КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра математичної інформатики

Кваліфікаційна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інформатика»
за спеціальністю 122 Комп’ютерні науки

на тему:

РОЗРОБКА IOS-FRAMEWORK ДЛЯ ПІДТРИМКИ ПОСТКВАНТОВИХ АЛГОРИТМІВ ШИФРУВАННЯ ТА ЦИФРОВОГО ПІДПИСУ

Виконав студент 4-го курсу
Юрій ПОБЕРЕЖНИЙ _____
(підпис)

Науковий керівник:
професор, доктор фіз.-мат. наук
Анатолій АНІСІМОВ _____
(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____
(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри математичної інформатики
«___»_____2023 р.
протокол №____
Завідувач кафедри
Василь ТЕРЕЩЕНКО _____
(підпис)

Київ – 2023

РЕФЕРАТ

Обсяг роботи 40 сторінок, 21 ілюстрація, 4 таблиці і 14 використаних джерел.

КВАНТОВО-СТІЙКІ АЛГОРИТМИ, IOS-FRAMEWORK, XCODE, ПОСТКВАНТОВА КРИПТОГРАФІЯ, SWIFT.

Об’єктом дослідження є існуючі постквантові алгоритми шифрування та цифрового підпису. Об’єктом розроблення є одна з таких схем шифрування з відкритим ключем та один алгоритм цифрового підпису, а також їх реалізація на мові програмування Swift у вигляді бібліотеки.

Метою роботи було дослідження та аналіз існуючих постквантових алгоритмів та вивчення їх характеристик та властивостей. Також метою була побудова та реалізація по одному алгоритму шифрування та цифрового підпису.

Методи дослідження: аналіз існуючих алгоритмів, проектування архітектури бібліотеки з деякими наявними протоколами.

Інструменти розробки: середовище розробки Xcode, мова програмування Swift.

Результат роботи: проведений детальний аналіз різноманітних алгоритмів, були досліджені основні властивості та характеристики. Була реалізована бібліотека з підтримкою одного алгоритму шифрування та одного алгоритму цифрового підпису.

ЗМІСТ

ВСТУП

Протягом останніх кількох років спостерігався постійний прогрес у створенні квантових комп'ютерів. Безпека багатьох широко використовуваних криптосистем із відкритим ключем опиниться під загрозою, якщо коли-небудь будуть реалізовані великомасштабні квантові комп'ютери. Зокрема, це включало б схеми обміну ключами і цифрові підписи, які базуються на факторизації, дискретних логарифмах і криптографії еліптичних кривих, проте симетричні криптографічні примітиви, такі як блочні шифри та хеш-функції, не зазнають такого різкого впливу. У результаті було активізовано дослідження щодо пошуку криптосистем із відкритим ключем, які були б захищені від зловмисників як за допомогою квантових, так і класичних комп'ютерів. Цю галузь часто називають постквантовою криптографією (PQC) або іноді квантово-стійкою криптографією. Мета полягає в тому, щоб розробити схеми, які можна розгорнути в існуючих комунікаційних мережах і протоколах без істотних змін.

На даний момент йде процес відбору квантово-стійких криптографічних алгоритмів з відкритим ключем. Нові стандарти криптографії з відкритим ключем визначать алгоритми для цифрових підписів, шифрування з відкритим ключем і обміну ключами.

Під час крайнього туру було проведено більш ретельний аналіз теоретичних та емпіричних доказів, які використовувалися для обґрунтування безпеки кандидатів. Було також ретельне дослідження їх продуктивності за допомогою оптимізованих реалізацій на різноманітних програмних і апаратних платформах.

У результаті досліджень було обрано перші алгоритми, які у майбутньому будуть стандартизуватися. Механізм інкапсуляції відкритого ключа (KEM), який буде стандартизовано, це CRYSTALS–KYBER.

Стандартизованими цифровими підписами будуть CRYSTALS–Dilithium, FALCON і SPHINCS+.

Хоча вибрано декілька алгоритмів підпису, рекомендується використовувати CRYSTALS–Dilithium як основний алгоритм. Крім того, існує ще чотири альтернативних KEM алгоритмів-кандидатів: BIKE, Classic McEliece, HQC і SIKE. Ці кандидати теж будуть розглянуті для майбутньої стандартизації.

Finalists	
Public-Key Encryption/KEMs	**Digital Signatures**
Classic McElliece	CRYSTALS-Dilithium
CRYSTALS-KYBER	FALCON
NTRU	Rainbow
Saber	

Alternate Candidates	
Public-Key Encryption/KEMs	**Digital Signatures**
BIKE	GeMSS
FrodoKEM	Picnic
HQC	SPHINCS^{+}
SIKE	

1. КРИТЕРІЇ ОЦІНЮВАННЯ

Було визначено три широкі аспекти критеріїв оцінки, які використовуватимуться для порівняння алгоритмів-кандидатів:

- безпека
- вартість і продуктивність
- характеристики алгоритму та реалізації

1.1. БЕЗПЕКА

Безпека є найважливішим критерієм, який перевіряється під час оцінки потенційних постквантових алгоритмів. Стандарти відкритого ключа наразі використовуються в широкому спектрі програм, включаючи протоколи інтернету, такі як TLS, SSH, IKE, IPsec і DNSSEC, а також для сертифікатів, підпису програмного коду та безпечних завантажувачів. Нові стандарти відкритого ключа забезпечать постквантову безпеку для кожної з цих програм.

З метою кількісної оцінки безпеки потенційних алгоритмів надано три можливі визначення безпеки — два для шифрування та одне для підписів. Також призначено п'ять категорій безпеки для класифікації обчислювальної складності атак, які порушують визначення безпеки:

1. застосування криптографії з відкритим ключем
2. визначення безпеки для шифрування/обміну ключами
3. визначення безпеки для ефемерного шифрування/обміну ключами
4. визначення безпеки для цифрових підписів
5. категорії міцності безпеки

Також зауважимо інші бажані властивості безпеки, такі як пряма секретність, стійкість до атак по стороннім каналам і атак з кількома ключами, а також стійкість до неправильного використання.

1.2. ВАРТІСТЬ І ПРОДУКТИВНІСТЬ

Вартість визначено як другий найважливіший критерій під час оцінювання алгоритмів. Вартість включає обчислювальну ефективність генерації ключів та операцій із відкритим і приватним ключами, витрати на передачу відкритих ключів і підписів або зашифрованих текстів, а також витрати на реалізацію з точки зору оперативної пам’яті або кількості шлюзів.

Під час порівняння загальної продуктивності алгоритмів враховувалися як обчислювальна вартість, так і вартість передачі даних. Для загального використання оцінка загальної продуктивності враховувала вартість передачі відкритого ключа на додаток до підпису чи зашифрованого тексту під час кожної транзакції. Для KEM також враховується вартість генерації ключів, оскільки багато програм використовують нову пару ключів KEM для кожної транзакції, щоб забезпечити пряму секретність. Для алгоритмів підпису вартість генерації ключів вважалася менш важливою.

Алгоритми KYBER, NTRU та Sabre мають гарну продуктивність як на процесорах x86-64 із розширеннями AVX2, так і на ARM Cortex-M4. Загальна продуктивність NTRU не така хороша, як у KYBER або Sabre, через повільнішу генерацію ключів і дещо більші відкриті ключі та зашифровані тексти. Однак загальна продуктивність будь-якого з цих KEM була б прийнятною для програм загального використання.

На рисунку 1 показано показники обчислювальної продуктивності для процесора x86-64 із розширеннями AVX2 для KYBER, NTRU і Sabre для категорій безпеки 1 і 3.

На рисунку 2 показано загальні витрати для KYBER, NTRU і Sabre, коли додається вартість передачі даних. Рисунок 2 було створено з використанням оціночної вартості 2000 циклів/байт.

Інкапсуляція та декапсуляція дуже швидкі за допомогою всіх трьох схем. Хоча Sabre має найнижчу загальну вартість через менші відкриті ключі та зашифровані тексти, різниця у вартості між KYBER і Sabre була недостатньою, щоб вважатися значною.

Вартість генерації ключів для ntruhps2048677 або ntruhrss701 приблизно в 11 разів більша, ніж для KYBER512. Однак, як показано на рисунку 2, у загальній вартості використання цих схем зазвичай домінує вартість передачі даних, і тому більша частина різниці в загальній вартості наборів параметрів NTRU порівняно з KYBER і Sabre пояснюється певною мірою тим, що NTRU великі відкриті ключі та зашифровані тексти. У результаті загальна вартість для ntruhps2048677 менш ніж на 30% більша, ніж для KYBER512. Крім того, оскільки відкриті ключі та шифртексти для наборів параметрів категорій 1 і 3 для всіх трьох схем, ймовірно, поміщаються в один інтернет-пакет, показники їх продуктивності можна вважати порівнянними. Можна також зазначити, що вартість генерації ключа для ntruhps2048677 або ntruhrss701 порівнянна з вартістю генерації ключа для еліптичної кривої криптографії P-256, яка широко використовується для ефемерного обміну ключами.

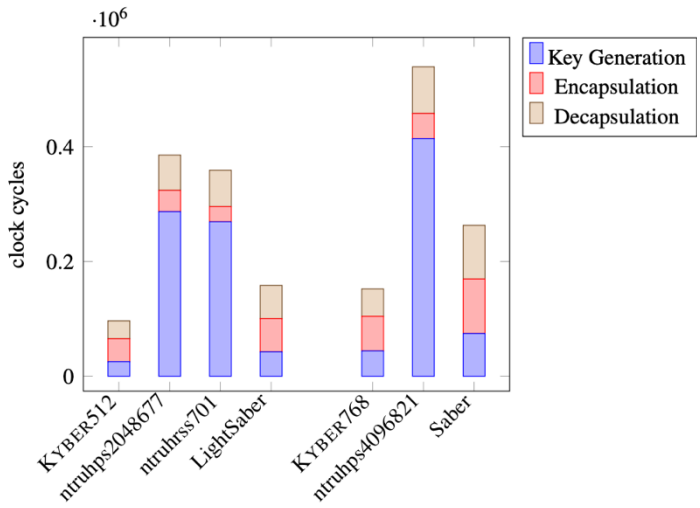

рисунок 1

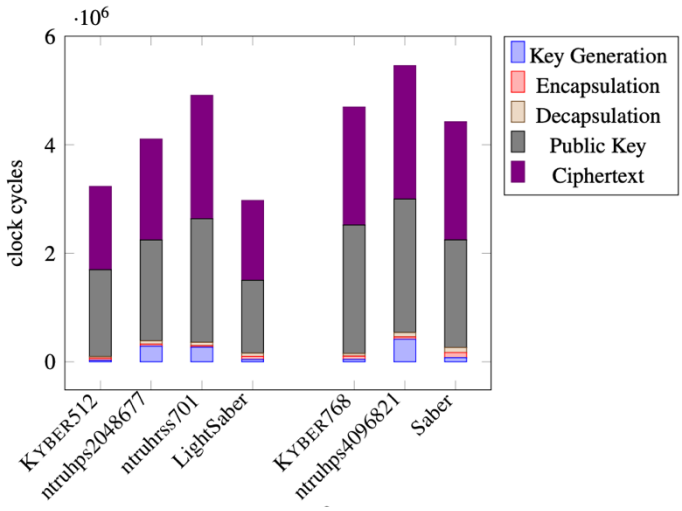

рисунок 2

На рисунку 3 показано показники обчислювальної продуктивності для ARM Cortex-M4, а на рисунку 4 показано «загальні витрати» з додаванням приблизної вартості передачі 2000 циклів/байт. У той час як у випадку процесора x86-64 загальна вартість визначається вартістю передачі даних, з ARM Cortex-M4, використовуючи ті самі оцінки циклів/байтів, вартість обчислень становить набагато більшу частину загальної суми вартість, особливо вартість генерації ключів із наборами параметрів NTRU. У результаті загальні витрати для ntruhps2048677 і ntruhrss701 більш ніж удвічі вищі, ніж для KYBER512. Однак більша частина додаткових витрат є результатом повільнішої генерації ключів NTRU, і обмежені пристрої з меншою ймовірністю виконуватимуть нову генерацію ключа для кожної транзакції. Якщо вартість генерації ключів вилучити із загальної вартості, то загальна вартість ntruhps2048677 буде менш ніж на 30% більшою, ніж для KYBER512. Отже, різниця в продуктивності між NTRU і KYBER або Sabre, яка буде спостерігатися на обмежених пристроях, ймовірно, буде набагато меншою, ніж показано на рисунках 3 і 4.

Результати порівняльного тесту показують, що і KYBER, і Sabre придатні для використання на обмежених пристроях, оскільки кожен із них може бути реалізований (принаймні без захисту від атак на бічних каналах), використовуючи менше 4 КіБ оперативної пам’яті з менш ніж 20 КіБ пам'яті для коду.

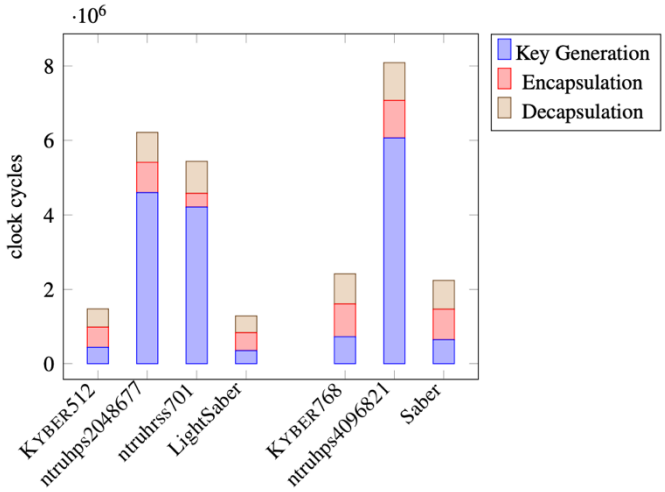

рисунок 3

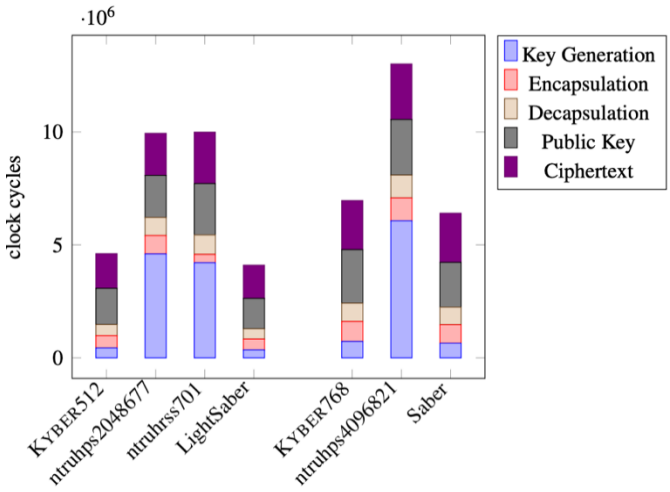

рисунок 4

Classic McEliece також був обраний як кандидат для можливої подальшої стандартизації. Класичний McEliece має профіль продуктивності, який відрізняється від інших KEM, що розглядаються, і, як наслідок, його продуктивність не порівнювалася безпосередньо з продуктивністю інших KEM. Класичний McEliece має повільну генерацію ключів і дуже великі відкриті ключі, але його швидкість інкапсуляції та декапсуляції порівнянна зі структурованою решіткою KEM, і він має дуже маленькі зашифровані тексти. Як наслідок, Classic McEliece може забезпечити найкращу продуктивність у програмах, де вартість генерації та передачі відкритого ключа не вважається частиною витрат, які входять у транзакцію, але його загальна вартість буде набагато більшою, ніж будь-який із інші кандидати KEM, якщо були включені витрати на передачу відкритого ключа.

Серед схем цифрового підпису загального призначення було обрано Dilithium, FALCON та Rainbow, хоч останній і має привабливий профіль продуктивності для додатків, що вимагають малих підписів або швидкої перевірки, зазнав втрат безпеки.

На рисунку 5 зображено показники обчислювальної продуктивності для процесора x86-64 із розширеннями AVX2 для Dilithium і FALCON. На відміну від рисунку 1, гістограма не включає вартість генерації ключів, оскільки ключі підпису не генеруються на основі кожної транзакції. На рисунку 6 показано «загальні витрати» для Dilithium і FALCON, якщо додати вартість передачі відкритого ключа та підпису. Як і на рисунках 2 і 4, використовується оціночна вартість 2000 циклів/байт. При використанні процесора x86-64 генерація підпису за допомогою Dilithium відбувається трохи швидше, ніж за допомогою FALCON. Однак передача даних домінує в загальних витратах на використання цих схем, тому загальна вартість FALCON нижча через його менший відкритий ключ і розмір підпису. Для більшості програм, які використовують процесор x86-64 або подібний,

показники продуктивності для Dilithium або FALCON мають бути прийнятними. Однак, на відміну від підписів FALCON, підписи Dilithium не можуть поміститися в один інтернет-пакет, тому це може ускладнити адаптацію деяких програм для використання Dilithium, ніж їх адаптацію для використання FALCON.

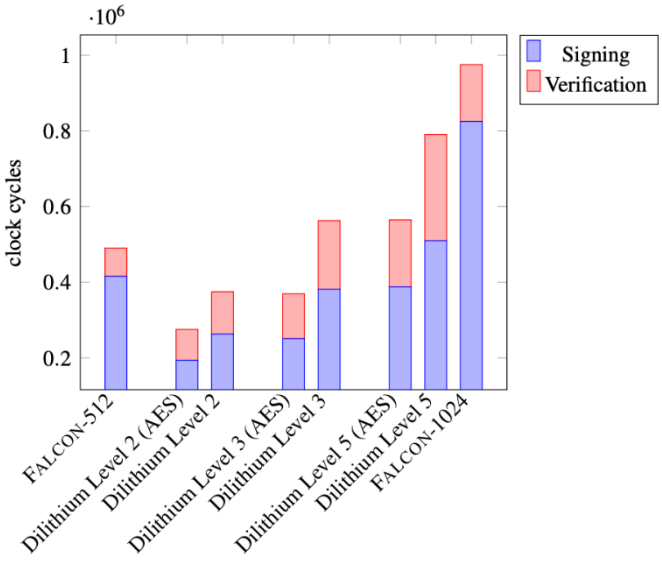

рисунок 5

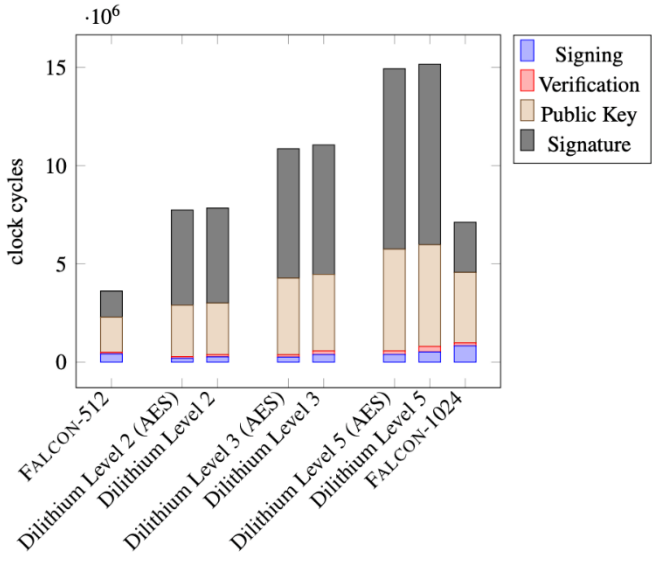

рисунок 6

На рисунку 7 показано показники обчислювальної продуктивності процесора ARM Cortex-M4 для наборів параметрів категорії безпеки 1, 2 і 3 наборів параметрів Dilithium і FALCON. На рисунку 8 показано «загальні витрати» з додаванням приблизної вартості передачі 2000 циклів/байт. Оскільки ARM Cortex-M4 не підтримує операції з плаваючою комою, генерація підпису за допомогою FALCON набагато повільніша, ніж генерація підпису за допомогою Dilithium, і різниця настільки велика, що загальна вартість використання Dilithium нижча, навіть якщо Dilithium передає високі дані.

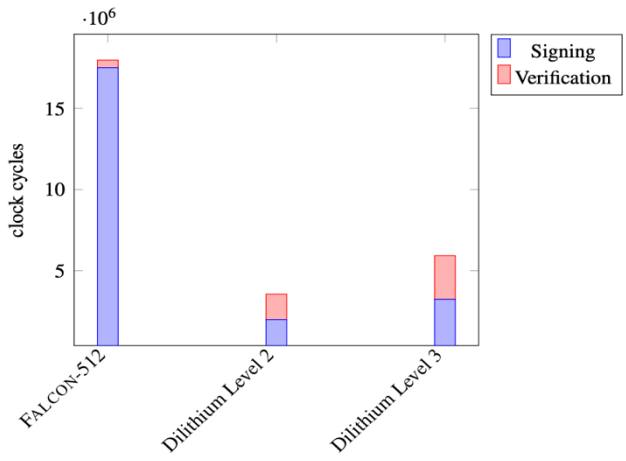

рисунок 7

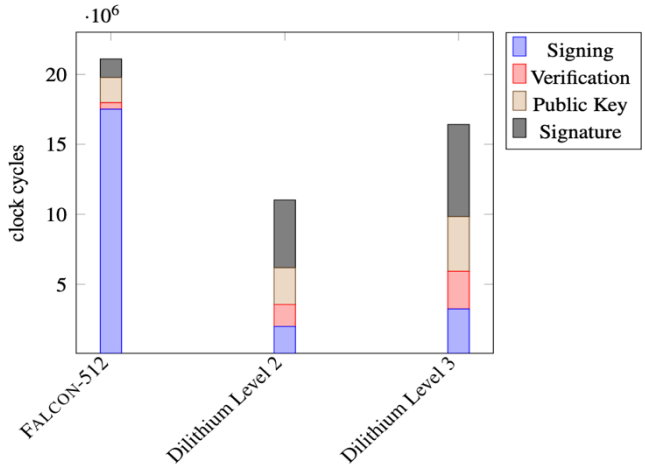

рисунок 8

Для схем цифрового підпису продемонстровано, що перевірка підпису для кожного з алгоритмів може бути реалізована з використанням менше 8 КіБ оперативної пам'яті та менш ніж 8 КіБ пам'яті для коду. Однак, у той час як генерація ключів і підписання за допомогою Dilithium можуть бути реалізовані з використанням менше ніж 9 КіБ оперативної пам'яті, FALCON, вимагає значно більше оперативної пам'яті, що може зробити FALCON неможливим для впровадження на обмежених пристроях, таких як смарт-карти.

На рисунках 9 і 10 показано результати для категорій безпеки 1 і 3 для альтернативних кандидатів KEM BIKE, FrodoKEM, HQC, NTRU Prime і SIKE. Як і у KYBER, NTRU і Sabre (див. рис. 2), за винятком SIKE, у загальній вартості використання цих схем на процесорах x86-64 домінує вартість передачі даних. Продуктивність NTRU Prime порівнянна з продуктивністю NTRU. Загалом, BIKE та HQC мають вищу загальну продуктивність, ніж FrodoKEM або SIKE. Використовуючи метрику 2000 циклів/байт, SIKE має дещо кращу загальну продуктивність, ніж FrodoKEM. Однак у багатьох випадках використання вартість передачі даних порівняно з обчисленнями буде нижчою, а FrodoKEM забезпечить кращу загальну продуктивність.

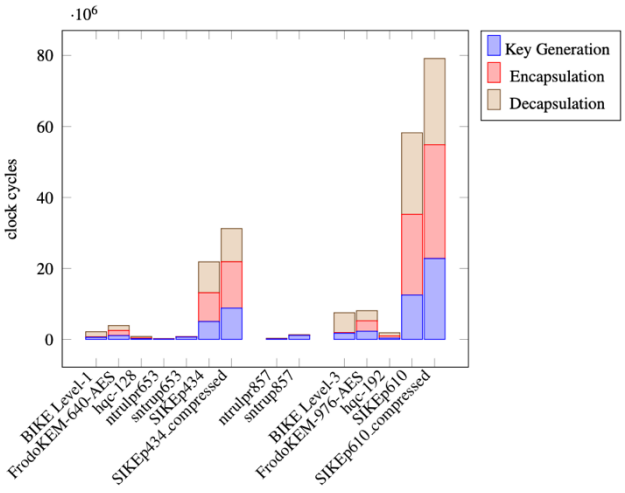

рисунок 9

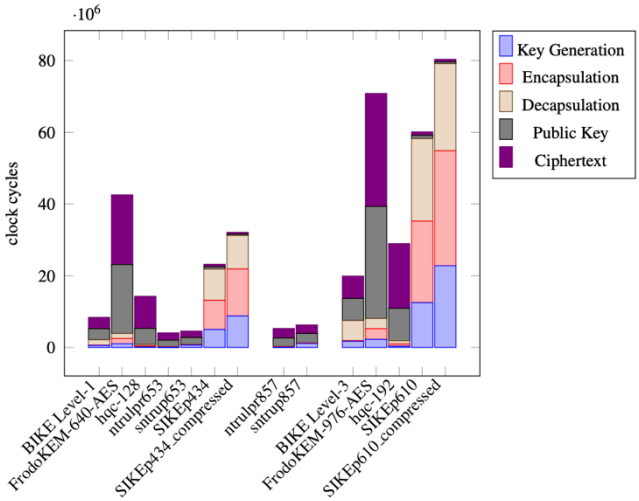

рисунок 10

1.3. АЛГОРИТМ І ХАРАКТЕРИСТИКИ РЕАЛІЗАЦІЇ

Розглядаючи інші критерії оцінки, окрім безпеки, вартості та ефективності, також розглядаються різні бажані алгоритми та характеристики реалізації. Специфічними характеристиками є гнучкість, простота та адаптація (відсутність факторів, які могли б перешкодити адаптації).

Іншою важливою характеристикою кандидатів є їхній потенційний вплив на продуктивність існуючих широко використовуваних протоколів (наприклад, TLS, IPSec і SSH) і сертифікатів. Кандидати структурованої решітки як для KEM, так і для підписів можуть бути замінені на ці протоколи для існуючих алгоритмів з відкритим ключем з відносно невеликими (або без) втратами продуктивності.

Хоча важко точно виміряти простоту, проте більш прості конструкції є кращими при порівнянні двох схожих схем. Зокрема, простота була важливим фактором у оцінці FALCON, який використовує арифметику з плаваючою комою та більш складна реалізація може призвести до помилок, які можуть вплинути на безпеку. На противагу цьому простіший дизайн Dilithium сприймається більш позитивно.

2. ОГЛЯД КРИПТОАНАЛІТИЧНИХ РЕЗУЛЬТАТІВ

Атака на GeMSS різко знизила його безпеку та підірвала впевненість щодо його використання. Цей результат призвів до виключення GeMSS з розгляду для стандартизації.

Rainbow також зазнала значних атак. Перша атака на початку спричинила втрату набору параметрів від 20 до 55 біт безпеки в моделі оперативної пам’яті, причому набори параметрів з вищим рівнем безпеки втрачали більше бітів безпеки. За цим послідувала більш серйозна атака, яка призвела до відновлення закритого ключа для параметрів категорії безпеки 1 трохи більше ніж за два дні часу обчислення на одному ноутбуці. Через брак впевненості в безпеці, Rainbow не було обрано для стандартизації.

Також було вилучено FrodoKEM, NTRU Prime і Picnic з розгляду для стандартизації. FrodoKEM — це кандидат, заснований на решітці, якого було обрано як альтернативу. FrodoKEM в основному вирізняється тим, що він не покладається на структуровані решітки (на відміну від KYBER, NTRU і Sabre). На меті є обрати принаймні один додатковий KEM, не заснований на структурованих решітках для стандартизації. Три інші альтернативи KEM (BIKE, HQC і SIKE) краще підходять для цієї ролі, ніж FrodoKEM. FrodoKEM загалом має гіршу продуктивність, ніж ці три, тому не розглядатиметься надалі для стандартизації. NTRU Prime також було висунуто як альтернативу, оскільки вважалося менш перспективним у порівнянні з іншими. Не було отримано результатів, які суттєво змінили б цю точку зору. NTRU Prime не було обрано для продовження процесу, оскільки буде стандартизуватися один із фіналістів KEM. Схожа ситуація була і з підписами. Picnic не було обрано, оскільки було вирішено стандартизувати SPHINCS+. Picnic і SPHINCS+ мають схожі профілі продуктивності (малі відкриті ключі та великі підписи) і підходять для однакових випадків використання. SPHINCS+ і Picnic мають кілька версій,

що робить пряме порівняння вартості та продуктивності більш складним. Однак кожен з них має набагато вищу вартість і набагато гіршу продуктивність порівняно з Dilithium і FALCON, що робить ці критерії менш важливими.

Під час вибору між подібними алгоритмами KEM вартість і продуктивність були важливими критеріями вибору. Як зазначено в розділі 2.2, під час порівняння кандидатів враховувалися як вартість передачі даних, так і ефективність обчислень.

Одним із важких виборів, був вибір між KYBER, NTRU та Sabre. Усі троє дуже схожі один на одного. Є впевненість у безпеці, яку забезпечує кожен. Більшість додатків зможуть використовувати будь-який із них без суттєвих втрат продуктивності, тому є намір стандартизувати лише одного з них, оскільки всі три були засновані на структурованих решітках. Основним чинником стали питання пов'язані з патентами. Однією з відмінностей між KYBER, Sabre і NTRU є конкретне припущення щодо безпеки. Вважається проблема MLWE, від якої залежить KYBER, є трохи переконливішою, ніж інші припущення, такі як MLWR або проблема NTRU.

Також високо оцінено специфікацію команди KYBER, яка включала ретельний і детальний аналіз безпеки. Що стосується продуктивності, KYBER був на вершині у більшості тестів.

Решта обраних кандидатів KEM (BIKE, Classic McEliece, HQC, SIKE) продовжуватимуть оцінюватися. І BIKE, і HQC базуються на структурованих кодах і підходять як KEM загального призначення, який не базується на решітках.

SIKE залишається привабливим кандидатом для стандартизації через малий розмір ключа та зашифрованого тексту. Класичний McEliece був фіналістом, але наразі не буде стандартизований, оскільки не передбачається його широкого використання через великий розмір

відкритого ключа. Таким чином, терміновості стандартизації Classic McEliece поки що немає.

Було обрано обидві схеми Dilithium і FALCON для стандартизації, оскільки обидва базуються на структурованих решітках і можуть використовуватися в більшості програм. Як зазначено в розділі 2.2, генерація ключів і підписів для FALCON вимагає більше ресурсів (шлюзів і оперативної пам’яті), ніж Dilithium, що може зробити FALCON непридатним для реалізації на обмежених пристроях, особливо у випадках, коли потрібний захист від атак. Крім того простіша конструкція генерації ключів і підписів Dilithium допоможе забезпечити безпечне впровадження. З цих причин обрано Dilithium як основний алгоритм підпису, який буде рекомендовано для загального використання.

Деякі програми не працюватимуть у тому вигляді, в якому вони зараз розроблені, якщо підпис і дані, що підписуються, не можуть поміститися в один інтернет-пакет. Для цих програм складність реалізації генерації підпису FALCON може не викликати занепокоєння, але складність модифікації програм для роботи з більшим розміром підпису Dilithium може створити перешкоду для переходу до схем постквантового підпису. З цієї причини також було вирішено стандартизувати FALCON. Враховуючи загальну кращу продуктивність FALCON, коли створення підпису не потрібно виконувати на обмежених пристроях, багато програм можуть віддати перевагу використанню FALCON над Dilithium, навіть у випадках, коли розмір підпису Dilithium не буде перешкодою для впровадження.

Щоб не покладатися повністю на безпеку решіток, також взято за мету стандартизувати SPHINCS+. Безпека SPHINCS+ добре зрозуміла, хоча вона набагато більша та повільніша, ніж підписи за допомогою решіток. SPHINCS+ — це зріла схема, і її стандартизація створює резервний варіант, який допомагає мінімізувати ризик того, що єдиний прорив у криптоаналізі залишить світ без життєздатного підпису. Зазначено, що SPHINCS+ може

не підходити для багатьох застосувань, враховуючи його профіль продуктивності.

SPHINCS+ було обрано зараз замість того, щоб продовжувати його подальший розгляд. Таким чином, це означає кінець поточного процесу для схем підпису. Усі кандидати на підписи були або відібрані для стандартизації, або вилучені з розгляду для стандартизації. У майбутньому може стандартизуватися більше підписів, але це займе кілька років, і немає гарантії кращих алгоритмів.

Підсумовуючи, було обрано чотирьох кандидатів для стандартизації та чотирьох для подальшої оцінки та вивчення. Перелік цих алгоритмів наведено в таблицях нижче.

Finalists	
Public-Key Encryption/KEMs	**Digital Signatures**
CRYSTALS-KYBER	CRYSTALS-Dilithium
	FALCON
	SPHINCS^{+}

Alternate Candidates	
Public-Key Encryption/KEMs	**Digital Signatures**
BIKE	GeMSS
Classic McElliece	Picnic
HQC	
SIKE	

Наразі обрано один KEM для стандартизації. Чотири додаткові KEM продовжуватимуть оцінюватися, і передбачається стандартизація принаймні одного з них після завершення повного огляду всіх.

Основними алгоритмами, рекомендованими для більшості випадків використання, є CRYSTALS–KYBER і CRYSTALS–Dilithium.

Крім того, схеми підписів FALCON і SPHINCS+ також будуть стандартизовані. Кандидати BIKE, Classic McEliece, HQC і SIKE будуть

розглянуті для подальшого вивчення. Причини такого вибору були наведені раніше у цій роботі.

Опираючись на результати досліджень та рекомендацій у межах даної роботи буде описано реалізацію CRYSTALS–KYBER і CRYSTALS–Dilithium.

3. ТЕХНІЧНІ ХАРАКТЕРИСТИКИ АЛГОРИТМУ CRYSTAL-KYBER

CRYSTALS–KYBER — це механізм інкапсуляції ключів (KEM), захищений IND-CCA2. Його безпека базується на складності вирішення проблеми навчання з помилками в модульних решітках (проблема MLWE). Створення Kyber відбувається за двоетапним підходом: спочатку представляється безпечна схема шифрування з відкритим ключем IND-CPA, яка шифрує повідомлення фіксованої довжини 32 байти, яку подалі будемо називати Kyber.CPAPKE. Далі використовується дещо змінене перетворення Фуджісакі–Окамото (FO) для побудови KEM, захищеного IND-CCA2. Щоразу, коли хочемо підкреслити, що ми говоримо про захищений IND-CCA2 KEM, то будемо називати його Kyber.CCAKEM.

3.1. ПОПЕРЕДНІ СЛОВА ТА ПОЗНАЧЕННЯ

Байти та байтові масиви. Вхідні та вихідні дані для всіх функцій є масивами байтів. Для спрощення позначень через $\mathcal{B}$ позначимо множину $\{0, \ldots, 255\}$, тобто набір 8-розрядних цілих чисел (байтів) без знаку. Отже, позначимо через $\mathcal{B}^k$ множину байтових масивів довжини k і через $\mathcal{B}^k$ множину байтових масивів довільної довжини (або потоків байтів). Для двобайтових масивів a і b позначаємо $(a||b)$ як конкатенацію a і b. Для

байтового масиву a позначаємо $a + k$ масив байтів, що починається з байта k масиву a (з індексацією, починаючи з нуля).

Наприклад, нехай a буде масивом байтів довжиною l, нехай b буде іншим масивом байтів і нехай $c = (a||b)$ буде конкатенацією a і b. Тоді $b = a + 1$.

Коли зручніше працювати з масивом бітів, робимо це перетворення явним за допомогою функції BytesToBits, яка приймає як вхідні дані масив з l байтів і створює як вихідний масив з 8ℓ бітів. Біт β_i у позиції i масиву вихідних бітів отримується з байта $b_{i/8}$ у позиції $i/8$ вхідного масиву шляхом обчислення $\beta_i = ((b_{i/8}/2^{(i\ mod\ 8)})mod\ 2)$.

Кільця поліномів і вектори. Позначимо як $\mathcal{R}$ кільце $\mathbb{Z}[X]/(X^n + 1)$, а через $\mathcal{R}_q$ — кільце $\mathbb{Z}_q[X]/(X^n + 1)$, де $n = 2n' - 1$ так, що $X^n + 1$ є $2n'$ циклотомічний поліном. Далі значення n, n' і q фіксуються як $n = 256, n' = 9$ і $q = 3329$.

За замовчуванням усі вектори будуть векторами-стовпцями. Великі літери є матрицями. Для вектора $\boldsymbol{v}$ (або матриці $\boldsymbol{A}$) ми позначимо через $\boldsymbol{v^T}$ (або $\boldsymbol{A^T}$) його транспонування. Для вектора $\boldsymbol{v}$ ми пишемо $\boldsymbol{v}[\boldsymbol{i}]$, щоб позначити його $\boldsymbol{i}$-й запис (з індексацією, починаючи з нуля). Для матриці $\boldsymbol{A}$ ми пишемо $\boldsymbol{A}[\boldsymbol{i}][\boldsymbol{j}]$, щоб позначити запис у рядку $\boldsymbol{i}$, стовпці $\boldsymbol{j}$.

Модульні скорочення. Для парного (відповідно непарного) додатного цілого числа $\boldsymbol{\alpha}$ ми визначаємо $\boldsymbol{r' = r\ mod^{\pm}\ \alpha}$ як унікальний елемент $\boldsymbol{r'}$ у діапазоні $\frac{-\alpha}{2} < r' \leq \frac{\alpha}{2}$ (відповідно $\frac{-\alpha-1}{2} \leq r' \leq \frac{\alpha-1}{2}$), такий що $\boldsymbol{r' = r\ mod\ \alpha}$.

Для будь-якого натурального число $\boldsymbol{\alpha}$, ми визначаємо $\boldsymbol{r' = r\ mod^{+}\ \alpha}$ як унікальний елемент $\boldsymbol{r'}$ в діапазоні $\boldsymbol{0 \leq r' < \alpha}$ такий, що $\boldsymbol{r' = r\ mod\ \alpha}$. Коли точне представлення неважливе, просто пишемо $\boldsymbol{r\ mod\ \alpha}$.

Округлення. Для елемента $x \in \mathbb{Q}$ позначимо через $[x]$ округлення x до найближчого цілого числа з округленням у більшу сторону.

Розміри елементів. Для елемента $w \in \mathbb{Z}_q$ ми пишемо $\|w\|_\infty$, що означає $|w \bmod^{\pm} q|$. Тепер ми визначимо $\boldsymbol{\ell}_\infty$ і $\boldsymbol{\ell}_\mathbf{2}$ норми для $w = w_0 + w_1X + \ldots + w_{n-1}X^{n-1} \in \mathcal{R}$:

$$\|w\|_\infty = \max_i \|w_i\|_\infty, \|w\| = \sqrt{\|w\|_\infty^2 + \cdots + \|w_{n-1}\|_\infty^2}.$$

Так само для $\boldsymbol{w} = (w_0, \ldots, w_k) \in \mathcal{R}^k$ визначаємо

$$\|\boldsymbol{w}\|_\infty = \max_i \|w_i\|_\infty, \|\boldsymbol{w}\| = \sqrt{\|w\|_\infty^2 + \cdots + \|w_k\|_\infty^2}.$$

Набори та розподіли. Для набору S ми пишемо $s \leftarrow S$, щоб позначити, що s вибрано рівномірно випадковим чином із S. Якщо S є розподілом ймовірностей, то це означає, що s вибрано відповідно до розподілу S.

Компресія та декомпресія. Тепер визначимо функцію $Compress_q(x, d)$, яка приймає елемент $x \in \mathbb{Z}_q$ і виводить ціле число в $\{0, \ldots, 2^d - 1\}$, де $d < \lceil \log_2 q \rceil$. Крім того, ми визначаємо функцію $Decompress_q$, таку, що

$$x' = Decompress_q(Compress_q(x, d), d)$$

є елементом, близьким до x, точніше

$$|x' - x \bmod^{\pm} q| \leq B_q := \left\lceil \frac{q}{2^d + 1} \right\rfloor.$$

Функції, що задовольняють цим вимогам, визначаються як:

$$Compress_q(x, d) = \lceil (2^d/q) \cdot x \rfloor \bmod^{+} 2^d,$$

$$Decompress_q(x, d) = \lceil (q/2^d) \cdot x \rfloor.$$

Якщо $Compress_q$ або $Decompress_q$ використовується з $x \in \mathcal{R}_q$ або $x \in \mathcal{R}_q^k$, процедура застосовується до кожного коефіцієнта окремо.

Основною причиною визначення функцій $Compress_q$ і $Decompress_q$ є можливість відкинути деякі біти молодшого порядку в зашифрованому тексті, які не мають великого впливу на вірогідність правильності дешифрування, таким чином зменшуючи розмір зашифрованих текстів.

$Compress_q$ і $Decompress_q$ також використовуються не для стиснення, а саме для виконання звичайного виправлення помилок LWE під час

шифрування та дешифрування. Точніше функція $Decompress_q$ використовується для створення проміжків із допуском до помилок, надсилаючи біт повідомлення 0 до 0 і 1 до $\lceil q/2 \rfloor$. Функція $Compress_q$ використовується для дешифрування до 1, якщо $v - s^T u$ ближче до $\lceil q/2 \rfloor$, ніж до 0, і дешифрування до 0 в іншому випадку.

Симетричні примітиви. Kyber використовує псевдовипадкову функцію PRF: $\boldsymbol{\mathcal{B}}^{32} \times \boldsymbol{\mathcal{B}} \to \boldsymbol{\mathcal{B}}^*$ та розширювану вихідну функцію XOF: $\boldsymbol{\mathcal{B}}^* \times \boldsymbol{\mathcal{B}} \times \boldsymbol{\mathcal{B}} \to \boldsymbol{\mathcal{B}}^*$. Kyber також використовує дві хеш-функції H: $\boldsymbol{\mathcal{B}}^* \to \boldsymbol{\mathcal{B}}^{32}$ і G: $\boldsymbol{\mathcal{B}}^* \to \boldsymbol{\mathcal{B}}^{32} \times \boldsymbol{\mathcal{B}}^{32}$, а також функцію визначення ключа KDF: $\boldsymbol{\mathcal{B}}^* \to \boldsymbol{\mathcal{B}}^*$.

NTT, множення та бітовий порядок. Дуже ефективним способом виконання множень у $\mathcal{R}_q$ є так зване теоретико-числове перетворення (NTT).

Для нашого простого числа $q = 3329$ з $q - 1 = 28 \cdot 13$ базове поле $\mathbb{Z}_q$ містить примітивні 256-і корені з одиниці, але не примітивні 512-і корені. Отже, визначальний поліном $X^{256} + 1$ $\mathcal{R}$ розкладається на 128 поліномів ступеня 2 за модулем q, а NTT полінома $f \in \mathcal{R}_q$ є вектором із 128 поліномів ступеня один. Проста реалізація NTT на місці без зміни порядку виводить ці поліноми в бітовому порядку, і таким чином визначаємо NTT. Конкретно, нехай $\zeta = 17$ — перший примітивний 256-й корінь з одиниці за модулем q, а $\{\zeta, \zeta^3, \zeta^5, \dots, \zeta^{255}\}$ — множина всіх 256-х коренів з одиниці. Тому поліном $X^{256} + 1$ можна записати як

$$X^{256} + 1 = \prod_{i=0}^{127} \left(X^2 - \zeta^{2i+1}\right) = \prod_{i=0}^{127} \left(X^2 - \zeta^{2br_7(127)+1}\right).$$

де $br7(i)$ для $i = 0, \dots, 127$ — розрядність 7-розрядного цілого числа i без знака. Тоді NTT для $f \in \mathcal{R}_q$ визначається як

$$(f \bmod X^2 - \zeta^{2br_7(0)+1}, \dots, f \bmod X^2 - \zeta^{2br_7(127)+1}.$$

Цей вектор лінійних поліномів потім серіалізується у вектор $\mathbb{Z}_q^{256}$ канонічним способом. Крім того, щоб не вводити додаткові типи даних і

полегшити реалізацію NTT на місці, визначаємо $NTT: \mathcal{R}_q \to \mathcal{R}_q$ як біекцію, яка відображає $f \in \mathcal{R}_q$ на поліном із вищезгаданим вектором коефіцієнтів. Отже,

$$NTT(f) = \hat{f} = \hat{f}_0 + \hat{f}_1 X + \cdots + \hat{f}_{255} X^{255}$$

$$\hat{f}_{2i} = \sum_{j=0}^{127} f_{2j}\, \zeta^{(2br_7(127)+1)j},$$

$$\hat{f}_{2i+1} = \sum_{j=0}^{127} f_{2j+1}\, \zeta^{(2br_7(127)+1)j}.$$

Хочеться підкреслити, що навіть якщо пишеться $\hat{f}$ як поліном від $\mathcal{R}_q$, він не має алгебраїчного значення як такого. Природне алгебраїчне представлення $NTT(f) = \mathcal{R}_q$ є 128 поліномами ступеня. Тобто, це

$$NTT(f) = \hat{f} = \left(\hat{f}_0 + \hat{f}_1 X,\ \hat{f}_2 + \hat{f}_3 X, \ldots, \hat{f}_{254} + \hat{f}_{255} X\right).$$

Використовуючи NTT та його обернений NTT^{-1}, ми можемо дуже ефективно обчислити добуток $f \cdot g$ двох елементів $f, g \in \mathcal{R}_q$ як $NTT^{-1}(NTT(f) \circ NTT(g))$, де $NTT(f) \circ NTT(g) = \hat{f} \circ \hat{g} = \hat{h}$ позначає базове множення, що складається з 128 добутків лінійних поліномів вигляду

$$\hat{h}_{2i} + \hat{h}_{2i+1} X = \left(\hat{f}_{2i} + \hat{f}_{2i+1} X\right)\left(\hat{g}_{2i} + \hat{g}_{2i+1} X\right) mod\ X^2 - \zeta^{2br_7(i)+1}.$$

Коли застосовуємо NTT або NTT^{-1} до вектора або матриці елементів $\mathcal{R}_q$, це означає, що відповідна операція застосовується $\circ$ до кожного запису окремо. Коли застосовуємо до матриць або векторів, це означає, що виконуємо звичайне множення матриць, але що окремі добутки записів є наведеними вище множеннями в базовому випадку.

Рівномірна вибірка в $\mathcal{R}_q$. Kyber використовує детермінований підхід до вибірки елементів у $\mathcal{R}_q$, які статистично близькі до рівномірно випадкового розподілу. Для цієї вибірки ми використовуємо функцію $Parse: \mathcal{B}^* \rightarrow \mathcal{R}_q$, яка отримує на вхід потік байтів $B = b_0, b_1, b_2, \ldots$ і обчислює NTT-представлення $\hat{a} = \hat{a}_1 X + \cdots + \hat{a}_{n-1} X^{n-1} \in \mathcal{R}_q$ для $a \in \mathcal{R}_q$. Розбір описано на рисунку 11 (в цьому описі передбачається, що $q = 3329$).

Algorithm 1 Parse: $\mathcal{B}^* \rightarrow R_q^n$

Input: Byte stream $B = b_0, b_1, b_2 \cdots \in \mathcal{B}^*$
Output: NTT-representation $\hat{a} \in R_q$ of $a \in R_q$

```
i := 0
j := 0
while j < n do
    d1 := b_i + 256 · (b_{i+1} mod+ 16)
    d2 := ⌊b_{i+1}/16⌋ + 16 · b_{i+2}
    if d1 < q then
        â_j := d1
        j := j + 1
    end if
    if d2 < q and j < n then
        â_j := d2
        j := j + 1
    end if
    i := i + 3
end while
return â_0 + â_1 X + ··· + â_{n-1} X^{n-1}
```

рисунок 11

Логіка функції Parse полягає в тому, що якщо вхідний масив байтів статистично близький до рівномірно випадкового масиву байтів, то вихідний поліном статистично близький до рівномірно випадкового елемента $\mathcal{R}_q$. Він представляє рівномірно випадковий поліном в $\mathcal{R}_q$, оскільки NTT є біективним і, таким чином, відображає поліноми з рівномірно випадковими коефіцієнтами на поліноми зі знову рівномірно випадковими коефіцієнтами.

Вибірка з біноміального розподілу. Шум у Kyber вибирається з центрованого біноміального розподілу B_η для $\eta = 2$ або $\eta = 3$. Ми визначаємо B_η наступним чином:

$$Sample(a_1, \dots, a_\eta, b_1, \dots, b_\eta) \leftarrow \{0,1\}^{2\eta}$$

$$\sum_{i=1}^{\eta}(a_i - b_i).$$

Коли ми пишемо, що поліном $f \in \mathcal{R}_q$ або вектор таких поліномів вибирається з B_η, ми маємо на увазі, що кожен коефіцієнт вибирається з B_η.

Для специфікації Kyber нам потрібно визначити, як поліном $f \in \mathcal{R}_q$ детерміновано відбирається відповідно до B_η з 64_η байтів виводу псевдовипадкової функції (фіксуємо $n = 256$ у цьому описі). Це робиться за допомогою функції CBD (для «центрованого біноміального розподілу»), визначеної, як описано на рисунку 12.

Algorithm 2 $\mathsf{CBD}_\eta : \mathcal{B}^{64\eta} \to R_q$

Input: Byte array $B = (b_0, b_1, \dots, b_{64\eta-1}) \in \mathcal{B}^{64\eta}$
Output: Polynomial $f \in R_q$

$(\beta_0, \dots, \beta_{512\eta-1}) \coloneqq \mathsf{BytesToBits}(B)$
for i from 0 to 255 **do**
  $a \coloneqq \sum_{j=0}^{\eta-1} \beta_{2i\eta+j}$
  $b \coloneqq \sum_{j=0}^{\eta-1} \beta_{2i\eta+\eta+j}$
  $f_i \coloneqq a - b$
end for
return $f_0 + f_1 X + f_2 X^2 + \cdots + f_{255} X^{255}$

рисунок 12

Кодування та декодування. Існує два типи даних, які Kyber має серіалізувати в байтові масиви: байтові масиви та вектори поліномів. Масиви байтів тривіально серіалізуються за допомогою ідентичності, тому нам потрібно визначити, як ми серіалізуємо та десеріалізуємо поліноми.

На рисунку 13 ми надаємо опис псевдокоду функції $Decode_\ell$, яка десеріалізує масив із 32_ℓ байтів у поліном $f = f_0 + f_1 X + \cdots + f_{255} X^{255}$f (знову фіксуємо $n = 256$ у цьому описі) з кожним коефіцієнтом f_i у $\{0, \dots, 2^\ell - 1\}$. Ми визначаємо функцію $Encode_\ell$ як зворотну до $Decode_\ell$. Щоразу, коли ми застосовуємо $Encode_\ell$ до вектора поліномів, ми кодуємо кожен поліном окремо та об’єднуємо вихідні масиви байтів.

Algorithm 3 $\mathsf{Decode}_\ell\colon \mathcal{B}^{32\ell} \to R_q$

Input: Byte array $B \in \mathcal{B}^{32\ell}$
Output: Polynomial $f \in R_q$
$(\beta_0, \ldots, \beta_{256\ell-1}) \coloneqq \mathsf{BytesToBits}(B)$
for i from 0 to 255 **do**
$f_i \coloneqq \sum_{j=0}^{\ell-1} \beta_{i\ell+j} 2^j$
end for
return $f_0 + f_1 X + f_2 X^2 + \cdots + f_{255} X^{255}$

рисунок 13

3.2. СПЕЦИФІКАЦІЯ KYBER.CPAPKE

Kyber.CPAPKE схожий на схему шифрування LPR, яку представили для Ring-LWE. Коріння цієї схеми сходять до першої схеми шифрування на основі LWE, з основною відмінністю в тому, що базове кільце не є $\mathbb{Z}_q$, і як секретний вектор, так і вектор помилки мають малі коефіцієнти. Ідея використання поліноміального кільця (замість $\mathbb{Z}_q$) сягає криптосистеми NTRU, тоді як симетрія між секретом і помилкою вже використовувалась у інших дуже схожих криптографічних схемах.

Основною відмінністю від схеми шифрування LPR є використання Module-LWE замість Ring-LWE. Крім того, ми скорочуємо зашифровані тексти, округлюючи молодші біти, як у схемах на основі навчання з округленням, що є загальною технікою для зменшення розміру зашифрованого тексту також у схемах на основі LWE.

Параметри. Kyber.CPAPKE параметризовано цілими числами $n, k, q, \eta_1, \eta_2, d_u, d_v$. Як зазначалося раніше, у цьому документі n завжди дорівнює 256, а q завжди дорівнює 3329.

Використовуючи позначення підрозділу 3.1, ми надаємо визначення генерації ключів, шифрування та дешифрування схеми шифрування з відкритим ключем Kyber.CPAPKE на рисунках 14, 15, 16 відповідно.

Algorithm 4 KYBER.CPAPKE.KeyGen(): key generation

Output: Secret key $sk \in \mathcal{B}^{12 \cdot k \cdot n/8}$
Output: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$

$d \leftarrow \mathcal{B}^{32}$
$(\rho, \sigma) := \mathrm{G}(d)$
$N := 0$
for i from 0 to $k-1$ **do** ▷ Generate matrix $\hat{\mathbf{A}} \in R_q^{k \times k}$ in NTT domain
  for j from 0 to $k-1$ **do**
    $\hat{\mathbf{A}}[i][j] := \mathsf{Parse}(\mathsf{XOF}(\rho, j, i))$
  end for
end for
for i from 0 to $k-1$ **do** ▷ Sample $\mathbf{s} \in R_q^k$ from B_{η_1}
  $\mathbf{s}[i] := \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(\sigma, N))$
  $N := N + 1$
end for
for i from 0 to $k-1$ **do** ▷ Sample $\mathbf{e} \in R_q^k$ from B_{η_1}
  $\mathbf{e}[i] := \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(\sigma, N))$
  $N := N + 1$
end for
$\hat{\mathbf{s}} := \mathsf{NTT}(\mathbf{s})$
$\hat{\mathbf{e}} := \mathsf{NTT}(\mathbf{e})$
$\hat{\mathbf{t}} := \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
$pk := (\mathsf{Encode}_{12}(\hat{\mathbf{t}} \bmod^+ q) \| \rho)$ ▷ $pk := \mathbf{As} + \mathbf{e}$
$sk := \mathsf{Encode}_{12}(\hat{\mathbf{s}} \bmod^+ q)$ ▷ $sk := \mathbf{s}$
return (pk, sk)

рисунок 14

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

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

$N := 0$
$\hat{\mathbf{t}} := \mathsf{Decode}_{12}(pk)$
$\rho := pk + 12 \cdot k \cdot n/8$
for i from 0 to $k-1$ **do** ▷ Generate matrix $\hat{\mathbf{A}} \in R_q^{k \times k}$ in NTT domain
  for j from 0 to $k-1$ **do**
    $\hat{\mathbf{A}}^T[i][j] := \mathsf{Parse}(\mathsf{XOF}(\rho, i, j))$
  end for
end for
for i from 0 to $k-1$ **do** ▷ Sample $\mathbf{r} \in R_q^k$ from B_{η_1}
  $\mathbf{r}[i] := \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(r, N))$
  $N := N + 1$
end for
for i from 0 to $k-1$ **do** ▷ Sample $\mathbf{e}_1 \in R_q^k$ from B_{η_2}
  $\mathbf{e}_1[i] := \mathsf{CBD}_{\eta_2}(\mathsf{PRF}(r, N))$
  $N := N + 1$
end for
$e_2 := \mathsf{CBD}_{\eta_2}(\mathsf{PRF}(r, N))$ ▷ Sample $e_2 \in R_q$ from B_{η_2}
$\hat{\mathbf{r}} := \mathsf{NTT}(\mathbf{r})$
$\mathbf{u} := \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$ ▷ $\mathbf{u} := \mathbf{A}^T\mathbf{r} + \mathbf{e}_1$
$v := \mathsf{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \mathsf{Decompress}_q(\mathsf{Decode}_1(m), 1)$ ▷ $v := \mathbf{t}^T\mathbf{r} + e_2 + \mathsf{Decompress}_q(m, 1)$
$c_1 := \mathsf{Encode}_{d_u}(\mathsf{Compress}_q(\mathbf{u}, d_u))$
$c_2 := \mathsf{Encode}_{d_v}(\mathsf{Compress}_q(v, d_v))$
return $c = (c_1 \| c_2)$ ▷ $c := (\mathsf{Compress}_q(\mathbf{u}, d_u), \mathsf{Compress}_q(v, d_v))$

рисунок 15

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

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

$\mathbf{u} \coloneqq \mathsf{Decompress}_q(\mathsf{Decode}_{d_u}(c), d_u)$
$v \coloneqq \mathsf{Decompress}_q(\mathsf{Decode}_{d_v}(c + d_u \cdot k \cdot n/8), d_v)$
$\hat{\mathbf{s}} \coloneqq \mathsf{Decode}_{12}(sk)$
$m \coloneqq \mathsf{Encode}_1(\mathsf{Compress}_q(v - \mathsf{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \mathsf{NTT}(\mathbf{u})), 1))$ ▷ $m \coloneqq \mathsf{Compress}_q(v - \mathbf{s}^T\mathbf{u}, 1))$
return m

рисунок 16

3.3. СПЕЦИФІКАЦІЯ KYBER.CCAKEM

Ми створюємо Kyber.CCAKEM, захищений IND-CCA2, KEM із захищеної IND-CPA схеми шифрування з відкритим ключем, описаної в попередньому підрозділі, за допомогою дещо зміненого перетворення Фудзісакі–Окамото. На рисунках 17, 18 і 19 ми визначаємо генерацію ключів, інкапсуляцію та декапсуляцію Kyber.CCAKEM відповідно.

Algorithm 7 KYBER.CCAKEM.KeyGen()

Output: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8 + 32}$
Output: Secret key $sk \in \mathcal{B}^{24 \cdot k \cdot n/8 + 96}$

$z \leftarrow \mathcal{B}^{32}$
$(pk, sk') \coloneqq$ KYBER.CPAPKE.KeyGen()
$sk \coloneqq (sk' \| pk \| \mathrm{H}(pk) \| z)$
return (pk, sk)

рисунок 17

Algorithm 8 KYBER.CCAKEM.Enc(pk)

Input: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8 + 32}$
Output: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
Output: Shared key $K \in \mathcal{B}^*$

$m \leftarrow \mathcal{B}^{32}$
$m \leftarrow \mathrm{H}(m)$ ▷ Do not send output of system RNG
$(\bar{K}, r) \coloneqq \mathrm{G}(m \| \mathrm{H}(pk))$
$c \coloneqq$ KYBER.CPAPKE.Enc(pk, m, r)
$K \coloneqq \mathsf{KDF}(\bar{K} \| \mathrm{H}(c))$
return (c, K)

рисунок 18

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

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

$pk := sk + 12 \cdot k \cdot n/8$
$h := sk + 24 \cdot k \cdot n/8 + 32 \in \mathcal{B}^{32}$
$z := sk + 24 \cdot k \cdot n/8 + 64$
$m' := \text{KYBER.CPAPKE.Dec}(\mathbf{s}, (\mathbf{u}, v))$
$(\bar{K}', r') := \text{G}(m' \| h)$
$c' := \text{KYBER.CPAPKE.Enc}(pk, m', r')$
if $c = c'$ **then**
    return $K := \text{KDF}(\bar{K}' \| \text{H}(c))$
else
    return $K := \text{KDF}(z \| \text{H}(c))$
end if
return K

рисунок 19

4. ТЕХНІЧНІ ХАРАКТЕРИСТИКИ АЛГОРИТМУ CRYSTAL-DILITHIUM

CRYSTALS-Dilithium цей криптографічний алгоритмом підпису забезпечує стійкість до квантових обчислювальних атак. Цей алгоритм базується на проблемі решітки (Lattice Problem) і використовує математичні операції для створення підписів, які є стійкими до атак з використанням квантових комп'ютерів. Основна мета CRYSTALS-Dilithium полягає в забезпеченні підпису повідомлень з високою стійкістю до квантових атак. Він забезпечує конфіденційність, цілісність та автентичність даних шляхом генерації підписів, які можуть бути перевірені іншими учасниками системи. Алгоритм CRYSTALS-Dilithium має важкість на основі математичних проблем решітки, що робить його стійким до атак з використанням квантових обчислювальних ресурсів. Він є одним з промислових стандартів для криптографічних алгоритмів підпису, які мають стійкість до квантових атак.

Генерація ключів. Алгоритм генерації ключа генерує $k \times \ell$ матрицю A, кожен із записів якої є поліномом у кільці $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$. Завжди матимемо $q = 2^{23} - 2^{13} + 1$ і $n = 256$. Після цього алгоритм вибирає вектори випадкових секретних ключів s_1 і s_2. Кожен коефіцієнт цих

векторів є елементом $\mathcal{R}_q$ з малими коефіцієнтами розміру не більше η. Нарешті, друга частина відкритого ключа обчислюється як $t = As_1 + s_2$. Передбачається, що всі алгебраїчні операції в цій схемі виконуються над кільцем поліномів $\mathcal{R}_q$.

Порядок підписання. Алгоритм підпису генерує маскувальний вектор поліномів y з коефіцієнтами, меншими за γ_1. Параметр γ_1 встановлюється стратегічно – він достатньо великий, щоб остаточний підпис не розкривав секретний ключ (тобто алгоритм підпису є нульовим розголошенням), але досить малий, щоб підпис було важко підробити. Потім підписувач обчислює Ay і встановлює w_1 як «старші» біти коефіцієнтів у цьому векторі. Зокрема, кожен коефіцієнт w в Ay можна записати канонічним способом у вигляді $w = w_1 \cdot 2\gamma_2 + w_0$, де $|w_0| \leq \gamma_2$; тоді w_1 є вектором, що містить усі w_1. Потім створюється виклик c як хеш повідомлення та w_1. Вихід c є поліномом від $\mathcal{R}_q$ з $\tau \pm 1$, а решта 0. Причина такого розподілу полягає в тому, що c має малу норму та походить із інтервалу розміру $\log_2(\frac{256}{\tau})$, який ми б хотіли мати між 128 і 256. Потенційна сигнатура потім обчислюється як $z = y + cs_1$.

Якщо z було б безпосередньо виведено в цей момент, то схема підпису була б незахищеною через те, що секретний ключ би витік. Щоб уникнути залежності z від секретного ключа, ми використовуємо вибірку відхилення. Параметр β встановлюється як максимально можливий коефіцієнт cs_i. Оскільки c має $\tau \pm 1$, а максимальний коефіцієнт у s_i дорівнює η, легко побачити, що $\beta \leq \tau \cdot \eta$. Якщо будь-який коефіцієнт при z більший за $\gamma_1 - \beta$, тоді ми відхиляємо та перезапускаємо процедуру підписання. Крім того, якщо будь-який коефіцієнт молодших бітів $Az - ct$ більший за $\gamma_2 - \beta$, ми перезапускаємо. Перша перевірка необхідна для безпеки, тоді як друга необхідна як для безпеки, так і для правильності. Цикл у процедурі підписання продовжує повторюватися, доки не будуть виконані

дві попередні умови. Параметри встановлюються так, щоб очікувана кількість повторень не була надто високою (наприклад, близько 4).

Перевірка. Верифікатор спочатку обчислює w'_1 як старші біти $Az - ct$, а потім приймає, якщо всі коефіцієнти z менші за $\gamma_1 - \beta$ і якщо c є хешем повідомлення та w'_1. Давайте розглянемо, чому перевірка працює, зокрема, чому $HighBits(Az - ct, 2\gamma_2) = HighBits(Ay, 2\gamma_2)$. Перше, на що слід звернути увагу, це те, що $Az - ct = Ay - cs_2$. Отже, все, що нам дійсно потрібно показати, це

$$HighBits(Ay, 2\gamma_2) = HighBits(Az - ct, 2\gamma_2)$$

Причиною цього є те, що дійсний підпис матиме $\|LowBits(Ay - cs_2, 2\gamma_2)\|_\infty < \gamma_2 - \beta\|$. І оскільки ми знаємо, що коефіцієнти cs_2 менші за β, ми знаємо, що додавання cs_2 недостатньо, щоб викликати будь-які переноси шляхом збільшення будь-якого нижчого коефіцієнта, щоб мати величину принаймні γ_2. Таким чином, рівняння вище є істинним, і підпис перевіряється правильно.

4.1. БАЗОВІ ОПЕРАЦІЇ

4.1.1. КІЛЬЦЕВІ ОПЕРАЦІЇ

Ми позначаємо $\mathcal{R}$ і $\mathcal{R}_q$ відповідно кільця $\mathbb{Z}[\boldsymbol{X}]/(\boldsymbol{X^n + 1})$ і $\mathbb{Z}_{\boldsymbol{q}}[\boldsymbol{X}]/(\boldsymbol{X^n + 1})$, де q ціле число. Надалі значення n завжди буде 256, а q буде простим числом $8380417 = 2^{23} - 2^{13} + 1$. Звичайні літери шрифту позначають елементи в $\mathcal{R}$ або $\mathcal{R}_q$ (що включає елементи в $\mathbb{Z}$ і $\mathbb{Z}_{\boldsymbol{q}}$), а літери нижнього регістру представляють вектори-стовпці з коефіцієнтами в $\mathcal{R}$ або $\mathcal{R}_q$. За замовчуванням усі вектори будуть векторами-стовпцями. Жирні великі літери є матрицями. Для вектора v позначимо через v^T його транспонування. Логічний оператор $[\![\ldots]\!]$ оцінюється як 1, якщо оператор істинний, і до 0 в іншому випадку.

Модульні скорочення. Для парного (відповідно непарного) натурального числа α ми визначаємо $\boldsymbol{r' = r \bmod^{\pm} \alpha}$ як унікальний елемент $\boldsymbol{r'}$ у діапазоні $\frac{-\boldsymbol{\alpha}}{\boldsymbol{2}} < \boldsymbol{r'} \leq \frac{\boldsymbol{\alpha}}{\boldsymbol{2}}$ (відповідно $\frac{-\boldsymbol{\alpha-1}}{\boldsymbol{2}} \leq \boldsymbol{r'} \leq \frac{\boldsymbol{\alpha-1}}{\boldsymbol{2}}$) такий, що $\boldsymbol{r' \equiv r \bmod \alpha}$. Іноді ми називатимемо це центрованим скороченням за модулем α. Для будь-якого натурального числа α ми визначаємо $\boldsymbol{r' = r \bmod \alpha}$ як унікальний елемент $\boldsymbol{r'}$ у діапазоні $0 \leq \boldsymbol{r'} < \alpha$, такий що $\boldsymbol{r' \equiv r \bmod \alpha}$. Коли точне представлення неважливе, ми просто пишемо $r \bmod \alpha$.

Розміри елементів. Для елемента $w \in \mathbb{Z}_q$ ми пишемо $\|w\|_\infty$, що означає $|w \bmod^{\pm} q|$. Визначимо ℓ_∞ та ℓ_2 норми для $w = w_0 + w_1 X + \ldots + w_{n-1} X^{n-1} \in \mathcal{R}$:

$$\|w\|_\infty = \max_i \|w_i\|_\infty, \|w\| = \sqrt{\|w\|_\infty^2 + \cdots + \|w_{n-1}\|_\infty^2}.$$

Так само для $\boldsymbol{w} = (w_0, \ldots, w_k) \in \mathcal{R}^k$ визначаємо

$$\|\boldsymbol{w}\|_\infty = \max_i \|w_i\|_\infty, \|\boldsymbol{w}\| = \sqrt{\|w\|_\infty^2 + \cdots + \|w_k\|_\infty^2}.$$

Будемо писати S_η для позначення всіх елементів $w \in \mathcal{R}$ таких, що ‖w $\|w\|_\infty \leq \eta$. Будемо писати $\widetilde{S_\eta}$, щоб позначити множину $\{w \bmod^{\pm} 2\eta : w \in \mathcal{R}\}$. Зауважте, що S_η і $\widetilde{S_\eta}$ дуже подібні, за винятком того, що $\widetilde{S_\eta}$ не містить поліномів із коефіцієнтами $-\eta$.

4.1.2. БІТИ ВИЩОГО ТА НИЖЧОГО ПОРЯДКУ

Щоб зменшити розмір відкритого ключа, нам знадобляться кілька простих алгоритмів, які витягають «вищий» і «нижчий» біти елементів у $\mathbb{Z}_{\boldsymbol{q}}$. Мета полягає в тому, щоб, якщо задано довільний елемент $r \in \mathbb{Z}_{\boldsymbol{q}}$ та інший невеликий елемент $z \in \mathbb{Z}_{\boldsymbol{q}}$, ми хотіли б мати можливість відновити біти вищого порядку $r + z$ без необхідності зберігати z. Тому ми визначаємо алгоритми, які беруть r, z і створюють однобітну підказку h, яка дозволяє

обчислювати біти вищого порядку $r + z$ просто за допомогою r і h. Ця підказка, по суті, є «перенесенням», викликаним z у додаванні.

Є два різні способи, якими ми розбиваємо елементи в $\mathbb{Z}_q$ на біти «старшого порядку» та біти «нижчого порядку». Перший алгоритм, $Power2Round_q$, є простим побітовим способом розбиття елемента $r = r_1 \cdot 2^d + r_0$, де $r_0 = r \bmod^{\pm} 2^d$ і $r_1 = (r - r_0)/2^d$.

Зауважте, що якщо ми вибираємо представниками r_1 невід'ємні цілі числа від 0 до $\lfloor q/2^d \rfloor$, тоді відстань (за модулем q) між будь-якими двома $r_1 \cdot 2^d$ і $r'_1 \cdot 2^d$ зазвичай $\geq 2^d$. Зокрема, відстань за модулем q між $\lfloor q/2^d \rfloor \cdot 2^d$ і 0 може бути дуже малою. Це проблематично у випадку, якщо ми хочемо створити однобітну підказку, оскільки додавання невеликого числа до r може призвести до зміни старших бітів r більш ніж на 1.

Ми уникаємо зміни старших бітів більш ніж на 1 за допомогою простого налаштування. Ми вибираємо α, який є дільником $q - 1$, і пишемо $r = r_1 \cdot \alpha + r_0$ так само, як і раніше. Для простоти припустимо, що α парне (що можливо, оскільки q непарне). Можливі $r_1 \cdot \alpha$ тепер дорівнюють $\{0, \alpha, 2\alpha, \ldots, q - 1\}$. Зауважте, що відстань між $q1$ і 0 дорівнює 1, тому ми видаляємо $q - 1$ із набору можливих $r_1 \cdot \alpha$ і просто округлюємо відповідні r до 0. Оскільки $q - 1$ і 0 відрізняються на 1, усе це це, можливо, збільшить величину залишку r_0 на 1. Ця процедура називається $Decompose_q$. Використовуючи цю процедуру як підпрограму, ми можемо визначити підпрограми $MakeHint_q$ і $UseHint_q$, які створюють підказку та, відповідно, використовують підказку для відновлення старших бітів суми. Для зручності нотації ми також визначаємо підпрограми HighBitsq і LowBitsq, які просто витягують r_1і r_0відповідно з результату $Decompose_q$.

Наведені нижче леми визначають властивості цих допоміжних алгоритмів, які необхідні для коректності та безпеки нашої схеми.

Лема 1. Нехай q і α є натуральними числами, що задовольняють $q > 2\alpha$, $q \equiv 1\ (mod\ \alpha)$ і α парні. Нехай r і z — вектори елементів у $\mathcal{R}_q$, де $\|z\|_\infty \leq \alpha/2$, і нехай h, h' — вектори бітів. Тоді алгоритми $HighBits_q$, $MakeHint_q$ і $UseHint_q$ задовольняють такі властивості:

1. $UseHint_q\big(MakeHint_q(\mathrm{z}, \mathrm{r}, \alpha), \mathrm{r}, \alpha\big) = HighBits_q(\mathrm{r} + \mathrm{z}, \alpha)$.

2. Нехай $v_1 = UseHint_q(\mathrm{h}, \mathrm{r}, \alpha)$. Тоді $\|r - v_1 \cdot \alpha\|_\infty$. Крім того, якщо кількість одиниць у h дорівнює w, то всі, окрім щонайбільше w коефіцієнтів $r - v_1 \cdot \alpha$ матимуть величину не більше $\alpha/2$ після центрованого скорочення по модулю q.

3. Для будь-яких h, h', якщо $UseHint_q(h, r, \alpha) = UseHint_q(h', r, \alpha)$, то $h = h'$.

Лема 2. Якщо $\|s\|_\infty \leq \beta$ та $\left\|LowBits_q(r, \alpha)\right\|_\infty \leq \frac{\alpha}{2} - \beta$, то

$$HighBits_q(r, \alpha) = HighBits_q(r + s, \alpha)$$

Лема 3. Нехай $(r_1, r_0) = Decompose_q(r, \alpha)$, $(w_1, w_0) = Decompose_q(r + s, \alpha)$ і $\|s\|_\infty \leq \beta$. Потім $\|s\|_0 \leq \frac{\alpha}{2} - \beta \Leftrightarrow w_1 = r_1 \wedge \|w\|_\infty \leq \frac{\alpha}{2} - \beta$.

$\underline{\mathsf{Power2Round}_q(r, d)}$
08 $r := r \bmod^+ q$
09 $r_0 := r \bmod^\pm 2^d$
10 **return** $\big((r - r_0)/2^d, r_0\big)$

$\underline{\mathsf{MakeHint}_q(z, r, \alpha)}$
11 $r_1 := \mathsf{HighBits}_q(r, \alpha)$
12 $v_1 := \mathsf{HighBits}_q(r + z, \alpha)$
13 **return** $[\![r_1 \neq v_1]\!]$

$\underline{\mathsf{UseHint}_q(h, r, \alpha)}$
14 $m := (q - 1)/\alpha$
15 $(r_1, r_0) := \mathsf{Decompose}_q(r, \alpha)$
16 **if** $h = 1$ and $r_0 > 0$ **return** $(r_1 + 1) \bmod^+ m$
17 **if** $h = 1$ and $r_0 \leq 0$ **return** $(r_1 - 1) \bmod^+ m$
18 **return** r_1

$\underline{\mathsf{Decompose}_q(r, \alpha)}$
19 $r := r \bmod^+ q$
20 $r_0 := r \bmod^\pm \alpha$
21 **if** $r - r_0 = q - 1$
22 **then** $r_1 := 0$; $r_0 := r_0 - 1$
23 **else** $r_1 := (r - r_0)/\alpha$
24 **return** (r_1, r_0)

$\underline{\mathsf{HighBits}_q(r, \alpha)}$
25 $(r_1, r_0) := \mathsf{Decompose}_q(r, \alpha)$
26 **return** r_1

$\underline{\mathsf{LowBits}_q(r, \alpha)}$
27 $(r_1, r_0) := \mathsf{Decompose}_q(r, \alpha)$
28 **return** r_0

рисунок 20

4.2. СПЕЦИФІКАЦІЯ АЛГОРИТМУ

Опираючи на означення наведені у попередніх підрозділах, запишемо декілька базових операцій:

Алгоритми генерації, підписання та перевірки ключів для нашої схеми підпису представлені на рисунку 21. Представлено детерміновану версію схеми, у якій випадковість, що використовується в процедурі підписання, генерується (з використанням SHAKE-256) як детермінована функція повідомлення та маленький секретний ключ. Оскільки можливо нашу процедуру підписання знадобитися повторити кілька разів, доки не буде створено підпис, ми також додаємо лічильник, щоб вихід SHAKE-256 відрізнявся при кожній спробі підписання того самого повідомлення. Крім того, у зв'язку з тим, що кожне повідомлення може потребувати кількох ітерацій для підписання, ми обчислюємо початковий дайджест повідомлення за допомогою хеш-функції, стійкої до зіткнень, і використовуємо цей дайджест замість повідомлення під час процедури підписання.

Основна складність, пов'язана з відсутністю всього t у відкритому ключі, полягає в тому, що алгоритм верифікації більше не може точно обчислити w'_1. Щоб зробити це, алгоритм верифікації потребуватиме високого порядку бітів $Az - ct$, але він може лише обчислити $Az - ct_1 \cdot 2d = Az - ct + ct_0$. Незважаючи на те, що старші біти ct_0 дорівнюють 0, його присутність у сумі створює «перенесення», які можуть впливати на старші біти. Таким чином, підписувач надсилає ці переноси як підказку верифікатору. Виходячи з вибору параметрів, не повинно бути більше ніж ω позицій, у яких викликається перенесення. Таким чином, підписувач просто надсилає позиції, в яких відбуваються ці переноси (це додаткові байти в $Dilithium$), що дозволяє верифікатору обчислити старші біти $Az - ct$.

Gen

$\zeta \leftarrow \{0,1\}^{256}$

$(\rho, \varsigma, K) \in \{0,1\}^{256\times 3} := \mathrm{H}(\zeta)$ ▷ H is instantiated as SHAKE-256 throughout

$(\mathbf{s}_1, \mathbf{s}_2) \in S_\eta^\ell \times S_\eta^k := \mathrm{H}(\varsigma)$

$\mathbf{A} \in R_q^{k\times\ell} := \mathsf{ExpandA}(\rho)$ ▷ $\mathbf{A}$ is generated and stored in NTT Representation as $\hat{\mathbf{A}}$

$\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ ▷ Compute $\mathbf{A}\mathbf{s}_1$ as $\mathtt{NTT}^{-1}(\hat{\mathbf{A}} \cdot \mathtt{NTT}(\mathbf{s}_1))$

$(\mathbf{t}_1, \mathbf{t}_0) := \mathsf{Power2Round}_q(\mathbf{t}, d)$

$tr \in \{0,1\}^{384} := \mathsf{CRH}(\rho \parallel \mathbf{t}_1)$

return $(pk = (\rho, \mathbf{t}_1), sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0))$

Sign(sk, M)

$\mathbf{A} \in R_q^{k\times\ell} := \mathsf{ExpandA}(\rho)$ ▷ $\mathbf{A}$ is generated and stored in NTT Representation as $\hat{\mathbf{A}}$

$\mu \in \{0,1\}^{384} := \mathsf{CRH}(tr \parallel M)$

$\kappa := 0$, $(\mathbf{z}, \mathbf{h}) := \perp$

$\rho' \in \{0,1\}^{384} := \mathsf{CRH}(K \parallel \mu)$ (or $\rho' \leftarrow \{0,1\}^{384}$ for randomized signing)

while $(\mathbf{z}, \mathbf{h}) = \perp$ do ▷ Pre-compute $\hat{\mathbf{s}}_1 := \mathtt{NTT}(\mathbf{s}_1)$, $\hat{\mathbf{s}}_2 := \mathtt{NTT}(\mathbf{s}_2)$, and $\hat{\mathbf{t}}_0 := \mathtt{NTT}(\mathbf{t}_0)$

  $\mathbf{y} \in \tilde{S}_{\gamma_1}^\ell := \mathsf{ExpandMask}(\rho', \kappa)$

  $\mathbf{w} := \mathbf{A}\mathbf{y}$ ▷ $\mathbf{w} := \mathtt{NTT}^{-1}(\hat{\mathbf{A}} \cdot \mathtt{NTT}(\mathbf{y}))$

  $\mathbf{w}_1 := \mathsf{HighBits}_q(\mathbf{w}, 2\gamma_2)$

  $\tilde{c} \in \{0,1\}^{256} := \mathsf{H}(\mu \parallel \mathbf{w}_1)$

  $c \in B_\tau := \mathsf{SampleInBall}(\tilde{c})$ ▷ Store c in NTT representation as $\hat{c} = \mathtt{NTT}(c)$

  $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$ ▷ Compute $c\mathbf{s}_1$ as $\mathtt{NTT}^{-1}(\hat{c} \cdot \hat{\mathbf{s}}_1)$

  $\mathbf{r}_0 := \mathsf{LowBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)$ ▷ Compute $c\mathbf{s}_2$ as $\mathtt{NTT}^{-1}(\hat{c} \cdot \hat{\mathbf{s}}_2)$

  if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$, then $(\mathbf{z}, \mathbf{h}) := \perp$

  else

    $\mathbf{h} := \mathsf{MakeHint}_q(-c\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0, 2\gamma_2)$ ▷ Compute $c\mathbf{t}_0$ as $\mathtt{NTT}^{-1}(\hat{c} \cdot \hat{\mathbf{t}}_0)$

    if $\|c\mathbf{t}_0\|_\infty \geq \gamma_2$ **or** the # of 1's in $\mathbf{h}$ is greater than ω, then $(\mathbf{z}, \mathbf{h}) := \perp$

  $\kappa := \kappa + \ell$

return $\sigma = (\mathbf{z}, \mathbf{h}, \tilde{c})$

Verify$(pk, M, \sigma = (\mathbf{z}, \mathbf{h}, \tilde{c}))$

$\mathbf{A} \in R_q^{k\times\ell} := \mathsf{ExpandA}(\rho)$ ▷ $\mathbf{A}$ is generated and stored in NTT Representation as $\hat{\mathbf{A}}$

$\mu \in \{0,1\}^{384} := \mathsf{CRH}(\mathsf{CRH}(\rho \parallel \mathbf{t}_1) \parallel M)$

$c := \mathsf{SampleInBall}(\tilde{c})$

$\mathbf{w}_1' := \mathsf{UseHint}_q(\mathbf{h}, \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2)$ ▷ Compute as $\mathtt{NTT}^{-1}(\hat{\mathbf{A}} \cdot \mathtt{NTT}(\mathbf{z}) - \mathtt{NTT}(c) \cdot \mathtt{NTT}(\mathbf{t}_1 \cdot 2^d))$

return $[\![\|\mathbf{z}\|_\infty < \gamma_1 - \beta]\!]$ **and** $[\![\tilde{c} = \mathsf{H}(\mu \parallel \mathbf{w}_1')]\!]$ **and** $[\![\#\text{ of 1's in } \mathbf{h} \text{ is } \leq \omega]\!]$

рисунок 21

5. ПРОГРАМНА РЕАЛІЗАЦІЯ БІБЛІОТЕКИ

5.1. ІНСТРУМЕТИ РЕАЛІЗАЦІЇ

Для реалізації бібліотеки алгоритмів CRYSTAL-KYBER і CRYSTAL-DILITHIUM було обрано мову програмування Swift. Було використано менеджер залежностей CocoaPods. В якості інтегрованого середовища розробки було використано Xcode. Це інтегроване середовище має велике кількість плагінів для розробки, але мова програмування Swift є вбудованною в цьому середовищі, тож додаткових плагінів не використовувалося.

5.2. ОПИС РЕАЛІЗАЦІЇ

Було створено наступні допоміжні структури для роботи з поліномами та ключами:

- **Polynomial** – має в собі поле coefficients, яке є масивом, що складається зі значень Int. У масиві впорядковані значення коефіцієнтів. Наприклад, на місці першого елемента стоїть значення коефіцієнта біля найбільшого степення.
- **PolynomialVector** – поле polynomials (масив з типом елементів Polynomial)
- **PolynomialMatrix** – поле polynomials (масив масивів з типом елементів Polynomial)
- **PrivateKey** – поле vector (PolynomialVector)
- **PublicKey** – перше поле vector (PolynomialVector), друге – matrix (PolynomialMatrix)
- **Keys** – перше поле publicKey (PublicKey), друге – privateKey (PrivateKey)

Було створено два основних класи Kyber і Dilithium для інкапсуляції роботи з алгоритмами.

Ці класи мають по три функції:

- **keyGen** – генерує ключі та повертає кортеж зі двох значень (відкритий та закритий ключ)
- **decode** – закодовує повідомлення за допомогою ключів.
- **encode** – розкодовує повідомлення за допомогою ключів.

Також була реалізована допоміжна логіка для роботи з поліномами, генерацією випадкових поліномів і базовими операціями над ними.

Отже, реалізовану бібліотеку, яку можливо підключити за допомогою менеджера залежностей CocoaPods. Для цього у файлі Podfiles потрібно дописати **pod 'PQCryptoKit'**.

ВИСНОВОК

У даній роботі було проведено дослідження та розробка iOS-фреймворку для підтримки постквантових алгоритмів шифрування та цифрового підпису. Об’єктом дослідження були існуючі постквантові алгоритми, а об’єктом розробки – реалізація одного алгоритму шифрування та одного алгоритму цифрового підпису на платформі iOS з використанням мови програмування Swift.

Метою роботи було проведення детального аналізу існуючих постквантових алгоритмів та їх характеристик, а також побудова та реалізація iOS-фреймворку, який би забезпечував підтримку цих алгоритмів на платформі iOS. Для досягнення цієї мети були використані методи аналізу алгоритмів та проектування архітектури бібліотеки.

У результаті проведеного дослідження було отримано глибоке розуміння різноманітних постквантових алгоритмів та їх основних властивостей. Застосування мови програмування Swift та середовища розробки Xcode дозволило успішно реалізувати підтримку CRYSTAL-KYBER та CRYSTAL-DILITHIUM в рамках розробленого iOS-фреймворку.

Отриманий iOS-фреймворк має великий потенціал для застосування в сфері криптографії на платформі iOS. Він забезпечує надійну та ефективну підтримку постквантових алгоритмів шифрування та цифрового підпису, що сприяє забезпеченню безпеки та захисту інформації. Розроблений фреймворк може бути використаний у реальних програмних продуктах, що потребують високого рівня безпеки.

Цей фреймворк може бути початковою основою для розширення функціональності, включення додаткових алгоритмів та оптимізації роботи з постквантовими протоколами. Такі подальші вдосконалення можуть значно поліпшити ефективність та надійність криптографічних застосунків

на платформі iOS і сприяти їх широкому використанню у сучасному цифровому світі.

Загалом, результати цієї роботи вносять вагомий внесок у розвиток постквантової криптографії на платформі iOS. Отриманий iOS-фреймворк демонструє успішну реалізацію постквантових алгоритмів шифрування та цифрового підпису, відкриваючи нові можливості для захисту інформації в програмних продуктах на iOS.

СПИСОК ДЖЕРЕЛ

1. CSRC - Проекти постквантової криптографії:
   https://csrc.nist.gov/Projects/post-quantum-cryptography/
2. Classic McEliece:
   https:classic.mceliece.org/
3. PQCRYPTO - Проект з постквантової криптографії:
   https://pq-crystals.org/
4. NTRU Cryptosystems:
   https://ntru.org/
5. SABER - Криптографічна система ESAT KU Leuven:
   https://www.esat.kuleuven.be/cosic/pqcrypto/saber/
6. FALCON - Криптографічна система Falcon:
   https://falcon-sign.info/
7. PQCrypto Rainbow:
   https://www.pqcrainbow.org/
8. BIKE - Криптографічна система BIKE:
   https://bikesuite.org/
9. FrodoKEM - Криптографічна система FrodoKEM:
   https://frodokem.org/
10. PQC-HQC:
    http://pqc-hqc.org/
11. NTRU Prime:
    https://ntruprime.cr.yp.to/
12. SIKE - Криптографічна система SIKE:
    https://sike.org/
13. GeMSS - Генератори еквідистантних мультиполів:
    https://www-polsys.lip6.fr/Links/NIST/GeMSS.html
14. SPHINCS - Криптографічна система SPHINCS:
    https://sphincs.org/