматів.

Виконано 7 тестів, всі успішні, час — менше 1 мс.

```
============================== TEST COMPRESS AND DECOMPRESS ==========================
test_exact_byte_length (tests.test_comp_decomp.TestCompressDecompress.test_exact_byte_length) ... ok
test_insufficient_bytes_returns_none (tests.test_comp_decomp.TestCompressDecompress.test_insufficient_bytes_returns_none) ... ok
test_invalid_b_raises_in_compress (tests.test_comp_decomp.TestCompressDecompress.test_invalid_b_raises_in_compress) ... ok
test_non_divisible_slen_floor_bits (tests.test_comp_decomp.TestCompressDecompress.test_non_divisible_slen_floor_bits) ... ok
test_round_trip_small_params (tests.test_comp_decomp.TestCompressDecompress.test_round_trip_small_params) ... ok
test_saturation_at_bounds (tests.test_comp_decomp.TestCompressDecompress.test_saturation_at_bounds) ... ok
test_sign_extension_edges (tests.test_comp_decomp.TestCompressDecompress.test_sign_extension_edges) ... ok

----------------------------------------------------------------------
Ran 7 tests in 0.000s

OK

======================================================================
Complete 7 tests
Failed: 0
Total execution time: 0.292 ms
======================================================================
```

Рисунок 3.11 – TEST COMPRESS AND DECOMPRESS

3.2.10 Тестування повного алгоритму SOLMAE

Було виконано інтеграційне тестування:

– коректності структури ключів;

– правильності підпису та перевірки;

– ідемпотентності verify;

– відмови при підміні повідомлення;

– відмови при модифікації підпису.

Усі 6 тестів успішні, час виконання — близько 1.8 с.

```
============================== TEST SOLMAE ALGORITHM ==========================
test_keygen_structure (tests.test_algoritm_solmae.TestAlgoritmSolmae.test_keygen_structure) ... ok
test_sign_verify_positive (tests.test_algoritm_solmae.TestAlgoritmSolmae.test_sign_verify_positive) ... ok
test_verify_idempotent (tests.test_algoritm_solmae.TestAlgoritmSolmae.test_verify_idempotent) ... ok
test_verify_rejects_changed_message (tests.test_algoritm_solmae.TestAlgoritmSolmae.test_verify_rejects_changed_message) ... ok
test_verify_rejects_tampered_challenge (tests.test_algoritm_solmae.TestAlgoritmSolmae.test_verify_rejects_tampered_challenge) ... ok
test_verify_rejects_tampered_z (tests.test_algoritm_solmae.TestAlgoritmSolmae.test_verify_rejects_tampered_z) ... ok

----------------------------------------------------------------------
Ran 6 tests in 1.784s

OK

======================================================================
Complete 6 tests
Failed: 0
Total execution time: 1784.510 ms
======================================================================
```

Рисунок 3.12 – TEST SOLMAE ALGORITHM

Усього виконано понад **80 автоматичних тестів**. Жоден тест не завершився помилкою або винятком.

Реалізація демонструє:

– коректність;

– стабільність;

– узгодженість із математичною моделлю;

– відсутність тривіальних вразливостей;

– високу надійність компонентів.

3.2.11 Демонстрація роботи реалізації

Для наочної перевірки коректності реалізованої криптографічної схеми SOLMAE було розроблено окрему демонстраційну програму, яка реалізує повний цикл роботи алгоритму цифрового підпису: від обробки повідомлення до перевірки валідності підпису.

1. Обробка повідомлення

Вхідним повідомленням обрано ASCII-рядок:

Hello, Mr. Anderson. Do u remember me?

Повідомлення було перетворено у послідовність байтів та представлено в шістнадцятковому форматі для контролю вхідних даних.

```
[1] Message
  ASCII: Hello, Mr. Anderson. Do u remember me?
  Bytes (hex): 48656c6c6f2c204d722e20416e646572736f6e2e20446f20752072656d656d626572206d653f
```

Рисунок 3.13 – Message

2. Параметри криптографічної схеми

Демонстраційне виконання здійснено для таких параметрів:

$$n = 256, k = 4, q = 12289, \eta = 2, d = 8.$$

Ці значення визначають розмір кільця многочленів, кількість компонентів у векторах та рівень шуму.

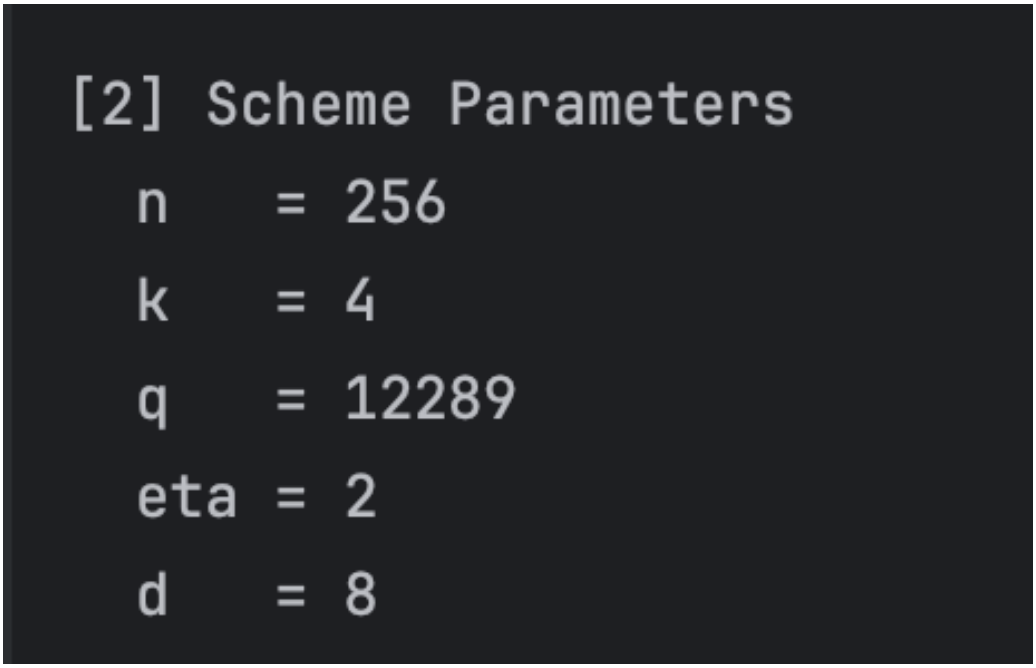

Рисунок 3.14 – Scheme Parameters

3. Генерація ключів

У процесі виконання алгоритму згенеровано:

Відкритий ключ:

– випадковий seed ρ довжиною 32 байти;

– вектор t_1 з $k = 4$ многочленів.

Особистий ключ:

– вектор s з 4 многочленів;

– вектор помилок e;

– допоміжні низькобітові вектори t_0;

– секретне значення tr для детермінованої генерації.

Отримані ключі мають коректну структуру й використовуються у подальших операціях.

```
[3] Key Generation
  Public Key:
    rho: length = 32 bytes
    rho (first 16 bytes hex): b31d13b7836f9750503a2b919adc7848
    t1: list of length k = 4
    t1[0]: length = 256 bytes, first 16 bytes: 2c12081d0e010a042f0009120d030918
  Secret Key:
    s: 4 polynomials, each of size 256
    e: 4 polynomials, each of size 256
    t0: 4 low-bit vectors (first 8 of t0[0]):
      [198, 205, 178, 67, 14, 172, 27, 191] ... (total 256 elements)
    tr: length = 32 bytes, first 16 bytes: 73288e7e4c31a982d2aa226469e1feba
```

Рисунок 3.15 – Key Generation

4. Формування підпису

У ході обчислення було сформовано підпис:

$$\sigma = (z, c, w_1),$$

де:

– c — хеш-виклик довжиною 32 байти;

– z — вектор з k многочленів по 256 коефіцієнтів;

– w_1 — стиснене представлення проміжної величини.

Для контролю результатів виведено перші коефіцієнти вектора z та байти w_1.

```
[4] Signature Generation
  Signature σ = (z, c, w1)
    c: length = 32 bytes
    c (hex): 4bc7ae358bc511ae3096205e085090d485fc1a958ed23cd2e55260b404057519
    z: list of k = 4 polynomials, each with 256 coefficients
      z[0] (first 8 coefficients): [12287, 0, 2, 12288, 0, 2, 0, 0] ... (total 256 elements)
    w1: list of k = 4 byte arrays of length 256
      w1[0] (first 16 bytes hex): 02040d231c16160f23292810001a1d27
```

Рисунок 3.16 – Signature Generation

5. Перевірка підпису

Функція перевірки повернула:

`verify_solmae(pk, msg, sig)=True`,

що підтверджує: підпис є валідним і відповідає відкритому ключу та повідомленню.

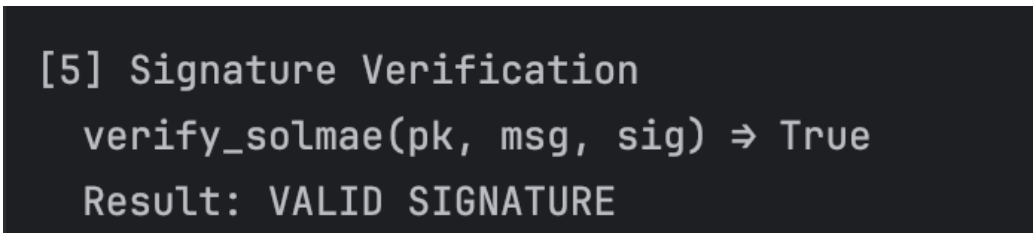

Рисунок 3.17 – Signature Verification

6. Демонстрація захисту від підробки

Для перевірки стійкості схеми до модифікації було змінено текст повідомлення:

Hello, Mr. Anderson. Do u remember me? (That's Mr. Smith)

Перевірка:

`verify_solmae(pk, tampered_msg, sig)=False,`

що підтверджує чутливість підпису до цілісності повідомлення та неможливість повторного використання підпису з іншим текстом.

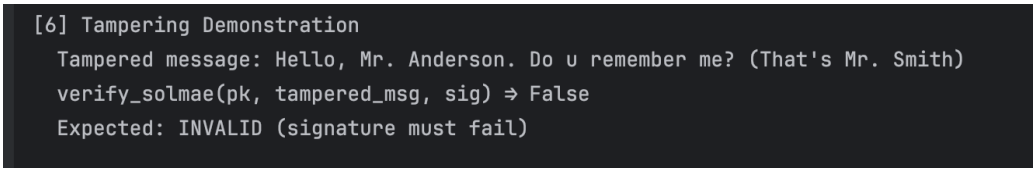

Рисунок 3.18 – Tampering Demonstration

ВИСНОВКИ

У ході виконання лабораторної роботи було проведено повне теоретичне та практичне дослідження решіткової схеми електронного підпису SOLMAE на базі NTRU-решіток. На теоретичному рівні детально розглянуто математичні основи алгоритму: побудову NTRU-решітки, якість базису, дискретні гаусівські розподіли, парадигму GPV-підписів, а також конкретну конструкцію процедур KeyGen, Sign та Verify для SOLMAE. Проаналізовано відомі результати щодо безпеки схеми, її зв'язок із задачами NTRU-search/decision, а також порівняно SOLMAE з близькими за ідеологією схемами (зокрема Falcon), виокремивши її переваги у простоті реалізації, можливості паралелізації та ефективності стиснення підписів.

На практичному етапі було розроблено модульну Python-реалізацію всіх основних компонентів схеми: арифметики в кільці R_q, операцій над многочленами, перетворень NTT/FFT, генераторів випадкових чисел і хеш-функцій, процедур семплінгу, генерації NTRU-пар (PairGen), розв'язувача NTRU-рівнянь (NtruSolve), модулів стиснення/декомпресії та власне алгоритму підпису SOLMAE. Архітектура коду побудована так, щоб окремі блоки можна було незалежно тестувати й за потреби замінювати або оптимізувати, що спрощує подальший розвиток реалізації.

Коректність реалізації підтверджено системою модульних та інтеграційних тестів для всіх ключових підсистем: модульної арифметики, операцій над многочленами, NTT/FFT-перетворень, RNG та хешування, процедур семплінгу, генерації пар NTRU-ключів, попередніх обчислень, стиснення/декомпресії та повного алгоритму підпису SOLMAE. Усі тести завершилися успішно, без єдиного збою, а час їх виконання знаходиться в межах мілісекунд, що свідчить про як коректність, так і достатню ефективність реалізованих алгоритмів для експериментальних цілей. Додаткова демонстрація показала, що згенерований підпис коректно проходить верифікацію для оригінального повідомлення і відхиляється

після найменшої модифікації тексту, що ілюструє цілісність та нефальсифікованість підпису в практичному сценарії.